Diophantine Satisfiability Arguments for Private Blockchains

Master Thesis

Author(s):
Meier, Jonas

Publication date:
2022

Permanent link:
https://doi.org/10.3929/ethz-b-000571918

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Diophantine Satisfiability Arguments for Private Blockchains

Master Thesis

Jonas Meier

August 20, 2022

Advisors: Prof. Dr. Kenny Paterson, Dr. Patrick Towa

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

Abstract

Zero-knowledge proofs have been introduced in the 1980s, and most of the initial interest was of theoretical nature. Since the development of Bitcoin [25], however, blockchain technology continues to produce exciting use-cases for zero-knowledge proofs. Prominent examples include privacy-coins, such as Zcash [2] and Monero [1], where zero-knowledge proofs serve as a tool to build confidential transactions.

Zcash utilizes the the Groth16 proof system [19]. Groth16 offers small, constant-sized proofs, but it requires an elaborate setup and its security is proven only in the generic group model.

In this thesis, we consider an alternative proof system developed by Towa and Vergnaud. [29] The proof system allows to argue knowledge of a solution to a Diophantine equation and its proof size grows logarithmically with the size of the equation. Our main contribution is to evaluate the performance of this proof system as a replacement of Groth16 in Zcash. To do so, the security statements of Zcash are expressed as a Diophantine equation, and the proof size is estimated using the theoretical upper bounds derived by Towa and Vergnaud. In addition, we implement a prototype version of TV20 that allows to observe the actual proof size in practice.

As a final contribution, we show how the verification time of Towa and Vergnaud's proof system can be lowered if one is willing to rely on the non-standard *adaptive root assumption*, introduced by Wesolowski in 2018 [30].

Contents

Acknowledgement

First, I would like to thank my co-supervisor Dr. Patrick Towa for his guidance and support during the thesis. He always took the time to make our meetings helpful and relaxed, and managed to get me back on track whenever I was stuck. Countless of his ideas and insights shaped this thesis into what it is now.

I would also like to thank Prof. Dr. Kenny Paterson, not only for giving me the opportunity to write a thesis in the Applied Cryptography group, but also for teaching the "Applied Cryptography" lecture in such a fun and engaging way.

Finally, a great thank you to my family, my girlfriend and all my friends for everything that they have done for me.

Chapter 1

Introduction

When zero-knowledge proofs were introduced for the first time, the main focus was on theoretical results. Research was primarily concerned with feasibility and other complexity-theoretic insights while actual applications were sparse. The advent of blockchain technology has changed the zero-knowledge landscape, in the sense that it gave (and continues to give) rise to numerous practical use-cases. Prominent examples include privacy coins such as Zcash [2] or Monero [1], where zero-knowledge proofs enable transactions that protects user's data.

For simple problems, like proving knowledge of a discrete logarithm in a group, zero-knowledge protocols specifically designed for the problem are available [26]. However, the statements that are proven in zero-knowledge in blockchain applications differ, as they are extremely complex. It is therefore impractical to design a zero-knowledge proof system directly for such a statement. Instead, proof systems for very expressive problems (say, NP-complete ones) are used, and the concrete statements are transformed to the appropriate input format using a reduction-style approach.

As an additional complication, most applications in the context of blockchain are performance sensitive, with the most important parameter being the proof size. Proof size is important when the proofs are stored on the blockchain, as it is desirable to minimize the amount of data that is stored on the blockchain.

When proving statements in the above reduction-style way, there are two orthogonal factors that affect the performance. First, the efficiency of encoding the concrete problem to the input format of the proof system and second, the efficiency of the underlying proof system.

Circuit satisfiability. A popular choice of such an expressive problem is circuit satisfiability over a finite field $\mathbb{Z}_p$, which is NP-complete. As such, a proof system for circuit satisfiability allows to prove any NP-statement,

making it an extremely versatile choice. For example, the proof system can be used to prove knowledge of a preimage to some hash value, by encoding the hash computation as a boolean circuit, but it can also be used to demonstrate knowledge of a discrete logarithm in an elliptic curve group, in this case by encoding the group operation as a circuit.

One of the most efficient proof systems for circuit satisfiability over a finite field was developed by Jens Groth in 2016, and is commonly referred to as *Groth16* [19]. Groth16 is the proof system that is currently used in the Zcash cryptocurrency.

Groth16 is based on pairing-based cryptography and has constant-sized proofs (i.e. the proof size does not depend on the particular circuit). However, it requires an elaborate trusted setup to be performed for each individual circuit. An additional drawback of Groth16 is that its security is proven in the so-called generic group model. The generic group model is an idealized model of computation introduced by Shoup [28]. In the generic group model, an adversary can perform group operations only via an oracle and does not have access to the concrete representation of a group. In practice, however, the representation of a group is always known to the adversary, and offers potential attack vectors that are not covered when proving security in the generic group model.

In this thesis, we consider a proof system developed by Towa and Vergnaud [29], which we refer to as *TV20*. TV20 is not a proof system for circuit satisfiability over a finite field, but instead allows to prove knowledge of a solution to a Diophantine equation. The proof size grows logarithmically with the size of the equation, and the security of TV20 is based on standard assumptions about hidden order groups (section 2.5).

The goal of this thesis is to evaluate TV20 as a possible replacement for Groth16 in the Zcash cryptocurrency. As Groth16 has constant-sized proofs, it is improbable that TV20 (having logarithmically sized-proofs) improves on the proof size. Nevertheless, the weaker security assumptions of TV20 might render it an attractive alternative to Groth16.

Additionally, the fact that TV20 argues about Diophantine satisfiability and therefore works directly with integers instead of finite field elements, is advantageous in certain scenarios. This is because in general, when proving statements over the integers using finite circuit satisfiability, an upper-bound on the numbers occurring during the computation must be known, and an appropriately large finite field has to be chosen. As TV20 operates directly on integers, it does not suffer from the same problem.

Organization. Chapter 2 introduces notation and preliminary concepts that are used in the rest of the thesis. Chapter 3 describes the Zcash cryptocurrency and its transaction structure. Our main contribution is chapter 4,

which shows how to encode the Zcash zero-knowledge statements as a Diophantine equation and evaluates the performance of TV20 when proving these statements. It also contains benchmark results of a prototype implementation of TV20. Chapter 5 develops a technique to speed up the verification time of TV20, and chapter 6 concludes the thesis by giving a summary of the results and proposing future work.

Chapter 2

Preliminaries

2.1 Notation

All logarithms are in base 2. For a natural number $n \in \mathbb{N}$, we define $[n] = \{1, \ldots, n\}$. Generic groups are denoted $\mathbb{G}$, and the number of bits required to represent a group element is written as $b_{\mathbb{G}}$. Notation is abused, as $\mathbb{G}$ is used both for the set of elements of a group, as well as the algebraic structure. For a value x, $|x|$ denotes the absolute value of x. For a set S, $|S|$ denotes the cardinality of S. Vectors are written as bold lowercase letters, for example $\mathbf{v}$. Subscripts are used to access individual entries of vectors, e.g. $\mathbf{v}_1$. Matrices are written as bold uppercase letters, for example $\mathbf{M}$. For a vector $\mathbf{v}$ or a matrix $\mathbf{M}$, $|\mathbf{v}|_\infty$ resp. $|\mathbf{M}|_\infty$ denotes the maximum entry. For a vector $\mathbf{v} = \{v_1, \ldots, v_n\}$ of even length, $\mathbf{v}_1$ and $\mathbf{v}_2$ denote the first and second halves respectively, i.e. $\mathbf{v}_1 = \{v_1, \ldots, v_{n/2}\}$ and $\mathbf{v}_2 = \{v_{n/2+1}, \ldots, v_n\}$. For a vector $\mathbf{v} = \{v_1, \ldots, v_n\}$ and a value x, $\mathbf{v}^x$ is the result of raising the individual entries of $\mathbf{v}$ to the power x, i.e. $\mathbf{v}^x = \{v_1^x, \ldots, v_n^x\}$. For two vectors $\mathbf{v} = \{v_1, \ldots, v_n\}, \mathbf{u} = \{u_1, \ldots, u_n\}$ of the same length, $\mathbf{v} \circ \mathbf{u}$ denotes their Hadamard product, i.e. $\mathbf{v} \circ \mathbf{u} = \{v_1 \cdot u_1, \ldots, v_n \cdot u_n\}$. For any two vectors $\mathbf{v} = \{v_1, \ldots, v_{n_1}\}, \mathbf{u} = \{u_1, \ldots, u_{n_2}\}$, $[\mathbf{v}, \mathbf{u}] = \{v_1, \ldots, v_{n_1}, u_1, \ldots, u_{n_2}\}$ denotes their concatenation.

2.2 Commitment Schemes

This section introduces non-interactive commitment schemes in an informal fashion. A non-interactive commitment scheme enables a party to *commit* to a secret value, and then later *open* the commitment and reveal the committed value. Intuitively, there are two properties that the scheme should satisfy. First, the act of committing should not reveal any information about the committed value. Second, it should only be possible to open a commitment to a single value, namely the one that was committed in the first place.

A bit more formally, a non-interactive commitment scheme is a pair of functions commit and open. commit takes as input a value m to be committed to and outputs a commitment cm and decommitment information d. The function open takes as input a value m, a commitment cm and decommitment information d and outputs a bit to indicate whether the opening is accepted. The following conditions must hold.

- **Correctness.** If $(cm, d) = \mathsf{commit}(m)$, then $\mathsf{open}(m, cm, d) = 1$
- **Hiding.** If $(cm, d) = \mathsf{commit}(m)$ and $(cm', d') = \mathsf{commit}(m')$, the distributions of cm and cm' are indistinguishable.
- **Binding.** It is infeasible to find a commitment cm and pairs $(m, d), (m', d')$ with $m \neq m'$ such that $\mathsf{open}(m, cm, d) = 1$ and $\mathsf{open}(m', cm, d') = 1$.

In practice, the majority of non-interactive commitment schemes (for example, all the commitment schemes used in Zcash, see sections 2.8.5, 2.8.6) stick to the following pattern. The function commit is a randomized function and the randomness r is sampled uniformly at random from some randomness space. The randomness r is often written explicitly as a subscript, i.e. a commitment to m is computed as $\mathsf{commit}_r(m)$. The decommitment information is then set to (m, r). To open, m and r are revealed and the commitment is recomputed using the fixed randomness. If it matches the original commitment, the opening is accepted, otherwise it is rejected.

Homomorphic commitments. A special type of commitment schemes are so-called *homomorphic* commitment schemes. A commitment scheme is called additively homomorphic, if commitments can be added to get a commitment for the sum of the values, i.e.

$$\mathsf{commit}_r(m) + \mathsf{commit}_{r'}(m') = \mathsf{commit}_{r+r'}(m + m'),$$

where we assume that the commitment scheme follows the aforementioned pattern. Essentially, if a party is committed to two values using a homomorphic commitment scheme, the party is automatically committed to the sum of the two values. This can be useful in various cryptographic scenarios.

2.3 Merkle Trees

A Merkle tree [24] is a data structure that can be used to quickly demonstrate membership in a set. It is constructed in the following way.

Let $S = \{v_1, \ldots, v_n\} \subset \{0,1\}^*$ be a set of n bitstrings, and let k be such that $n \leq 2^k$. Also, let $H : \{0,1\}^* \mapsto \{0,1\}^l$ be a collision-resistant hash function. A Merkle tree for the set S is a binary tree of depth k in which the leaves contain the hashes of members of S, and empty leaves (if $n \neq 2^k$) are initialized with a default value. Inner nodes contain the hash of the

concatenation of their two child nodes (see figure 2.1). The value of the root node is called the *root value*, denoted rt.

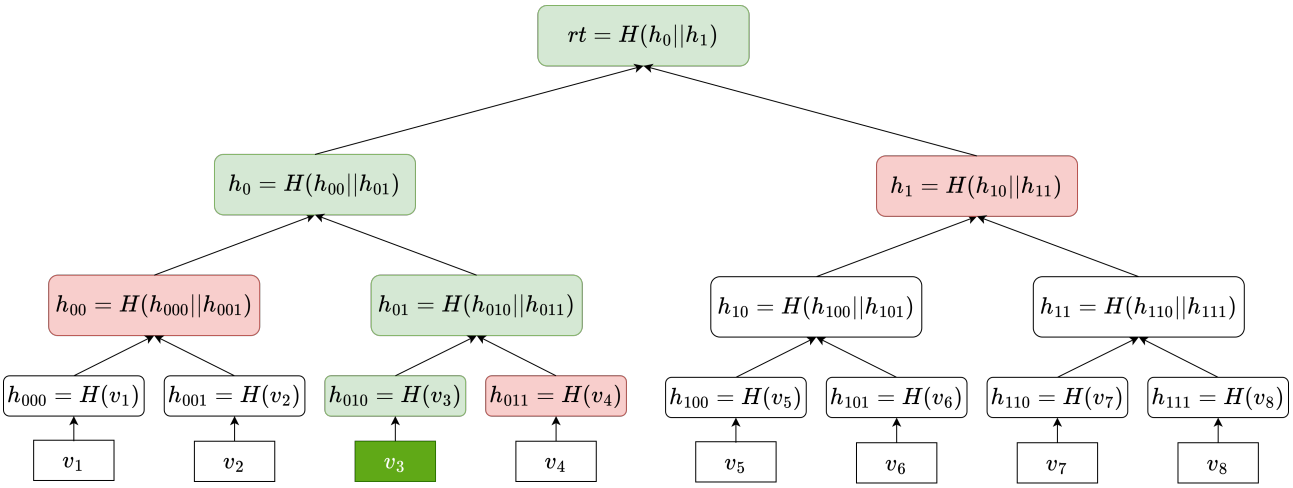

Figure 2.1: An example Merkle tree for the set $S = \{v_1, \ldots, v_8\}$. The path of v_3 towards the root is marked in green, its co-path in red.

One of the main features of a Merkle tree is that it is easy to proof that a value is in the set S. Assume that a party has an authentic copy of the root value rt. Then, a Merkle tree allows one to convince this party that some value v is contained in S in time $\mathcal{O}(k)$ as follows.

The idea is to prove that there is a leaf in the Merkle tree that contains v. To do this, one would provide the so-called *co-path* of the leaf, as well as the position of the leaf. Essentially, the co-path includes the contents of all intermediate nodes that are needed to re-compute the root hash value, which are exactly the sibling nodes of the nodes on the path from v towards the root (see figure 2.1). Note that the co-path has length linear in k, resp. logarithmic in n. Then, the verifying party re-computes the root value, by iterated hashing. If the re-computed root value matches the actual root value, the party is convinced that v is contained in the set S.

In order to do fraudulent proofs, i.e. convince somebody that a value v' is contained in the tree even though it is not, one would have to break the collision resistance of the underlying hash function. Intuitively, this is because the process of iterated hashing would have to result in the same output (the recomputed root value and real root value match), even though the input is different. This implies that one can find a collision in some step of the process.

2.4 Zero-Knowledge Proofs

This section introduces zero-knowledge proofs. Subsection 2.4.1 provides an overview over interactive proofs in general and gives the reader some broader context. In subsection 2.4.2, zero-knowledge proofs of knowledge are formally defined.

2.4.1 A Brief History of Zero-knowledge Proofs

Interactive Proofs

Interactive proof systems were introduced back in the mid-80s by Babai [5] and Goldwasser, Micali and Rackoff [17]. An interactive proof of a statement is an interactive protocol between a prover and a verifier. During the protocol, the prover proves the validity of a mathematical statement to the verifier. To be a bit more formal, for a language $\mathcal{L}$ and a value x, the prover convinces the verifier that $x \in \mathcal{L}$. While the prover is allowed unbounded computing power, the verifier is constrained to run in probabilistic polynomial-time. Intuitively, an interactive proof system must satisfy the following conditions:

- **Completeness:** For an $x \in \mathcal{L}$, the honest prover convinces the verifier.
- **Soundness:** For an $x \notin \mathcal{L}$, no prover can convince the verifier.

The above statements are of course informal and the precise definitions are probabilistic. In particular, a small probability that a malicious prover convinces a verifier of a false statement must be allowed.

Recall that NP can be defined as the set of languages for which efficiently verifiable witnesses exists. Interactive proofs can be seen as a generalization of NP, in the sense that the witness is not a string, but instead an interactive algorithm. Accordingly, the validity of a witness is checked interactively as well. The set of all languages for which an interactive proof system exists is denoted IP.

A natural question is how IP relates to other complexity classes. It is easy to see that $\mathsf{NP} \subseteq \mathsf{IP}$. This is because it is possible to build an interactive proof system for any NP-language: For an $x \in \mathcal{L}$, the (unbounded) prover simply computes and sends over a witness w to the verifier, which can then be verified efficiently by the verifier. No witnesses exist for non-members of the language, which implies soundness.

A major complexity-theoretic result for interactive proofs is that $\mathsf{IP} = \mathsf{PSPACE}$, where PSPACE is the set of all languages that can be decided using polynomial amount of space, but unbounded time. This result has been proven by Shamir [27], building on previous work by Lund et. al [22].

Zero-Knowledge Proofs

Zero-knowledge proofs [17] are interactive proof systems that satisfy an additional property, called the *zero-knowledge* property. Intuitively, the zero-knowledge property states the following

- **Zero-Knowledge:** During the proof of a statement, no information is revealed besides the fact that the statement is true.

To formalize the zero-knowledge property (section 2.4.2), one makes use of a simulator. Essentially, the simulator must be able to efficiently generate a transcript that looks indistinguishable from the real interaction between the prover and the verifier. The indistinguishability can be perfect, statistical or computational, giving rise to the notions of perfect, statistical or computational zero-knowledge.

Reconsider the simple interactive proof for a language $\mathcal{L} \in \mathsf{NP}$, where the prover simply sends over a witness w and the verifier checks whether this is a valid witness for x. This protocol is not necessarily zero-knowledge. The verifier does not just learn that $x \in \mathcal{L}$, but also receives a valid witness w, which can be difficult to compute in general.

Proofs of Knowledge

Besides interactive proofs of statements, there is a second kind of interactive proofs, called interactive proofs *of knowledge*. Instead of proving validity of a mathematical statement, in an interactive proof of knowledge, the prover convinces the verifier that it knows some value satisfying a mathematical relation. More formally, for a relation $\mathcal{R}$ and a public value x, the prover convinces the verifier that it knows a value w such that $(x, w) \in \mathcal{R}$.

Often, $\mathcal{R}$ is and NP-relation, containing pairs of problem instances and witnesses. For example, consider the boolean satisfiability problem. In this case, the respective relation would be defined as all pairs (x, w) of formulas x and assignments w, so that w satisfies x. A proof of knowledge would allow a prover to demonstrate that it knows a satisfying assignment to a given boolean formula.

Notice the conceptual difference between the two types of interactive proofs: In an interactive proof of statement, the prover may prove that a boolean formula is satisfiable, i.e. there exists a satisfying assignment, while in an interactive proof of knowledge, the prover proves that it *knows* a satisfying assignment (in this case, implicitly proving the satisfiability of the formula as well).

Formalizing the knowledge property is achieved using a so-called *knowledge extractor*. Intuitively, a knowledge extractor is an algorithm that interacts

with the prover and extracts the value that the prover is supposed to know. Conceptually, the extractability property states the following.

- **Extractability:** There exists an efficient algorithm which, upon interaction with any successful prover, outputs w such that $(x, w) \in \mathcal{R}$.

The extractability property replaces the soundess property. So, a proof of knowledge must be complete and extractable, while a proof of statement must be complete and sound. In a rigorous definition of extractability (originally by Bellare and Goldreich [6]), the running time of the extractor must depend on the success probability of the prover (see section 2.4.2).

Non-interactive Zero-knowledge Proofs

One of the most attractive features of interactive proofs and arguments is the zero-knowledge property: The ability to prove the validity of a statement without revealing additional information. A natural question to ask is whether zero-knowledge is also achievable for non-interactive proofs or, under which assumptions an interactive proof can be turned into a non-interactive one, without sacrificing the zero-knowledge property. The ability to do so would be of great value as requiring interaction is challenging in many settings. As it turns out, it is possible. The following paragraphs describe two of the most popular options to remove interaction.

CRS model. Blum, Feldman and Micali were to first to consider non-interactive zero-knowledge [11]. They use the so-called *common reference string* model, or CRS model. In the CRS model, the prover and verifier have access to a common string, generated by a trusted party according to some probability distribution. Then, to create a proof non-interactively, the prover essentially simulates the verifier, using of the shared randomness. When verifying the proof later, one checks in particular whether the simulation was done properly by the prover.

A challenge in practice is the trusted generation of the common reference string. A straightforward option would be to use an actual trusted party, but in many cases, this is undesirable. A popular choice is mulitparty computation, which enables generating a CRS in a provably secure manner, as long as at least one party behaves honestly.

Fiat-Shamir heuristic. The Fiat-Shamir heuristic [16] uses a hash-function to turn an interactive, so-called *public-coin* proof into a non-interactive one in the random oracle model. A protocol is called public-coin if the verifier picks its messages uniformly at random and independently from the messages of the prover. To generate a proof non-interactively, the prover simulates the verifier by computing its messages as the hash of the current transcript. Modelling the hash function as a random oracle (with appropriate output range), the messages from the simulated verifier look *as if* generated by

the real verifier and in particular, are unpredictable for the prover. When verifying the proof, one again has to make sure that the challenges were properly generated.

Interactive Arguments

The soundness and extractability properties are defined with respect to an unbounded prover, i.e. even an all-powerful prover cannot break them. It is possible to relax this notion by requiring soundness and extractability to hold only for efficient provers. The resulting protocols are called *argument systems* and were introduced by Brassard, Chaum and Crépeau [13]. Today, in many settings, interactive arguments are preferred over interactive proofs, as their communication complexity can be much lower.

In common language, argument systems are still referred to as proof systems, even though there is a technical difference. In fact, both "proof" systems in this thesis, Groth16 and TV20, are actually argument systems.

Performance

There are three important aspects of performance that can be considered for interactive proofs: The prover complexity, the verifier complexity and the communication complexity.

The prover/verifier complexity simply refers to the amount on computation that prover/verifier must do during a protocol execution.

The Communication complexity is the amount of data that is sent between the two parties during a protocol run. In case of a non-interactive proof, the parallel notion to communication complexity is the proof size, which, in case of the Fiat-Shamir heuristic or CRS model, is equal to the prover communication complexity of the interactive version of the protocol. This is because a non-interactive proof only contains the messages sent by the prover, while the verifier's messages can be computed.

2.4.2 Defining Zero-Knowledge Proofs of Knowledge

This section formally defines zero-knowledge proofs of knowledge.

Definition 2.1 *A zero-knowledge proof of knowledge $(\mathcal{P}, \mathcal{V})$ between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ for a language $\mathcal{L}$ with relation $R_{\mathcal{L}}$, containing pairs of members of $\mathcal{L}$ and witnesses, is a pair of algorithms such that $\mathcal{V}$ runs in probabilistic polynomial time and the following conditions hold:*

1. ***Completeness*** *$(\mathcal{P}, \mathcal{V})$ is complete, if for any $x \in \mathcal{L}$ with a membership witness w:*

$$Pr[(\mathcal{P}(w), \mathcal{V})(x) = 1] \geq 2/3$$

2. ***Knowledge Extraction.*** *$(\mathcal{P}, \mathcal{V})$ is knowledge-extractable with knowledge error κ, if there exists an efficient algorithm Ext[1] and a polynomial p such that for any input x, for any prover P^*, $Ext^{[P^*]}$ runs in expected polynomial time and satisfies*

$$Pr[w \leftarrow Ext^{[P^*]} : R_{\mathcal{L}}(x, w') = 1] \geq \frac{\epsilon - \kappa}{p(|x|)}$$

where ϵ denotes the probability that $\mathcal{V}$ accepts when interacting with P^* on common input x.

3. ***Zero-Knowledge.*** *$(\mathcal{P}, \mathcal{V})$ is zero-knowledge, if for every probabilistic polynomial time V^* there exists a probabilistic simulator Sim running in expected polynomial time, such that for every $x \in \mathcal{L}$, the output of Sim is indistinguishable from a transcript obtained by running the interactive protocol between P and V^*.*

2.5 Hidden-order Group Assumptions

This section defines standard assumptions about hidden order groups. These are the assumptions on which the security of the TV20 proof system is based. The definitions are taken more or less directly from [29].

A hidden order group generator G is an algorithm that takes as input a security parameter 1^λ and outputs a description of a finite abelian group $(\mathbb{G}, \cdot)$ and an integer $P \geq 2$. The integer P is mostly used as an upper bound when sampling challenges during an interactive proof and is assumed to be superpolynomial in λ, but smaller than $|\mathbb{G}|$.

Definition 2.2 ***Strong-Root Assumption.*** *A group generator G satisfies the (T, ε)-strong-root assumption, if for all $\lambda \in \mathbb{N}$ and for every adversary $\mathcal{A}$ that runs in time at most $T(\lambda)$,*

$$Pr\left[\begin{array}{c} g^n = h \wedge n > 1 \end{array} : \begin{array}{c} (\mathbb{G}, P) \leftarrow \mathsf{G}(1^\lambda) \\ h \leftarrow_{\$} \mathbb{G} \\ (g, n) \leftarrow \mathcal{A}(\mathbb{G}, P, h) \end{array}\right] \leq \varepsilon(\lambda)$$

The strong root assumption is a generalization of the strong-RSA assumption to generic hidden-order groups.

Definition 2.3 ***Small-Order Assumption.*** *A group generator G satisfies the (T, ε)-small-order assumption if for all $\lambda \in \mathbb{N}$, for every adversary $\mathcal{A}$ that runs in time at most $T(\lambda)$,*

$$Pr\left[\begin{array}{c} g^n = 1_{\mathbb{G}} \wedge g^2 \neq 1_{\mathbb{G}} \\ 0 < n < P \end{array} : \begin{array}{c} (\mathbb{G}, P) \leftarrow \mathsf{G}(1^\lambda) \\ (g, n) \leftarrow \mathcal{A}(\mathbb{G}, P) \end{array}\right] \leq \varepsilon(\lambda)$$

[1] *Ext* must be able to interact with P^*, and in particular, restart P^* with chosen randomness

The small-order assumption simply states that it is hard to find low order elements in the group, except elements of order 2.

Definition 2.4 ***Orders with Low Dyadic Valuation.*** *A group generator* G *satisfies the low-dyadic-valuation assumption on orders if for all* $\lambda \in \mathbb{N}$*, for every* $(\mathbb{G}, P) \leftarrow \mathsf{G}(1^\lambda)$*, for every* $g \in \mathbb{G}$*,* $ord(g)$ *is divisible by 2 at most once.*

Definition 2.5 ***Many Rough-Order Elements Assumption.*** *An integer is said to be P-rough if all its prime factors are greater than or equal to P. A group generator* G *satisfies the* μ*-assumption that there are many rough-order elements in the groups generated by* G *if for all* $\lambda \in \mathbb{N}$*,*

$$Pr\left[\ ord(h) \text{ is } P\text{-rough} \ : \ \begin{array}{c}(\mathbb{G}, P) \leftarrow \mathsf{G}(1^\lambda) \\ h \leftarrow_\$ \mathbb{G}\end{array}\ \right] \geq \mu(\lambda)$$

All these assumptions are believed to hold in an RSA group $\mathbb{Z}_N^*$ for a modulus $N = pq$, with prime numbers p, q such that $p, q = 3 \ mod\ 4$, if the number of prime factors of $p-1$ and $q-1$ that are $\leq P$ is of order $\mathcal{O}(\lambda)$.

2.6 Diophantine Equations

A *Diophantine equation* is a multivariate polynomial equation with integer coefficients, where one is interested in integer solutions only. In this thesis, the number of variables of a Diophantine equation is usually denoted ν and the degree is denoted δ. An example Diophantine equation in $\nu = 3$ variables of degree $\delta = 4$ would be

$$2xy^2z - 4xz + y + 1 = 0$$

A solution for the above equation is $x = 1,\ y = 1,\ z = 1$. A Diophantine equation that has an integer solution is called satisfiable. Davis, Putnam, Robinson and Matiyasevich [23] proved that there is no algorithm that can determine whether any Diophantine equation has a solution, demonstrating that the Diophantine satisfiability problem is undecidable. A large class of problems can be reduced to Diophantine satisfiability.

2.7 The TV20 Proof System

The proof system of Towa and Vergnaud [29] allows to prove knowledge of a solution to a Diophantine equation. Its security is based on the standard assumptions about hidden order groups introduced in section 2.5. For a Diophantine equation in ν variables of total degree δ with coefficients of size at most 2^H for some $H \in \mathbb{N}$, the communication-complexity of TV20 is

$\mathcal{O}(\delta\ell + min(\nu, \delta)\log(\nu + \delta) \cdot b_G + H)$ bits, if all integers of the solution are upper-bounded by 2^ℓ, for $\ell \in \mathbb{N}$.

This section starts off by giving a coarse grained overview of how TV20 works. Then it discusses the main building blocks of TV20 in more detail.

2.7.1 Overview

In TV20, the first step is always to convert the input Diophantine equation into a Hadamard product of the form $\mathbf{a}_L \circ \mathbf{a}_R = \mathbf{a}_O$ and integer linear constraints over the entries of $\mathbf{a}_L$, $\mathbf{a}_R$ and $\mathbf{a}_O$. Towa and Vergnaud describe a generic procedure to perform this step. After the reduction step, the prover and verifier then run a protocol for the new problem, in which the prover proves that it knows a solution to the Hadamard product and linear constraints. At the heart of this protocol is another sub-protocol, for proving knowledge of values that satisfy an inner product relation. The following paragraphs describe each of these building blocks, in reverse order.

2.7.2 Inner Product Argument

At the core of TV20 is an inner product argument for the following problem. For a hidden-order group $\mathbb{G}$, a length n, an upper bound on the witness entries 2^ℓ and parameters

$$e, f, C \in \mathbb{G} \quad \mathbf{g}, \mathbf{h} \in \mathbb{G}^n,$$

the inner-product protocol allows to prove knowledge of $\mathbf{a}, \mathbf{b} \in \mathbb{Z}^n$ and $r \in \mathbb{Z}$, such that

$$C^2 = \left(\mathbf{g}^{\mathbf{a}}\mathbf{h}^{\mathbf{b}} e^{\langle \mathbf{a}, \mathbf{b}\rangle} f^r\right)^4 \wedge |[\mathbf{a}, \mathbf{b}, r]|_\infty < 2^\ell$$

Overview. The protocol works recursively. To ease the explanation, assume that n is a power of two. Also, note that the subscripts denote the first and second half of a vector, respectively.

During the i-th recursion step, the prover samples two integers s_u, s_v and computes the two group elements

$$U_i = \left(\mathbf{g}_1^{\mathbf{a}_2}\mathbf{h}_2^{\mathbf{b}_1} e^{\langle \mathbf{a}_2 \mathbf{b}_1\rangle} f^{s_u}\right)^2$$
$$V_i = \left(\mathbf{g}_2^{\mathbf{a}_1}\mathbf{h}_1^{\mathbf{b}_2} e^{\langle \mathbf{a}_1 \mathbf{b}_2\rangle} f^{s_v}\right)^2.$$

The prover sends over U_i and V_i and receives a challenge x_i, sampled uniformly at random by the verifier from $\{0, \ldots, P-1\}$.

Then, the new parameters for the next recursion step are computed. Both the prover and the verifier compute new base vectors $\mathbf{g}', \mathbf{h}'$ and a new C-value as

$$\mathbf{g}' = \mathbf{g}_1^{x_i} \circ \mathbf{g}_2, \quad \mathbf{h}' = \mathbf{h}_1^{x_i} \circ \mathbf{h}_2, \quad C_{i+1} = U_i^{x_i^2} C_i^{x_i} V_i$$

In addition, the prover also computes a new witness as

$$\mathbf{a}' = \mathbf{a}_1 + x_i \mathbf{a}_2, \quad \mathbf{b}' = x_i \mathbf{b}_1 + \mathbf{b}_2, \quad r' = s_v + r x_i + s_u x_i^2.$$

Crucially, the length n of the base vectors $\mathbf{g}, \mathbf{h}$ and witness vectors $\mathbf{a}, \mathbf{b}$ is halved in each recursion step. As soon as these vectors only contain a single element (after $n' = \log(n)$ steps), the parties run a dedicated 3-move protocol for the case $n = 1$.

In this protocol, the prover sends over two group elements, Γ and Δ, receives a challenge integer $x_{n'+1}$ (sampled u.a.r. from $\{0, \ldots, P-1\}$) and then computes and sends over integers a', b', u. At last, the verifier evaluates the verification equation

$$\left(g^{x_{n'+1} a'} h^{x_{n'+1} b'} e^{a' b'} f^u\right)^4 \stackrel{?}{=} \left(C_{n'+1}^{x_{n'+1}^2} \Gamma^{x_{n'+1}} \Delta\right)^2,$$

and accepts or rejects the protocol run.

Verification Complexity The lion's share of the computation by the verifier is done in between recursion steps, to compute the parameters $\mathbf{g}', \mathbf{h}'$ and C_{i+1} for the next step. However, at any recursion level, the verifier simply samples a challenge from the same challenge space $\{0, \ldots, P-1\}$. In particular, the challenge space only depends on P, which means that the verifier can delay the computation of the new parameters until the very end, when it has to evaluate the verification equation. This way, all the verifier's computation can be aggregated into a single multi-exponentiation. The explicit expression for the multi-exponentiation is

$$\begin{aligned}
&\left(\prod_{i=1}^{n} g_i^{\prod_{j \in S_i} x_j}\right)^{4 x_{n'+1} a'} \left(\prod_{i=1}^{n} h_i^{\prod_{j \in [n] \setminus S_i} x_j}\right)^{4 x_{n'+1} b'} e^{4 a' b'} f^{4u} \\
&= \left(U_{n'}^{x_{n'}} \prod_{i=1}^{n'-1} U_i^{x_i x_{i+1} \ldots x_{n'}} C^{x_1 \ldots x_{n'}} \prod_{i=1}^{n'-1} V_i^{x_{i+1} \ldots x_{n'}} V_{n'}\right)^{2 x_{n'+1}^2} \Gamma^{2 x_{n'+1}} \Delta^2,
\end{aligned}$$

where

$$S_i = \{j \in [n'] : n' + 1 - j\text{th bit of } i - 1 \text{ is } 0\}.$$

The total number of bases in the multi-exponentiation is $2n + 2\lceil \log_2(n) \rceil + 5$. The value of the maximum exponent is at most

$$4 \max \left(2^{\ell} P^{2n'+1},\ 2^{2\ell} P^{2n'},\ (2n' 2^{b_G+\lambda} P^{n'+1} + 2^{\ell}(P-1)^{n'+2})(1+2^{\lambda}) \right),$$

implying that the logarithm of the maximum exponent is $\mathcal{O}(\ell + b_G + \log(n)\log(P))$.

Therefore, using a square-and-multiply approach, the verifier has to do $\mathcal{O}(n(l + b_G + \log(n)\log(P)))$ group operations in the RSA group. Using pre-computed tables and sliding-window methods [4], one can reduce the number of group operations that have to be performed, at the cost of increased memory requirements. In the extreme case of using $\mathcal{O}(2^n \cdot b_G)$ bits of memory, one can bring down the number of necessary group operations to $\mathcal{O}(l + b_G + \log(n)\log(P))$. In practice, however, it is unlikely that this amount of memory is available. Using less memory will still decrease the number of group operations, and different memory/time trade-offs are available.

Communication Complexity. During the whole protocol, the prover sends over $2n' + 2$ group elements and three integers, a', b' and u. The integers a', b' have absolute value less than $2^{\ell} P^{n'}$ and the integer u has absolute value less than $\left(2n' 2^{b_G+\lambda} P^{n'+3} + 2^{\ell}(P-1)^{n'+2}\right) \cdot \left(1 + 2^{\lambda}\right)$. This results in an asymptotic communication complexity of $\mathcal{O}(l + \log(n) \cdot b_G)$ bits. Note that the communication complexity scales logarithmically with the length of the inner product, n.

2.7.3 Hadamard Product Argument

The statement being proven in the Hadamard product argument is the following[2]. For parameters

$$\mathbf{W}_L, \mathbf{W}_R, \mathbf{W}_O \in \mathbb{Z}^{Q \times n}, \quad \mathbf{c} \in \mathbb{Z}^{Q},$$

where n is the length of the Hadamard product and Q is the number of linear constraints, the protocol allows to prove knowledge of $\mathbf{a}_L, \mathbf{a}_R, \mathbf{a}_O \in \mathbb{Z}^n$ such that

$$\mathbf{a}_L \circ \mathbf{a}_R = \mathbf{a}_O \ \wedge \ \mathbf{W}_L \mathbf{a}_L^T + \mathbf{W}_R \mathbf{a}_R^T + \mathbf{W}_O \mathbf{a}_O^T = \mathbf{c}^T$$

Description. The core idea of the protocol is to reduce the Hadamard product and linear constraints to an inner product argument. Then, the inner product protocol is invoked as a sub-protocol. The reduction is so that the length n of the inner product is equal to the length of the Hadamard product.

[2] In fact, the full argument is more general and allows some variables to be committed to. However, the simplified version suffices for our case.

Besides the inner-product argument, there is also a call to a so-called *base-switching argument*. The base switching argument is not described here (as the details are irrelevant for us), but the communication complexity of the call is estimated in section 4.2.2.

Communication Complexity. Overall, the communication complexity of the protocol is $\mathcal{O}(\ell + \log(n)b_{\mathbb{G}})$ bits. The bulk of the communication complexity originates in the calls to the two sub-protocols (inner-product argument and base-switching argument).

2.7.4 Diophantine Equation Argument

This section describes the full TV20 argument system to prove knowledge of a solution to a Diophantine equation. The protocol builds on the Hadamard product argument. Using a reduction, knowing a solution to the input equation is demonstrated to be equivalent to knowing vectors $\mathbf{a}_L, \mathbf{a}_R, \mathbf{a}_O$ that satisfy a Hadamard product, i.e $\mathbf{a}_L \circ \mathbf{a}_R = \mathbf{a}_O$, as well as a number of linear constraints over the entries of the vectors. Then, the Hadamard product protocol is invoked.

Reduction Procedure.

The reduction procedure described by Towa and Vergnaud works by transforming any Diophantine equation to an equation of degree at most 4. Additionally, the resulting equation is easily seen to be equivalent to a Hadamard product and linear constraints, i.e. knowing a solution to the former allows computing a solution to the latter and vice-versa.

Example. In the following, the procedure is illustrated with an example. Let the input polynomial be $4x_1x_2 + 2x_1^5 + 3$. The core idea of the reduction is to introduce new variables that correspond to the product of two previous variables.

In a first step, the polynomial is transformed such that all variables have degree 1. In the example polynomial, the only term that violates this condition is the second term, $2x_1^5$. Two new variables, $v_1 = x_1^2$ and $v_2 = v_1^2 = x_1^4$ are introduced and the term can be rewritten as $2x_1v_2$. However, the relations $v_1 = x_1^2$ and $v_2 = v_1^2$ also need to be enforced. To do so, the terms $(v_1 - x_1^2)^2$ and $(v_2 - v_1^2)^2$ are introduced into the polynomial, and the original polynomial is squared to get a new polynomial

$$(4x_1x_2 + 2x_1v_2 + 3)^2 + (v_1 - x_1^2)^2 + (v_2 - v_1^2)^2$$

The squaring makes sure that in a satisfying solution, all of the individual terms are zero, not just the whole polynomial. Observe that a solution to the original polynomial can be transformed to a solution for the new one and vice-versa.

In a second step, the polynomial is further transformed so that the part that corresponds to the original polynomial (currently $4x_1x_2 + 2x_1v_2 + 3$) becomes of degree one, i.e. no products of variables appear. Two new variables, $u_1 = x_1x_2$ and $u_2 = x_1v_2$ are introduced. Just as before, terms $(u_1 - x_1x_2)^2$ and $(u_2 - x_1v_2)^2$ are added to constrain the variables. The resulting polynomial is

$$(4u_1 + 2u_2 + 3)^2 + (v_1 - x_1^2)^2 + (v_2 - v_1^2)^2 + (u_1 - x_1x_2)^2 + (u_2 - x_1v_2)^2$$

Note that the first term, $4u_1 + 2u_2 + 3$, is now an affine equation.

Third step. In this form, the polynomial is ready to be transformed to a Hadamard product and linear constraints over the entries of the Hadamard product. The idea is that each of the last four terms, which are of the form $(a - bc)^2$, becomes an entry in the Hadamard product, so that the witness of the prover is

$$\mathbf{a}_L = [x_1\ v_1\ x_1\ x_1], \quad \mathbf{a}_R = [x_1\ v_1\ x_2\ v_2], \quad \mathbf{a}_O = [v_1\ v_2\ u_1\ u_2].$$

To make sure that the vectors are actually of this form, linear constraints are used to guarantee consistency in a single vector as well as between different vectors. For example, to make sure that the first, third and fourth entry of the witness vector $\mathbf{a}_L$ all contain the same value, the two linear constraints $\mathbf{a}_{L,1} = \mathbf{a}_{L,3}$ and $\mathbf{a}_{L,3} = \mathbf{a}_{L,4}$ are used. For $\mathbf{a}_R$ and $\mathbf{a}_O$, there are no in-vector consistencies to be enforced (i.e. no variable appears twice). However, one also needs to enforce the between-vector consistencies $\mathbf{a}_{L,2} = \mathbf{a}_{R,2}$, $\mathbf{a}_{L,2} = \mathbf{a}_{O,1}$ and $\mathbf{a}_{R,4} = \mathbf{a}_{O,2}$.

Finally, it must be made sure that the actual equation is satisfied, by introducing the linear constraint $4\mathbf{a}_{O,3} + \mathbf{a}_{O,4} + 3 = 0$.

In Summary, we have shown that knowing a solution to the Diophantine equation $4x_1x_2 + 2x_1^5 + 3$ is equivalent to knowing three vectors $\mathbf{a}_L, \mathbf{a}_R, \mathbf{a}_O \in \mathbb{Z}^4$ such that $\mathbf{a}_L \circ \mathbf{a}_R = \mathbf{a}_O$ and

$$\begin{aligned} &\mathbf{a}_{L,1} = \mathbf{a}_{L,3} \ \wedge\ \mathbf{a}_{L,3} = \mathbf{a}_{L,4} \ \wedge\ \mathbf{a}_{L,2} = \mathbf{a}_{R,2} \\ \wedge\ &\mathbf{a}_{L,2} = \mathbf{a}_{O,1} \ \wedge\ \mathbf{a}_{R,4} = \mathbf{a}_{O,2} \ \wedge\ 4\mathbf{a}_{O,3} + \mathbf{a}_{O,4} + 3 = 0 \end{aligned}$$

Generic Procedure. In general, the reduction procedure consists of the following steps

1. Replace variables x^k of degree $k \geq 2$ by introducing variables $v_1, \ldots, v_{\lfloor \log k \rfloor}$ where $v_1 = x^2$ and $v_i = v_{i-1}^2$ for $i \geq 2$ and replacing x^k with the appropriate subset of the new variables. Constrain the new variables using terms of the form $(v_i - v_{i-1}^2)^2$.

2. Replace products of the form $\prod_{i=1}^{k} x_i$ with a single variable, by introducing new variables $u_1, \ldots, u_{k-1}$ that correspond to the prefixes of the products, i.e. $u_j = \prod_{i=1}^{j} x_i$, and replacing the product by u_{k-1}. Constrain the new variables with terms of the form $(u_j - x_j u_{j-1})^2$.
3. Transform the equation into a Hadamard product and linear constraints, by adding Hadamard product entries for terms of the form $(a - bc)^2$, using linear constraints for consistencies and to enforce that the actual equation is satisfied.

For a polynomial in $\mathbb{Z}[x_1, \ldots, x_\nu]$ of total degree δ and μ monomials, the procedure guarantees that the resulting Hadamard product is of length at most $\nu\lfloor\log(\delta)\rfloor + (\delta - 1)\mu$ and that there are at most $1 + 2\nu(\lfloor\log(\delta)\rfloor - 1) + (\delta - 2)\mu$ linear constraints.

Communication Complexity.

The overall communication complexity of the protocol is equal to the communication complexity of a call to the Hadamard product protocol with parameters generated by the reduction procedure. For an input equation with ν variables of total degree δ, coefficients less than 2^H and witness entries less than 2^ℓ, the communication complexity in bits is of order $\mathcal{O}(\ell + min(\nu, \delta)\log(\nu + \delta)b_{\mathrm{G}} + H)$.

2.7.5 How to write Equations

The above discussion demonstrates that the communication complexity of the Diophantine equation argument is dictated by the size of the resulting Hadamard product. While the generic reduction procedure gives us a bound on this size, for many problems it is advantageous to not rely on it. Instead, it is often possible to apply the ideas of the reduction procedure 'by hand', and directly write down the equation in a form where the Hadamard product and linear constrains can be read off immediately. Most of the examples in the original paper follow this approach, which has the following two benefits.

Immediate feedback. One can directly see the size of the resulting Hadamard product. Otherwise, the generic bound from the reduction procedure would have to be applied, which prevents immediate feedback, and furthermore might not be tight.

Composability. This approach makes sure that encodings for two different problems are efficiently composable. This point is now illustrated with an example.

Consider two Diophantine equations, say $a + b = 0$ and $c + a^2 = 0$. Note that the same variable a appears in both equations. If both equations should

be satisfied at the same time, one cannot simply add them, as a solution to $(a+b)+(c+a^2)=0$ does not necessarily give solutions to the original two equations. Instead, one would have to first square the individual equations and then add them, to get $(a+b)^2+(c+a^2)^2=0$. This works, as the squaring makes sure that in a satisfying solution of the composed equation, all the individual terms are zero. However, this generic approach might not be optimal in the size of the resulting Hadamard product when subsequently applying the reduction procedure.

On the other hand, if Hadamard products and linear constraints for the subproblems are available, they can be composed in the following way. The Hadamard products are concatenated and additional linear constraints are introduced that 'tie' together variables that are shared by the two subproblems. The new set of linear constraints is the union of the linear constraints for the subproblems and the ones that are newly introduced. We go back to our example, but this time using Hadamard product and linear constraints directly.

The first subproblem, $a+b=0$, can be reduced to a Hadamard product $\mathbf{a}_L^{(1)} \circ \mathbf{a}_R^{(1)} = \mathbf{a}_O^{(1)}$ of length two with linear constraint

$$\mathbf{a}_{O,1}^{(1)} + \mathbf{a}_{O,2}^{(1)} = 0.$$

The prover sets

$$\mathbf{a}_L^{(1)} = [1\ 1], \quad \mathbf{a}_R^{(1)} = [a\ b], \quad \mathbf{a}_O^{(1)} = [a\ b]$$

As a side note, notice that "dummy entries" need to be introduced in the Hadamard product, to be able to have a linear constraint corresponding to the original equation. This is needed whenever the original equation contains a summand of only a single variable of degree one (in the above case, both a and b).

The second subproblem, $c+a^2=0$ is reduced to a Hadamard product $\mathbf{a}_L^{(2)} \circ \mathbf{a}_R^{(2)} = \mathbf{a}_O^{(2)}$ of length 2 with linear constraints

$$\mathbf{a}_{L,1}^{(2)} = \mathbf{a}_{R,1}^{(2)}, \quad \mathbf{a}_{O,1}^{(2)} + \mathbf{a}_{O,2}^{(2)} = 0.$$

The prover sets

$$\mathbf{a}_L^{(2)} = [a\ 1], \quad \mathbf{a}_R^{(2)} = [a\ c], \quad \mathbf{a}_O^{(2)} = [a^2\ c].$$

Again, a dummy entry must be introduced for c.

To prove knowledge of a solution for both subproblems, we compose it in the following way. The new Hadamard product $\mathbf{a}_L \circ \mathbf{a}_R = \mathbf{a}_O$ will be of length 4, and the linear constraints are

$$\mathbf{a}_{O,1} + \mathbf{a}_{O,2} = 0, \quad \mathbf{a}_{L,3} = \mathbf{a}_{R,3}, \quad \mathbf{a}_{O,3} + \mathbf{a}_{O,4} = 0, \quad \mathbf{a}_{R,1} = \mathbf{a}_{R,3},$$

The witness vector of the prover is

$$\mathbf{a}_L = [1\ 1\ a\ 1], \quad \mathbf{a}_R = [a\ b\ a\ c], \quad \mathbf{a}_O = [a\ b\ a^2\ c]$$

The first three linear constraints are simply the ones from the subproblems, shifted to account for the concatenation of the witness vectors. The fourth linear constraint, $\mathbf{a}_{R,1} = \mathbf{a}_{R,3}$, was introduced to tie the a-values that appear in both equation together. This is important, as otherwise the prover could use different values for a in the first and second equation. Crucially, when composing like this, the length of the resulting Hadamard product is the sum of the lengths of the Hadamard products for the subproblems.

2.7.6 Example encoding

This section shows how to encode two classical NP-complete problems as Diophantine equations.

k-Coloring of Graphs

Let $G = (V, E)$ be a graph with n vertices and m edges. A valid k-coloring of G is a labeling of the vertices in V with labels from $\{1, \ldots, k\}$, such that no neighbouring vertices have the same color. The goal is to find a Diophantine equation such that knowing a k-coloring of G allows computing a solution to the Diophantine equation and vice-versa.

Intuition. The idea is the following. For each vertex $v \in V$, k variables $x_{v,1}, \ldots, x_{v,k}$ are introduced. The fact that v has a color j is indicated by setting $x_{v,j} = 1$ and $\forall i \neq j : x_{v,i} = 0$. Additionally, the variables are constrained in such a way that a satisfying solution must correspond to a valid coloring and vice-versa. Concretely, the following three things must hold:

1. Every variable is set to either 0 or 1. For all $v \in V$ and for all $j \in [k]$, this is enforced by the equation $x_{v,j}^2 - x_{v,j} = 0$
2. As every node can only have one color, for every node v, only one of the the variables $x_{v,j}$ can be set to 1. This is enforced by the equation $\sum_{i=1}^{k} x_{v,k} = 1$.
3. The assignment must be a legal coloring, meaning no two neighbouring nodes have the same color. This is equivalent to saying that for every edge $(u, v) \in E$ and for every color $j \in [k]$, $x_{v,j} \cdot x_{u,j} = 0$.

The first two points make sure that *some* coloring can be extracted from a solution to the equation. The third point additionally confirms that the coloring is actually valid.

Composing. The above equations need to be composed so they all hold at the same time, and this is exactly where the aforementioned composability

(section 2.7.5) comes into play. Technically, one could square all the equations and add them up to get a single equation. However, the equations are purposefully written in such a way that Hadamard product and linear constraints can be easily read off.

Every equation of the first type, $x_{v,j}^2 - x_{v,j} = 0$, can be transformed to a single Hadamard product entry of the form $(x_{v,j}) \circ (x_{v,j}) = (x_{v,j})$. Additionally, two linear constraints are needed to guarantee consistency in entries. As we have k entries like this for every variable, we have kn entries in the Hadamard product and $2kn$ linear constraints from the first point above.

The second type of equation can be translated into a single linear equation over the entries of the Hadamard product. This takes n linear constraints.

For the third point, an equation $x_{v,j} \cdot x_{u,j} = 0$ translates to a single Hadamard product entry of the form $(x_{v,j}) \circ (x_{u,j}) = (x_{(v,u),j})$ where a new variable $(x_{(v,u),j})$ is introduced to store the result of the product. An additional linear constraint $(x_{(v,u),j}) = 0$ then enforces the product to be 0. Furthermore, another two linear constraints are needed to tie $x_{v,j}$ and $x_{u,j}$ to previous Hadamard product entries. For every edge, we have k equations as above. Therefore, km entries to the Hadamard product and $3km$ linear constraints are added.

In total, this results in a Hadamard product of length $k(n+m)$, as well as $(2k+1)n + 3km$ linear constraints.

Hamiltonian Cycle

Let $G = (V, E)$ be a graph with n vertices, $V = \{v_1, \ldots, v_n\}$, and m edges. A Hamiltonian cycle in G is a cycle where each vertex is visited exactly once. The goal is to find a Diophantine equation such that a solution can be found whenever one knows a Hamiltonian cycle in G and vice-versa.

Intuition. The idea is as follows. There are n^2 variables $x_{i,j}$ for $i \in \{1 \ldots, n\}$, $j \in \{1 \ldots, n\}$. A variable $x_{i,j}$ being set to 1 indicates that v_j is the i-th vertex in the Hamiltonian cycle. The variables need to be constrained such that:

1. All variables are either 0 or 1. For a variable $x_{i,j}$ this is enforced by the equation $x_{i,j}^2 - x_{i,j} = 0$
2. At every position of the cycle, only one vertex can occur. For a position $i \in [n]$, this is enforced by $\sum_{j=1}^{n} x_{i,j} = 1$.
3. The implied cycle must actually be in G. This is checked by ensuring for all positions $i \in [n]$, the vertices at the i-th and $(i+1)$-th position are connected by an edge in the graph, i.e $\sum_{\{u,v\} \in E} x_{i,u} \cdot x_{i+1 \bmod n,v} = 1$
4. A vertex occurs exactly once in the cycle (i.e. the cycle is Hamiltonian). For every $j \in [n]$, this is enforced by $\sum_{i=1}^{n} x_{i,j} = 1$

Intuitively, the first three points make sure that for any solution of the Diophantine equation, one can actually get a valid cycle of length n in G. The fourth point additionally confirms that the cycle is Hamiltonian.

Composing. The equations above are written in a way such that reading off Hadamard product and linear constraints directly is possible.

The first point adds n^2 entries of the form $(x_{i,j}) \circ (x_{i,j}) = (x_{i,j})$ to the Hadamard product. $2n^2$ linear constraints are needed to guarantee consistency between entries.

The second point contributes n linear equations over the entries of the Hadamard product.

For the third point, a little more work is required. The idea is to introduce new variables that essentially store the results of the products. For every $i \in [n]$ and $\{u, v\} \in E$, an entry of the form $(x_{i,u}) \circ (x_{i+1 \bmod n,v}) = (x_{i,\{u,v\}})$ is added to the Hadamard product to store the result of $x_{i,u} \cdot x_{i+1 \bmod n,v}$ in a new variable called $x_{i,\{u,v\}}$. To guarantee consistency of the values $x_{i,u}$ and $x_{i+1 \bmod n,v}$ with previous Hadamard product entries, two linear constraints are added. Also, for every $i \in [n]$ the linear equation $\sum_{\{u,v\} \in E} x_{i,\{u,v\}} = 1$ is included. This adds nm entries to the Hadamard product and $2nm + n$ linear constraints.

The fourth point adds another n linear equations.

In total, this results in a Hadamard product of size $n^2 + nm$, as well as $2n^2 + 3n + 2nm$ linear constraints.

2.8 Zcash-specific Preliminaries

This section introduces background specific to Zcash. This includes a brief introduction to elliptic curves, as well as concrete descriptions of functions such as hash functions or commitment schemes that are used in Zcash. Also, some Zcash-specific notation is introduced. All of the content can be found in the Zcash specification [20].

2.8.1 Conversions between Integers and Bitstrings

There are several functions in Zcash that convert between bits and integers. These include:

- $\mathsf{I2LEBSP} : (\ell : \mathbb{N}) \times \{0, \ldots, 2^\ell - 1\} \to \{0,1\}^\ell$. $\mathsf{I2LEBSP}_\ell(x)$ is defined as the sequence of ℓ bits representing x in little-endian order.
- $\mathsf{LEBS2OSP} : (\ell : \mathbb{N}) \times \{0,1\}^\ell \to \{0,1\}^{8 \cdot \lceil \ell/8 \rceil}$ is defined as follows. Pad the input on the right with $8 \cdot \lceil \ell/8 \rceil - \ell$ zero bits so that its length is a multiple of 8 bits. Then, convert each group of 8 bits to a byte value

with the least significant bit first and concatenate the resulting bytes in the same order as the groups.

- $\mathsf{LEOS2IP} : (\ell : \mathbb{N} | \ell \mod 8 = 0) \times \{0,1\}^\ell \to \{0, \ldots, 2^\ell - 1\}$. $\mathsf{LEOS2IP}_\ell(S)$ is defined as the integer represented in little-endian order by the byte sequence S of length $\ell/8$.

2.8.2 Elliptic Curve Background

Zcash makes extensive use of elliptic curve cryptography. It is out-of-scope for this thesis to provide a complete overview of elliptic curve theory. Instead, we choose to focus only on the parts that are directly relevant for the project.

Zcash uses two distinct forms of representation of an elliptic curve: The *complete twisted Edwards* form (referred to as *ctEdwards*) and the *Montgomery* form. This section describes the two types of representation, the arithmetic on them, and the specific curve that is used in Zcash. An understanding of this is necessary, as the elliptic curve operations are encoded as Diophantine equations in chapter 4.

The ctEdwards Form

A ctEdwards curve [9, 10] is an elliptic curve over a field $\mathbb{F}$, parameterized by values $a, d \in \mathbb{F}$, where a is a square and d is non-square. The curve consists of all points $(u, v) \in \mathbb{F} \times \mathbb{F}$ that satisfy the equation $a \cdot u^2 + v^2 = 1 + d \cdot u^2 \cdot v^2$. The neutral element on a ctEdwards curve is $(0, 1)$, the inverse of an element (u, v) is $(-u, v)$. Addition of two points (u_1, v_1), (u_2, v_2) is computed using the formula

$$(u_1, v_1) + (u_2, v_2) = \left(\frac{u_1 v_2 + u_2 v_1}{1 + d u_1 u_2 v_1 v_2}, \frac{v_1 v_2 - a u_1 u_2}{1 - d u_1 u_2 v_1 v_2} \right)$$

The denominator is never 0, meaning that the above addition formula can be used for any pair of points. This is one reason why the ctEdwards form is convenient to use in a zero-knowledge setting, because it means that no case-distinctions have to be implemented.

The Montgomery Form

A Montgomery curve [10] is a curve over a field $\mathbb{F}$, parameterized by $A, M \in \mathbb{F}$, where $B(A^2 - 4) \neq 0$. The curve contains all points $(x, y) \in \mathbb{F} \times \mathbb{F}$ that satisfy the equation $By^2 = x^3 + Ax^2 + x$. The neutral element is denoted $\mathcal{O}$ and cannot be represented as a pair of coordinates. The negation of a point (x, y) is $(x, -y)$. To perform an addition $(x_3, y_3) = (x_1, y_1) + (x_2, y_2)$, one uses the formula

$$
\begin{aligned}
x_3 &= B \cdot \lambda^2 - A - x_1 - x_2 \\
y_3 &= (x_1 - x_2) \cdot \lambda - y_1 \\
\text{where } \lambda &= \begin{cases} \frac{3 \cdot x_1^2 + 2 \cdot A \cdot x_1 + 1}{2 \cdot B y_1}, & \text{if } x_1 = x_2 \\ \frac{y_2 - y_1}{x_2 - x_1}, & \text{otherwise} \end{cases}
\end{aligned}
$$

Notice that a case distinction is needed, depending on the input points, which hinders the usage of the Montgomery form in a zero-knowledge proof setting. However, if one can guarantee that the side condition $x_1 \neq x_2$ always holds, the Montgomery addition can actually be more efficient to implement in zero-knowledge than a ctEdwards addition (see section 4.1.16).

Jubjub

The concrete elliptic curve that is used in Zcash is called *Jubjub* [20]. Jubjub has been specifically designed for Zcash and to be efficiently implementable in finite field circuits. The specification defines both a ctEdwards and a Montgomery form of Jubjub as well as a way to convert between the two. For completeness, we include a description of both forms here.

Jubjub ctEdwards Form. Let

$q_{\mathbb{J}} = 52435875175126190479447740508185965837690552500527637822603658699938581184513$,

$r_{\mathbb{J}} = 65544843968907738099309675635232457297059212658723172813653591623921832541 99$,

where both $q_{\mathbb{J}}$ and $r_{\mathbb{J}}$ are prime. Furthermore, let

$$a_{\mathbb{J}} = -1, \quad d_{\mathbb{J}} = -10240/10241 \; mod \; q_{\mathbb{J}}.$$

The ctEdwards-Jubjub curve, denoted $\mathbb{J}$, is defined as the ctEdwards curve over $F_{q_{\mathbb{J}}}$ with parameters $a = a_{\mathbb{J}}$ and $d = d_{\mathbb{J}}$. Jubjub has order $8 \cdot r_{\mathbb{J}}$, i.e. the curve has a co-factor of 8. The neutral element is denoted $\mathcal{O}_{\mathbb{J}}$. $\mathbb{J}^{(r)}$ denotes the $r_{\mathbb{J}}$-order subgroup of $\mathbb{J}$ and $\mathbb{J}^{(r)*}$ denotes $\mathbb{J}^{(r)} \setminus \{\mathcal{O}_{\mathbb{J}}\}$.

Notation is abused, in the sense that $\mathbb{J}$ is used for both the elliptic curve group as well as the set of points on the curve, i.e. $\{(u, v) \in F_{q_{\mathbb{J}}} \times F_{q_{\mathbb{J}}} : a_{\mathbb{J}} \cdot u^2 + v^2 = 1 + d_{\mathbb{J}} \cdot u^2 \cdot v^2\}$. Generally, points on the ctEdwards form of Jubjub will be denoted (u, v). A point can also be stored in a compressed form by storing the full v-coordinate, but only the sign of the u-coordinate. This is because the v-coordinate already determines the absolute value of the u coordinate.

Jubjub Montgomery form. Let

$$q_{\mathbb{M}} = q_{\mathbb{J}}, \quad A_{\mathbb{M}} = 40962, \quad B_{\mathbb{M}} = 1$$

The Montgomery-Jubjub curve, $\mathbb{M}$, is defined as the Montgomery curve over $\mathbb{F}_{q_{\mathbb{M}}}$, with parameters $A = A_{\mathbb{M}}$ and $B = B_{\mathbb{M}}$. Points on the Montgomery Jubjub curve are denoted (x, y).

Again, notation is abused as $\mathbb{M}$ is used for both the elliptic curve group as well as the set of points on the curve.

Conversion. The curves $\mathbb{J}$ and $\mathbb{M}$ are birationally equivalent. In Zcash, conversion between the two is done in the following way. Let s be the square root of -40964 in $F_{q_{\mathbb{J}}}$, in the range $\{0, \ldots, \frac{q_{\mathbb{J}}-1}{2}\}$. Then, to convert from ctEdwards to Montgomery form, Zcash uses the function

$$\mathsf{CtEdwardsToMont} : \mathbb{J} \to \mathbb{M}$$
$$\mathsf{CtEdwardsToMont}(u, v) = \left(\frac{1+v}{1-v}\ ,\ s \cdot \frac{1+v}{(1-v)\cdot u}\right), \quad [1 - v \neq 0, u \neq 0]$$

To convert in the other direction, i.e. from Montgomery to ctEdwards, Zcash uses the function

$$\mathsf{MontToCtEdwards} : \mathbb{M} \to \mathbb{J}$$
$$\mathsf{MontToCtEdwards}(x, y) = \left(s \cdot \frac{x}{y}\ ,\ \frac{x-1}{x+1}\right), \quad [y \neq 0, x + 1 \neq 0]$$

Miscellaneous Elliptic Curve Functions

The Zcash specification uses many helper functions around the Jubjub curve. The most relevant ones are described in the following.

- The function $\mathsf{repr}_{\mathbb{J}} : \mathbb{J} \to \{0,1\}^{256}$ is an injective function used to get a bit-representative from a ctEdwards Jubjub point. It is defined as

  $$\mathsf{repr}_{\mathbb{J}}((u, v)) = \mathsf{I2LEBSP}_{256}(v \ mod \ q_{\mathbb{J}}) + 2^{255} \cdot \tilde{u}), \text{ where } \tilde{u} = u \mod 2.$$

- The function $\mathsf{Extract}_{\mathbb{J}^{(r)}}$ is used to convert a point in $\mathbb{J}^{(r)}$ to a bit string. It is defined as

  $$\mathsf{Extract}_{\mathbb{J}^{(r)}} : \mathbb{J}^{(r)} \to \{0,1\}^{255}$$
  $$\mathsf{Extract}_{\mathbb{J}^{(r)}}(u, v) = \mathsf{I2LEBSP}_{255}(u)$$

  Even though the v-coordinate is dropped, the function is still injective as we are working in the subgroup $\mathbb{J}^{(r)}$. A proof for this fact is given in the Zcash specification.

- The function $\mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}} : \{0,1\}^{64} \times \{0,1\}^* \to \mathbb{J}^{(r)*}$ is a group hash function that maps bit strings onto arbitrary ("random looking") group elements of $\mathbb{J}^{(r)*}$. It is mostly used to get some fixed generators, as for example in the windowed or homomorphic Pedersen commitment schemes (sections 2.8.5, 2.8.6). Its full definition can be found in the Zcash specification. Some of the most frequently used fixed points on Jubjub are

$$\mathcal{G}^{\text{Sapling}} := \mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(\text{"}Zcash\_G\_\text{"}, \text{""})$$
$$\mathcal{H}^{\text{Sapling}} := \mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(\text{"}Zcash\_H\_\text{"}, \text{""})$$
$$\mathcal{J}^{\text{Sapling}} := \mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(\text{"}Zcash\_J\_\text{"}, \text{""})$$

2.8.3 Pedersen Hash

Intuition. The Pedersen hash function is an algebraic hash function with collision resistance derived from the hardness of the discrete logarithm problem on an elliptic curve. It is based on various works [14][15][7] and has additionally been tailored to fit well within the Zcash design. The collision resistance only holds for inputs of the *same* length. It is not collision resistant for inputs of different lengths.

The high level idea is the following. The message (a bitstring) is split into n segments. Each of the segments is injectively encoded as an integer that is smaller than the elliptic curve group order (the length of the segments in the first step is chosen so that such an encoding is possible). Then, each encoded segment serves as a scalar in a scalar-multiplication with some fixed point on the elliptic curve (a different fixed point for each segment) and all the resulting points are added together. The hash output is the first coordinate of this final point.

Definition. Let us formally define the function as it is used in Zcash, with Jubjub as the underlying elliptic curve. Let $c = 63$. First, pad the input message $M \in \{0,1\}^N$ with zeros so that the padded message M' is a multiple of 3 bits long. Then, split M' into segments $M' = M_1||M_2||\ldots||M_n$, where $M_1, \ldots, M_{n-1}$ are $3 \cdot c = 189$ bits long, and M_n is at most 189 bits long (but the length of every segment, including M_n, is a multiple of 3).

Now, define the encoding $\langle\cdot\rangle : \{\{0,1\}^3\}^* \to \{-\frac{r_{\mathbb{J}}-1}{2}, \ldots, \frac{r_{\mathbb{J}}-1}{2}\} \setminus \{0\}$ of a segment M_i as follows:

1. Let $k_i = length(M_i)/3$.
2. Split M_i into k_i chunks of 3 bits, $M_i = [m_1, m_2, \ldots, m_{k_i}]$.
3. Write a chunk m_j as $m_j = [s_0, s_1, s_2]$, and let $enc(m_j) = (1 - 2s_2) \cdot (1 + s_0 + s_1)$.

4. Let $\langle M_i \rangle = \sum_{j=1}^{k_i} enc(m_j) \cdot 2^{4(j-1)}$.

Note that the maximum length of the segments (189 bits) has been chosen exactly so that the encoding lies in the specified range.

Now, define

$$\mathsf{PedersenHashToPoint} : \{0,1\}^{64} \times \{0,1\}^* \to \mathbb{J}^{(r)}$$
$$\mathsf{PedersenHashToPoint}(D, M) = \sum_{i=1}^{n} [\langle M_i \rangle] \cdot \mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(D, i)$$

and then

$$\mathsf{PedersenHash} : \{0,1\}^{64} \times \{0,1\}^* \to \{0,1\}^{255}$$
$$\mathsf{PedersenHash}(D, M) = \mathsf{Extract}_{\mathbb{J}^{(r)}}(\mathsf{PedersenHashToPoint}(D, M))$$

2.8.4 Mixing Pedersen Hash

The mixing Pedersen hash function is defined as

$$\mathsf{MixingPedersenHash} : \mathbb{J} \times \{0, \ldots, r_{\mathbb{J}} - 1\} \to \mathbb{J}$$
$$\mathsf{MixingPedersenHash}(P, x) = P + [x] \cdot \mathcal{J}^{\text{Sapling}}$$

2.8.5 Windowed Pedersen Commitment

The windowed Pedersen commitment scheme is a commitment scheme with commitments on Jubjub. It is defined as

$$\mathsf{WindowedPedersenCommitment} : \{1, \ldots, r_{\mathbb{J}}\} \times \{0,1\}^* \mapsto \mathbb{J}$$
$$\mathsf{WindowedPedersenCommitment}_r(s) =$$
$$\mathsf{PedersenHashToPoint}(\text{"}Zcash\_PH\text{"}, s) + [r]\ \mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(\text{"}Zcash\_PH\text{"}, \text{"}r\text{"})$$

2.8.6 Homomorphic Pedersen Commitment

The homomorphic Pedersen commitment scheme is a homomorphic commitment scheme with commitments on Jubjub. It is defined as

$$\mathsf{HomomorphicPedersenCommitment} : \{1, \ldots, r_{\mathbb{J}}\} \times \{0,1\}^{64} \times \{1, \ldots, 2^{64} - 1\} \mapsto \mathbb{J}$$
$$\mathsf{HomomorphicPedersenCommitment}_{rcv}(D, v) =$$
$$[v]\ \mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(D, \text{"}v\text{"}) + [rcv]\ \mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(D, \text{"}r\text{"})$$

2.8.7 BLAKE2 Hash Function

Zcash uses the BLAKE2 hash function. The BLAKE2 hash function was proposed in 2012 by Aumasson et al. [3]. It is assumed to be collision

resistant. Zcash uses the BLAKE2s variant to construct a function

$$\mathsf{BLAKE2s} - \ell : \{0,1\}^{64} \times \{0,1\}^{*} \to \{0,1\}^{\ell},$$

that is assumed to be collision resistant. More information on the inner workings of BLAKE2 is given in section 4.1.21.

Chapter 3

Zcash Description

This chapter describes the Zcash cryptocurrency [2]. Zcash was developed to enhance privacy of cryptocurrencies such as Bitcoin [25] and in fact, is an extension of the Bitcoin protocol. The first Zcash block was mined on october 28th, 2016. Zcash is based on ideas from Zerocash [8], a protocol proposed in 2014 by Ben-Sasson et. al. However, the first version of Zcash (called *Sprout*) already differs substantially from the original Zerocash protocol. In 2018, another major update (*Sapling*) introduced further considerable changes. The next large update, called *Orchard*, is currently running on Zcash's testing network.

The first section of this chapter, section 3.1, gives a simplified overview of how Bitcoin works. A basic understanding of this is necessary in order to understand the design choices made in Zcash.

Section 3.2 contains an intuitive description of the main ideas used in Zcash to increase privacy. This is done by starting with the simplified Bitcoin protocol of section 3.1 and strengthening its privacy step-by-step until ending up with something very similar to Sprout, the first version of Zcash. Section 3.3 then describes the current version of Zcash (Sapling) and presents the precise statements that are being proven in zero-knowledge.

3.1 Bitcoin Transaction Structure

This section gives a simplified description of how a transaction is conducted in Bitcoin. Note that this is not a full description of the Bitcoin protocol, but suffices for our purposes.

Consensus. The Bitcoin network is a peer-to-peer network. Users send bitcoin between each other by broadcasting *transactions* to the network. Depending on the location of some node in the network, it will receive transactions in a particular order. Another node might see the transactions in a

different order, or receive different transactions altogether. However, for the protocol to function, nodes must have a way to agree on which transactions were made in the past, and in what order. This is referred to as the *consensus* problem. A solution for it, utilizing *proof of work*, is the main contribution of the original Bitcoin paper. As a direct consequence of achieving consensus, nodes are able to maintain the *blockchain*, a distributed ledger that contains a list of all past transactions. For our purposes, the concrete technicalities of the consensus mechanism are irrelevant, so the details are abstracted. In the discussion that follows, it is simply assumed that every node has access to the same, authentic ledger that contains a list of all past transactions.

Addresses. Each participant of the Bitcoin network holds a signing/verifying-key pair of a digital signature scheme. The verifying key essentially serves as an address to direct a payment, while the signing key is used to prove being the recipient of a particular payment.

Transactions. In general, a *transaction* consists of multiple *inputs* and *outputs*. An output is a pair (v, vk) of a value v (the amount of bitcoin being sent) and a recipient verifying key, vk. On the other hand, an input is a pointer to an output of some past transaction, essentially spending the money that was received in that specific output [1]. If Alice creates a transaction, she must prove that for each input, the referenced output was really sent to her. She does this by providing a digital signature over the hash of the new transaction, using the signing key belonging to the verifying key in that input. Anybody can verify the signature using the verifying key and will be convinced that the signing key is owned by Alice. To prevent creating or destroying money, the sum of all input amounts must be equal to the sum of all output amounts. This can easily be checked by anybody receiving the transaction. A transaction that violates this balancing property will be rejected by the network.

Double Spending. An output of a transaction that has not yet been used as the input to another transaction is called an *unspent transaction output*, or *UTXO*. Whenever a transaction is made, the nodes have to check that all inputs are UTXOs. To do this, each node needs to be aware of the set of all current UTXOs, called the *UTXO set*. As we assume that nodes are in consensus about the current state of the blockchain, it follows that they also agree on the UTXO set as the UTXO set can be computed from the blockchain. Hence, just as with the blockchain itself, we treat the current UTXO set as a shared global state, to which every node has access to. Now, to prevent double spending, every transaction that includes an input which is not contained in the UTXO set is rejected by the network.

[1] In practice, this pointer consists of a *transaction id* and an output index. The transaction id is essentially the hash of a transaction and is used to uniquely identify it.

Complete Transaction Example. We now go through a sample transaction to see how it all comes together. Assume Alice wants to send 10 bitcoin to Bob. Alice holds a key pair (sk_a, vk_a) and Bob holds the key pair (sk_b, vk_b). We assume that Alice has an authentic copy of vk_b. To send the money to Bob, Alice must now create a transaction and send it to the Bitcoin network.

It is likely that Alice does not have an unspent output containing exactly 10 bitcoin. In such a case, she can aggregate funds by specifying multiple inputs. Assume that Alice has two unspent outputs, $out_1^{old} = (5, vk_a)$ and $out_2^{old} = (7, vk_a)$, containing 5 and 7 bitcoin respectively. Alice creates a transaction with inputs out_1^{old}, out_2^{old}. As she wants to send 10 bitcoin to bob, she would include an output $out_1^{new} = (10, vk_b)$. The inputs contain a total of 12 bitcoin, but she only wants to send 10 bitcoins to Bob, so she needs to balance the transaction. To do so, she would also create a change output, $out_2^{new} = (2, vk_a)$, sending the spare money back to herself. In addition, for each of the two inputs, she provides a digital signature over the transaction hash.

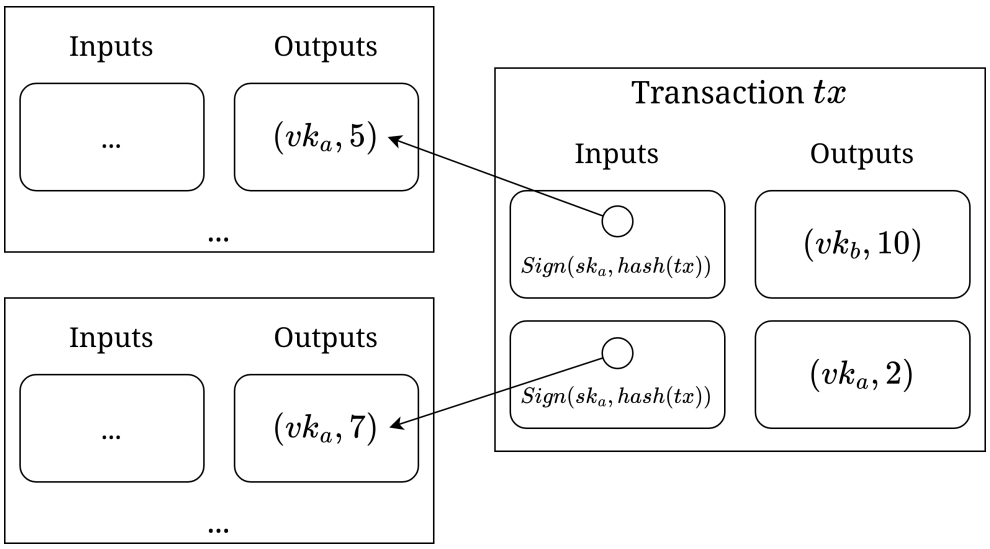

Figure 3.1: An example transaction where Alice is sending 10 bitcoin to Bob.

A node that receives the transaction will check that:

1. Inputs and outputs balance, i.e. value is preserved.
2. The inputs are in the UTXO set.
3. Each signature verifies, using the verification key of the respective input.

If any of the above checks did not go through, the transaction is rejected.

If everything checks out, the transaction is accepted and appended to the blockchain. Implicitly, this will remove the inputs from and add the outputs to the UTXO set. At this point, the transaction has been completed.

3.2 Design of an Anonymous Currency

This section intuitively explains the core ideas used in Zcash, by building a privacy-enhanced transaction structure step-by-step. The idea of an incremental presentation is taken from the Zerocash paper [8], but has been adapted for our purposes. The simplified version of the Bitcoin protocol from section 3.1 will serve as a starting point.

Consider a transaction in which Alice is sending some v bitcoin to Bob. Assume that she has an unspent output containing exactly the value v. In this case, she would send the money using a Bitcoin transaction with exactly one input and one output, as in figure 3.2.

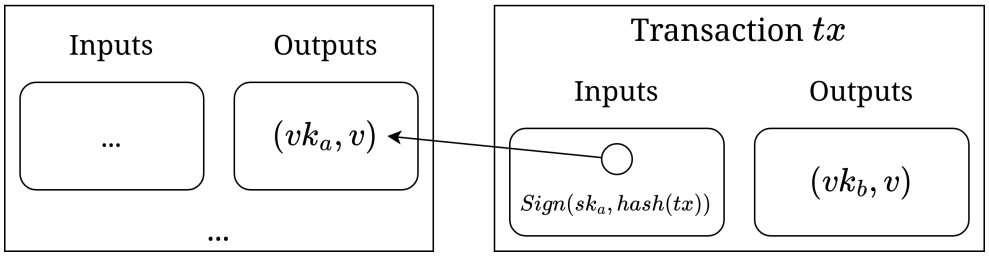

Figure 3.2: An example transaction where Alice is sending v bitcoin to Bob using an unspent output that contains exactly v bitcoin.

The following data is revealed by the transaction:

- The unspent output that is being spent in this transaction, which includes Alice's public key
- The output that is being created in this transaction, which includes Bob's public key
- The transaction amount v.

Using cryptographic tools, the transaction format is to be modified so that it avoids leaking all of the above information, while still supporting the same functionality as an ordinary Bitcoin transaction. The final transaction format will be very similar to *JoinSplit*, the original format used in Zcash. We start off by looking at how we can prevent revealing Alice's unspent output and her verifying key.

Prevent revealing Alice's unspent output and verifying key

Recall that in a Bitcoin transaction, Alice would reference the UTXO

she is spending as an input in the transaction. Additionally, she provides a signature proving that this UTXO really belongs to her. This is changed in the following way. Instead of revealing an UTXO with value v and providing a signature, Alice proves the following statement in zero-knowledge:

"For a given value v, I know a verifying key vk_a and a signing key sk_a such that

- there exists a transaction output (v, vk_a) in the set of all past transaction outputs
- sk_a is the signing key corresponding to vk_a"

In order to be able to verify the proof, nodes need a list of all transaction outputs that have ever been made. Just as with the UTXO set, we can assume that such a list is part of the shared global state, as every node is able to compute such a list from the public ledger.

Using zero-knowledge proofs in the above way, Alice neither reveals vk_a nor the exact output that she is spending, but still convinces everybody that an appropriate output exists.

However, an immediate problem is that our new design makes it impossible to compute the UTXO set, as nodes cannot tell which outputs have been spent, and which outputs have not. This means that the current version is susceptible to double spends.

A comparatively smaller problem is that the proof that a transaction is contained in a list is terribly inefficient. We put a pin in the double-spend problem and first concern ourselves with proof efficiency.

Improve zero-knowledge proof efficiency

Keeping a list of all transaction outputs is unwieldy and makes the ZK proof prohibitively complex. Instead, to prove that a certain transaction was conducted in the past, a Merkle tree of fixed height is used, instantiated with a collision-resistant hash function. The leaves of the Merkle tree contain all past transaction outputs, as in figure 3.3.

In Zcash, this Merkle tree is called the *note commitment tree* (why will become apparent in a couple of paragraphs). As the note commitment tree can be computed from the public ledger, we treat it as part of the shared global state. In particular, the current root value rt, is known by every node. To prove that there is some suitable transaction output, Alice proves that such an output is part of the note tree. The zero-knowledge statement changes accordingly to:

"For a given note commitment tree root rt and value v, I know a verifying key vk_a, a signing key sk_a such that

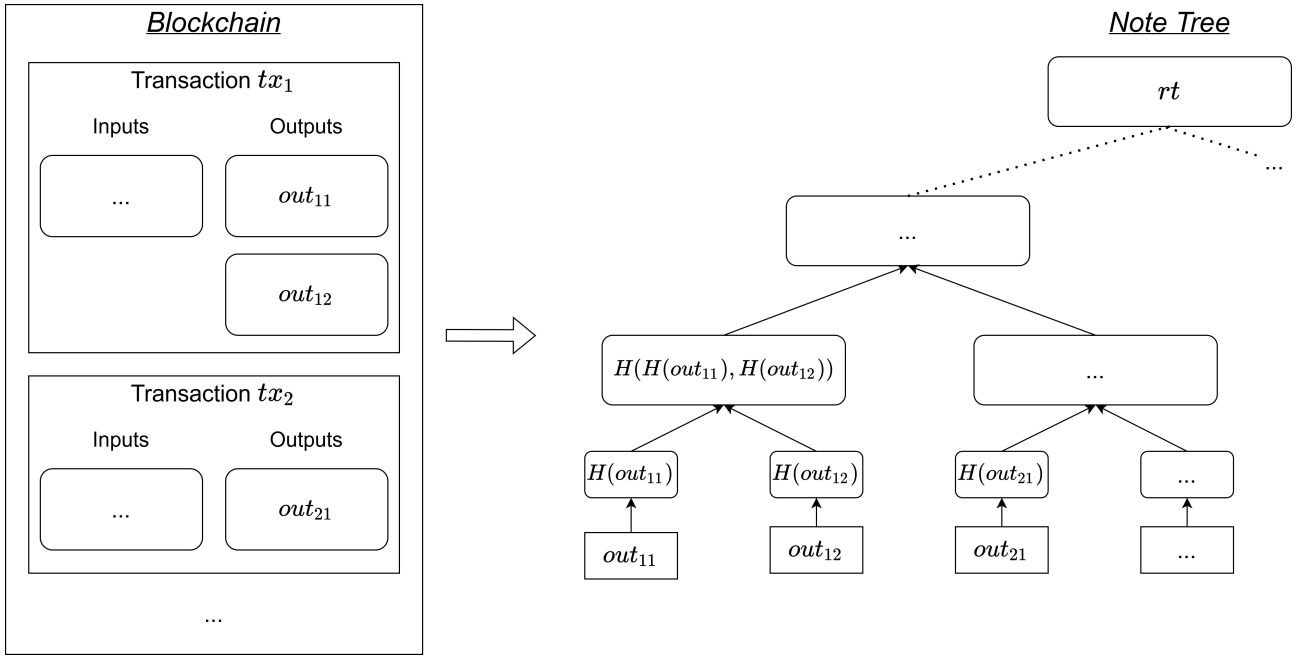

Figure 3.3: An illustration of how the note tree is computed from the blockchain.

- (v, vk_a) is a leaf of a Merkle tree with root rt
- sk_a is the signing key corresponding to vk_a"

This proof improves exponentially on the previous one using the simple list. Intuitively, this is because to prove that a transaction output is part of a Merkle tree with root rt, it suffices to provide the co-path of the leaf that contains the transaction. As the co-path has logarithmic length in the number of past transaction outputs, we get an exponential improvement.

Changing the keys

For an arbitrary signature scheme, it is not necessarily easy to prove that some key is the signing key corresponding to a verifying key. To address this problem, Zcash does not use a signing/verifying key pair. Instead, each participant possesses a *spending key* sk_a and a *paying key* pk_a. The spending key is generated as a uniformly random bit string. The paying key pk_a is computed as $pk_a = \mathsf{PRF}_{sk_a}(0)$, for a pseudorandom function PRF. Note that the security properties of the PRF render it infeasible to compute the spending key from the paying key. The paying key takes over the function of the verifying key before, while the spending key takes the function of the signing key. The zero-knowledge statement is changed to:

"For given note commitment tree root rt and value v, I know a paying

key pk_a, a spending key sk_a such that

- (v, pk_a) is a leaf of a Merkle tree with root rt
- $pk_a = \mathsf{PRF}_{sk_a}(0)$"

Prevent Bob's public key from being revealed

In Bitcoin, Alice includes Bob's verifying key in the output so that when he spends the output at a later point in time, he is able to prove that he knows the corresponding signing key. In Zcash, a commitment scheme is used to hide Bob's paying key.

Concretely, when creating the output, Alice picks a random value r^{new} and computes $cm = commit_{r^{new}}(pk_b)$. Now, set the output to be (v, cm). Additionally, Alice proves (in zero-knowledge) that cm is computed correctly.

Note that now Alice's input has the format $(v, commit_{r^{old}}(pk_a))$ which we need to consider in our zero-knowledge proof. Crucially, Alice is still able to prove in zero-knowledge that she knows the spending key corresponding to the paying key that is contained inside the commitment. The zero-knowledge proof changes to:

"For given note commitment tree root rt, value v and output commitment cm, I know paying keys pk_a, pk_b, a spending key sk_a and values r^{old}, r^{new} such that

- $(v, commit_{r^{old}}(pk_a))$ is a leaf of a Merkle tree with root rt
- $pk_a = \mathsf{PRF}_{sk_a}(0)$
- $cm = commit_{r^{new}}(pk_b)$"

The hiding property of the commitment scheme makes sure that no one is able to learn Bob's public key just from the commitment itself. The binding property guarantees that the output uniquely determines the recipient, i.e. Bob can be sure that no one else is able to spend his money.

There is a problem with this approach, though. Alice needs to know r^{old} in order to complete the proof. However, it was not explained how Alice received r^{old} in the first place, or how she communicates the value r^{new} to Bob. Certainly, the r-value should not be put in the transaction output in plaintext, because then everybody learns r, potentially undermining security properties of the commitment scheme. Instead, addresses are extended to not only contain the paying key pk, but also a public key pk^{enc} of a public key encryption scheme, where the corresponding secret key sk^{enc} is known only to the holder of the address. In practice, these keys are derived from the spending key sk. When Alice sends money to Bob, she creates the output

as $(v, commit_{r^{new}}(pk_b), Enc_{pk_b^{enc}}(r^{new}))$. Bob can decrypt r^{new} using sk_b^{enc} and is able to use it when he wants to spend the output.

To ensure that Alice actually encrypted the real r^{new} value, Bob decrypts the ciphertext and re-computes cm. He accepts the payment only if it matches the commitment that was released by Alice.

Conceal transaction amount v

In the current construction, a transaction still contains the value v in plaintext. Fortunately, it is easy to extend the construction to hide v, by simply putting it inside the commitment. To communicate v to Bob, Alice sends it in encrypted form alongside r^{new}. The zero-knowledge statement will have to be altered accordingly. The updated statement is:

"For given note commitment tree root rt, and output commitment cm, I know paying keys pk_a, pk_b, a spending key sk_a and values r^{old}, r^{new}, v such that

- $commit_{r^{old}}(pk_a, v)$ is a leaf of a Merkle tree with root rt
- $pk_a = \mathsf{PRF}_{sk_a}(0)$
- $cm = commit_{r^{new}}(pk_b, v)$"

In this way, v is never revealed but it is proven implicitly that input and output commitment contain the same v, i.e. the transaction balances.

Prevent double spending

This paragraph addresses the double spending problem. Remember that one needs to be able to determine whether some input has already been spent. The idea is the following. When creating a transaction, Alice also includes a random value ρ^{new} in the output commitment. In addition, she sends ρ^{new} to Bob in encrypted form alongside r^{new}.

Then, when Bob spends the output at a later point in time, he reveals what's called the *nullifier* $nf = \mathsf{PRF}_{sk_b}(\rho^{new})$, for a pseudorandom function PRF. In addition, Bob proves in zero-knowledge that the nullifier is computed correctly.

A node accepts the transaction only if the nullifier has never been revealed before. This works because if Bob attempts to spend an output twice, he must necessarily reveal the same nullifier twice, and the network will reject one of the transactions as invalid.

In order to check whether a nullifier has been revealed before, every node must have a list of all revealed nullifiers. This list can be computed from the blockchain, and we therefore consider it as part of the shared state.

Because Bob computes the nullifier using a pseudorandom function indexed with his secret key, he is the only one that is able to compute it. In particular Alice, who knows ρ^{new}, cannot tell when Bob spends the money by checking whether a specific nullifier is revealed.

The zero knowledge statement is updated to include the nullifier computation as follows:

"For given note commitment tree root rt, output commitment cm and nullifier nf, I know paying keys pk_a, pk_b, a spending key sk_a and values $r^{old}, r^{new}, v, \rho^{old}, \rho^{new}$ such that

- $commit_{r^{old}}(pk_a, v, \rho^{old})$ is a leaf of a Merkle tree with root rt
- $pk_a = \mathsf{PRF}_{sk_a}(0)$
- $cm = commit_{r^{new}}(pk_b, v, \rho^{new})$
- $nf = \mathsf{PRF}_{sk_a}(\rho^{old})$"

In Zcash, the tuple (pk, v, ρ, r) is called a *note*. The corresponding commitment $commit_r(pk, v, \rho)$ is called a *note commitment*. This explains why the Merkle tree is called note commitment tree, as the outputs that are contained in the leaves are actually note commitments.

Integrity of the transaction

In the protocol of section 3.1, transactions are integrity protected. This is because for each input, a signature over the whole transaction is provided. Concretely, this means that anyone who does not know the secret keys to *all* inputs cannot tamper with the transaction, as they would be unable to create a new signature.

In the current design, no such integrity protection mechanism is in place. The creator of a transaction proves knowledge of the secret keys to the inputs, but does not provide a signature. In case the zero-knowledge proof is malleable, this becomes a problem.

To get similar integrity protection as in Bitcoin, Zcash makes use of a pseudorandom function as well as a one-time signature scheme. Concretely, when creating a transaction, Alice performs the following additional steps:

1. Sample a key pair (vk^{sig}, sk^{sig}) of a one-time signature scheme.
2. Compute $h = PRF_{sk_a}(vk^{sig})$.
3. Sign the whole transaction (including h) with sk^{sig} to obtain a signature σ.

She includes vk^{sig}, h and σ in the transaction. Also, the zero-knowledge statement is extended to contain a proof that h was computed correctly. The updated zero-knowledge statement is

"For given note commitment tree root rt, output commitment cm, nullifier nf, verifying key vk^{sig} and value h, I know paying keys pk_a, pk_b, a spending key sk_a and values $r^{old}, r^{new}, v, \rho^{old}, \rho^{new}$ such that

- $commit_{r^{old}}(vk_a, v, \rho^{old})$ is a leaf of a Merkle tree with root rt
- $pk_a = \mathsf{PRF}_{sk_a}(0)$
- $cm = commit_{r^{new}}(pk_b, v, \rho^{new})$
- $nf = \mathsf{PRF}_{sk_a}(\rho^{old})$
- $h = \mathsf{PRF}_{sk_a}(vk^{sig})$"

Intuitively, we can think about the value h as a way for Alice to approve a particular one-time verifying key.

To see why this works, let us think about an adversary Charlie trying to tamper with Alice's transaction. Charlie knows neither the spending key of the input, sk_a, nor the one-time signing key sk^{sig}. Charlie might attempt to change the content of the transaction while keeping the one-time key pair the same. In this case, Charlie's transaction verifying correctly would mean he broke the security properties of the one-time signature scheme, because he must provide a signature on a new message by just knowing the verifying key vk_{sig}. If, on the other hand, he changes the one-time key pair, he has no way of computing h, as the spending key sk_a is required to compute the PRF (and he has to prove that h is compute correctly). Thus, Charlie has no way of modifying the transaction.

Preventing the Faerie Gold attack

The current scheme is susceptible to the so-called *Faerie Gold Attack*. In the attack, an adversary would send coins to the victim several times, and purposefully include duplicate ρ-values. This means that when spending the output notes, the same nullifier must be revealed and the victim can only spend one of the notes. Notice that if ρ was picked truly at random, the probability that the same value was picked twice is negligible.

To prevent the attack, Alice does not simply pick ρ^{new} u.a.r anymore, but instead picks a value φ u.a.r and compute the nullifier as $\rho^{new} = \mathsf{PRF}_{\varphi}(h)$, for the value $h = \mathsf{PRF}_{sk_a}(vk^{sig}, nf)$. Note that the computation of the h-value was updated to also include the nullifier nf. This ensures that the h-values for two different transactions differ, hence the ρ^{new}-values will differ as well.

The zero-knowledge statement is extended to contain a proof that the nullifier was computed in this way. The full statement is now

"For given note commitment tree root rt, output commitment cm, nullifier nf and value h, I know paying keys pk_a, pk_b, a spending key sk_a and values $r^{old}, r^{new}, v, \rho^{old}, \rho^{new} \varphi$ such that

- $commit_{r^{old}}(pk_a, v, \rho^{old})$ is a leaf of a Merkle tree with root rt
- $pk_a = \mathsf{PRF}_{sk_a}(0)$
- $cm = commit_{r^{new}}(pk_b, v, \rho^{new})$
- $nf = \mathsf{PRF}_{sk_a}(\rho^{old})$
- $h = \mathsf{PRF}_{sk_a}(vk^{sig}, nf)$
- $\rho^{new} = \mathsf{PRF}_{\varphi}(h)$"

Note that Alice can still pick φ freely. To perform the Faerie Gold attack with the new design, Alice would have to pick two values φ_1, φ_2 so that $\mathsf{PRF}_{\varphi_1}(h_1) = \mathsf{PRF}_{\varphi_2}(h_2)$ for $h_1 \neq h_2$. This is assumed to be infeasible.

Allowing multiple inputs and outputs

Up until this point, we have considered the simplified situation where a transaction contains exactly one input and output. Of course, transactions with multiple inputs and outputs need to be supported. To do so, the following changes need to be implemented.

- For inputs, perform the same steps as before, but *for each* input separately. Compute and release a nullifier nf as well as a value h. In addition, prove (in zero-knowledge) knowledge of the spending key, and that nf and h were computed correctly.
- For each output, generate ρ^{new} and r^{new} as in the one-output case. For every output commitment, prove it has been computed correctly.
- Prove that the sum of input values is equal to the sum of output values. This was proven implicitly before, as there was only a single value v for both input and output.

These changes will conclude the journey of creating an anonymous transaction type. To summarize everything, the next section goes through a complete example.

Complete transaction example

Suppose that Alice wants to spend n inputs, where the i-th input is of the form $cm_i^{old} = commit_{r_i^{old}}(pk_{a,i}, v_i^{old}, \rho_i^{old})$. For each input, Alice knows r_i^{old}, $pk_{a,i}$, v_i^{old}, ρ_i^{old} as well as the spending key $sk_{a,i}$ corresponding to the paying key $pk_{a,i}$. She would like to send the money to m different addresses, $(pk_{b,1}, pk_{b,1}^{Enc}), \ldots, (pk_{b,m}, pk_{b,m}^{Enc})$, sending an amount

v_i^{new} to $pk_{b,i}$. She needs to make sure that the transaction balances, i.e. $\sum_{i=1}^{n} v_i^{old} = \sum_{j=1}^{m} v_j^{new}$.

First, Alice samples a one-time signature pair (vk^{sig}, sk^{sig}). Then, she proceeds as follows.

for each $i \in \{1, \ldots, n\}$:

1. Compute $nf_i = \mathsf{PRF}_{sk_{a,i}}(\rho_i)$
2. Compute $h_i = \mathsf{PRF}_{sk_{a,i}}(vk_{sig}, nf_i)$

Then she picks a random value φ and for each $j \in \{1, \ldots, m\}$:

1. Pick random values r_j^{new}
2. Compute $\rho_j^{new} = \mathsf{PRF}_{\varphi}(i, h_1, \ldots, h_n)$
3. Compute $cm_j = commit_{r_j^{new}}(pk_{b,j}, v_j^{new}, \rho_j^{new})$
4. Compute $C_j = Enc_{pk_{b,j}^{Enc}}(\rho_j^{new}, r_j^{new}, v_j^{new})$

Then, she computes a zero-knowledge proof π for the statement:

"Given input:

- note commitment tree root rt
- output commitments $cm_1, \ldots, cm_m$
- nullifiers $nf_1, \ldots, nf_n$
- values $h_1, \ldots, h_n$

I know

- paying keys $vk_{a,1}, \ldots, vk_{a,n}$, $vk_{b,1}, \ldots, vk_{b,m}$
- spending keys $sk_{a,1}, \ldots, sk_{a,n}$
- values $r_1^{old}, \ldots, r_n^{old}$, $r_1^{new}, \ldots, r_m^{new}$
- values $v_1^{old}, \ldots, v_n^{old}$, $v_1^{new}, \ldots, v_m^{new}$
- values $\rho_1^{old}, \ldots, \rho_n^{old}, \rho_1^{new}, \ldots, \rho_m^{new}$
- value φ

such that the following statements hold:

- $\forall i \in \{1, \ldots, n\}$: $commit_{r_i^{old}}(pk_{a,i}, v_i^{old}, \rho_i^{old})$ is a leaf of a merkle tree with root rt
- $\forall i \in \{1, \ldots, n\}$: $pk_{a,i} = \mathsf{PRF}_{sk_{a,i}}(0)$
- $\forall i \in \{1, \ldots, n\}$: $nf = \mathsf{PRF}_{sk_{a,i}}(\rho_i^{old})$

- $\forall j \in \{1, \ldots, m\}$: $cm_i = commit_{r_i^{new}}(pk_{b,i}, u_i, \rho_i^{new})$
- $\forall i \in \{1, \ldots, n\}$: $h_i = \mathsf{PRF}_{sk_{a,i}}(vk_{sig}, nf_i)$
- $\forall j \in \{1, \ldots, m\}$: $\rho_j^{new} = \mathsf{PRF}_{\varphi}(i, h_1, \ldots, h_n)$
- $\sum_{i=1}^{n} v_i = \sum_{j=1}^{m} u_j$ "

Finally, she creates a signauture σ over all public values of the transaction, using sk^{sig}. The final transaction contains:

- commitments $cm_1^{new}, \ldots, cm_m^{new}$
- nullifiers $nf_1, \ldots, nf_n$
- values $h_1, \ldots, h_n$
- ciphertexts $C_1, \ldots, C_m$
- the one-time-signature verifying key vk^{sig}
- signature σ
- the zero-knowledge proof π.

A node receiving the transaction would verify the proof π as well as the signature σ using the one-time verifying key pk^{sig}. If any verification fails, the transaction would be rejected. Otherwise, the transaction is accepted and appended to the blockchain. As a result, the new commitments $cm_1, \ldots, cm_m$ are inserted into the note commitment tree, and the revealed nullifiers $nf_1, \ldots, nf_n$ are added to the nullifier set.

3.2.1 Summary

This section showed how Zcash uses zero-knowledge proofs to increase privacy of transactions. The zero-knowledge statements became increasingly complex towards the final design. Except for some subtle differences, this final design is essentially the original Zcash transaction design, called *JoinSplit*. The associated zero-knowledge proof statement is called the *JoinSplit statement*. In 2018, a major update called Sapling introduced considerable changes. These changes are discussed in section 3.3.

3.3 The Sapling Update

This section discusses the changes to the transaction structure that were introduced in the Sapling update. While section 3.2 serves to give an intuition to the reader of how an anonymous currency can work, the purpose of this section is do introduce the exact statements that are used in the Sapling version of Zcash. It will also use the same notations for keys, functions, etc.

as the Zcash specification, to give the reader an opportunity to cross-check easily.

Section 3.3.1 describes the most important changes and the motivation behind them on a conceptual level. Section 3.3.2 defines the primitives that are to be used in the zero-knowledge proof, such as commitment schemes and hash functions. Then, sections 3.3.3 and 3.3.4 will define the exact zero-knowledge statements that are proven in the Sapling version of Zcash.

3.3.1 Conceptual Changes

This section describes the changes that the Zcash design has undergone in the Sapling update. As all the changes are interconnected, it is difficult to find an order for presentation that is really satisfactory. We have opted to first describe the changes to the key schedule (i.e. the procedure in which cryptographic keys and addresses are generated), as it serves as a catalyst for numerous other changes down the line.

Keys and Addresses

The format of cryptographic keys and addresses has changed considerably in Sapling. To understand the motivation for this change, recall the key schedule of Sprout, depicted in figure 3.4.

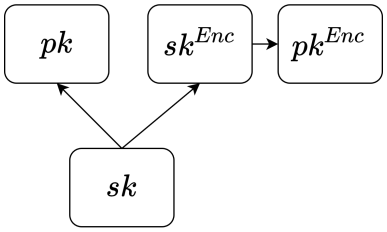

Figure 3.4: The Sprout key schedule. Figure adapted from the Zcash specification.

Essentially, the spending key sk serves as a master and is used to derive both the paying key $pk = \mathsf{PRF}_{sk}(0)$ as well as a public/private key pair (sk^{enc}, pk^{enc}) of an encryption scheme. Additionally, whenever spending funds, knowledge of sk has to be proven in the zero-knowledge proof.

One reason why this key schedule is suboptimal is that it does not allow for fine grained access. To give an example, the spending key is needed even just to identify outgoing transactions on the blockchain. This means that there is no way to allow someone to see one's transactions without giving full spending authority as well.

The Sapling key schedule, on the other hand, allows for much finer grained access, as there is a hierarchy of keys, and each level provides increasing functionality.

The new key schedule. Figure 3.5 shows the new key schedule. The next couple of paragraph explain the format of the different keys, how to derive them and their function.

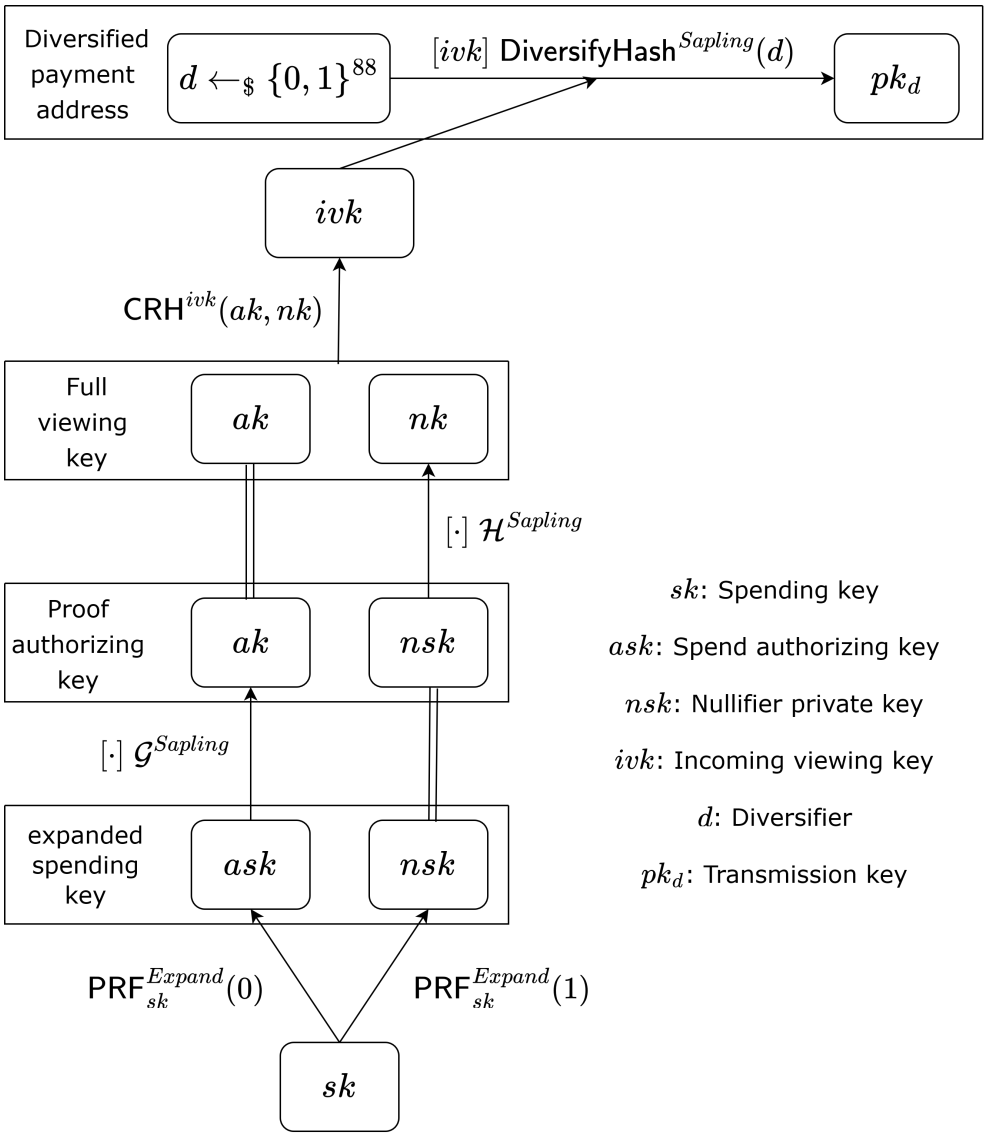

Figure 3.5: The Sapling key schedule. Two parallel lines indicate that the same key is used as part of different key tuples. Figure adapted from the Zcash specification.

The ***spending key*** $sk \in \{0,1\}^{256}$ functions as a master key that is used to

derive all other keys. It is picked as a uniformly random string of bits. The spending key is *only* used to derive keys, but never used directly in a zero-knowledge proof or anywhere else.

Utilizing a PRF called PRF^{Expand}, the spending key is used to derive a tuple called the ***expanded spending key***, which consists of two keys: The *spend authorizing key* $ask \in \mathbb{Z}_{r_{\mathbb{J}}}$, which is derived as $ask = \mathsf{PRF}^{Expand}_{sk}(0)$ and the *nullifier private key* $nsk \in \mathbb{Z}_{r_{\mathbb{J}}}$, which is derived similarly to *ask* as $nsk = \mathsf{PRF}^{Expand}_{sk}(1)$.[2]

It is helpful to think of the expanded spending key as a bisection of the old spending key *sk* in Sprout. Knowledge of both *ask* and *nsk* must be demonstrated when spending funds. However, while knowledge of *nsk* is proven directly in zero-knowledge (as with *sk* in Sprout), knowledge of *ask* is demonstrated outside the zero-knowledge proof using a so-called *spend authorization signature*. The signature scheme is further explained below, in the paragraph "Spend Authority".

The reason for this separation is that computationally limited devices can delegate the creation of the zero-knowledge proof without sharing the spend authorizing key *ask*. The third party is able to create the zero-knowledge proof, but does not have the power to spend any notes on its own. Note that this was impossible in Sprout: To delegate the creation of the zero-knowledge proof, one must share the spending key *sk*, giving full power to the external party.

The next key tuple is called the ***proof authorizing key*** and contains *nsk* as well as a new key $ak \in \mathbb{J}$ derived as $ak = [ask]\ \mathcal{G}^{\text{Sapling}}$. The proof authorizing key allows to create the zero-knowledge proof (see paragraph "spend authority" below for how *ak* is used in the zero-knowledge proof). However, the proof authorizing key does not give the capability to spend any notes, as *ask* would be required for this.

The last key tuple is called the ***full viewing key***[3]. It contains the key *ak* as well as a key $nk \in \mathbb{J}$ derived from *nsk* as $nk = [nsk]\ \mathcal{H}^{\text{Sapling}}$. The key *nk* is used to compute the nullifier as $nf = \mathsf{PRF}^{nfSapling}_{nk}(\rho)$, using a PRF called $\mathsf{PRF}^{nfSapling}$. Note that the nullifier is computed in a similar way in Sprout, except that there, the PRF is indexed directly by the spending key *sk*. However, ρ is derived differently in Sapling (see below). Knowledge of *nk* allows to identify outgoing transactions by recomputing the nullifier and comparing it with the published nullifier. If they match, the transaction was sent from the party owning *nk*.

[2] Technically, the PRF does not directly map to $\mathbb{Z}_{r_{\mathbb{J}}}$. Instead, one interprets the PRF output (which is a bitstring) as a number, and then squishes to a scalar in $\mathbb{Z}_{r_{\mathbb{J}}}$

[3] Technically, the full viewing key contains another key called *ovk*. The role of this key is not discussed here, as it is irrelevant for our purposes.

The ***incoming viewing key*** ivk is derived from ak and nk using a collision resistant hash function CRH^{ivk} as $ivk = \mathsf{CRH}^{ivk}(ak, nk)$. The incoming viewing key allows to view all *incoming* transactions. How it works is that in Sapling, the incoming viewing key is used to decrypt the ciphertext that contains the randomness of commitments, etc. (see "Key Agreement" below). Therefore, to find incoming transactions on the blockchain, one goes through all transactions one by one and tries to decrypt the associated ciphertexts. A successful decryption indicates that a transaction was sent to an address derived from ivk.

Finally, the incoming viewing key is then used to derive ***diversified addresses***, which are the counterpart of the paying key in Sprout. The advantage of diversified addresses over paying keys is that one can create many diversified addresses for a single spending key, where in Sprout, a new spending key must be sampled for every paying key.

A diversified address is derived using $\mathsf{DiversifyHash}^{Sapling}$, a function that essentially maps bit strings onto arbitrary elements on the Jubjub curve. To compute a diversified address, pick a random diversifier $d \leftarrow \{0,1\}^{88}$, and compute $g_d = \mathsf{DiversifyHash}^{Sapling}(d)$. Then, compute $pk_d = [ivk]\ g_d$ and set the address to be (d, pk_d). Multiple addresses can be generated by repeatedly picking new d values.

Note Format

In Sprout, a note contains four values: The paying key pk, the value v, a value ρ to compute the nullifier, and randomness r which is used to compute the commitment.

In Sapling, the paying key pk is replaced by a diversified address (d, pk_d). This change is reflected in the note format, as a Sapling note contains the two values $g_d = \mathsf{DiversifyHash}^{Sapling}(d)$ and pk_d instead of pk.

The ρ-value is also no longer part of the note. Instead, ρ is computed from the note commitment and its position on the note commitment tree as $\rho = \mathsf{MixingPedersenHash}(cm^{old}, pos)$, using a hash function called $\mathsf{MixingPedersenHash}$. Computing ρ in this fashion makes sure that each transaction contains a unique ρ-value and is essentially a replacement for the previous defense against the Faerie Gold attack.

The values v and r are still part of the note. To align with the Zcash specification, r will be denoted rcm from now on.

Note commitments are computed as $cm = \mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}}(\mathsf{repr}_{\mathbb{J}}(g_d), \mathsf{repr}_{\mathbb{J}}(pk_d), v)$, using a commitment function called $\mathsf{NoteCommit}^{\mathsf{Sapling}}$.

Spend Authority

As was already described, knowledge of both *ask* and *nsk* is required to spend any notes. Knowledge of *nsk* is proven directly in the zero-knowledge proof, in the same manner as knowledge of *sk* is proven in Sprout. However, knowledge of *ask* is demonstrated outside the zero-knowledge proof. Concretely, one has to provide a separate *spend authorization signature* using *ask* as the signing key.

The spend authorization signature is a Schnorr-based signature scheme that allows so-called *re-randomization* of the verifying key $ak = [ask]\ \mathcal{G}^{\text{Sapling}}$. The verifying key *ak* cannot be revealed directly, because this would allow linking payments as it would essentially be equivalent to the signing procedure in Bitcoin. Rather, a re-randomization of the verifying key, $rk = ak + [\alpha]\ \mathcal{G}^{\text{Sapling}}$ is published, and it is proven in zero-knowledge that *rk* is a valid re-randomization of *ak*.

Then, the whole transaction is signed using the re-randomized signing key $rsk = ask + \alpha$. The signature can be verified using only the re-randomized verifying key *rk*, but still convinces the verifying party that the signer is in possession of the original signing key *ask*.

As a bonus, Alice not only proves that she knows the spend authorization key *ask*, but the signature additionally provides integrity protection, rendering the previously used integrity protection mechanism redundant.

As was already mentioned, the main reason to have two separate keys *ask* and *nsk*, and proving knowledge of *ask* outside the zero-knowledge proof is that it allows computationally limited devices to delegate the zero-knowledge proof by only sharing the proof authorizing key (ak, nsk). The external party is still able to re-randomize *ak* in the zero-knowledge proof, but does not gain capability to spend any notes by itself, as it does not know *ask*.

Tying the Address to the Extended Spending Key

In Sprout, the paying key *pk* is derived directly from the spending key as $pk = PRF_{sk}(0)$. Proving knowledge of *sk* therefore immediately proves that an output note containing *pk* belongs to Alice, as she proves that the payment was directed at a paying key that was derived from her spending key.

The new key schedule introduces several layers of indirection between the proof authorizing key and the diversified address. This makes it necessary to prove in several steps that the address was actually derived from the particular values of *ak* and *nsk* that are used in the proof. To do so, the address is essentially re-computed from *ak* and *nsk* in the zero-knowledge proof and

asserted to be equal to the address contained in the input commitment (see "diversified address integrity" in section 3.3.3).

Splitting up the Zero-Knowledge Proof

Sapling introduces a big change in the format of the zero-knowledge proof, by splitting up the large JoinSplit statement that is used in Sprout. Instead, Sapling defines two smaller, more modular statements called *Spend* and *Output*. While this drastically changes the format, it is conceptually straightforward:

The Spend statement contains those parts of the zero-knowledge proof that concern inputs. Concretely, the following things are part of the Spend statement:

- The Merkle path check
- Integrity of the value commitment (see "Value commitments" below)
- Nullifier computation
- Re-randomization of *ak*
- Tying the address to the extended spending key

In contrast to the JoinSplit statement, a single Spend statement only covers a single input. If a transaction has several inputs, several proofs for Spend statements have to be included.

The Output statement, on the other hand, includes all parts that are concerned with outputs. Specifically, the following things are proven:

- Integrity of the output commitment
- Integrity of the output value commitment (see "Value Commitments" below)
- Integrity of the ephemeral public key (see "Key Agreement" below)

Just as the Spend statement, the Output statement also only covers a single output. Several outputs require several separate Spend statement proofs.

Consequences. Most parts of the zero-knowledge proof are relatively unaffected by the split-up. The only part that needs to be changed is the balancing property as the new proof format makes it impossible to prove the balancing property in zero knowledge. This is because the new statements only concern themselves with a single input or output, but are unaware of other inputs or outputs in the transaction. The next paragraph "Value Commitments" discusses how this problem is solved in Sapling.

Value Commitments

Splitting up the zero-knowledge proof renders it impossible to prove the balancing property directly in zero-knowledge. To guarantee balance in a transaction, Sapling uses a separate (homomorphic) commitment scheme $\mathsf{ValueCommit}^{\mathsf{Sapling}}$ and a so-called *Sapling binding signature*.

Concretely, when creating a transaction, Alice does the following. For every input note containing some value v_i^{old}, she samples randomness rcv_i^{old} uniformly at random and publishes the commitment $\mathsf{ValueCommit}^{\mathsf{Sapling}}_{\mathsf{rcv}_\mathsf{i}^\mathsf{old}}(v_i^{old})$. Similarly, for every output note with value v_j^{new}, she generates randomness rcv_j^{new} uniformly at random and publishes the commitment $\mathsf{ValueCommit}^{\mathsf{Sapling}}_{\mathsf{rcv}_\mathsf{j}^\mathsf{new}}(v_j^{new})$. Additionally, for each input and output she proves in the associated Spend/Output-statement that the published value commitment contains the same value as the note commitments.

Then, a new commitment c is computed by adding up all input value commitments and subtracting all output values commitments as

$$\begin{aligned} c &= \sum_i \mathsf{ValueCommit}^{\mathsf{Sapling}}_{\mathsf{rcv}_\mathsf{i}^\mathsf{old}}(v_i^{old}) - \sum_j \mathsf{ValueCommit}^{\mathsf{Sapling}}_{\mathsf{rcv}_\mathsf{j}^\mathsf{new}}(v_j^{new}) \\ &= \mathsf{ValueCommit}^{\mathsf{Sapling}}_{\sum_\mathsf{i} \mathsf{rcv}_\mathsf{i}^\mathsf{old} - \sum_\mathsf{j} \mathsf{rcv}_\mathsf{j}^\mathsf{new}} \left(\sum_i v_i^{old} - \sum_j v_j^{new} \right) \end{aligned}$$

Note that if the transaction balances, c is a commitment to the value 0 as $\mathsf{ValueCommit}^{\mathsf{Sapling}}$ is additively homomorphic. The sapling binding signature allows Alice to prove exactly this, that she is able to open c to the value 0. Therefore, by providing a sapling binding signature, Alice demonstrates that the transaction balances.

Key Agreement

Recall that Alice needs to communicate the values ρ^{new}, r^{new}, and v to Bob, so that he is later able to spend the money. In the design of section 3.2, this is achieved using Bob's public encryption key pk_b^{Enc} and including a ciphertext $C = Enc_{pk_b^{Enc}}(\rho^{new}, r^{new}, v)$ in the transaction.

Instead of directly using public key encryption, Sapling uses a key agreement scheme as follows. When sending funds to an address (d, pk_d), one picks an ephemeral key esk and computes a shared secret

$$[esk]\ pk_d = [esk][ivk]\ g_d = [esk \cdot ivk]\ g_d.$$

From this shared secret, one derives a symmetric key that is then used to do the encryption. The sender also includes their ephemeral public key

$epk = [esk]\ g_d$ in the output, and in addition, proves in zero-knowledge that epk is computed correctly.

The receiver (who knows ivk) is able to compute the shared secret as

$$[ivk]\ epk = [ivk]\ [esk]\ g_d = [esk \cdot ivk]\ g_d.$$

Which allows them to derive the symmetric key and do the decryption.

3.3.2 Instantiations

This section discusses how the primitives which, up to this point, were described in abstract terms, like commitments and pseudorandom functions, are actually instantiated in Zcash. Following a convention in the Zcash specification, a $\star$-symbol is used to mark variables that are bit-representatives of scalars or Jubjub curve points.

Note Commitment

The note commitment function $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ is a commitment function instantiated as a windowed Pedersen commitment. It is defined as

$$\mathsf{NoteCommit}^{\mathsf{Sapling}} : \{0,\ldots,2^{252}-1\} \times \{0,1\}^{256} \times \{0,1\}^{256} \times \{0,\ldots,2^{64}-1\} \to \mathbb{J}$$

$$\mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}}(g\star_d, pk\star_d, v) =$$

$$\quad \mathsf{WindowedPedersenCommitment}_{rcm}(111111||\mathsf{I2LEBSP}_{64}(v)||g\star_d||pk\star_d)$$

Value Commitment

The value commitment function $\mathsf{ValueCommit}^{\mathsf{Sapling}}$ is a commitment function instantiated using the homomorphic Pedersen commitment scheme. It is defined as

$$\mathsf{ValueCommit}^{\mathsf{Sapling}} : \{0,1\}^{252} \times \left\{-\frac{r_{\mathbb{J}}-1}{2},\ldots,\frac{r_{\mathbb{J}}-1}{2}\right\} \to \mathbb{J}$$

$$\mathsf{ValueCommit}^{\mathsf{Sapling}}_{\mathsf{rcv}}(v) = \mathsf{HomomorphicPedersenCommitment}_{rcv}(\text{"}Zcash\_cv\text{"}, v)$$

Pseudorandom function

A pseudorandom function $\mathsf{PRF}^{\mathsf{nfSapling}}$ is used to compute the nullifier from a value ρ. $\mathsf{PRF}^{\mathsf{nfSapling}}$ is instantiated using the BLAKE2s hash function. It is defined as

$$\mathsf{PRF}^{\mathsf{nfSapling}} : \{0,1\}^{256} \times \{0,1\}^{256} \to \{0,1\}^{256}$$

$$\mathsf{PRF}^{\mathsf{nfSapling}}(nk\star, \rho\star) = \mathsf{BLAKE2s}-256(\text{"}Zcash\_nf\text{"}, \mathsf{LEBS2OSP}_{256}(nk\star)||\mathsf{LEBS2OSP}_{256}(\rho\star))$$

Collision Resistant Hash Function

The collision resistant hash function $\mathsf{CRH}^{\mathsf{ivk}}$ is defined as

$$\mathsf{CRH}^{\mathsf{ivk}} : \{0,1\}^{256} \times \{0,1\}^{256} \to \{0, \ldots, 2^{251} - 1\}$$

$$\mathsf{CRH}^{\mathsf{ivk}}(ak\star, nk\star) =$$

$$\mathsf{LEOS2IP}_{256}(\mathsf{BLAKE2s} - 256(\text{"}Zcashivk\text{"}, \mathsf{LEBS2OSP}_{256}(ak\star)||\mathsf{LEBS2OSP}_{256}(nk\star))) \mod 2^{251}$$

3.3.3 Spend Statement

As described in section 3.3.1, the zero-knowledge proof in Sapling is divided into two parts. The first part, called the Spend statement, is associated with spending an old note $(d, pk_d, v^{old}, rcm^{old})$. The Spend statement is defined as follows.

Spend Statement

Given public input

$$\begin{aligned}&(rt^{Sapling} \in \{0,1\}^{255},\\ &\ cv^{old} \in \mathbb{J},\\ &\ nf^{old} \in \{0,1\}^{256},\\ &\ rk \in \mathbb{J}),\end{aligned}$$

the prover knows

$$\begin{aligned}&(path \in \{0,1\}^{255 \cdot 28},\\ &\ pos \in \{0,\ldots,2^{28}-1\},\\ &\ g_d \in \mathbb{J},\\ &\ pk_d \in \mathbb{J},\\ &\ v^{old} \in \{0,\ldots,2^{64}-1\},\\ &\ rcv^{old} \in \{0,\ldots,2^{252}-1\},\\ &\ cm^{old} \in \mathbb{J},\\ &\ rcm^{old} \in \{0,\ldots,2^{252}-1\},\\ &\ \alpha \in \{0,\ldots,2^{252}-1\},\\ &\ ak \in \mathbb{J},\\ &\ nsk \in \{0,\ldots,2^{252}-1\}),\end{aligned}$$

such that the following holds:

Note commitment integrity $cm^{old} = \mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm^{old}}}(\mathsf{repr}_{\mathbb{J}}(g_d), \mathsf{repr}_{\mathbb{J}}(pk_d), v^{old})$.

Merkle path validity Either $v^{old} = 0$ or $(path, pos)$ is a valid Merkle path from $\mathsf{Extract}_{\mathbb{J}^{(r)}}(cm^{old})$ to the root $rt^{Sapling}$.

Value commitment integrity $cv^{old} = \mathsf{ValueCommit}^{\mathsf{Sapling}}_{\mathsf{rcv^{old}}}(v^{old})$.

Small order checks g_d and ak are not of small order, i.e. $[h_{\mathbb{J}}]\ g_d \neq \mathcal{O}_{\mathbb{J}}$ and $[h_{\mathbb{J}}]\ ak \neq \mathcal{O}_{\mathbb{J}}$.

Nullifier integrity $nf^{old} = \mathsf{PRF}^{nfSapling}_{nk\star}(\rho\star)$, where
$nk\star = \mathsf{repr}_{\mathbb{J}}([nsk]\ \mathcal{H}^{\text{Sapling}})$,
$\rho\star = \mathsf{repr}_{\mathbb{J}}(\mathsf{MixingPedersenHash}(cm^{old}, pos))$

Spend authority $rk = ak + [\alpha]\ \mathcal{G}^{\text{Sapling}}$

Diversified address integrity $pk_d = [ivk]\ g_d$, where
$ivk = \mathsf{CRH}^{\mathsf{ivk}}(ak\star, nk\star)$,
$ak\star = \mathsf{repr}_{\mathbb{J}}(ak)$

Discussion

This section discusses the different parts of the statement and their purpose.

Note commitment integrity This part of the proof is essentially the same as in JoinSplit, except that the note commitments have changed in format.

Merkle path validity. This is a slightly more concrete version of what in section 3.2 was simply described as "$cm(..)$ *is a leaf of a Merkle tree with root* rt". In reality, the prover provides a merkle tree co-path *path* and the position of the leaf, *pos*. These two values allow to recompute the root value and compare it with rt inside the zero-knowledge circuit. For more details, see section 4.1.18. Additionally, the Merkle check is only needed if any money is spent, i.e. $v^{old} \neq 0$.

Value commitment integrity Similar to note commitment integrity - It is simply proven that cv^{old} is computed correctly and contains the same value v^{old} as the note commitment.

Small order checks These small order checks are a technical necessity to defend against some attacks. We don't go into details here.

Nullifier integrity This part is similar to the nullifier integrity of the JoinSplit statements, but adapted for the new nullifier computation. In addition, notice that nk is not directly provided as an auxiliary input, but rather computed as $nk = [nsk]\ \mathcal{H}^{\text{Sapling}}$. This is a crucial detail, as it makes sure that the prover knows the value nsk.

Spend authority Here it is proven that rk is a valid re-randomization of the key ak, i.e. was computed by picking a value α and computing $rk = ak + [\alpha]\ \mathcal{G}^{\text{Sapling}}$.

Diversified address integrity The purpose of this statement is to show that pk_d is really a diversified address derived from the keys ak and nk. This is necessary to tie the values together, as otherwise, an attacker could potentially spend a note containing pk_d using a different spending key ask'.

3.3.4 Output Statement

The second type of zero-knowledge statement is called the Output statement. Every proof of an Output statement is associated with the creation of an output note $(d, pk_d, v^{new}, rcm^{new})$. The Output statement is:

Output statement

Given public input

$$\begin{aligned}&(cv^{new} \in \mathbb{J},\\ &\ cm_u \in \{0,1\}^{255},\\ &\ epk \in \mathbb{J}),\end{aligned}$$

the prover knows

$$\begin{aligned}&(g_d \in \mathbb{J},\\ &\ pk\star_d \in \{0,1\}^{256},\\ &\ v^{new} \in \{0,\ldots,2^{64}-1\},\\ &\ rcv^{new} \in \{0,\ldots,2^{252}-1\},\\ &\ rcm^{new} \in \{0,\ldots,2^{252}-1\},\\ &\ esk \in \{0,\ldots,2^{252}-1\}),\end{aligned}$$

such that the following holds:

Note commitment integrity $cm_u = \mathsf{Extract}_{\mathbb{J}^{(r)}}(\mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}^{\mathsf{new}}}(g\star_d, pk\star_d, v^{new}))$, where $g\star_d = \mathsf{repr}_{\mathbb{J}}(g_d)$

Value commitment integrity $cv^{new} = \mathsf{ValueCommit}^{\mathsf{Sapling}}_{\mathsf{rcv}^{\mathsf{new}}}(v^{new})$

Small order check g_d is not of small order i.e. $[h_{\mathbb{J}}]\ g_d \neq O_{\mathbb{J}}$

Ephemeral public key integrity $epk = [esk]\ g_d$

Discussion

Note commitment integrity This point proves that the output commitment is well formed. As a technical detail, note that the output commitment cm_u is given as a bitstring instead of a point on Jubjub, which is why $\mathsf{Extract}_{\mathbb{J}^{(r)}}$ has to be used in the proof.

Value commitment integrity Here, it is proven the the output value commitment is well formed and contains the same value v^{new} as the note commitment.

Small order check It is required to demonstrate that g_d is not a point of small order on the elliptic curve. This helps to prevent some attacks, but we do not go into details here.

Ephemeral public key integrity It is proven that the ephemeral public key, used by the receiver to decrypt the ciphertext, is well formed.

Chapter 4

Spend and Output Encodings

This chapter first shows how to encode the Zcash Spend and Output statements as Diophantine equations, so that they can be proven using TV20. Then, a comparison is drawn between the proof size of the current system (Groth16) and the proof size when using TV20. The last section contains benchmarks on the proof size, obtained with a prototype implementation of the inner product argument of TV20.

4.1 Encodings

Appendix A of the Zcash specification [20] contains a detailed description of how the Spend and Output statements are encoded as Quadratic Constraint Programs or *QCPs*, which is the input format of Groth16. Their presentation is incremental, in that they first provide constraints for simple building blocks, that are then assembled into the more involved statements. We follow this approach exactly. All equations for the building blocks are written as discussed in section 2.7.5, so that the Hadamard product and linear constraints can easily be read off. Additionally, for every building block, we include the cost of tying input variables to previous variables in the Hadamard product, as in the majority of the cases where the buildings blocks are used, this cost needs to be included (see section 2.7.5). Helper variables that are introduced to store intermediate values will generally be denoted z_i.

4.1.1 Boolean Constraint

A variable b is constrained to be a bit, i.e. $b \in \{0, 1\}$ using the equation

$$(b^2 - b) = 0$$

This adds a single entry to the Hadamard product and two linear constraints to guarantee consistency between entries. When used in composition, we

need an additional linear constraint to tie b to previous variables in the Hadamard product. In total, this amounts to one entry in the Hadamard product and three linear constraints.

4.1.2 Conditional Equality

To implement the statement "either $a = 0$ or $b = c$", one can use the equation

$$(ab - z_1)^2 + (ac - z_2)^2 + (z_1 - z_2)^2 = 0$$

This adds two entries to the Hadamard product and two linear constraints: One for consistency between a-values and another one for the last term. When used in composition, three linear constraints are needed to bind the input value a, b, c to previous variables. In total, these are 2 entries in the Hadamard product and 5 linear constraints.

4.1.3 Selection Constraint

Assume b has already been boolean-constrained. Then "$z = (b\ ?\ x : y)$" can be encoded using the equation

$$(by - z_1)^2 + (bx - z_2)^2 + (z_1 - z_2 - y + z)^2 = 0$$

This adds two entries to the Hadamard product, a linear constraint for consistency between b values and another linear constraint for the second term. To compose, three linear constraints are needed to bind b, x, y to previous values. In total, 2 entries to the Hadamard product and 5 linear constraints are needed.

4.1.4 Nonzero Constraint

To prove that a value a is nonzero modulo r, it suffices to provide an inverse, e.g. values a_{inv} and k such that $a_{inv} \cdot a = 1 + kr$. This can be implemented as the equation

$$(a \cdot a_{inv} - z_1)^2 + (1 + kr - z_1)^2 = 0$$

The first term adds an entry to the Hadamard product, the second one is a single linear constraint. To compose, another linear constraint is needed to tie a to a previous output. In total, this amounts to 1 entry in the Hadamard product and 2 linear constraints.

4.1.5 Exclusive-or Constraint

Assume that a, b are boolean-constrained. One can implement "$a \oplus b = c$" with the equation

$$(ab - z_1)^2 + (2z_1 - a - b + c)^2 = 0$$

This adds an entry to the Hadamard product as well as a linear constraint. In addition, 2 linear constraints are needed to tie a, b when composing. In total, 1 entry to the Hadamard product and 3 linear constraints are needed.

4.1.6 Unpacking

Let $n \in \mathbb{N}^+$ be a constant. The operation of converting a field element $a \in \mathbb{F}_r$ to boolean variables $b_0, \ldots, b_{n-1} \in \{0, 1\}$ such that $a = \sum_{i=0}^{n-1} b_i \cdot 2^i \ mod \ r$ is called unpacking. The inverse operation is called packing. Assuming that the boolean variables are constrained separately, this can be implemented with a single linear constraint,

$$\sum_{i=0}^{n-1} b_i \cdot (2^i \ mod \ r) = a + kr.$$

4.1.7 Range Check

Assume we want to constrain $a \leq c$ for some constant $c \in \mathbb{N}$. We can use the fact that for every natural number x, $4x + 1$ can be written as the sum of 3 squares (as suggested by Groth [18]), and prove that $a \leq c$ by providing three numbers such that the sum of their squares is equal to $4(c - a) + 1$. This is expressed by the equation

$$(z_1 - x_1^2)^2 + (z_2 - x_2^2)^2 + (z_3 - x_3^2)^2 + (c - a - z_1 - z_2 - z_3)^2 = 0$$

3 entries in the Hadamard product and three linear constraints for consistency are needed for the first three terms. Additionally, another single linear constraint is needed for the last term. When composing, one more linear constraint is needed to tie a to a previous value. In total, this amounts to 3 Hadamard product entries and 5 linear constraints.

4.1.8 Check that Affine-ctEdwards Coordinates are on the Curve

The ctEdwards Jubjub curve has been introduced in section 2.8.2. To check that a point (u, v) is on the curve, we simply need to check that $a_{\mathbb{J}} \cdot u^2 + v^2 = 1 + d_{\mathbb{J}} \cdot u^2 \cdot v^2 + kr$, for some k. This can be implemented with the equation

$$(u^2 - z_1)^2 + (v^2 - z_2)^2 + (z_1 \cdot z_2 - z_3)^2 + (a_J \cdot z_1 + z_2 - 1 - d_J \cdot z_3 - kr)^2 = 0,$$

The first three terms add 3 entries to the Hadamard product, 2 linear constraints to provide consistency for the u and v values, and 2 linear constraints for consistency between the z_1 and z_2 values. The last term is a single linear constraint. When composed, another 2 linear constraints are needed to bind u, v to previous Hadamard product entries. In total, these are 3 Hadamard product entries and 7 linear constraints.

4.1.9 ctEdwards (de)Compression and Validation

On the ctEdwards Jubjub curve (section 2.8.2), given a compressed point $(\tilde{u}, v)$ and u, check that (u, v) is a valid decompression of $(\tilde{u}, v)$ or vice-versa. To do so, we need to perform three separate checks.

1. Check that (u, v) is on the curve.
2. Unpack u modulo $q_{\mathbb{J}}$, as $u = \sum_{i=0}^{254} u_i \cdot 2^i$. Additionally, boolean-constrain $u_0, \ldots, u_{254}$. Equate $\tilde{u}$ with u_0.
3. Check that $u \leq r_{\mathbb{J}} - 1$.

The first part adds 3 entries to the Hadamard product and 7 linear constraints. The second part adds 255 entries to the Hadamard product and 256 linear constraints. The third part adds 3 entries to the Hadamard product and another 5 linear constraints. In total, this adds 261 entries to the Hadamard product and 268 linear constraints.

4.1.10 Conversion between ctEdwards and Montgomery Forms

Given a point (u, v) on the ctEdwards Jubjub curve, we want to convert it to the point (x, y) on the Montgomery Jubjub curve and vice-versa, using the functions introduced in section 2.8.2. Let s be defined as in section 2.8.2. The conversion can be implemented using the following equation

$$(yu - z_1)^2 + (xv - z_2)^2 + (z_1 - sx)^2 + (z_2 + v - x + 1)^2 = 0,$$

These add a total of 2 entries to the Hadamard product, as well as 2 linear constraints. To tie u, v (or x, y respectively when converting in the other direction) to previous variables when composing, another 2 linear constraints are needed. In total, 2 entries to the Hadamard product and 4 linear constraints are needed.

Note that the equations above only implement the conversion correctly if no exceptional cases occur. The Zcash specification contains a proof that conversions are only used whenever exceptional cases cannot occur (Appendix A, section 3.3.3).

4.1.11 Affine Montgomery Arithmetic

To add two points $(x_1, y_1), (x_2, y_2)$ on a Montgomery curve, we directly implement the formulas from section 2.8.2. Actually, we only implement the case where $x_1 \neq x_2$, as the Montgomery addition will only be used in cases where this can be assumed (there is a proof on this in the Zcash specification, appendix A, section 3.3.4). To add the two points in this case we use

the equation

$$\begin{aligned}&(x_1\lambda - z_1)^2 + (x_2\lambda - z_2)^2 + (x_3\lambda - z_3)^2 + (\lambda^2 - z_4)^2\\ &+(z_2 - z_1 - y_2 - y_1 + k_1 r)^2\\ &+(B_{\mathbb{M}} z_4 - A_{\mathbb{M}} - x_1 - x_2 - x_3 + k_2 r)^2\\ &+(z_1 - z_3 - y_3 - y_1 + k_3 r)^2 = 0\end{aligned}$$

The first four terms add 4 entries to the Hadamard product and 4 linear constraints to guarantee consistency between λ values. The last three terms add three 3 constraints. When composing, we need another 4 linear constraints to bind the input values to previous variables. In total, 4 entries to the Hadamard product as well as 11 linear constraints are needed.

4.1.12 Affine ctEdwards Arithmetic

The formulas for adding two points (u_1, v_1), (u_2, v_2) on a ctEdwards curve that were introduced in section 2.8.2 can be turned into Diophantine equations directly as

$$\begin{aligned}(1 + d_J u_1 u_2 v_1 v_2) \cdot u_3 &= u_1 v_2 + v_1 u_2 + k_1 r\\ (1 - d_J u_1 u_2 v_1 v_2) \cdot v_3 &= v_1 v_2 - a_J u_1 u_2 + k_2 r\end{aligned}$$

To read off a Hadamard product and linear constraints, the above is transformed into

$$\begin{aligned}&(u_1 v_2 - z_1)^2 + (v_1 u_2 - z_2)^2 + (v_1 v_2 - z_3)^2 + (u_1 u_2 - z_4)^2\\ &+(z_1 z_2 - z_5)^2 + (z_5 u_3 - z_6)^2 + (z_5 v_3 - z_7)^2\\ &+(u_3 + d_J z_6 - z_1 - z_2 + k_1 r)^2\\ &+(v_3 - d_J z_7 - z_3 + a_j z_4 + k_2 r)^2 = 0\end{aligned}$$

The first seven terms add 7 entries to the Hadamard product and need 4 linear constraints to check consistency for the values u_1, u_2, v_1, v_2. The last two terms add a linear constrain each. To bind the input values to previous variables, another 4 linear constraints are needed. In total, these are 7 Hadamard product entries and 10 linear constraints.

4.1.13 Affine ctEdwards nonsmall-Order Check

The Jubjub curve has a cofactor of $h_{\mathbb{J}} = 8$ and we want to be able to check whether a point is of order $h_{\mathbb{J}}$ or less. To do so, one can double the point three times and check if the u-coordinate is 0 (recall that the neutral element

is $\mathcal{O}_{\mathbb{J}} = (0, 1)$). Doubling three times adds 21 entries to the Hadamard product and 30 linear constraints (section 4.1.12). A nonzero check needs a single Hadamard product entry and two linear constraints (section 4.1.4). In total, 22 Hadamard product entries and 32 linear constraints are needed for a nonsmall-order check.

4.1.14 Fixed-base Affine ctEdwards Scalar Multiplication

We want to compute $[k]B$ for a fixed point B on the ctEdwards Jubjub curve and a scalar k. For now, we assume the scalar k to be as large as the group order, which is 251 bits in the case of Jubjub (if it is larger, we can add one additional linear constraint to reduce it modulo the group order, but it could be smaller). Also assume that the scalar has been boolean-constrained separately.

To do the operation, we will use pre-computed 3-bit window tables. The technique is based on the following idea. The scalar can be expressed in base 8 as $k = \sum_{i=0}^{83} k_i \cdot 8^i$ (one could choose any other base and proceed similarly, but base 8 results in the most efficient encoding). Therefore we can write

$$[k]B = \sum_{i=0}^{83} [k_i \cdot 8^i] \cdot B = \sum_{i=0}^{83} w_{(B,i,k_i)}$$

where $w_{(B,i,k_i)} := [k_i \cdot 8^i] \cdot B$. The values $w_{(B,i,s)}$ are pre-computed for $i \in \{0, \ldots, 83\}$ and $s \in \{0, \ldots, 7\}$. Then, for each 3-bit window, we will use an equation that 'selects' the correct pre-computed value depending on the bits of that window. Finally, we add up all the selected values to get the results.

Window selection. To see how the selection works, let us fix a value i and look only at the i-th window, k_i. Denote the three bits of this window as $k_i = [s_0, s_1, s_2]$. We want to select the value $w_{(B,i,s)}$ such that $s = 4s_2 + 2s_1 + s_0$. Recall that a value $w_{(B,i,s)} =: (u_s, v_s)$ is a point on an elliptic curve, and therefore a pair of coordinates. We use the following equations to select the correct value. The first equation defines "AND"-style variables for subsets of the three bits. The remaining two equations select the u_s and v_s values according to the inclusion-exclusion principle using the previously computed variables. Note that these last two equations are linear.

$$(s_{01} - s_0 s_1)^2 + (s_{02} - s_0 s_2)^2 + (s_{12} - s_1 s_2)^2 + (s_{123} - s_{01} s_2)^2 = 0$$

$$
\begin{aligned}
u_s = u_0 &+ (u_1 - u_0) \cdot s_0 + (u_2 - u_0) \cdot s_1 + (u_4 - u_0) \cdot s_2 \\
&+ (u_3 + u_0 - u_1 - u_2) \cdot s_{01} \\
&+ (u_5 + u_0 - u_1 - u_4) \cdot s_{02} \\
&+ (u_6 + u_0 - u_2 - u_4) \cdot s_{12} \\
&+ (u_7 - u_6 - u_5 - u_3 + u_1 + u_2 + u_4 - u_0) \cdot s_{123}
\end{aligned}
$$

$$
\begin{aligned}
v_s = v_0 &+ (v_1 - v_0) \cdot s_0 + (v_2 - v_0) \cdot s_1 + (v_4 - v_0) \cdot s_2 \\
&+ (v_3 + v_0 - v_1 - v_2) \cdot s_{01} \\
&+ (v_5 + v_0 - v_1 - v_4) \cdot s_{02} \\
&+ (v_6 + v_0 - v_2 - v_4) \cdot s_{12} \\
&+ (v_7 - v_6 - v_5 - v_3 + v_1 + v_2 + v_4 - v_0) \cdot s_{123}
\end{aligned}
$$

The first equation adds four entries to the Hadamard product and 5 linear constraints to guarantee consistency between entries. The other equations are two linear constraints.

The selection has to be done for all of the 84 windows. Thereafter, we need to add the selected values using 83 ct-Edward additions, each of which takes 7 entries in the Hadamard product and 10 linear constraints (section 4.1.12). When composed, another 255 linear constrains are needed to bind the input bits to previous variables. In total, this amounts to $84 \cdot 4 + 83 \cdot 7 = 917$ entries in the Hadamard product as well as $84 \cdot 7 + 84 \cdot 10 + 255 = 1683$ linear constraints.

In case the scalar does not have the full length of 252 bits, the number of windows decreases. The scalar multiplication is also used with scalars of 64 and 32 bits. In the case of 64 bits, we have $\lceil 64/3 \rceil = 22$ windows. This results in a Hadamard product of length $22 \cdot 4 + 21 \cdot 7 = 235$ and $22 \cdot 7 + 21 \cdot 10 + 64 = 428$ linear constraints. In the case of 32 bits, we have $\lceil 32/3 \rceil = 11$ windows. There, a Hadamard product length of $11 \cdot 4 + 10 \cdot 7 = 114$ and $11 \cdot 7 + 10 \cdot 10 + 32 = 209$ linear constraints suffice.

4.1.15 Variable-base Affine-ctEdwards Scalar Multiplication

If we want to do a scalar multiplication $R = [k] \cdot B$, where the base point B on the ctEdwards Jubjub curve is not fixed (i.e. it is only computed during the zero-knowledge proof and depends on the witness), we cannot use the method from 4.1.14, as pre-computation is impossible. Instead, we use a double-and-add approach. Assume that k is a full length scalar (251 bits) and has already been boolean-constrained into bits as in $k = \sum_{i=0}^{250} k_i \cdot 2^i$.

For $i \in \{0, \ldots, 250\}$, we denote by b_i the current base (i.e. $b_i = [2^i] \cdot B$) and by a_i the current accumulator (i.e $a_i = \sum_{j=0}^{i} [k_j \cdot 2^j] \cdot B$). We use superscripts to access individual coordinates, for example a_i^u denotes the u-coordinate of

the accumulator a_i. We will make use of the building block for the selection constraint, described in section 4.1.3. First, do an initialization step using

$$b_0 = B, \quad a_0^u = k_0 \; ? \; b_0^u : 0, \quad a_0^v = k_0 \; ? \; b_0^v : 1$$

In this first step, a_0 is set to $\mathcal{O}_{\mathbb{J}} = (0, 1)$ if the lowest bit of k is 0 and to B otherwise. Now, for $i \in \{1, \ldots, 250\}$, we perform the double-and-add using the following equations

$$b_i = [2] \cdot b_{i-1}, \quad u_i = k_i \; ? \; b_i^u : 0, \quad v_i = k_i \; ? \; b_i^v : 1, \quad a_i = a_{i-1} + (u_i, v_i)$$

Finally, we let $R = a_{250}$. Each of the 502 selections (section 4.1.3) adds two entries to the Hadamard product as well as five linear constraints. The 250 doublings and the 250 additions each add 7 entries to the Hadamard product and 10 linear constraints (section 4.1.12). In total, these are 4504 Hadamard product entries as well as 7510 linear constraints.

4.1.16 Pedersen Hash

The Pedersen hash is defined in section 2.8.3. Assume that the input bits are already boolean-constrained separately. Also, assume that the label D, and therefore the points $G_i := \mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(D, i)$ are fixed too. A straightforward approach would be to first evaluate all values $\langle M_i \rangle$, then do several scalar multiplications and finally add up all the resulting points. However, this is not the best way to do it, if we want to minimize the size of the Hadamard product. In fact, Zcash has several optimizations in place to reduce the number of constraints of their QCP. We take the same approach but adapt it to our setting.

If we expand the definition, we see that

$$\begin{aligned}
\mathsf{Extract}_{\mathbb{J}^{(r)}}\left(\sum_{i=1}^{n}[\langle M_i \rangle] \cdot G_i\right) &= \mathsf{Extract}_{\mathbb{J}^{(r)}}\left(\sum_{i=1}^{n}\left[\sum_{j=1}^{k_i} enc(m_j) \cdot 2^{4(j-1)}\right] \cdot G_i\right) \\
&= \mathsf{Extract}_{\mathbb{J}^{(r)}}\left(\sum_{i=1}^{n}\sum_{j=1}^{k_i}\left[enc(m_j) \cdot 2^{4(j-1)}\right] \cdot G_i\right).
\end{aligned}$$

Intuition. Now, on a high level, the approach is as follows. To evaluate the inner sums, we use window tables, similarly to section 4.1.14. Concretely, for each chunk, we will have an equation to select the correct value for that chunk from a set of pre-computed values. To optimize further, the base points G_i (respectively the pre-computed values) are actually points on

the Montgomery Jubjub curve, instead of the ctEdwards Jubjub curve. After having done the selection for individual chunks, we add up the selected values to evaluate the inner sum using Montgomery additions. Using Montgomery additions here is advantageous over ctEdwards additions, as it adds fewer entries to the Hadamard product. However, Montgomery addition is incomplete, and can only be used if we are certain that the points that we are adding satisfy the respective side condition. This is proven to be true for the terms in the inner sums in the Zcash specification ([20], Appendix A, section 3.3.9). Unfortunately, for the terms of the outer sum, the side conditions do not necessarily hold. This implies that before evaluating the outer sum, we first have to convert all the points back to ctEdwards form. Then, these points are added to each other using ctEdwards additions and the u-coordinate is extracted to get the final hash output.

Details. Let us look concretely at how the inner sums are evaluated, as this operation is the main contributor in terms of complexity and size. First, we focus on the selection that is performed for a single summand. Notice that the function *enc* has the range $\{-4,\dots,4\} \setminus \{0\}$ and that for a single chunk of 3 bits, the lower two bits determine the absolute value, while the uppermost bit negates the result of *enc* whenever it is set to 1. Note also that as we are operating on the Montgomery curve, we invert a point by flipping the second coordinate.

The above facts yield to following strategy for computing a term of the inner sum. First, disregard the sign of the scalar and simply select the correct absolute value in a manner similar to the fixed-base scalar multiplication (section 4.1.14). Then, conditionally flip the y-coordinate of the selected point if the uppermost bit of the chunk is set to 1.

As a segment has a maximum length of 189 bits, the maximum number of chunks in a segment is $189/3 = 63$. Therefore, to implement the above idea, we need to pre-compute the values $\left[s \cdot 2^{4(j-1)}\right] \cdot G_i$, for all $s \in \{1,\dots,4\}$, $j \in \{0,\dots,63\}$, $i \in \{1,\dots,n\}$. Now, to explain how the selection is performed, fix j and i and let $[s_0, s_1, s_2]$ be the bits of the j-th chunk in segment i. Denote $(x_k, y_k) = \left[k \cdot 2^{4(j-1)}\right] \cdot G_i$, for $k \in \{1,2,3,4\}$. First, we introduce a new variable s_{01} using a single Hadamard product and two linear constraints to tie s_0 and s_1 to the input bits.

$$(s_0 s_1 - s_{01})^2 = 0$$

Next, we use two linear constraints to select the two coordinates x_s, y_r from pre-computed values.

$$x_s = x_1 + (x_2 - x_1) \cdot s_0 + (x_3 - x_1) \cdot s_1 + (x_4 + x_1 - x_2 - x_3) \cdot s_{01}$$

$$y_r = y_1 + (y_2 - y_1) \cdot s_0 + (y_3 - y_1) \cdot s_1 + (y_4 + y_1 - y_2 - y_3) \cdot s_{01}$$

As a third step, compute the real y-coordinate y_s by conditionally negating y_r.

$$y_r = s_2?(-y_r) : y_r$$

Which takes a single Hadamard product and five single linear constraint (see section 4.1.3). In total, for each chunk, we add 2 entries to the Hadamard product and 9 linear constraints to compute the elements of the inner sum. After having selected the summands, we add them all up using Montgomery additions (section 4.1.11). For a segment M_i with q_i chunks, we need to do $q_i - 1$ additions, each of which takes 4 entries in the Hadamard product and 11 linear constraints.

To finish up the computation after the inner sums have been evaluated, for each segment, we need to convert the obtained point to a ctEdwards point (section 4.1.10). If the number of segments is n, we do n such conversions, and each conversion takes 2 entries in the Hadamard product and 4 linear constraints. Finally, we need to add these points using ctEdwards additions. Here, $n - 1$ additions are done, each addition takes 7 Hadamard product entries and 10 linear constraints.

To summarize the cost, let $n = \lceil \frac{N}{189} \rceil$ be the number of segments, and $q = \lceil \frac{N}{3} \rceil$ be the total number of chunks. The cost of the selections and the Montgomery additions can be aggregated over all segments. Therefore, $2q + 4(q - n) + 2n + 7(n - 1) = 6q + 5n - 7$ entries in the Hadamard product, as well as $9q + 11(q - n) + 4n + 10(n - 1) = 20q + 3n - 10$ linear constraints are needed.

4.1.17 Mixing Pedersen Hash

The function MixingPedersenHash is defined in section 2.8.4. This function is only used in one place in the Zcash circuit. There, the second input x, which is used to do a fixed-base scalar multiplication, has a size of 32 bits. Therefore, the scalar multiplication requires 114 entries to the Hadamard product and 209 linear constraints (section 4.1.14). The ctEdwards addition needs another 7 entries to the Hadamard product as well as 10 linear constraints (section 4.1.12). In total, MixingPedersenHash adds 121 entries to the Hadamard product and 253 linear constraints.

4.1.18 Merkle Path Check

The note commitment tree is instantiated as a binary Merkle tree with 28 layers and the Pedersen hash function as the underlying hash. Internal nodes store values of the set $\{0,1\}^{255}$. Leaves do not store note commitments directly, instead, $\mathsf{Extract}_{\mathbb{J}^{(r)}}$ is used to convert note commitments (which are Jubjub curve points) to bitstrings that are then stored in the leaves.

Remember that we can use the co-path to check whether an element is part of a Merkle tree, by re-computing the root value (see section 2.3). In addition, to do the re-computation, one must know the location of the leaf containing the value. This information is needed to compute the correct hashes at each layer, as one needs to know whether the value in the co-path is the left or the right input to the hash function. Thus, the prover provides two secret inputs:

1. The co-path, containing only the hash values
2. The position of the leaf

It is assumed here that the position of the leaf is given as a bitstring of length 28 (the number of layers in the note tree), where every bit indicates whether the co-path contains the left or the right child of the node. If you recompute the root value using the two items, it must be equal to the public root value. To encode this computation as a Diophantine equation, four computational steps are needed on each layer of the tree.

- Boolean constrain the path-bit
- Conditionally swap the accumulated hash and the hash contained in the co-path, depending on the path-bit
- Unpack the inputs into 255 bits each and boolean-constrain the bits
- Compute the Pedersen Hash

Boolean constraining the path bit can be done by using a single Hadamard product and three linear constraints (section 4.1.1).

The conditional swapping can be implemented with 2 selection constraints (section 4.1.3), taking 4 Hadamard products as well as 10 linear constraints in total.

The unpacking and boolean-constraining takes $2 \cdot 255 = 510$ Hadamard product entries as well as $2 \cdot 2 \cdot 255 + 2 = 1022$ linear constraints (section 4.1.6 and 4.1.1).

The input length of the hash is 512 bits (two 256 bit inputs which are the outputs of the previous layer). For a 512 bit input, the number of segments is 3 and the number of chunks is 172. Therefore, the hash evaluation needs 1040 Hadamard product entries and 3439 linear constraints. In total, this means that for each layer of the Merkle tree, 1555 Hadamard product entries as well as 4474 linear constraints are needed.

As the note commitment tree has 28 layers, the total cost to prove that a note commitment is contained in a leaf are 43540 Hadamard product entries and 125272 linear constraints.

4.1.19 Windowed Pedersen Commitment

The function WindowedPedersenCommitment is defined in section 2.8.5. The windowed Pedersen commitment scheme is used only when computing note commits. There, s has a size of 582 bits. This means that we need 1177 Hadamard product entries and 3882 linear constraints for PedersenHashToPoint (see section 4.1.16). For the scalar multiplication, we need 917 entries in the Hadamard product as well as 1683 linear constraints (section 4.1.14). Finally, to add the two points, we need another 7 entries in the Hadamard product and 10 linear constraints (section 4.1.12). In total, these are 2101 Hadamard product entries as well as 5575 linear constraints.

4.1.20 Homomorphic Pedersen Commitment

The function HomomorphicPedersenCommitment is defined in section 2.8.6. Note that s is constrained to be in $\{1, \ldots, 2^{64} - 1\}$, and can therefore be represented by 64 bits. Hence, the scalar multiplication by s needs only 235 Hadamard product entries and 428 linear constraints (section 4.1.14). The scalar multiplication by (the full length scalar) r needs 917 Hadamard product entries as well as 1683 linear constraints (section 4.1.14). The final ctEdwards addition needs 7 Hadamard product entries and 10 linear constraints (section 4.1.12). In total, these are 1159 Hadamard product entries and 2121 linear constraints.

4.1.21 BLAKE2s Hash Function

The BLAKE2s hash function was proposed in 2012 by Aumasson et al. [3]. We will not look at the full definition of the functions, instead focusing on the parts that are relevant for us. The full definition can either be found in the original paper or the Zcash specification. At its core, BLAKE2s uses a function G that is defined as follows.

$$
\begin{aligned}
&G : \{0, \ldots, 2^{32} - 1\}^6 \mapsto \{0, \ldots, 2^{32} - 1\}^4 \\
&G(a, b, c, d, x, y) = (a'', b'', c'', d''), \text{ where} \\
&\quad a' = (a + b + x) \ mod \ 2^{32} \\
&\quad d' = (d \oplus a') \gg 16 \\
&\quad c' = (c + d') \ mod \ 2^{32} \\
&\quad b' = (b \oplus c') \gg 12 \\
&\quad a'' = (a' + b' + y) \ mod \ 2^{32} \\
&\quad d'' = (d' + a'') \gg 8 \\
&\quad c'' = (c' + d'') \ mod \ 2^{32} \\
&\quad b'' = (b' \oplus c'') \gg 7
\end{aligned}
$$

The function G is used to shuffle parts of the input (x, y), into the state (a, b, c, d). Next, it is explained how to encode G as a Diophantine equation.

As several xor operations need to be done during the function, one cannot work with integer values but instead has to operate directly on the bits. The xor operations followed by right-shifts can be done as described in section 4.1.5 and then simply using only those bits that where not 'discarded' by the shift operation. For 32-bit values (as in the G function), this needs 32 Hadamard product entries and 96 linear constraints per xor operation.

An addition of two 32-bit values, say $c = (a + b) \ mod \ 2^{32}$, is performed as follows. As we are operating on bits at all times, the binary representation of $a = \sum_{i=0}^{31} a_i \cdot 2^i$ and $b = \sum_{i=0}^{31} b_i \cdot 2^i$ is available. To do the addition, 33 new variables $c_0, \ldots, c_{32}$ are introduced and boolean-constrained. An equality check is used to make sure that they correspond to the binary representation of the output of the addition:

$$\sum_{i=0}^{31} (a_i + b_i) \cdot 2^i = \sum_{i=0}^{32} c_i \cdot 2^i$$

As the operation is done modulo 2^{32}, the bit c_{32} is then discarded (i.e. simply not used in the subsequent computation). With this approach, 33 Hadamard product entries and 67 linear constraints are needed.

An addition with three 32-bit values, say $c = (a + b + m) \ mod \ 2^{32}$, can be implemented by following exactly the same approach, but we need to use 34 bits (instead of 33) $c_0, \ldots, c_{33}$, to account for a "double overflow". In this case, 34 Hadamard product entries and 69 linear constraints are needed.

Therefore, for a single evaluation of G, $4 \cdot 32 + 2 \cdot 33 + 2 \cdot 34 = 262$ entries in the Hadamard product and $4 \cdot 96 + 2 \cdot 67 + 2 \cdot 69 = 656$ linear constraints are needed.

A BLAKE2s evaluation is subdivided into ten rounds in total, in each round G is evaluated eight times. Therefore, G will be evaluated a total of $10 \cdot 8 = 80$ times. This means that in total, $80 \cdot 262 = 20960$ entries in the Hadamard product and $80 \cdot 656 = 52480$ linear constraints are needed for G evaluations.

In addition to the G-evaluations, at the very end of BLAKE2s, one also needs to evaluate eight expressions of the form $(a \oplus x \oplus y)$, where x, y are 32-bit variables and a is a 32-bit constant. As a is constant, each expression can be encoded using 32 entries in the Hadamard product and 96 linear constraints. For the whole BLAKE2s hash function, $20960 + 8 \cdot 32 = 21216$ entries in the Hadamard product as well as $52480 + 8 \cdot 96 = 53248$ linear constraints are needed.

4.2 Spend and Output Encoding Costs

This section computes the full cost of encoding the Zcash Spend and Output statements as Diophantine equation. Computing the cost of the statements is essentially done by adding all the costs of individual operations. These include evaluating functions like $\mathsf{NoteCommit}^{\mathsf{Sapling}}$, $\mathsf{ValueCommit}^{\mathsf{Sapling}}$, etc. and doing various elliptic curve operations such as scalar multiplications. Additionally, some operations are needed to convert between different representation of values, such as going from an integer to its binary representation and furthermore, some inputs need to be constrained to lie in the correct range. The Zcash Spend and Output statements are defined in section 3.3.3 and 3.3.4. The encoding costs are summarized in tables 4.1 and 4.2.

4.2.1 Discussion

Unpacking. There are several operations that unpack values into their binary representation. For example, the scalar to re-randomize the public key, α is unpacked into its binary representation as $\alpha\star$, or the integer value v^{old} is unpacked into its binary representation as $v^{old}_{\star}$. Each unpacking operation includes the cost of boolean-constraining the bits, as subsequent operations (such as the scalar multiplication) rely on their inputs to be boolean-constrained already.

Computing representatives. Computing $\mathsf{repr}_{\mathbb{J}}(ak)$, $\mathsf{repr}_{\mathbb{J}}(nk)$ and $\mathsf{repr}_{\mathbb{J}}(g_d)$ essentially amounts to doing a compression operation (section 4.1.9), as one has to check that the points lie on the curve as well as boolean-constrain the output bits.

Merkle path validity. To check the Merkle path validity, a value $pos\star = \mathsf{I2LEBSP}_{28}(pos)$ is computed. This simply converts the integer position pos to a string of bits as required by 4.1.18. Note that the bits do not need to be boolean-constrained when computing $pos\star$, as the cost for this is already included in the Merkle tree check.

Operation	Statement Part	HPE	LC	Ref.
g_d is on the curve	$g_d \in \mathbb{J}$	3	7	4.1.8
Unpack v^{old} into $v^{old}_{\star} \in \{0,1\}^{64}$	$v^{old} \in \{0,\ldots,2^{64}-1\}$	64	193	4.1.1, 4.1.6
Unpack rcv into $rcv\star \in \{0,1\}^{252}$	$rcv \in \{0,\ldots,2^{252}-1\}$	252	757	4.1.1, 4.1.6
Unpack rcm into $rcm\star \in \{0,1\}^{252}$	$rcm \in \{0,\ldots,2^{252}-1\}$	252	757	4.1.1, 4.1.6
Unpack α into $\alpha\star \in \{0,1\}^{252}$	$\alpha \in \{0,\ldots,2^{252}-1\}$	252	757	4.1.1, 4.1.6
ak is on the curve	$ak \in \mathbb{J}$	3	7	4.1.8
Unpack nsk into $nsk\star \in \{0,1\}^{252}$	$nk \in \{0,\ldots,2^{252}-1\}$	252	757	4.1.1, 4.1.6
$cm = \mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}}(g_d, pk_d, v^{old})$	Note commitment integrity	2101	5575	2.8.5
$cm_u = \mathsf{Extract}_{\mathbb{J}^{(r)}}(cm)$	Merkle path validity	0	0	
rt' is the root of a Merkle tree with leaf cm_u with co-path $path$ and path bits $pos\star$	Merkle path validity	43540	125272	4.1.18
$pos\star = \mathsf{I2LEBSP}_{28}(pos)$	Markle path validity	0	1	4.1.6
If $v^{old} \neq 0$ then $rt' = rt$	Merkle path validity	2	5	4.1.2
$cv = \mathsf{ValueCommit}^{\mathsf{Sapling}}_{\mathsf{rcv}}(v^{old})$	Value commitment integrity	1159	2121	2.8.6
g_d is not of small order	Small order checks	22	32	4.1.8
ak is not of small order	Small order check	22	32	4.1.8
$nk = [nsk\star]\ \mathcal{H}^{\mathrm{Sapling}}$	Nullifier integrity	917	1683	4.1.14
$nk\star = \mathsf{repr}_{\mathbb{J}}(nk)$	Nullifier integrity	261	268	4.1.9
$\rho = \mathsf{MixingPedersenHash}(cm^{old}, pos)$	Nullifier integrity	121	253	4.1.17
$\rho\star = \mathsf{repr}_{\mathbb{J}}(\rho)$	Nullifier integrity	262	268	4.1.9
$nf^{old} = \mathsf{PRF}^{nfSapling}_{nk\star}(\rho\star)$	Nullifier integrity	21216	53248	4.1.21
$\alpha' = [\alpha\star]\ \mathcal{G}^{\mathrm{Sapling}}$	Spend authority	917	1683	4.1.14
$rk = \alpha' + ak$	Spend authority	7	10	4.1.12
$ak\star = \mathsf{repr}_{\mathbb{J}}(ak)$	Diversified address integrity	261	268	4.1.9
$ivk\star = \mathsf{I2LEBSP}_{251}(\mathsf{CRH}^{\mathsf{ivk}}(ak, nk))$	Diversified address integrity	21216	53248	4.1.21
$pk_d = [ivk\star]g_d$	Diversified address integrity	4504	7510	4.1.15
Total		97607	254714	

Table 4.1: Cost of encoding the Spend statement as a Diophantine equation. The Operation columns describes the actual operation that is performed. The statement part column specifies which part of the Spend statement is implemented. HPE is the number of Hadamard product entries and LC is the number of linear constraints.

Operation	Statement Part	HPE	LC	Ref.
Boolean constrain $pk\star_d$	$pk\star_d \in \{0,1\}^{256}$	256	768	4.1.1
Unpack v^{new} into $v^{new}_{\star} \in \{0,1\}^{64}$	$v^{new} \in \{0,\ldots,2^{64}-1\}$	64	193	4.1.1, 4.1.6
Unpack rcv into $rcv\star \in \{0,1\}^{252}$	$rcv \in \{0,\ldots,2^{252}-1\}$	252	757	4.1.1, 4.1.6
Unpack rcm into $rcm\star \in \{0,1\}^{256}$	$rcm \in \{0,\ldots,2^{252}-1\}$	252	757	4.1.1, 4.1.6
Unpack esk into $esk\star$	$esk \in \{0,\ldots,2^{252}-1\}$	252	757	4.1.1, 4.1.6
$g\star_d = \mathsf{repr}_{\mathbb{J}}(g_d)$	Note commitment integrity	261	268	4.1.9
$cm = \mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}}(g_d, pk_d, v^{old})$	Note commitment integrity	2101	5575	2.8.5
$cv = \mathsf{ValueCommit}^{\mathsf{Sapling}}_{\mathsf{rcv}}(v^{old})$	Value commitment integrity	1159	2121	2.8.6
g_d is not of small order	Small order checks	22	32	4.1.8
$epk = [esk\star]\ g_d$	Ephemeral public key integrity	4504	7510	4.1.15
Total		9123	18738	

Table 4.2: Cost of encoding the Output statement as a Diophantine equation. The Operation columns describes the actual operation that is performed. The statement part column specifies which part of the Output statement is implemented. HPE is the number of Hadamard product entries and LC is the number of linear constraints.

4.2.2 Proof Sizes

This section estimates the proof size of TV20 when proving the Spend and Output statements. Section 2.7 already discusses how the communication complexity of the Diophantine equation argument comes together. The length n of the Hadamard product is the determining factor, as the resulting asymptotic prover communication complexity (in bits) is $\mathcal{O}(\ell + \log(n)b_{\mathbb{G}})$. Note that in case of using the Fiat-Shamir Heuristic to make the protocol non-interactive, the proof size will be equal to the prover communication complexity.

The prover-communication complexity originates mainly in two calls to sub-protocols: A call to the inner product argument and a called to a so-called base-switching argument. The cost of the two calls are estimated below. Instead of relying on the asymptotic communication complexity, the following computation considers the exact amount of communication required. We set $\lambda = 128$, $b_{\mathbb{G}} = 3072$ and $P = \lambda^{\log(\lambda)}$.

Inner product argument. To compute the proof size of the inner product argument, recall from section 2.7.2 that in the inner product argument with vectors of length n and witness entries bound by 2^{ℓ}, the prover sends over $2n' + 2$ RSA group elements, two integers with absolute value less than $2^{\ell}P^{n'}$ and an integer with absolute value less than $\left(2n'2^{b_G+\lambda}P^{n'+3} + 2^{\ell}(P-1)^{n'+2}\right) \cdot \left(1 + 2^{\lambda}\right)$, where $n' = \lceil \log n \rceil$. Therefore, the total amount of communication in bits is

$$
\begin{aligned}
&(2n'+2)\cdot b_{\mathbb{G}} + 2\log(2^{\ell}P^{n'}) + \log\left(\left(2n'2^{b_G+\lambda}P^{n'+3} + 2^{\ell}(P-1)^{n'+2}\right)\cdot\left(1+2^{\lambda}\right)\right)\\
\leq&(2n'+2)\cdot b_{\mathbb{G}} + 2\ell + 2n'\cdot\log(P) + 1 + \lambda + \log(P^{n'+3}(2n' + 2^{b_{\mathbb{G}}+\lambda} + 2^{\ell}))\\
\leq&(2n'+2)\cdot b_{\mathbb{G}} + 2\ell + 2n'\cdot\log(P) + 1 + \lambda + (n'+3)\cdot\log(P) + 1 + \log(n') + b_{\mathbb{G}} + \lambda + \ell\\
=&(2n'+3)\cdot b_{\mathbb{G}} + 3\ell + 2n'\cdot\log(P) + 2\lambda + (n'+3)\cdot\log(P) + \log(n')
\end{aligned}
$$

When being called as a subroutine in the Hadamard product argument, n is equal to the length of the Hadamard product. Also, Towa and Vergnaud show that the maximum bit-length of a witness is $\ell = b_{\mathbb{G}} + \lambda + 4$. Plugging in the lengths of the Hadamard products for the Spend and Output statement results in a proof size for the inner-product proof of about 15.3 kilobytes for the Spend statement, and 12.9 kilobytes for the Output statement.

Base-switching argument As the base switching argument was not discussed, we do not go into detail of how its proof size comes together. The main part is due to a call to a sub-protocol that works in a similarly recursive fashion as the inner-product argument. All in all, when being called from the Hadamard product argument, the proof size of the base-switching

argument is at most

$$\log(P) + \ell' + \lambda + (2\log(n) + 2) \cdot b_{\mathbb{G}},$$

where

$$\ell' = \max\Big[\log(\max(\ell, 2 + \log(P) + b_{\mathbb{G}} + \lambda + 3) + n' \log(P)),$$
$$2\lambda + 2 + (n' + 3)\log(P) + \log(n') + b_{\mathbb{G}} + \max(\ell, 2 + \log(P) + b_{\mathbb{G}} + \lambda + 3)\Big]$$

This results in a size of about 14.4 kilobytes for the Spend and 11.8 kilobytes for the Output statement. Note that the two sub-protocols are of comparable size.

Final proof size. The final proof size is equal to the sum of the proof sizes for the two sub-protocols as well as two additional group elements that are sent during the proof. This results in a maximum size of 30.5 kilobytes for the Spend statement and 25.5 kilobytes for the Output statement.

4.3 Comparison and Discussion

Section 4.2.2 shows that the expected proof size for TV20 is about 30.5 kilobytes for the Spend statement and 25.5 kilobytes for the Output statements. The current proof size (using Groth16 at the same 128-bit security level) for any one of the two statements is 192 bytes. The size therefore increases by a factor of 133 to 159.

It is clear that the difference in size is substantial, and this is also expected. In addition to the logarithmic growth (in contrast to Groth16's constant size proofs), another factor increasing the proof size of TV20 is the reliance on hidden-order groups with much larger bit-size of elements (3072 bits) for the same security level compared to the elliptic curve groups underlying Groth16 (384 bits).

Both the Spend and Output statements can be considered rather large. However, note that the (about) tenfold difference in the lengths of the Hadamard products only translate to a growth factor of about 1.2 in the proof sizes, which is due to the logarithmic growth. This also means that even for a much smaller statement, the proof size of TV20 exceeds the proof size of Groth16. For example, setting $n = 50$ (and λ, $b_{\mathbb{G}}$, P as above) yields a proof size of about 12.7 kilobytes.

Recall once more that the benefit of using TV20 is the fact its security relies only on standard assumptions (section 2.5), while the security of Groth16 is proven only in the generic group model. So in spite of the large difference

in proof size, there is still a trade-off to be considered between efficiency and strength of assumptions. The question whether the weaker security assumptions compensate for this substantial increase in proof size is difficult to answer in general.

4.4 Benchmarks

This section presents proof sizes obtained via a prototype implementation of the inner product proof of TV20. The implementation is written in Rust and can be found on Github [21]. The evaluated security levels are 80, 112 and 128 bits, which require RSA groups of 1024, 2048 and 3072 bits. For each security level, the proof size for different vector lengths is evaluated. For a given security level λ and $b_{\mathbb{G}}$ bit RSA group, the bit-length ℓ of the maximum witness entry was set to be $\ell = b_{\mathbb{G}} + \lambda + 4$, to emulate a call as a subroutine from the Hadamard product argument (see section 4.2.2).

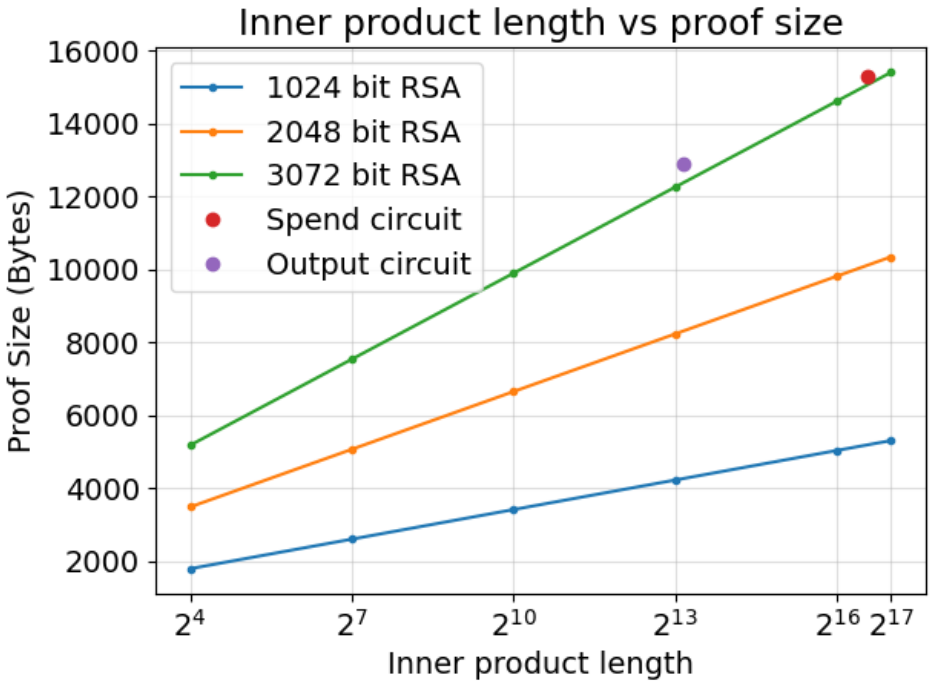

Figure 4.1: Proof sizes for different RSA strengths and inner product lengths obtained via a prototype implementation of the inner product argument. Note the log-scale on the x-axis. For reference, the figure also includes the proof sizes of the inner-product sub-protocol computed in section 4.2.2 for the Spend and Output circuit (red and purple dot).

Methodology. For each pair of security level and vector length, a new RSA group with appropriate bit-size is generated. Then, an element f is sampled

uniformly at random from the group. The entries of $\mathbf{g}, \mathbf{h}$ and the element e are generated by picking random exponents from $\{0, \ldots, 2^\ell\}$ and raising f to the respective power. Similarly, witness entries of $\mathbf{a}, \mathbf{b}$ as well as r are simply sampled randomly from $\{0, \ldots, 2^\ell\}$. Finally, C is computed as $C = \left(\mathbf{g}^{\mathbf{a}}\mathbf{h}^{\mathbf{b}}e^{\langle a,b\rangle}f^r\right)^2$, and the inner product proof is generated.

The results are shown in figure 4.1. The figure also contains two data points for the proof size of the inner-product sub-protocol of the Spend and Output circuit that was derived in section 4.2.2. These estimates should be directly comparable to the proof sizes obtained via the prototype implementation. Indeed, the computed proof sizes match the real size closely, while slightly overestimating due to the upper bounds used.

In other aspects, the proof sizes behave as expected. The figure depicts nicely how the proof size scales logarithmically with the length of the inner product (note the log-scale on the x-axis).

Chapter 5

Lowering the Verification Time of the Inner Product Argument

This chapter shows how the verification of the inner product argument due to Towa and Vergnaud [29] can be improved from $\mathcal{O}(n(\ell + b_G + \log(n)\log(P)))$ RSA group operations to $\mathcal{O}(n\log(\lambda))$ RSA group operations, $\mathcal{O}(n(\ell + b_G + \log(n)\log(P))/\lambda)$ multiplications in $\mathbb{Z}_p$, where p is a λ-bit prime, and $\mathcal{O}(n\log(n))$ integer multiplications with integers less than $\max\left(2^{\ell}P^{2n'+1},\ 2^{2\ell}P^{2n'}\right)$. This is achieved using techniques based on non-standard assumptions and developed by Wesolowski [30] and then Boneh, Bünz and Fisch [12].

5.1 The Adaptive-Root Assumption

The adaptive root assumption was introduced by Wesolowski in 2018 [30]. Intuitively, the adaptive-root assumption states that it is hard to find random roots of arbitrary group elements. Let $Primes(\lambda)$ denote the set of all λ-bit prime numbers. The adaptive root assumption is defined as follows.

Definition 5.1 ***Adaptive-Root Assumption*** *A group generator* G *satisfies the* (T, ε)*-adaptive-root assumption, if for all* $\lambda \in \mathbb{N}$ *and for all adversaries* $(\mathcal{A}_0, \mathcal{A}_1)$ *with running time at most* $T(\lambda)$*,*

$$Pr\left[\begin{array}{l} u^{\ell} = 1 \wedge w \neq 1 \ : \end{array} \begin{array}{c} (\mathbb{G}, P) \leftarrow_{\$} \mathsf{G}(1^{\lambda}) \\ (w, state) \leftarrow_{\$} \mathcal{A}_0(\mathbb{G}) \\ \ell \leftarrow_{\$} Primes(\lambda) \\ u \leftarrow_{\$} \mathcal{A}_1(\ell, state) \end{array}\right] \leq \varepsilon(\lambda)$$

An important remark is that this assumption does *not* hold in an RSA group Z_N^*, as it is easy to find roots of the element -1 in Z_N^*. Instead, one must work in the group $Z_N^* \setminus \{-1, 1\}$, where the assumption is believed to hold.

5.2 Proofs of Exponentiation

Wesolowski proposed the following interactive proof, whose security relies on the adaptive root assumption. Let $\mathbb{G}$ be a group and let $y, g \in \mathbb{G}$ and $t \in \mathbb{Z}^+$ be public values. The protocol allows a prover to prove to a verifier that $y = g^{2^t}$. Note that as both the prover and the verifier know the values g, and t, the verifier could compute g^{2^t} on its own. Concretely, using a square-and-multiply method, y can be computed in t group operations. The protocol allows, however, to verify the computation in time $polylog(t)$. The setting for which the protocol was originally proposed is one where t is super-polynomial and the verifier wants to verify quickly that the prover has performed the expensive computation.

Boneh, Bünz and Fisch generalized the above protocol in 2018 to support exponents that are not powers of two, i.e. allow to prove that $w = u^x$ for $w, u \in G$ and arbitrary $x \in \mathbb{Z}$. In the same work, they further generalize the protocol to support proving that $w = \phi(x)$, for an arbitrary function $\phi : \mathbb{Z}^n \mapsto G$ that is homomorphic, i.e. ϕ has the property that $\forall a, b, \in \mathbb{Z},\ x, y \in \mathbb{Z}^n : \phi(ax + by) = \phi(x)^a \cdot \phi(y)^b$. They called this protocol *Proof of Homomorphism Preimage*, or PoHP. Still, just as with the original protocol, the only purpose of PoHP is to reduce the verifier's computation time. This most general version is also applicable in our case, as we will show below.

All these protocols can be proven secure (i.e. the prover cannot convince the verifier of a false statement) under the adaptive root assumption.

5.3 Applications to the Inner Product Argument

This section shows how to use the PoHP to reduce the verification time of the inner product argument from $\mathcal{O}(n(\ell + b_G + \log(n)\log(P)))$ RSA group operations to $\mathcal{O}(n\log(\lambda))$ RSA group operations, $\mathcal{O}(n(\ell + b_G + \log(n)\log(P))/\lambda)$ multiplications in $\mathbb{Z}_p$, where p is a λ-bit prime and $O(n\log(n))$ integer multiplications with integers less than $\max\left(2^{\ell}P^{2n'+1},\ 2^{2\ell}P^{2n'}\right)$.

5.3.1 Overview

Recall that in the inner-product argument of TV20 (section 2.7), the verifier's work is aggregated into a single multi-exponentiation of the form

$$\left(\prod_{i=1}^{n} g_i^{\prod_{j\in S_i} x_j}\right)^{4x_{n'+1}a'} \left(\prod_{i=1}^{n} h_i^{\prod_{j\in[n]\setminus S_i} x_j}\right)^{4x_{n'+1}b'} e^{4a'b'} f^{4u} \tag{5.1}$$

$$= \left(U_{n'}^{x_{n'}} \prod_{i=1}^{n'-1} U_i^{x_i x_{i+1}\ldots x_{n'}} C^{x_1\ldots x_{n'}} \prod_{i=1}^{n'-1} V_i^{x_{i+1}\ldots x_{n'}} V_{n'}\right)^{2x_{n'+1}^2} \Gamma^{2x_{n'+1}} \Delta^2 \tag{5.2}$$

Now, instead of the verifier evaluating the multi-exponentiation, we propose to run a PoHP, shifting the burden of the heavy computation from the verifier to the prover.

Before going into more detail, it is briefly demonstrated that any multi exponentiation in a group $\mathbb{G}$ using some fixed bases $g_1, \ldots, g_n$ is a homomorphic function and therefore eligible for a PoHP. Let

$$\begin{aligned} \phi \colon \mathbb{Z}^n &\to \mathbb{G} \\ (x_1, \ldots, x_n) &\mapsto \prod_{i=1}^{n} g_i^{x_i}. \end{aligned}$$

Then, for $a, b \in \mathbb{Z}$ and $x, y \in \mathbb{Z}^n$,

$$\begin{aligned} \phi(ax + by) &= \prod_{i=1}^{n} g_i^{ax_i + by_i} = \prod_{i=1}^{n} g_i^{ax_i} \cdot \prod_{i=1}^{n} g_i^{by_i} \\ &= \left(\prod_{i=1}^{n} g_i^{x_i} \right)^a \cdot \left(\prod_{i=1}^{n} g_i^{y_i} \right)^b = \phi(x)^a \cdot \phi(y)^b. \end{aligned}$$

Therefore, the PoHP can be applied with a multi-exponentiation ϕ.

5.3.2 Detailed Changes to the Protocol

This section describes the changes to the protocol in detail. There are two possible ways to integrate the PoHP. For both variants, the respective PoHP are run *after* the final values of the parent protocol (i.e. a', b', u) have been received. This is important as for example, integrating the PoHP with the last two steps of the parent protocol would violate the assumptions of PoHP.

The first approach mimics the design of the original protocol, where the verifier evaluates 5.1 and 5.2 separately, and then asserts that they evaluate to the same value w. To do so, the prover first sends over w. Then, two instances of PoHP are run in parallel to prove that both sides evaluate to w.

In the second approach, equations 5.1 and 5.2 are rearranged by moving all the terms to one side of the equation. Then, a PoHP is run to prove that this evaluates to 1.

The advantage of the latter approach is that less communication is required, as the protocol is only run once and w does not have to be sent over. The advantage of the former approach is that all the exponents are positive, and the prover does not have to do any inversions of group elements. We discuss the second approach below, but both can in practice be considered.

5.3.3 Effects on Verification Time

Before it is argued that extractability of the parent protocol still holds, this section shows how the verification performance is affected.

Previous Performance. The verifier's performance in the original inner product argument has already been discussed in section 2.7.2. To recap, using a square-and-multiply approach, one has to do $\mathcal{O}(n(l + b_G + \log(n)\log(P)))$ group operations in the RSA group. The number of required group operations can be reduced by using pre-computation, at the cost of increased memory requirements.

New Performance. In the alternative version of the protocol, the verifier does not compute the multi-exponentiation on its own, but still does a fair amount of computation during the PoHP [12]. First, the verifier must compute the exponents, which requires $O(n\log(n))$ integer multiplications with integers less than max $\left(2^{\ell}P^{2n'+1},\ 2^{2\ell}P^{2n'}\right)$ in absolute value. Thereafter, the verifier computes, for each base of the multi-exponentiation, the residue of the exponent of that base modulo p, a random λ-bit prime. Computing the residues takes $\mathcal{O}(n(\ell + b_G + \log(n)\log(P))/\lambda)$ multiplications in $\mathbb{Z}_p$. Note that an operation in $\mathbb{Z}_p$ (where p is a λ-bit prime) is significantly faster than a group operation in an RSA group with security level λ. Then, it has to evaluate the multi-exponentiation with reduced exponents, i.e. exponents that are at most λ bits in size (plus another single exponentiation with an exponent of the same length and another single group operation). This takes $\mathcal{O}(n\log(\lambda))$ group operations in the RSA group using a square-and-multiply approach (and the same optimizations using pre-computation are applicable here).

5.3.4 Security

Finally, we provide proof sketches that the change breaks neither zero-knowledge nor extractability of the parent protocol.

Zero-knowledge

When running the PoHP at the end of the parent protocol as described above, the verifier only receives values that it can compute efficiently on its own. Therefore, the zero-knowledge property of the parent protocol continues to hold.

Extractability

This section describes how the original proof of extractability can be adapted to the new version. For the original proof, the concrete assumptions on the underlying hidden-order group are the $(T^{strg}, \varepsilon^{strg})$-strong-root assumption,

the $(T^{ord}, \varepsilon^{ord})$-small-order assumption, the low dyadic-valuation assumption and the μ-assumption. Newly, the security will also be relying on the $(T^{ar}, \varepsilon^{ar})$-adaptive root assumption.

The Original Proof. The proof of extractability of the inner product argument by Towa and Vergnaud can be divided into two parts.

In the first part, it is shown that the sub-protocol that is run at the end of the recursion, i.e. when $n = 1$, is extractable, given five distinct accepting transcripts.

The structure is roughly as follows. The proof starts off by considering five distinct accepting transcripts, i.e.

$$\left(\Gamma, \Delta, x_j, a'_j, b'_j, u_j\right)_{j=1}^{5} \text{ such that } (g^{x_j a'_j} h^{x_j b'} e^{a'_j a'_j} f^{u_j})^4 = (C^{x_j^2} \Gamma^{x_j} \Delta)^2 \text{ for all } j \in [5].$$

It is then shown that with these accepting transcripts, one is either able to extract a valid witness, i.e. values (a, b, r) so that $C^2 = (g^a h^b e^{\langle a,b \rangle} f^r)^4$, or break one of the security assumptions. All in all, the probability that extraction fails in this first step where $n = 1$ is determined to be at most

$$\varepsilon_1^{ext} := \varepsilon^{ord} + \varepsilon^{strg} + max\left(2^{-b_G - \lambda + 1}, (1/2 - 2^{-\lambda} - (1 - \mu))^{-1}(\varepsilon^{ord} + \varepsilon^{strg} + 1 - \mu)\right).$$

In the second part, it is proven that at each recursion layer, extraction of a solution for the previous recursion layer is possible, given five distinct witnesses for the current layer. The resulting extraction error for the full protcol, i.e. the probability that extraction fails at any step during the protocol is

$$\varepsilon^{ext} := \varepsilon_1^{ext} + \left(1/2 - 2^{-\lambda} - (1 - \mu)\right)^{-1} (\varepsilon^{ord} + \varepsilon^{strg} + 1 - \mu).$$

Necessary Adaptions. The second part of the proof is unaffected by the change as the multi-exponentiation only comes into play at the last step of the recursion. Therefore, we only discuss on how to adapt the case $n = 1$ of the proof to the updated protocol.

There, the very beginning of the proof has to be changed: Given the five accepting transcript, one cannot assume anymore that the verification equation holds for all of them. This is because it could also happen that instead, the security of the PoHP was broken. However, we do know that either all accepting transcripts satisfy the verification equation, or the security of the PoHP (and hence the adaptive root assumption) was broken, which happens with probability at most ε^{ar}. In the former case, we continue as in the original proof. To account for the latter case, we add a term ε^{ar} to the extraction error. The new extraction failure probability, for the step $n = 1$ is now $\varepsilon_1^{ext} + \varepsilon^{ar}$. The new extraction error for the full updated protocol is therefore $\varepsilon^{ext} + \varepsilon^{ar}$.

Chapter 6

Conclusion

Our contributions. In this thesis, we evaluated the TV20 proof system as a possible alternative to Groth16 in the Zcash cryptocurrency. The motivation to do a comparison of this kind is that while Groth16 has extremely small (constant sized) proofs, it relies on an intricate trusted setup and its security is only proven in the generic group model. On the other hand, the (trusted) setup of TV20 is simple, the security is based on standard assumptions, but the proofs are of logarithmic size.

To be able to draw a comparison, the concrete proof size of TV20 when proving the statements in Zcash (called the *Spend* and *Output* statement) had to be computed. To do so, we encoded the statements as Diophantine equations and then computed a theoretical upper bound on the proof size.

In the case of 128 bits of security, the resulting proof sizes are 30.5 kilobytes for the Spend statement and 25.5 kilobytes for the Output statement. For Groth16, the size for any statement is 192 bytes. This translates to an increase in proof size by a factor of 133 to 159.

Additionally, we implemented a prototype version of the inner product argument of TV20 the get empirical data on how the proof size changes with the length of the inner product. The proof sizes derived on paper fit the observed data of the prototype implementation well.

As a final contribution, we demonstrated how the verification time of the inner product argument of TV20 can be lowered. This is achieved using a protocol called *Proof of Homomorphic Preimage*.

Future work. The topic offers great opportunity for future work.

An aspect of TV20 that unfortunately did not come into play when doing the encoding in chapter 4.1 is the fact that it works directly with integers instead of finite field elements. A possible explanation is that the Spend and Output statements of Zcash were designed directly for a proof system

for circuit satisfiability in finite fields. As such, any operations that are not efficiently encodable as a circuit over a finite field (but might be well-encodable as a Diophantine equation), were avoided from the start. Therefore, it would be interesting to look into whether there are any applications, in the blockchain-space or else, that are able to profit from the additional expressability of TV20. Candidate problems include RSA operations or integer linear programming, which are both difficult to encode as a circuit over a finite field.

Also, verification in TV20 is still computationally-intensive, even with the techniques from Chapter 5, which rely on a non-standard assumption. It would be interesting to see whether there is a way to lower the verification time even more, or use standard assumptions instead of the adaptive root assumption.

Another research direction is to extend the prototype implementation to the full TV20 proof system, as it currently only covers the inner product argument. This would allow to measure the proof size for the complete protocol.

Bibliography

[1] The official monero website. https://www.getmonero.org/. Accessed: 2022-08-02.

[2] The official zcash website. https://z.cash/. Accessed: 2022-08-02.

[3] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. Blake2: Simpler, smaller, fast as md5. *IACR Cryptol. ePrint Arch.*, 2013:322, 2013.

[4] Roberto Maria Avanzi. The complexity of certain multi-exponentiation techniques in cryptography. *Journal of Cryptology*, 18:357–373, 2004.

[5] László Babai and Shlomo Moran. Arthur-merlin games: A randomized proof system, and a hierarchy of complexity classes. *J. Comput. Syst. Sci.*, 36:254–276, 1988.

[6] Mihir Bellare and Oded Goldreich. On defining proofs of knowledge. In *CRYPTO*, 1992.

[7] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography: The case of hashing and signing. In *CRYPTO*, 1994.

[8] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. *2014 IEEE Symposium on Security and Privacy*, pages 459–474, 2014.

[9] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted edwards curves. In *AFRICACRYPT*, 2008.

[10] Daniel J. Bernstein and Tanja Lange. Montgomery curves and the montgomery ladder. *IACR Cryptol. ePrint Arch.*, 2017:293, 2017.

[11] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *STOC '88*, 1988.

[12] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. *IACR Cryptol. ePrint Arch.*, 2018:1188, 2018.

[13] Gilles Brassard, David Chaum, and Claude Crépeau. Minimum disclosure proofs of knowledge. *J. Comput. Syst. Sci.*, 37:156–189, 1988.

[14] David Chaum, Ivan Damgård, and Jeroen van de Graaf. Multiparty computations ensuring privacy of each party's input and correctness of the result. In *CRYPTO*, 1987.

[15] David Chaum, Eugène van Heijst, and Birgit Pfitzmann. Cryptographically strong undeniable signatures, unconditionally secure for the signer. In *CRYPTO*, 1991.

[16] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, 1986.

[17] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems. In *STOC '85*, 1985.

[18] Jens Groth. Non-interactive zero-knowledge arguments for voting. In *ACNS*, 2005.

[19] Jens Groth. On the size of pairing-based non-interactive arguments. *IACR Cryptol. ePrint Arch.*, 2016:260, 2016.

[20] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. *Zcash Specification*. Electronic Coin Company, 2022. https://zips.z.cash/protocol/protocol.pdf.

[21] Jonas Meier and Patrick Towa. TV20 prototype implementation. https://github.com/JomeierFL/tv20, August 2022.

[22] Carsten Lund, Lance Fortnow, Howard J. Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, pages 2–10 vol.1, 1990.

[23] Ju. V. Matijasevich. Enumerable sets are diophantine. *Sov. Math., Dokl., 11:354– 358*, 1970.

[24] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, 1987.

[25] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[26] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4:161–174, 2004.

[27] Adi Shamir. Ip = pspace. *J. ACM*, 39:869–877, 1992.

[28] Victor Shoup. Lower bounds for discrete logarithms and related problems. In *EUROCRYPT*, 1997.

[29] Patrick Towa and Damien Vergnaud. Succinct diophantine-satisfiability arguments. *IACR Cryptol. ePrint Arch.*, 2020:682, 2020.

[30] Benjamin Wesolowski. Efficient verifiable delay functions. In *IACR Cryptol. ePrint Arch.*, 2018.