b
UNIVERSITÄT
BERN

Cryptographic Primitives for On-chain Tumbler Designs

Improving the Cost of Autonomous Cryptocurrency Mixers

Bachelor Thesis

Marko Cirkovic

from

Bern, Switzerland

Faculty of Science, University of Bern

September 6, 2022

Prof. Christian Cachin
Dr. Duc V. Le
Cryptology and Data Security Group
Institute of Computer Science
University of Bern, Switzerland

Abstract

Permissionless blockchains have attracted substantial interest from the both research and practitioner communities because of their openness, transparency, and decentralization. There are various applications utilizing permissionless blockchains in different fields, ranging from simple financial payments to decentralized finance (DeFi) applications (e.g., lending, trading, and asset management), gaming platforms, and decentralized autonomous organization (DAO). However, the growth of DeFi has raised a number of privacy problems, as DeFi applications often reveal financial positions and users' balances in plain text. Therefore, blockchain privacy solutions are becoming increasingly vital. In particular, add-on privacy services on existing non-private blockchains are becoming de facto solution for privacy for many blockchain users. One prominent example of add-on solutions is Tornado Cash. Tornado Cash allows blockchain users to effectively hide the on-chain linkage between the source and the destination addresses, hence, allow them to securely obfuscate their transaction graphs. However, Tornado Cash has one major drawback. The cost of interacting with Tornado cash contract is still very expensive for standard users, due the use of computationally expensive cryptographic machineries.

In this work, we investigate two different approaches for reducing the cost of using an on-chain mixer. In our first approach, we keep the number of operations in a deposit transaction constant by introducing a RSA accumulator. In the second approach we will introduce Merkle Pyramid Builder, a method to batch deposit transactions and therefore reducing the overall cost of a mixer. We include a formal specification of our two systems, as well as a discussion of privacy and security. Furthermore, we implemented and evaluated our two systems.

Contents

Chapter 1

Introduction

Popular permissionless blockchains, such as Bitcoin, provide only pseudonymity, not anonymity. Every transaction reveals the amount transferred, the time, the transaction fees and the user's addresses. As a result, multiple works have aimed to deanonymize clients [18], cluster addresses [28] and develop privacy solutions to protect client's privacy [17]. While privacy-preserving blockchains are capable of protecting their users' privacy, retrofitting a blockchain with privacy has been proven to be difficult and is still a work in progress. There are two available privacy solutions. The first option is to utilise private-by-design blockchains that hide transactions data from monitoring parties. As a second option, there are add-on privacy solutions that attempt to provide privacy for customers of current blockchains that do not protect privacy.

Inspired by the design of Zerocash [13], on-chain zero-knowledge-proof (ZKP) mixers are one of the most popular add-on privacy solutions, in which users deposit a fixed amount of coins into a pool, and withdraw these coins into a new address. A decentralized mixer should not be able to identify the link between deposit and withdrawal addresses. Zero-knowledge-proof-based mixers are decentralized applications (dApps) that run on a blockchain with smart contracts. For instance, Tornado Cash [5] is one of the most active ZKP-based mixers. Tornado Cash has thus far processed $2,547,731$ ETH worth of transactions for $27,631$ users. However, a known problem with Tornado Cash is the high gas cost of depositing. Concretely, the cost of depostiting, up to August 22'[1], is approximately $1.1m$ gas which currently corresponds to about 130\$ but was also about 600\$.

The focus of this thesis is to design new methods that help reduce the overall cost related to the mixer of Tornado Cash. Our work consists of two primary approaches. The first approach uses a RSA accumulator [15]. We attempt to replace the $\mathcal{O}(\log n)$ updating cost of the Merkle tree (where n is the number of leaves) with the $\mathcal{O}(1)$ updating cost of the RSA accumulator. This allows us to reduce the number of operations in a deposit transaction, which leads to a reduction in gas costs as well. In our second approach, we introduce a more efficient way of updating the Merkle tree in this setting. Instead of updating the entire Merkle tree after every single deposit, we batch the deposits together. This results in a decrease in the average number of times the Merkle tree is utilized, hence reducing costs. Depending on the amount of deposits we batch together, we can have up to eight times less costs for depositing than Tornado Cash.

Our contributions can be summarized as follows.

- Inspired by Zerocoin [36], we formalize and present a practical RSA accumulator-based mixer, which breaks the linkability between deposited and withdrawn coins of a client on a smart contract enabled blockchain and we provide a security and privacy discussion of the proposed system.
- We formulate and present our developed method Merkle Pyramid Builder, which makes the method used by Tornado Cash more cost effective and we also provide a security and privacy discussion of the proposed system.

[1] US Government bans Tornado Cash [6]

- We implement our two approaches and show that the systems can be deployed and operated efficiently on a permissionless blockchain.
- We find that the RSA accumulator approach allows unlimited deposits. However, the cost of a withdrawal transaction is too high and because of that not suitable for the blockchain environment.
- We observe that by using Merkle Pyramid Builder Method, we can reduce the deposit cost from two to eight times, depending on the batching size.
- We present a method how to lower the cost even more using verifiable computation, but we leave the implementation to future exploration.

Thesis Organization. Chapter 2 outlines the necessary background before introducing our two approaches in the following two chapters. In chapter 3, the RSA accumulator mixer is described. We begin by introducing further theoretical fundamentals and continue then displaying the algorithm and engaging in a discussion. The chapter will conclude with practical outcomes and suggestions for additional cost reduction. In chapter 4, similar to the preceding chapter, we present further theoretical basics. Then, we introduce the algorithm of the Merkle Pyramid Builder. We will show the outcomes of the benchmarks of our implementation after a discussion. At the end of the chapter, we will provide an approach that might cut expenses even more. In chapter 5 we will provide a conclusion and some additional remarks. Lastly, an appendix A with additional information and code snippets is included.

Chapter 2

Background

2.1 Background on Smart Contract Blockchains and On-Chain Mixers

Ethereum Blockchain. The Ethereum blockchain functions as a distributed virtual machine that supports decentralized applications. The ability of Ethereum to execute extremely expressive languages enables developers to design smart contracts. Nick Szabo [38] first introduced the concept of smart contracts in 1994. Ethereum keeps track of the state of each account, including externally-owned accounts (EoA) that are managed by a private key and contract accounts that are owned by the contract's code. The state changes of the virtual machine are determined by EoA transactions.The purpose of a transaction is either to transfer Ether or to initiate the execution of smart contract code. The expenses associated with performing functions are given in gas units. In Ethereum, the sender of a transaction pays for the execution of any contract actions caused by that transaction.

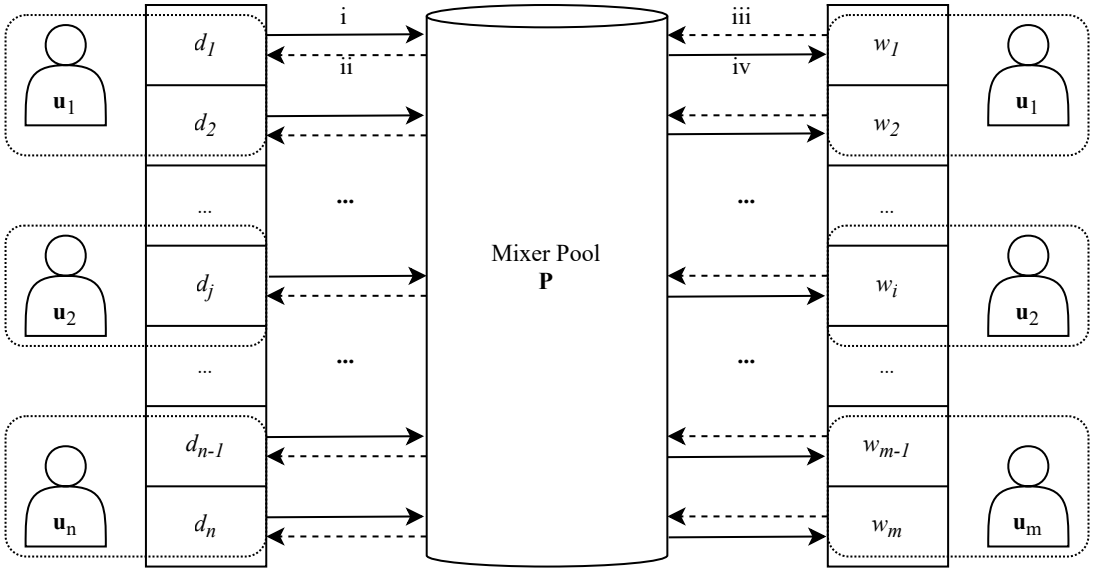

Figure 2.1. Concept of a mixer. $\{d_1, \ldots, d_n\}$ and $\{w_1, \ldots, w_m\}$ represent a pool of depositors and a pool of withdrawers, accordingly. An arrow indicates the transfer of coins, whereas a dotted arrow indicates the transfer of notes. When an user **u** deposits coins into pool **P** (i), **u** obtains a note from **P** (ii); to withdraw, **u** provides this note to **P** (iii) and receives the coins once **P** verifies the note (iv). A user can control many addresses. An address can be used several times to deposit or withdraw funds.

On-Chain Mixers. On-chain mixers are one of the most often used add-on privacy solutions. A user can initiate transactions and deposit funds to the mixer. In addition, the user needs to provide a fresh account

address to which the funds will be transferred to. This is an anonymous address, and only the user should know who it belongs to. The user is able to withdraw funds at a later time. However, it is advisable to wait as long as possible, since anonymity may be guaranteed more securely in this manner. In the mean time other users can deposit funds into the mixer. If a user wishes to withdraw his funds, he must prove to the mixer that he already deposited funds. The purpose of a mixer is to mix the coins of various users to break the link between the old and the new address. Figure 2.1 illustrates how a mixer works. For a more thorough background on blockchain mixers, we refer the interested reader to [41].
The anonymity of the mixer improves as more users utilize it. Using zero-knowledge proofs, such as zk-SNARK, on-chain mixers attempt to ensure unlinkability of the addresses. Tornado Cash [5] is one of the most popular mixers in use today.

2.2 Cryptographic Primitives

Notation. We refer 1^λ as the security parameter and $\mathsf{negl}(\lambda)$ as the negligible function in λ. We define the pair of public and secret key as $(\mathsf{pk}, \mathsf{sk})$. In addition, we suppose that pk can be computed from sk using the deterministic function $\text{EXTRACTPK}(\mathsf{sk}) = \mathsf{pk}$. The concatenation of two string k and r is denoted by $k||r$. With $\mathbb{Z}_{\geq a}$ we define a set of integers, which are greater or equal to a, i.e., the $\{a, a+1, \dots\}$. Let PPT be probabilistic polynomial time. We define an instance of statements, i.e., a boolean expression as $st[a, b, c \dots]$ where $a, b, c, \dots$ have fixed public values. To denote private inputs in the statement we use shaded areas like $st[a, b, c; \; i, j, k \;]$.

Collision resistant hash function. We denote H to be a family of collision resistant hash functions, if for any PPT $\mathcal{A}$ given $h \xleftarrow{\$} H$, the probability that $\mathcal{A}$ learns x, x', such that $h(x) = h(x')$ is negligible. The cryptographic hash function h is referred to as the fixed function $h : \{0,1\}^* \to \{0,1\}^\lambda$.

Non-interactive Zero-Knowledge Proofs (NIZK). Zero Knowledge Proof (ZKP) is a cryptographic primitive that enables a prover to convince a verifier of the validity of certain assertions without exposing the verifier with any valuable information. Several ZKPs need several interactions between the prover and the verifier. NIZK is, on the other hand, one of the variants of ZKP in which there is no need for interaction between verifier and prover. As shown in [19], NIZK can be realized if there is a shared common reference string between verifer and prover. Adapting the definition from [22, 23], we define NIZK as follows.

Definition 2.2.1 (NIZK)**.** *A non-interactive zero-knowledge proof (NIZK) system consists of three algorithms* (NIZK.SETUP, NIZK.PROVE, NIZK.VERIFY) *relative to a hard relation R defining the language $\mathcal{L}_R := \{st \mid \exists w : (st, w) \in R\}$ where st is that statement and w is a witness, such that:*

- *params* $\leftarrow$NIZK.SETUP*(1^λ) takes as input the security parameter, and outputs the common reference string params containing* $(\mathsf{ek}, \mathsf{vk})$.
- $\pi \leftarrow$NIZK.PROVE*(params, st, w) takes as input the reference string params, any $(st, w) \in R$ and outputs a proof π that $(st, w) \in R$.*
- $0/1 \leftarrow$NIZK.VERIFY*(params, st, π) takes as input the reference string params, a statement st, and a proof π and outputs 1 if the proof π verifies that $s \in \mathcal{L}_R$, otherwise, returns 0.*

A zero-knowledge proof π for the relation

$$R : \{(a, b, c, \dots; \; x, y, z \dots \;) : f(a, b, c, \dots; \; x, y, z \dots \;)\}$$

means that the prover has a knowledge of $(x, y, z, \dots)$ such that $f(a, b, c, \dots; \; x, y, z \dots \;)$ is true, where $a, b, c \dots$ are public variables.

A zero-knowledge proof of some statement satisfy the following three properties:

- **Completeness:** If *st* is true an honest verifier will always be convinced by an honest prover.

- **Soundness:** For false statements, a prover cannot convince the verifier (even if the prover cheats and deviates from the protocol)
- **Zero-knowledge:** No verifier learns anything other than the fact that the statement *st* is true if it is true. To put it in another way, knowing the statement, but not the secret, is enough to construct a scenario in which the prover knows the secret.

We defer the formal definitions of these properties to Groth [27].

Commitment Scheme. During the committing round, a commitment scheme allows a client to commit to selected values while concealing them from others. During the revealing round, the client can choose to reveal the committed value.

Definition 2.2.2 (Commitment Scheme)**.** *A commitment scheme includes two algorithms* (COMMIT, VERIFY)

- $\mathsf{cm} \leftarrow \textsc{Commit}(m, r)$ *accepts a message* m *and a secret randomness* r *as inputs and returns the commitment string* cm.
- $0/1 \leftarrow \textsc{Verify}(m, \mathsf{cm}, r)$ *accepts a message* m*, a commitment* cm *and a decommitment value* r *as inputs, and returns* 1 *if the commitment is opened correctly and* 0 *otherwise.*

Our commitment scheme must satisfy two security requirements:

- **Binding:** Except for a negiligible probability, no adversary can efficiently produce cm, (m_1, r_1) and (m_2, r_2) such that $\textsc{Verify}(m_1, \mathsf{cm}, r_1) = \textsc{Verify}(m_2, \mathsf{cm}, r_2) = 1$ and $m_1 \neq m_2$.
- **Hiding:** Except for a negiligible probability, cm does not reveal anything about the committed data.

Authenticated Data Structure ADS. An authenticated data structure (ADS) is suitable for particular operations that an untrusted verifier is qualified of performing. The verifier can then efficiently verify the validity of the proof results. This is achieved by the prover generating a concise proof that the verifier can verify using the outcomes of each operation. We are solely interested in a data structure for set membership in this work.

Definition 2.2.3 (Authenticated Data Structure)**.** *An authenticated data structure consists of four algorithms* $\Pi = (\textsc{Init}, \textsc{Prove}, \textsc{Verify}, \textsc{Update})$.

- $y \leftarrow \textsc{Init}(1^\lambda, X)$ *takes the security parameter and a list* $X = (x_1, \ldots, x_n)$ *as inputs and outputs* $y \in \{0,1\}^\lambda$.
- $\pi \leftarrow \textsc{Prove}(i, x, X)$ *takes an element* $x \in \{0,1\}^*, 1 \leq i \leq n$ *and a* $X = (x_1, \ldots, x_n)$ *as inputs, and outputs the proof that* $x = x_i \in X$.
- $0/1 \leftarrow \textsc{Verify}(i, x, y, \pi)$ *takes an element* $x_i \in \{0,1\}^*, 1 \leq i \leq n, y \in \{0,1\}^\lambda$ *and a proof* π *as inputs. It outputs* 1 *if* $x = x_i$ *and* $y = \textsc{Init}(1^\lambda, X)$ *and* 0 *otherwise.*
- $y' \leftarrow \textsc{Update}(i, x, X)$ *takes an element* $x \in \{0,1\}^*, 1 \leq i \leq n$ *and* X *as inputs, and outputs* $y' = \textsc{init}(1^\lambda, X')$ *where* X' *is* X *but* $x_i \in X$ *is replaced by* x.

We require that the ADS be correct and secure. For the formal definitions of these qualities we refer to the book by Boneh and Shoup [21]. Merkle tree [34] or RSA-accumulator [31, 15] are examples of authenticated data structures.

2.3 System Overview

We will now describe the system's components, the setup phase, the client, and the smart contract algorithm.

2.3.1 System Components

The system consists of two components: the **client** and the **smart contract**. A client interacts with the smart contract via accounts held by external parties. A client can either deposit or withdraw coins. The smart contract manages both deposits and withdrawals. The contract keeps track of various data structures and parameters to verify the accuracy and validity of transactions received to it.

Deposit. The client can load a fixed amount of coins in the system, making a depositing transaction. If this transaction is valid, miners will record the transaction in a blockchain block. Each deposit transaction of the client reduces his credit by a fixed amount of coins on his address.

Withdraw. To withdraw coins a client must make a withdraw transaction. These transactions contain a cryptographic proof, which proves that the client has made a deposit transaction in the past without revealing which one it was.

2.3.2 Contract Setup

In the setup phase all public parameters, which are required by the client and the smart contract of the mixer, are generated. Furthermore, the contract will be initialized with different data structures to be protected from double-withdrawal. The deposit amt is specified as a fixed deposit amount of coins. Furthermore the smart contract is setup with two empty lists: a list, $\mathsf{DepositList}$, that includes all commitments cm contained in depositing transactions and another list, $\mathsf{NullifierList}$, that contains all unique identifiers (i.e., sn) appeared in withdrawal transactions. We refer to pp^h as the state of the contract in block h. The state includes all data structures of the contract which were initialized in the setup phase. Furthermore, this state is implicitly provided to all client and contract algorithms. At last, the contract is deployed in this phase.

2.3.3 Client Algorithm

A client can communicate with the smart contract using the following algorithms in a mixing system. Note that each transaction is automatically signed using the private key associated with the Ethereum account that executes the transaction.

- $(wit, \mathsf{tx}_{dep}) \leftarrow \textsc{CreateDepositTx}(\mathsf{sk}, \mathsf{amt})$ takes the secret key sk and the amount amt, specified in the setup phase as inputs and outputs a witness wit, which is needed to create withdrawals in the future, and a deposit transaction tx_{dep}.
- $\mathsf{tx}_{wdr} \leftarrow \textsc{CreateWithdrawTx}(\mathsf{sk}', wit)$ takes as input the secret key sk' and the witness wit and outputs a withdrawing transaction tx_{wdr}.

2.3.4 Smart Contract Algorithm

The smart contract should be able to handle deposits and withdrawals of coins. It consists of the following algorithms:

- $0/1 \leftarrow \textsc{AcceptDeposit}(\mathsf{tx}_{dep})$ takes as an input the deposit transaction tx_{dep} and outputs 1 if the transaction was successful and 0 otherwise.
- $0/1 \leftarrow \textsc{IssueWithdraw}(\mathsf{tx}_{wdr})$ takes the withdrawing transaction tx_{wdr} as the input and outputs 1 and deposits amt to the sender $\mathsf{tx}_{wdr}.\mathsf{sender}$ if the transaction was successful and 0 otherwise.

2.4 System Requirements

A secure autonomous mixer must satisfy certain properties. We use the properties from the AMR paper [30]. In the following sections, we need $\mathsf{tx}.sender$ to indicate the address of the sender from which tx originates.

Privacy. Clients' privacy must be protected by our system. Considering an adversary with access to the history of all depositing and withdrawing transactions made to our mixer contract, the system must ensure that depositing and withdrawing transactions cannot be linked.

Correctness. The goal of our system is to prevent clients from stealing coins from the contract or from other clients. The system must be able to guarantee that a customer cannot withdraw more from the contract than he deposited.

Availability. Clients should always be able to use the mixer. No one should be able to block clients from depositing or withdrawing coins.

Chapter 3

Improved Mixer: RSA + NIZK

In this chapter, we adapt the design of Zerocoin [36] to our design to reduce the costs associated with the use of an on-chain mixer. We use a RSA accumulator as our authenticated data structure, and as a NIZK proof, we will use a structure that already exists in Zerocoin [36]. Zerocoin employs a modified version of the proof provided by Camenisch and Lysyanskaya [25].

Zerocoin approach. As stated, we follow the same protocol as Zerocoin [36]. We implement a RSA accumulator to determine if a client has deposited funds. As proof of knowledge, we utilize the proof scheme proposed by Zerocash. Our objective is to reduce the deposit fee incurred when using the Tornado Cash service. This system makes use of a Merkle tree. The issue with this strategy is that it is expensive to update the Merkle tree, resulting in high transaction costs. Using a RSA accumulator eliminates the high cost of updating a Merkle tree after a transaction because it is significantly less computationally intensive. Tornado Cash takes $\mathcal{O}(k \log n)$ operations for k deposit transactions and a n leaves. We are now able to reduce the number of operation to $\mathcal{O}(1)$.
In the following sections we will explain how the system of the mixer is built.

3.1 Cryptographic Building Blocks

RSA accumulator. A cryptographic accumulator is an authenticated data structure that generates a brief binding commitment to a set of items as well as brief membership/non-membership proofs for each item in the set. A RSA accumulator is an universal accumulator that supports both batch-processed membership and non-membership proofs. Constructing the accumulator involves selecting a modulus N from a group of unknown order, which can be generated by a trusted third party. The RSA accumulator's starting state is the generator g sampled from the group of unknown order, implying that the accumulator's list of items is empty. Strong collision resistance is exhibited by an accumulator when the RSA strong assumption is hard. This concept was formally introduced by Benaloh and de Mare [15]. The RSA accumulator is described as in the Zerocoin paper [36].

Definition 3.1.1 (RSA accumulator)**.** *A RSA accumulator scheme consists of four algorithms:* (ACC.INIT, ACC.PROVE, ACC.VERIFY, ACC.UPDATE)*:*

- *params* $\leftarrow$ ACC.INIT(1^λ) *takes a security parameter as input. Two prime numbers* p *and* q *are generated to be the secret key. Further* $N = pq$ [1] *is calculated and a seed value* $u \in QR_N, u \neq 1$ *is chosen. The output is* (N, u) *as params.*
- $A \leftarrow$ ACC.UPDATE(*params*, P) *takes params* (N, u) *and a set of primes* $P = \{p_1, \ldots, p_n\}$ *as inputs and it computes the value of the accumulator* $A = u^{p_1 \cdots p_n} \mod N$.
- $\pi \leftarrow$ ACC.PROVE(*params*, p, P) *takes as an input params* (N, u)*, a prime* p *and a set of primes* P *and outputs* π *also called witness. The witness* π *is the accumulation of all the values in* P *besides* p*, i.e.,* $\pi =$ ACC.UPDATE(*params*, $P \setminus \{p\}$).

[1] p and q are going to be discarded after the initialization

- $0/1 \leftarrow$ ACC.VERIFY$(params, p, A, \pi)$ *takes as inputs params* (N, u)*,* p *prime, a value of the accumulator* A *and a witness* π*. It computes* $A' = \pi^p \mod N$ *and outputs 1 if* $A' = A$*. Otherwise 0.*

For simplicity, the description above uses the full calculation of A. It should be noted that accumulators can also be updated incrementally. This also implies that the witness must be updated for all other participants; however, because the commitments are available on-chain, participants may always compute the witness. If A_n is an accumulator, you can add an element x and determine A_{n+1} by computing $A_{n+1} = A_n^x \mod N$.

Primality test. The committed value of the client must be a prime number. Unless we keep control over this, the system's security will be compromised. An adversary may attempt to deposit a commitment that for instance is a composite of two prime numbers. This commitment is connected with a fixed number of coins. The commitment would be accumulated by the accumulator, but since there are two prime numbers involved, there would be two entries in the DepositList. On the other hand, the balance of the mixer increases only by one, as one deposit transaction has been done. If the adversary wishes to withdraw the coins, he proves to the system with one value of the composition that he knows one value of the accumulator, so regaining possession of the coins. Further, he can withdraw coins with the second value of the composition, as the second value does not display on the NullifierList and he has never withdrawn coins according to the system. Fermat primality test [32] and Miller-Rabin test [33] are the most used tests for prime numbers since they are quick and straightforward. These tests are probabilistic. Nevertheless, this form of test has a slight chance of incorrectly identifying a number as prime when it is not. In order to prevent this, the test is done several times to reduce the likelihood of mistakes.

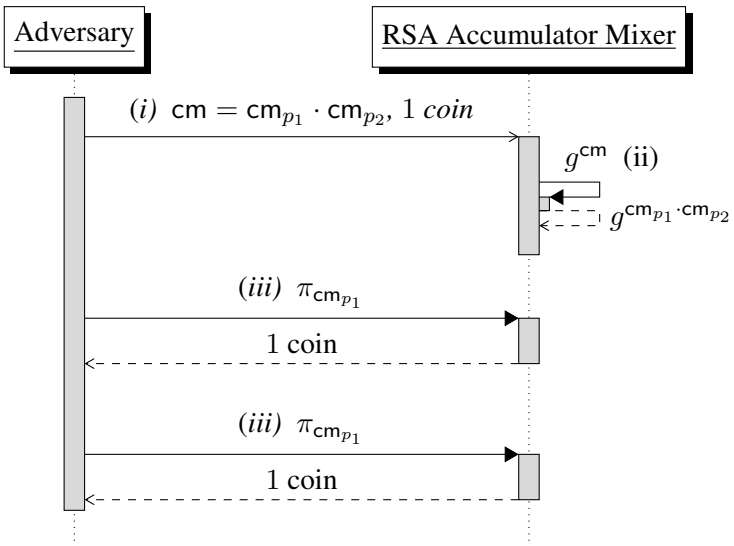

Figure 3.1. Possible attack on the mixer. An arrow represents a transfer. g is the current accumulator value. *(i)* An adversary can generate a commitment cm such that it is a combination of two commitments that are prime numbers; hence, the commitment sent to the mixer is not prime. Note that the adversary only sends one coin. *(ii)* the accumulator will increase his value to the power of the commitments, which is equivalent to raising it to the two part commitments. *(iii)* the adversary is now capable to provide proof $\pi_{\mathsf{cm}_{p_1}}$ and $\pi_{\mathsf{cm}_{p_2}}$ and withdraw twice the amount of the deposit.

Definition 3.1.2 (Primality test)**.** *The primality test consists of one algorithm:*

- $0/1 \leftarrow$ ISPRIME(n) *takes an integer* n *and returns* 1 *if it is probably a prime number and* 0 *otherwise.*

Withdrawal Proof. To withdraw coins from the system, a user must prove the following two conditions: (1) The client is aware of the committed values, and (2) the client has not previously withdrawn by passing a new nullifier value. Specifically, a client must provide a proof of the following relationship:

$$\mathcal{R}_{wdr} = \{(i, sn, N, A, g, h; \mathsf{cm}, r, wit) : \text{ACC.VERIFY}(i, sn, A, wit) = 1 \wedge \mathsf{cm} = g^{sn} \cdot h^r\} \quad (3.1)$$

The details of this proof can be found in Appendix A. Note that we have to commit our message to a prime, which is cm.

3.2 Detailed Construction

3.2.1 Contract Setup

CONTRACTSETUP(1^λ)

1 : Choose $\mathsf{amt} \in \mathbb{Z}_{>0}$ to be a fixed deposit amount
2 : Initialize the RSA accumulator
$(N, u) \leftarrow$ ACC.INIT(1^λ),
3 : Construct C_{wdr} for relation $\mathcal{R}_{wdr}$
4 : Let Π be the $NIZK$ instance for R_{wdr}
5 : Setup *params* $\leftarrow \Pi$.SETUP($1^\lambda, R$)
Initialize $\mathsf{DepositList} = \{\}, \mathsf{NullifierList} = \{\}$
6 : Deploy smart contract with parameters:
$\mathsf{pp} = \{\mathsf{amt}, N, u, \textit{params},$
$\mathsf{NullifierList}, \mathsf{DepositList}\}$

Figure 3.2. Pseudocode for the setup of the smart contract.

Initialization of the smart contract occurs during the setup phase. First, we impose a restriction on the number of coins a consumer may deposit in a single transaction. This is done so that an opponent cannot determine who owns the coins based on the quantity. The RSA accumulator is then initialized in accordance with Section 3.1.1. When configuring the $NIZK$ instance, we are provided with two keys $(\mathsf{ek}, \mathsf{vk})$. In addition, the lists DepositList and NullifierList are initialized. The first includes all commitments, while the second has all unique identifiers. The public parameter pp contains all the required information for a client to engage with the contract.

3.2.2 Deposit Interactions

Client. The client can deposit money into the smart contract using CREATEDEPOSITTX. By sampling random r and sn, the message will be committed to a prime number. This is required since the smart contract accepts only prime numbers as commitments. Finally, transactions must be signed using sk to prevent an attacker from modifying the transaction's recipient. In addition, a witness is produced that will be needed to withdraw the coins again in the future.

Contract. When a smart contract receives a deposit transaction from an external address, it first has to verify if the commitment is a prime number and it also has to verify the amount. Then DepositList is updated, the index is incremented by one, and the accumulator is updated using the ACC.UPDATE function.

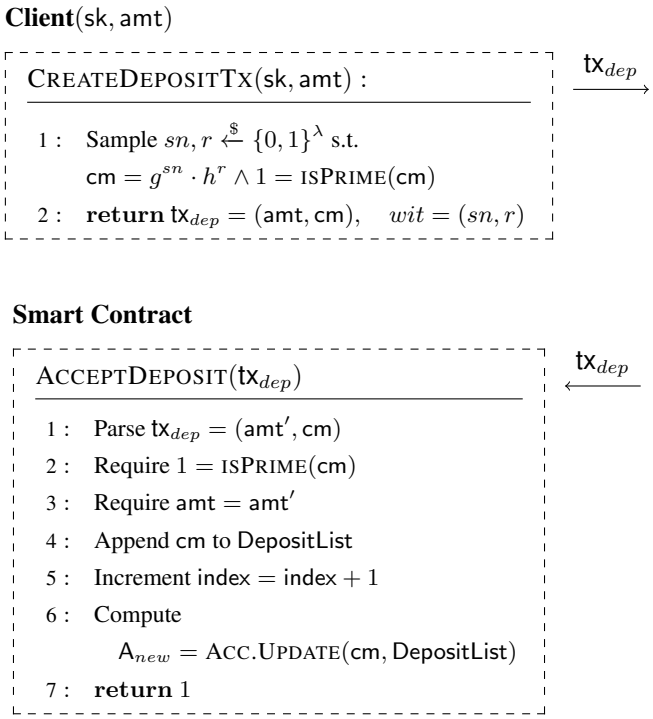

Figure 3.3. Deposit interactions between the client (CREATEDEPOSITTX algorithm) and the smart contract (ACCEPTDEPOSIT algorithm).

3.2.3 Withdraw Interactions

Client. Using the CREATEWITHDRAWTX function, a client may provide a proof to withdraw amt to the public key using the secret note, wit, and the secret key. The contract demands the client to provide a proof that the client has deposited coins in the past, as well as a nullifier value sn to verify that those coins have not been withdrawn previously, and to prevent customers from withdrawing coins without having contributed any to the smart contract. pp^h represents the contract's status at block height h. The withdrawal transaction, tx_{wdr}, includes the proof π_{wdr}, which proves to the smart contract the client's knowledge of a commitment, cm. Finally, tx_{wdr} is finally signed by sk'. Note that the client may compute the witness by taking the current block's accumulator checkpoint and take the cm-root from it, where cm is the commitement she had to provide by depositing.

Contract. When a withdraw transaction is received, the smart contract checks the proof and ensures that sn is not in the NullifierList. Further the given statement is being verified. The contract places sn in the NullifierList to prevent future double-withdrawal. Finally, amt is deposited to the user's address from the smart contract.

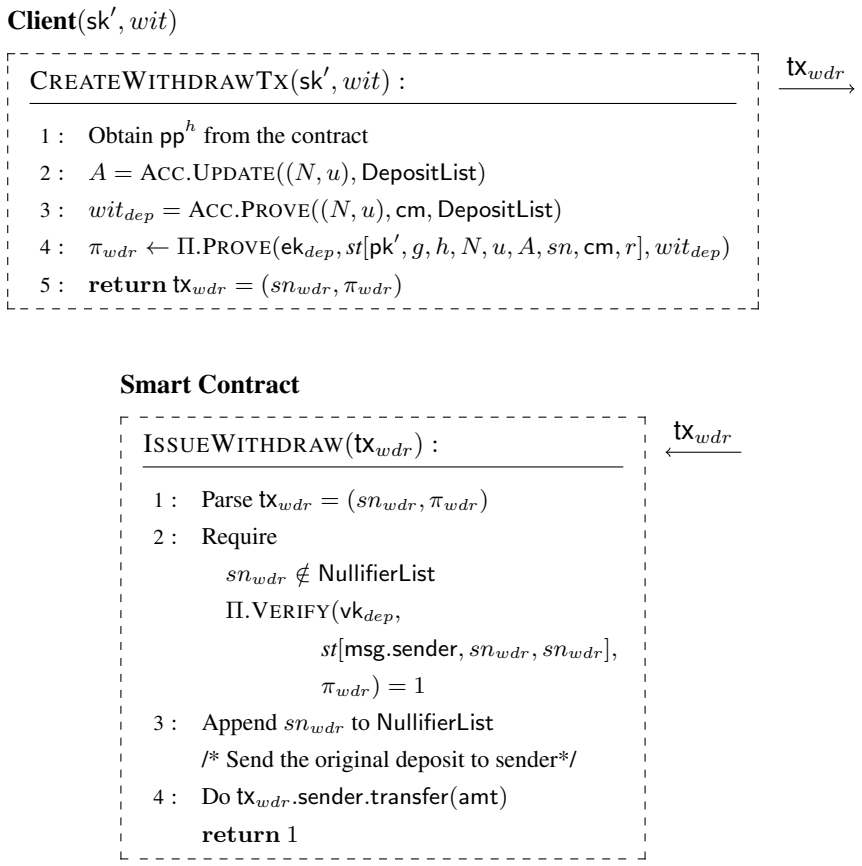

Figure 3.4. The system's deposit interactions between the client (CREATEWITHDRAWTX algorithm) and smart contract (ISSUEWITHDRAW algorithm).

3.3 Security Discussion

Privacy. Our system achieves privacy. Because we are using NIZK we ensure the unlinkability of addresses. Assuming the underlying crypography primitives are secure, an adversary can not link a deposit and a withdrawal transaction.

Correctness. Our system achieves correctness. There are two hypothetical scenarios in which an adversary might withdraw money without having previously deposited them: First, the attacker can either produce a new transaction for the current contract state or intercept a withdrawal transaction and substitute the recipient address with his own. The first result indicates that our primality test failed, while the second result indicates that the attacker violated the security of the $NIZK$ instance.

Availability. Because our smart contract runs autonomously on the blockchain, an attacker cannot prohibit clients from interacting with the blockchain, therefore our system satisfies availability.

3.4 Evaluation

3.4.1 Parameter

Choice of cryptographic primitives. We adapted the implementation of the primality test provided by riordant [4].

Hardware. We ran our experiment on a standard desktop system with an 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30G CPU and 16GB RAM.

3.4.2 Performance

Onchain Costs to deposit. Figure 3.5 illustrates the total deposit charges per person for various accumulator sizes. the cost is constant for each person. A deposit transaction costs $470k$ gas for a $1024bit$ accumulator, $762k$ gas for a $2048bit$ accumulator and $1230k$ gas for a $3072bit$ accumulator. In contrast to Tornado Cash, the price is based on the primality test and not the size of the Merkle tree. Although, for instance a $2048bit$ accumulator accumulates far more commitments than a Merkle tree with height 20, the cost is still less than with Tornado Cash. We used the Miller-Rabin primality test. This is relatively expensive and accounts for more than a third of the cost of depositing.

Onchain Costs to withdraw. Since the implementation of the withdrawing method and the proof is beyond the scope of this thesis, we refer to the GitHub post [2]. According to the developers of zeth [7], withdrawing would cost about $100m$ gas, since computing large exponents is computationally very expensive. However, Fiore [16] has shown in his paper how the costs could be reduced.

Onchain Costs to deploy. The cost of deploying the contract is the most expensive operation. The smallest accumulator costs around $2.1m$ gas, the $2048bit$ big accumulator $2.2m$ gas and the largest accumulator $2.4m$ gas. Figure 3.6 gives a visual representation of the deploying costs. However, we note that the deployment cost is a one-time cost which is amortized over the lifetime of the contract.

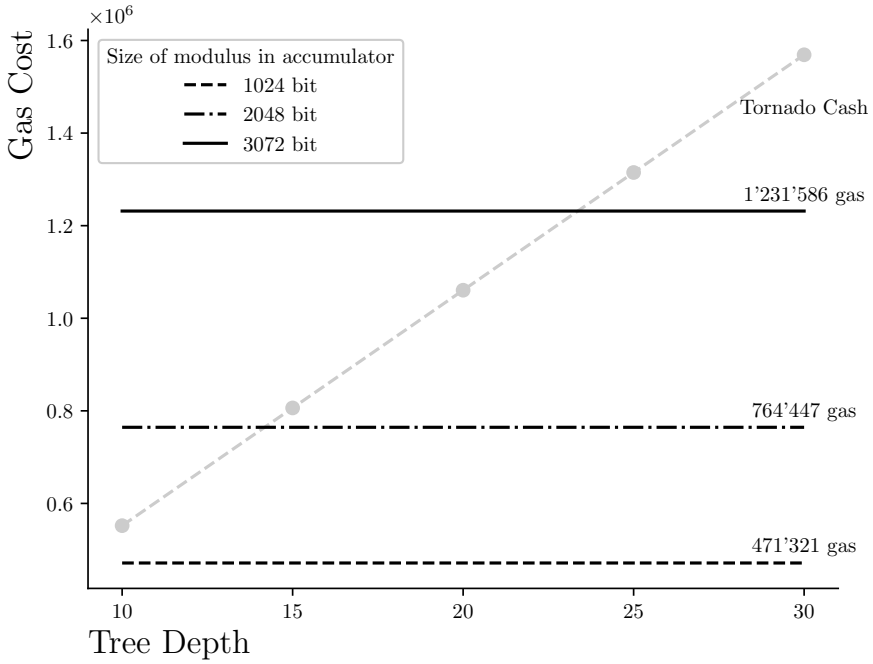

Figure 3.5. On-chain costs of deposits for different accumulator sizes in comparison with Tornado Cash. Each black line represents the indicated accumulator size. The grey line represents Tornado Cash with different Merkle tree heights.

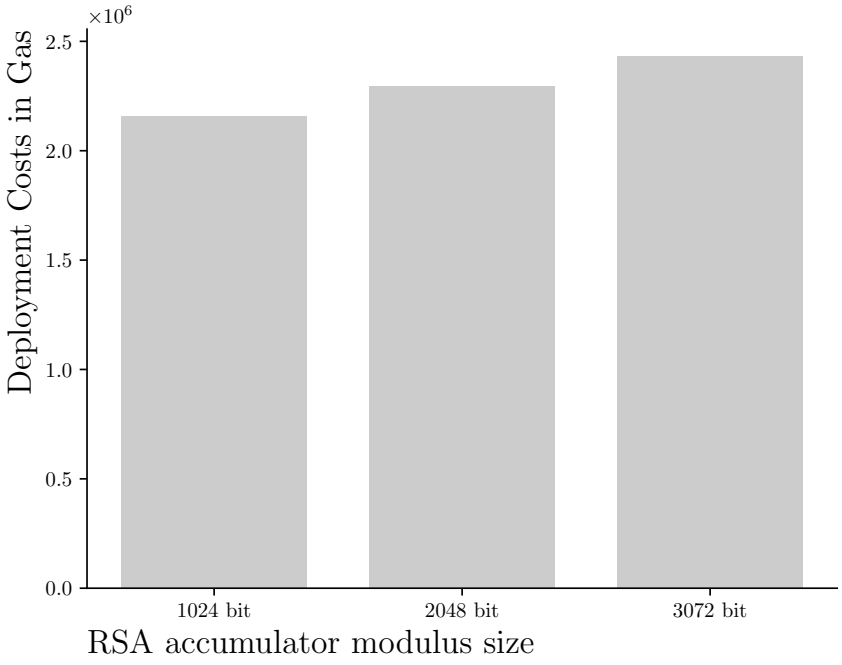

Figure 3.6. On-chain costs of deploying the different size accumulators.

Chapter 4

Improved Mixer: Merkle Pyramid Builder

4.1 System Overview

Traditional Ethereum mixers, such as Tornado Cash [5], have a significant drawback, namely that the cost to deposit funds is around $1.1m$ gas due to the need to update the Merkle tree after each deposit transaction. The need to update each time is redundant, as it is recommended to wait some time for other clients to deposit before withdrawing. Influenced by this fact, by the work of Reyzin and Yakoubov [37] and the practitioner-friendly version of this method, Merkle tree mountain range (MMR) [40] by Todd, we developed our method Merkle Pyramid Builder. The goal is to amotize the deposit cost like in the related method of Szydlo [39]. Note that the most relevant code components of our suggested method are available in appendix A.2

4.2 Merkle Pyramid Builder

Merkle Pyramid Builder Method. The fundamental concept is not to continually update the Merkle tree, but rather to gather deposit transactions and then update them collectively. Assuming we define a deposit-queue of length l. This would result in a subtree with a height of $\log(l)$. Every even deposit in the sequence does not have to pay for anything. In contrast, each odd deposit must hash all the hashes up to the tree until there are no values remaining on the left side of the deposit in the same sub-tree. Every l's deposit has to compute all hashes up to and including the root. Figure 4.1 shows a visual example.

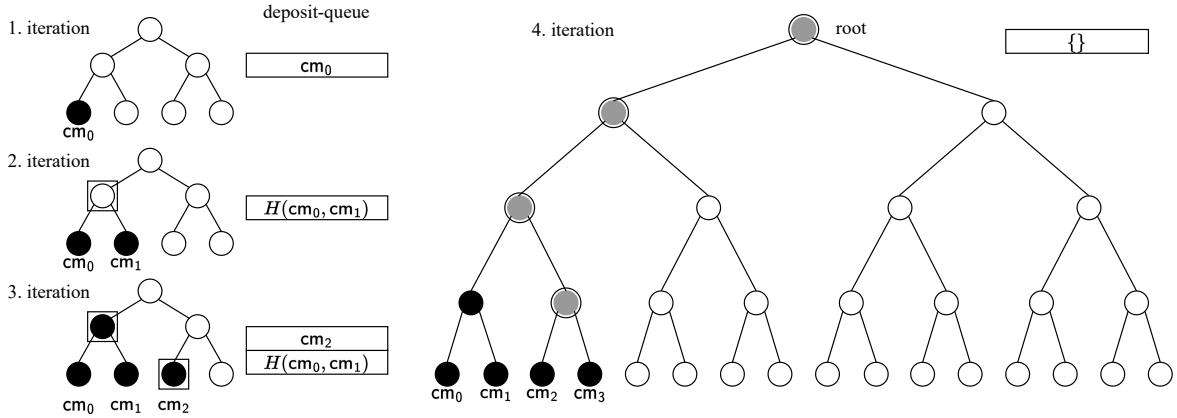

Figure 4.1. Graphical illustration of the MPB method with a deposit queue length of four. cm_i represents the deposit commitment. In the first iteration, a deposit is made. This is added to the deposit-queue. At the following deposit, clients must hash its deposit together with the previously stored deposit. This new value will be added to the queue, while the previous deposit will be removed. If a third deposit arrives, the client must do nothing further. The fourth must now compute all hashes up to and including the root using all deposit-queue values. The deposit-queue will then be deleted and the procedure will start again from the beginning at the next deposit.

4.3 Detailed Construction

4.3.1 Cryptographic Building Blocks

Hash Functions. $H_p : \{0,1\}^* \to \mathbb{F}$ is a preimage-resistant and collision-resistant hash function that maps a binary string to an element in $\mathbb{F}$, whereas $H_{2p} : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ is a collision-resistant hash function that maps two elements in $\mathbb{F}$ into one single element in $\mathbb{F}$. We refer to a hash function as "secure" if it is collision-resistant.

zk-SNARK. A succinct NIZK for arithmetic circuit satisfiability is considered as zero-knowledge succinct non-interactive argument of knowledge or zk-SNARK. For a given field $\mathbb{F}$, an arithmetic circuit C accepts items from $\mathbb{F}$ as inputs and returns elements from $\mathbb{F}$ as outputs. To define the arithmetic circuit satisfiability problem, we adapt a similar definition from Sasson et al.'s Zerocash paper [13].
For a field $\mathbb{F}$, an arithmetic circuit is defined as

$$C : \mathbb{F}^n \times \mathbb{F}^h \to \mathbb{F}^l$$

and the statisfiability problem is noted as

$$R_C = \{(st, \boxed{wit}) \in \mathbb{F}^n \times \mathbb{F}^h : C(st, \boxed{wit}) = 0^l\}$$

with the language

$$\mathcal{L}_R := \{st \mid \exists wit : (st, \boxed{wit}) \in R\}.$$

Definition 4.3.1 (zk-SNARK)**.** *zk-SNARK consists of three efficient algorithms* (SETUP, PROVE, VERIFY)

- $(\mathsf{ek}, \mathsf{vk}) \leftarrow \text{SETUP}(1^\lambda, C)$ *takes as inputs the security parameter and the circuit. As output, it returns a string containing the evaluation key* ek*, which is used later by the prover to generate the proof, and the verification key* vk*, which is used by the verifier to verify the proof.*
- $\pi \leftarrow \text{PROVE}(\mathsf{ek}, st, wit)$ *takes the evaluation key* ek *and* $(st, wit) \in R_C$ *as inputs and ouputs a proof* π *for the statement st.*

- $0/1 \leftarrow \text{VERIFY}(\mathsf{vk}, \pi, st)$ *takes as inputs the verification key* vk*, the proof* π*, and the statement st. It outputs* 1 *if the proof is valid for the statement.*

Withdrawal Proof. To withdraw coins from the smart contract a client has to satisfy three conditions:

1. The client has committed values for certain existing commitments that were utilized to construct the tree root using zk-SNARK.
2. The client did not withdraw previously, by passing a new nullifier value.
3. The secret key used to issue the withdrawal transaction is known by the client.

In other words, a client must give a proof showing the following relation for a Merkle tree T with a root root_{dep}:

$$\begin{aligned} R_{wdr} : \{&\mathsf{pk}, sn, \mathsf{root}_{dep}; \ \mathsf{sk}, k_{dep}, r, \mathsf{path}_i \ : \\ &\mathsf{pk} = \text{EXTRACTPK}(\mathsf{sk}) \wedge sn = H_p(k_{dep}) \wedge \\ &\mathsf{cm} = H_p(k_{dep}||r) \wedge T.\text{VERIFY}(i, \mathsf{cm}, \mathsf{root}_{dep}, \mathsf{path}_i))\} \end{aligned} \tag{4.1}$$

Where $\mathsf{pk}, sn, \mathsf{root}_{dep}$ are public values and $\mathsf{sk}, k_{dep}, r, \mathsf{path}_i$ are private values.

Merkle Tree. The Merkle tree is an example of authenticated set membership testing data structure. This is a complete binary tree. The leaves of the Merkle tree are initialized with zero values. The smart contract then preserves the Merkle tree T_{dep} over all commitments. When deposit transactions occur, the smart contract keeps track of the total number of deposit transactions and updates the tree using the ACCEPTDEPOSIT algorithm. We define the Merkle proof for commitment cm_i as path_{dep}. In addition, we define the root of the Merkle tree at block h as $\mathsf{root}_{wdr}^{\mathsf{curr}}$. We also denote root_{dep}.blockheight as the height of the blockchain block at the moment when root_{dep} is updated. Figure 4.2 presents a visual illustration.

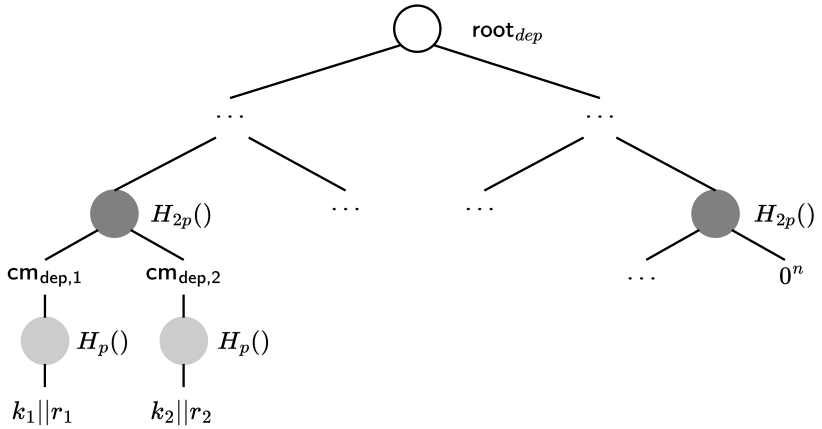

Figure 4.2. Illustrative example of the Merkle tree, T_{dep} . The tree keeps track of commitments from clients' deposit transactions. The root of the tree, root_{dep} is used to verify the NIZK proofs from withdrawing transactions.

Definition 4.3.2 (Merkle Tree)**.** *Merkle Hash tree consists of following algorithms:*

- $\mathsf{root}_{dep} \leftarrow \text{T.INIT}(1^\lambda, X)$ *takes as inputs the security parameter and a set* $X = \{0_1, \ldots, 0_n\}$*. The leaves are initialized with the set* X *and one initializes* $index = 1$ *to track the deposits. Further, the list* $\mathsf{RootList}_{wdr,k}$ *is initialized to be the list of the* k *most recent roots of* T*. The output is* root_{dep}*.*

- $\pi \leftarrow$ T.PROVE(i, x, X) *takes as input an element* x*, index* $1 \leq i \leq n$ *and a set* X *and outputs that* $x = x_i \in X$
- $0/1 \leftarrow$ T.VERIFY$(i, x, \mathsf{root}_{dep}, \pi)$ *takes an element* x*,* $1 \leq i \leq n$*,* $\mathsf{root}_{dep} \in \{0,1\}^\lambda$ *and a proof* π *as inputs. The output is 1 if* $x = x_i \in X$ *and* $\mathsf{root}_{dep} =$ T.INIT$(1^\lambda, X)$ *and 0 otherwise.*
- $\mathsf{root}_{new} \leftarrow$ T.UPDATE(Q) *takes a sub-tree* Q *containing all new hashes as input. The algorithm inserts the sub-tree into the existing Merkle tree and so the new root will be updated. The output is the new root* root_{new}

In order to deposit more cost-efficiently we define methods which allow us to batch deposit-transaction and therefore allow us to update the Merkle tree less frequently.

Definition 4.3.3 (Deposit-Queuing)**.** *The maintenance of the deposit-queue can be described with the following three algorithms:*

- $q \leftarrow$ CREATEQUEUE(l) *takes as input an integer* l*, which will determine how many deposit transactions we are batching. The output is an empty queue* q *with a fixed size* l*.*
- $q' \leftarrow$ UPDATEQUEUE(q, cm) *takes as input a list* q *containing all deposit transactions that have not yet been stored in the Merkle tree and a new commitment* cm*. The algorithm stores this value in the queue as long as it does not have enough elements to hash together. The output is updated queue which contains all already hashed commitments.*
- $q'' \leftarrow$ CLEARQUEUE(q) *takes as input a list containing all deposit transactions which are already stored in the Merkle tree. The algorithm then deletes all entries and returns an empty list.*

4.3.2 Contract Setup

Let $\mathbb{F}$ represent the finite field we want to use in the contract. In the setup phase the algorithm CONTRACTSETUP samples two secure hash functions H_p and H_{2p} from the collision-resistant hash families. The procedure further initializes amt for the fixed amount of coins that can be deposited.

CONTRACTSETUP(1^λ)

Sample $H_p : \{0,1\}^* \to \mathbb{F}$ and $H_{2p} : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$
Choose $\mathsf{amt} \in \mathbb{Z}_{>0}$ to be a fixed deposit amount
Choose $d \in \mathbb{Z}_{>0}$, Let $X = \{x_1, \ldots, x_{2^d}\}$
    where $x_i = 0^\lambda$ for all $x_i \in X$
Initialize an empty tree $\mathsf{root}_{dep} =$ T.INIT$(1^\lambda, X)$,
Choose $k \in \mathbb{Z}_{>0}$, set $\mathsf{RootList}_{wdr,k}[i] = \mathsf{root}_{dep}$,
    for $1 \leq i \leq k$
Construct C_{wdr} for statement described in equation 4.1
Let Π be the zk-SNARK instance.
    – Run $(\mathsf{ek}_{dep}, \mathsf{vk}_{dep}) \leftarrow \Pi$.SETUP$(1^\lambda, C_{wdr})$
Initialize: $\mathsf{DepositList} = \{\}, \mathsf{NullifierList} = \{\}$
Initialize: $\mathsf{DepositQueue} \leftarrow$ CREATEQUEUE(l)
Deploy smart contract with parameters :
$\mathsf{pp} = (\mathbb{F}, H_p, H_{2p}, \mathsf{amt}$
    $T, \mathsf{index}, \mathsf{RootList}_{wdr,k}, \mathsf{DespositQueue}$
    $(\mathsf{ek}_{dep}, \mathsf{vk}_{dep}), \mathsf{DepositList}, \mathsf{NullifierList})$

Figure 4.3. Pseudocode for the setup of the smart contract. pp can be queried by any client.

Setup Merkle Tree. Let's define T as the Merkle tree with depth d. The algorithm T.INIT initializes T as described.

Setup zk-SNARK parameters. We initialize a zk-SNARK-instance Π with the algorithm Π.SETUP with C_{wdr} as input. We obtain two keys $(\mathsf{ek}_{dep}, \mathsf{vk}_{dep})$.

Setup commitments and nullifier lists. The contract initializes two empty lists. DepositList contains all cm included in depositing transactions. NullifierList contains all unique identifiers sn, which commited in a withdrawing transaction.

Setup deposit-queue. The DepositQueue is initialized to keep track of how many deposit transactions are in the queue and how many of them were hashed to know if the next user has to pay for updating the Merkle tree. We define the size l of the queue to determine how big the subtree should be.

4.3.3 Deposit Interaction

Client. A client can deposit coins into the smart contract using CREATEDEPOSITTX. First, the client must randomly select k and r. Those are then hashed together and are used to construct the commitment. The transaction is currently comprised of the commitment and the tokens the client wishes to transmit. Additionally, the transaction must be signed with sk to prevent an attacker from just altering the transaction's recipient. In addition, a witness is issued which will be used in the future to withdraw the coins again.

Contract. At the beginning the smart contract verifies that the amount of coins is as requested and that there is still free space in the Merkle tree. If the conditions are met the commitment is added in the DepositList. With UPDATEQUEUE the DespositQueue will be updated. If this deposit-queue is full, the Merkle tree will be updated using the algorithm T.UPDATE. Furthermore the smart contract will empty the deposit-queue and finally add the new root to the $\mathsf{RootList}_{wdr,k}$

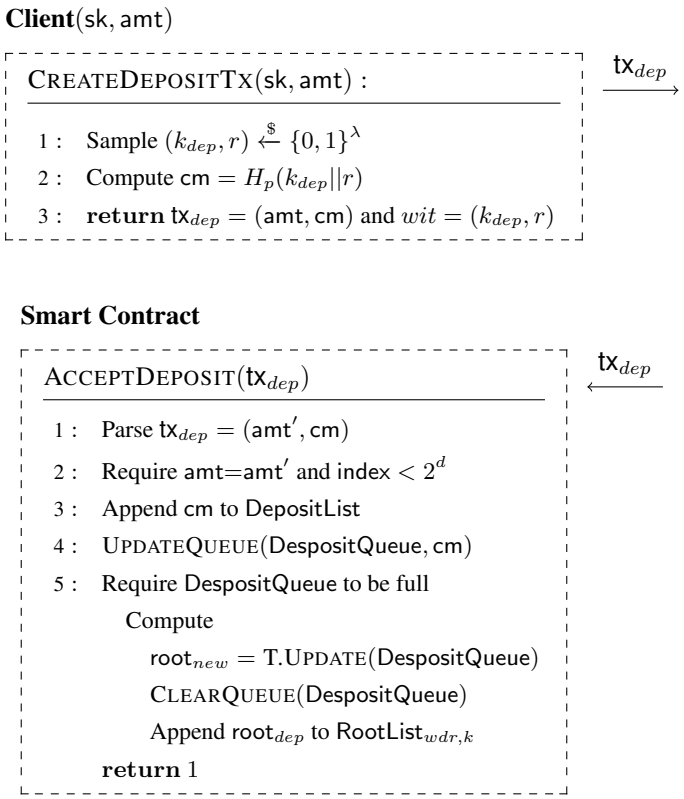

Figure 4.4. Deposit interactions between the client (CREATEDEPOSITTX algorithm) and smart contract (ACCEPTDEPOSIT algorithm). Transaction tx_{dep} is signed by sk.

4.3.4 Withdraw Interaction

Client. A client has to create a proof π_{wdr} with the secret note, wit, and the secrete key sk', to withdraw amt to the public key pk'. The client has to prove to the contract the three points we mentioned in 4.3.1.

Contract. The contract checks the proof, π_{wdr}, and confirms that the nullifier sn_{wdr} is not in the NullifierList when it receives a withdrawal transaction. The contract then appends sn_{wdr} to NullifierList to avoid future double withdrawals. Finally, the smart contract deposits amt to the address specified by the user.

Client($\mathsf{sk}, \mathsf{amt}$)

CREATEWITHDRAWTX($\mathsf{sk}, \mathsf{amt}$) : $\xrightarrow{\mathsf{tx}_{wdr}}$

1 : Parse $wit = (k_{dep}, r)$
2 : Obtain pp^h from the contract
3 : Compute $sn_{wdr} = H_p(k_{dep})$
4 : Compute $\mathsf{cm} = H_p(k_{dep}||r)$
5 : Get index i of cm from $\mathsf{DepositList}^h$
6 : Choose $\mathsf{root}_{dep} \in \mathsf{RootList}_{wdr,k}$
7 : Compute $\mathsf{path}^h_{dep,i}$ such that
- T.VERIFY$(i, \mathsf{cm}, \mathsf{root}_{dep}, \mathsf{path}^h_{dep,i}) = 1$
8 : Form: $wit_{dep} = (\mathsf{sk}', k_{dep}, r, \mathsf{path}^h_{dep,i})$
9 : $\pi_{wdr} \leftarrow \Pi$.PROVE$(\mathsf{ek}_{dep}, st[\mathsf{pk}', sn_{wdr}, \mathsf{root}_{dep}], wit_{dep})$
10 : **return** $\mathsf{tx}_{wdr} = (sn_{wdr}, \mathsf{root}_{dep}, \pi_{wdr})$

Smart Contract

ISSUEWITHDRAW(tx_{wdr}) : $\xleftarrow{\mathsf{tx}_{wdr}}$

1 : Parse $\mathsf{tx}_{wdr} = (sn_{wdr}, \mathsf{root}_{dep}, \pi_{wdr})$
2 : Require
- $\mathsf{root}_{dep} \in \mathsf{RootList}_{wdr,k}$
- $sn_{wdr} \notin \mathsf{NullifierList}$
- Π.VERIFY$(\mathsf{vk}_{dep}, \pi_{wdr},$
$st[\mathsf{msg.sender}, sn_{wdr}, sn_{wdr}, \mathsf{root}_{dep}]) = 1$
3 : Append sn_{wdr} to NullifierList
/* Send the original deposit to sender*/
4 : Do $\mathtt{tx}_{wdr}$.sender.transfer(amt)
return 1

Figure 4.5. Withdraw interactions between the client (CREATEWITHDRAWTX algorithm) and smart contract (ISSUEWITHDRAW algorithm). The state of the contract at block height h is denoted by pp^h. The withdraw transaction tx_{wdr} contains the proof π_{wdr} that proves the client's knowledge of $\mathsf{cm} = H_p(k_{dep}, r)$ which is a valid member of the Merkle tree with the root root_{wdr}. sn_{wdr} is used to nullify the old commitment, cm. tx_{wdr} is signed by sk'.

4.4 Security Discussion

Privacy. Our system achieves privacy. Because we are using zk-SNARK we ensure the unlinkability of addresses. Assuming the underlying crypography primitives are secure, an adversary can not link a deposit and a withdrawal transaction.

Correctness. MPB satisfies correctness. There are two conceivable outcomes if an attacker can deliver a withdrawal transaction that confirms without depositing any coins into the system. First, the adversary can generate a new legitimate transaction for the present state of the contract (i.e. by watching the commitment list), or it can intercept a withdrawal transaction and change the recipient address with its own. In the first scenario, this indicates that the adversary breaches the preimage-resistant security of the underlying hash function H_p, whereas in the second situation, the opponent breaches the security of the zk-SNARK instance.

Availability. MPB does not fully satisfy availability, since clients cannot withdraw until the final client in a sub-tree has updated the root. Nonetheless, we do not consider this a drawback. As previously stated, anonymity rises the longer funds are in the mixer. By imposing a mandatory delay, we provide more anonymity.

4.5 Evaluation

Choice of cryptographic primitives. We select Groth's zk-SNARK [27] as our instance of zk-SNARK owing to its efficiency in terms of the size of proofs and the calculations required by the verifier. We employ the Pedersen [1] hash function [35] for H_p and the MiMC hash function [9], for H_{2p} for cryptographic hash functions. Compared to arithmetic circuits that rely on other hash functions like Jubjub [3], arithmetic circuits that employ MiMC hash produce a smaller number of constraints and operations. In addition to being created exclusively for SNARK applications, MiMC hash functions are also very gas-efficient for Ethereum smart contract applications.

Software. For the arithmetic circuit design, the Circom library [11] is used to build the withdrawal circuit, C_{wdr}, for the relation specified in Equation 4.1. We utilize Groth's zk-SNARK proof system implemented by the snarkjs package [12] to construct the client's algorithms and to establish the trusted environment for getting the proving and evaluation keys for the MPB contract and clients. We deployed the MPB system on the EVM ganache [1].

Hardware. We ran our experiment on a standard desktop system with an 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30G CPU and 16GB RAM

4.5.1 Performance

We measured the performance and the cost of the MPB system using the tree depths $d = 10, 15, 20, 25, 30$ and the following deposit-queue length $l = 2, 4, 8, 16, 32, 64, 128$.

Onchain Cost to Deposit. Figure 4.6 illustrates how expanding the size of the deposit queue reduces expenses. We may also observe that by reducing or raising the depth of the Merkle tree, the cost falls or increases, accordingly. Note that reducing the depth of the Merkle tree reduces the number of users, while raising the depth significantly increases the time required to compute the withdrawal proof. We can also observe that for a deposit-queue of length 128, the gas costs are around $135k$, which is quite consistent amongst the various Merkle tree depths.

[1] Pedersen hash function is a secure hash function which maps a sequence of bits to a compressed point on an elliptic curve

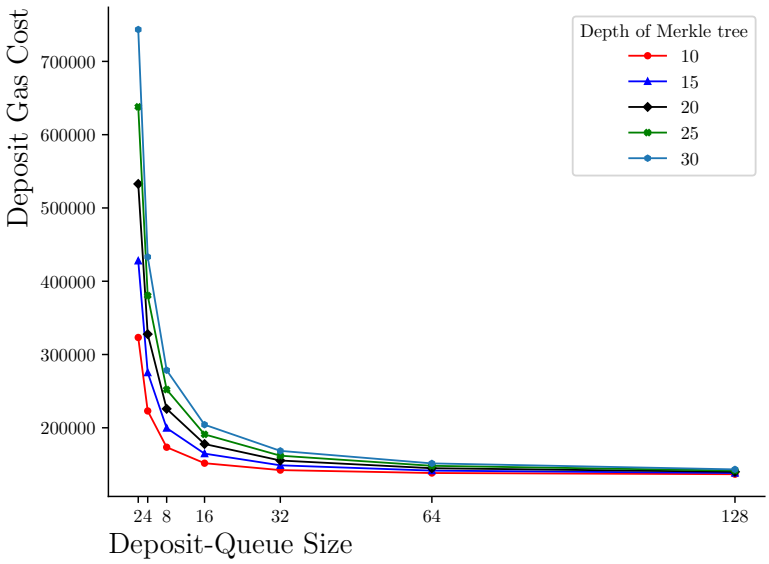

Figure 4.6. Average on-chain costs of deposits per client for different deposit-queue size and different Merkle tree depth.

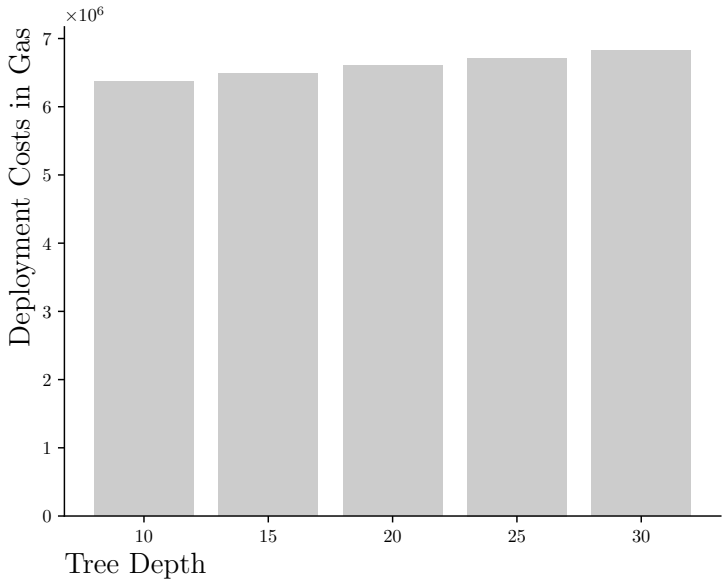

Figure 4.7. Cost to deploy the MPB smart contract with different Merkle tree sizes.

Onchain Cost to Deploy. Figure 4.7 shows the expense associated with deploying smart contracts. All expenditures exceed $6m$ gas, however the deployment cost is a one-time expense that is amortized over the duration of the contract.

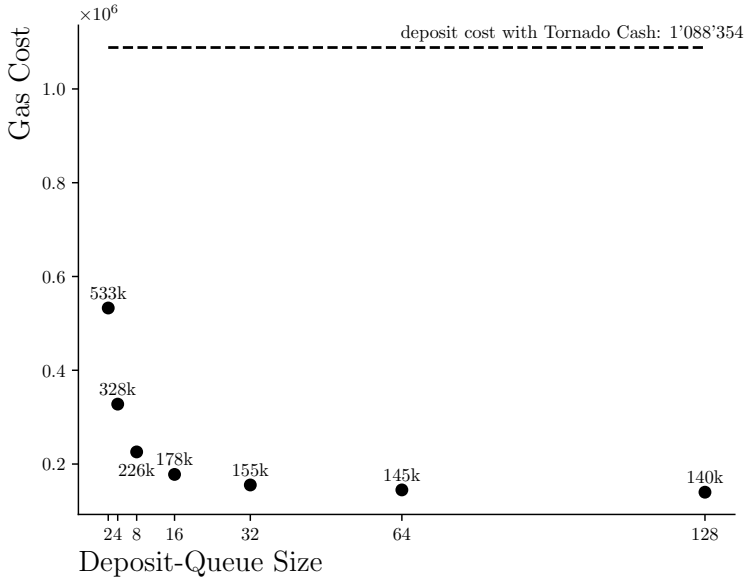

Figure 4.8. Average on-chain costs of deposits per client for different deposit-queue lengths in comparison with Tornado Cash. The dashed line represents the cost of depositing in Tornado Cash. It is about $1.1m$ gas.

Onchain Cost. Figure 4.8 illustrates the total deposit charges per person for various deposit queue-sizes. The costs reduce in inverse proportion to the length of the deposit queue. However, the cost will never go below $1.1k$ gas. Nevertheless, this is just a $\frac{1}{7}$ of the cost of depositing in Tornado Cash. Note that the costs associated with deposit-queue size one would roughly correspond to those of Tornado Cash. The gas cost for a withdrawing transaction is approximately $350k$, as the MPB contract needs to verify the zk-SNARK proof.

4.6 Reduce Deposit Cost via Verifiable Computation

To further minimize costs, one can use verifiable computation techniques [26]. The primary concept is to do the calculation off-chain and utilize the contract to validate its correctness. The client evaluates the root of the Merkle tree and delivers the result along with a proof that the calculation was performed correctly. However, since the implementation would go beyond this bachelor thesis, we will only introduce the theory. The technique of using verifiable computation to reduce on-chain cost was first introduced in Hawk [29].

4.6.1 Verifiable Computation

Verifiable Computation Scheme. In a verifiable computation scheme, the client selects a function and an input to send to the server. The server must evaluate the function on the input and return the result, along with proof that the result is valid. The client then verifies that the output given by the user server is, in fact, the result of the function computed on the specified input. The goal is to make such a verification very efficient respectively to make this verification faster than the computation of the function itself.

Definition 4.6.1 (Verifiable Computation Scheme)**.** *Let F be a function, expressed as an arithmetic circuit over a finite field $\mathbb{F}$ and λ be a security parameter.*

- $(\mathsf{ek}, \mathsf{vk}) \leftarrow \textsc{VcInit}(1^\lambda, F)$ *takes a security parameter and an arithmetic circuit as an input and generates two public keys: an evaluation key* ek *and a verification key* vk*.*

- $(y, \pi) \leftarrow \text{VCPROVE}(\mathsf{ek}, x)$ *takes as input an element* x *and the evaluation key* ek *and computes* $y = F(x)$ *and a proof* π *that* y *has been correctly computed.*
- $0/1 \leftarrow \text{VCVERIFY}(\mathsf{vk}, x, y, \pi)$ *takes the verification key* vk*, the input/output* (x, y) *of the computation* F *and the proof* π *and outputs* 1 *if* $y = F(x)$ *and* 0 *otherwise.*

A zk-SNARK instance can be used as the verifiable computation scheme. Groth16 can be used for this purpose.

Deposit Proof. To deposit coins to the smart contract a client has to give a proof showing the following relation for a Merkle tree T with a root, root_{dep}:

$$R_{dep} : \{\mathsf{pk}, \mathsf{cm}, \mathsf{root}_{dep}^{old}, \mathsf{root}_{dep}^{new}, \mathsf{path}_i : T.\text{VCVERIFY}(\mathsf{vk}, \mathsf{root}_{dep}^{old}, \mathsf{root}_{dep}^{new}, \mathsf{path}_i)\} \tag{4.2}$$

4.6.2 Contract Setup

The CONTRACTSETUP algorithm differs marginally from the MPB method in section 4.3.2. The only difference is that an additional instance must be initialized for the verifiable computation.

CONTRACTSETUP(1^λ)

1 : Sample $H_p : \{0,1\}^* \to \mathbb{F}$ and $H_{2p} : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$
2 : Choose $\mathsf{amt} \in \mathbb{Z}_{>0}$ to be a fixed deposit amount
3 : Choose $d \in \mathbb{Z}_{>0}$, Let $X = \{x_1, \ldots, x_{2^d}\}$
4 : where $x_i = 0^\lambda$ for all $x_i \in X$
5 : Initialize an empty tree $\mathsf{root}_{dep} = \text{T.INIT}(1^\lambda, X)$,
6 : Choose $k \in \mathbb{Z}_{>0}$, set $\mathsf{RootList}_{wdr,k}[i] = \mathsf{root}_{dep}$,
for $1 \leq i \leq k$
7 : Construct C_{dep} for statement described in 4.2
8 : Construct C_{wdr} for statement described in 4.1
9 : Let Π_{dep} be the Verifiable Computation instance.
– Run $(\mathsf{ek}_{dep}^{vc}, \mathsf{vk}_{dep}^{vc}) \leftarrow \Pi_{dep}.\text{VCINIT}(1^\lambda, C_{dep})$
10 : Let Π_{wdr} be the zk-SNARK instance.
– Run $(\mathsf{ek}_{dep}, \mathsf{vk}_{dep}) \leftarrow \Pi_{wdr}.\text{SETUP}(1^\lambda, C_{wdr})$
11 : Initialize: $\mathsf{DepositList} = \{\}, \mathsf{NullifierList} = \{\}$
12 : Initialize: $\mathsf{DepositQueue} \leftarrow \text{CREATEQUEUE}(l)$
13 : Deploy smart contract with parameters :
$\mathsf{pp} = (\mathbb{F}, H_p, H_{2p}, \mathsf{amt}$
$T, \mathsf{index}, \mathsf{RootList}_{wdr,k}, \mathsf{DespositQueue}$
$(\mathsf{ek}_{dep}, \mathsf{vk}_{dep}), (\mathsf{ek}_{dep}^{vc}, \mathsf{vk}_{dep}^{vc}), \mathsf{DepositList}, \mathsf{NullifierList})$

Figure 4.9. Pseudocode for the setup of the smart contract. pp can be queried by any client.

Setup Verifiable Computation Proof. We initialize a verifiable computation instance Π_{dep} with the algorithm $\Pi_{dep}.\text{VCINIT}$ with C_{dep} as input. We obtain two keys $(\mathsf{ek}_{dep}^{vc}, \mathsf{vk}_{dep}^{vc})$

4.6.3 Deposit Transaction

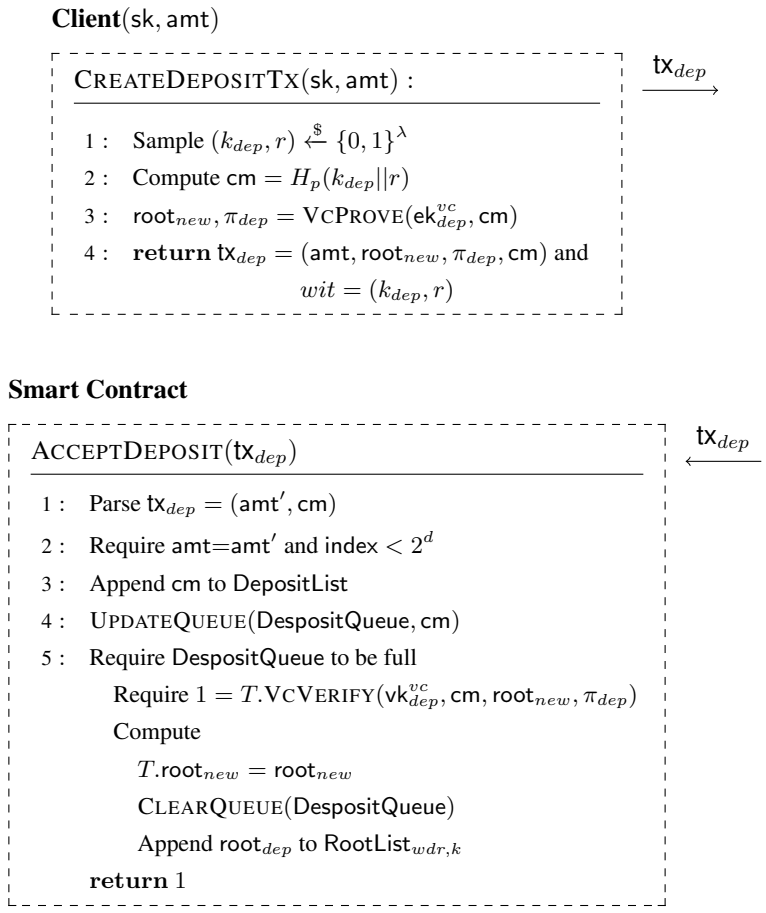

Figure 4.10. Deposit interactions between the client (CreateDepositTx algorithm) and smart contract (AcceptDeposit algorithm). Transaction tx_{dep} is signed by sk.

Client. In comparison to section 4.3.3, the client must calculate and prove the proper calculation of the new root in addition to the cm and amt. The rest remains unchanged.

Contract. In contrast to section 4.3.3, the smart contract must now do much fewer calculations. When the DespositQueue is full, the client's proof for the validity of the new root is checked. If this is the case, the root of the contract is modified to match the client's root. The rest remains the same as before.

4.6.4 Withdraw Interaction

Since verifiable computation only alters the method we deposit, the way we withdraw coins stays the same as in section 4.3.4.

Chapter 5

Conclusion

The lack of privacy features in open and permissionless blockchains can be mitigated to a degree via coin mixers. Their operations are expensive due to both transaction fees and the fact that "greater" privacy is more expensive than "weaker" privacy when privacy quality is measured quantitatively with the anonymity set size. In this thesis, we introduce two coin mixers.

The first mixer is based on a RSA accumulator, which reduces the cost by only needing a constant number of operations. Despite of the high cost of the primality test we were able to reduce the deposit cost. We implemented a simple RSA accumulator on-chain mixer. The deposit costs were measured on the EVM ganache and we could see how different modulus sizes affect the cost of depositing in comparison with Tornado Cash. For example by choosing a modulus of a $2048bit$ long number we could safe about $300k$ gas in comparison with the standard Tornado Cash mixer.

The second on-chain mixer, the MPB-mixer is a more cost efficient version of the Tornado Cash mixer. We introduced MPB and also implemented it in Solidity. We saw that by increasing the size of the deposit-queue the average deposit-cost per person would decrease until approximately $110k$ gas is reached. Furthermore there are not any addition costs for a withdrawal transaction, which costs about $350k$ gas. We observed that regardless of the Merkle tree depth, if we select a big deposit-queue, the gas cost will be around $135k$. In addition, we proposed a second method for reducing depositing expenses. Using a verified computation approach would reduce the on-chain cost associated with computing hash functions to update the Merkle root.

Both our mixers should attract privacy-seeking users, who are searching for a more cost efficient version of Tornado Cash. Therefore, we hope that such methods significantly broaden the number of users of the mixers and thereby enhancing the quality of the anonymity set for all users engaged. However, the recent U.S. legislation that made Tornado Cash illegal may reduce interest in these methods. Our implementations and assessments demonstrate the practicability of our mixers by enabling anonymity sets with more than thousands of members.

Our system uses the MiMC hash functions. MiMC might be replaced in the future by the Poseidon hash function, which is not only cheaper but also faster. Further, our last presented method in Section 4.6 can be implemented and evaluated for additional cost reductions.

Appendix A

Extra material

A.1 Contruction of a NIZK proof

In the following section we will provide you a detailed description of the NIZK proof used in section 3.1.

Consider that the commitment scheme's parameters are a group $\mathbb{G}_q$ and two generators g_1 and h_1. To commit to a value x, one choose a random $r \in \mathbb{Z}_q$ and outputs $\text{COMMIT}(msg, r) = g_1^x \cdot h_1^r$. k' and k'' are security parameters. Under the discrete-logarithm assumption, this information-theoretically hiding commitment scheme is binding. Furthermore we require that g_2 and $h_2 \in QR_N$ be available such that $\log_{g_2} h$ is unknown to the prover, where N is the value of the modular of the accumulator. Let e be the value we want to hide and let u be a value such that $u^e = A \mod N$. Let also $H : \{0,1\}^* \to \{0,1\}^k$ be a cryptographic hash function.

1. The prover chooses

$$\begin{gathered} r_{x_1} \in_R [-B2^{k'+k''+2}, B2^{k'+k''+2}] \\ r_{x_3}, r_{x_7}, r_{x_8}, r_{x_{10}}, r_{x_{11}} \in_R \mathbb{Z}_1 \\ r_{x_5}, r_{x_9}, r_{x_6} \in_R (-\lfloor \tfrac{N}{4} \rfloor \cdot 2^{k'+k''}, \ldots, \lfloor \tfrac{N}{4} \rfloor \cdot 2^{k'+k''}) \\ r_{x_2}, r_{x_4} \in_R (-\lfloor \tfrac{N}{4} \rfloor \cdot q \cdot 2^{k'+k''}, \ldots, \lfloor \tfrac{N}{4} \rfloor \cdot q \cdot 2^{k'+k''}) \end{gathered}$$

and computes

$$\begin{array}{llll} \mathbb{C} = g_1^e \cdot h_1^r & C_e = g_2^{r_1} \cdot h_2^e & C_u = u \cdot h_2^{r_2} & C_r = g_2^{r_2} \cdot h_2^{r_3} \\ t_1 = g_1^{x_1} \cdot h_1^{x_7} & t_2 = (\dfrac{\mathbb{C}}{g_1})^{r_{x_3}} \cdot h_1^{r_{x_8}} & t_3 = (g_1 \cdot \mathbb{C})^{r_{x_{10}}} h_1^{r_{x_{11}}} & t_1' = h_2^{x_5} \cdot g_2^{r_{x_6}} \\ t_2' = h_2^{r_{x_1}} \cdot g_2^{r_{x_9}} & t_3' = C_u^{r_{x_1}} \cdot (\dfrac{1}{h_2})^{r_{x_2}} & t_4' = C_r^{r_{x_1}} \cdot (\dfrac{1}{h_2})^{r_{x_4}} \cdot (\dfrac{1}{g_2})^{r_{x_2}} & \end{array}$$

$$\begin{array}{ll} s_{x_1} = r_{x_1} - c \cdot e & s_{x_7} = r_{x_7} - c \cdot r \mod q \\ s_{x_2} = r_{x_2} - c \cdot r_2 \cdot e & s_{x_8} = r_{x_8} - c \cdot r \cdot (e-1)^{-1} \mod q \\ s_{x_3} = r_{x_3} + c \cdot (e-1)^{-1} \mod q & s_{x_9} = r_{x_9} - c \cdot r_1 \\ s_{x_4} = r_{x_4} - c \cdot r_3 \cdot e & s_{x_{10}} = r_{x_{10}} - c \cdot (e+1)^{-1} \mod q \\ s_{x_5} = r_{x_5} - c \cdot r_3 & s_{x_{11}} = r_{x_{11}} + c \cdot r \cdot (e+1)^{-1} \mod q \\ s_{x_6} = r_{x_6} - c \cdot r_2 & \end{array}$$

and also $c = H(e, t_1, \ldots, t_3, t_1', \ldots, t_4')$
The signature of knowledge on e is $(c, g_1, h_1, g_2, h_2, \mathbb{C}, C_e, C_u, C_r, r_{x_1}, r_{x_2}, r_{x_4}, s_{x_1}, \ldots, s_{x_{11}})$

2. The verifier computes

$$\bar{t_1} = \mathbb{C} \cdot g_1^{s_{x_1}} \cdot h_1^{s_{x_7}} \qquad \bar{t_3'} = A^c \cdot C_u^{s_{x_1}} \cdot (\frac{1}{h_2})^{s_{x_2}}$$
$$\bar{t_2} = g_1^c (\frac{\mathbb{C}}{g_1})^{s_{x_3}} \cdot h_1^{s_{x_8}} \qquad \bar{t_4'} = C_r^{r_{x_1}} \cdot (\frac{1}{h_2})^{r_{x_4}} \cdot (\frac{1}{g_2})^{r_{x_2}}$$
$$\bar{t_3} = g_1^c \cdot (g_1 \cdot \mathbb{C})^{s_{x_{10}}} h_1^{s_{x_{11}}} \qquad \bar{s_{x_1}} \in [-B2^{k'+k''+2}, B2^{k'+k''+2}]$$
$$\bar{t_1'} = C_r^c \cdot h_2^{s_{x_5}} \cdot g_2^{s_{x_6}}$$

$c' = H(e, \bar{t_1}, \ldots, \bar{t_3}, \bar{t_1'}, \ldots, \bar{t_4'})$
and verify $c \stackrel{!}{=} c'$

We will define the first part of the proof as

$$\pi_1 = (c, g_1, h_1, g_2, h_2, \mathbb{C}, C_e, C_u, C_r, r_{x_1}, r_{x_2}, r_{x_4}, s_{x_1}, \ldots, s_{x_{11}})$$

The part (2) is a double discrete log signature of knowledge, that is described in the paper of Zerocoin [36]. It is constructed as follows:
Let's denote $y_1 = g^{a^x \cdot b^z} \cdot h^w$ and let $l \leq k$ two security parameter. Let $H : \{0,1\}^* \to \{0,1\}^k$ be a cryptographic hash function.

1. The prover chooses $2l$ numbers $r_1, \ldots, r_l, v_1', \ldots, v_l'$ and computes
   - $t_i = g^{a^x \cdot b^{r_i}} \cdot h^{v_i}$, for $1 \leq i \leq l$
   - $s_i = \begin{cases} r_i & \text{if } c[i] = 0 \\ r_i - z & \text{otherwise.} \end{cases}$
   - $s_i' = \begin{cases} v_i & \text{if } c[i] = 0 \\ v_i - w \cdot b^{r_i - z} & \text{otherwise.} \end{cases}$
   - $c = H(e||y_1||a||b||g||h||x||t_1|| \ldots ||t_l)$

   The signature of knowledge on e is $(c, s_1, \ldots, s_l, s_1', \ldots, s_l')$

2. the verifier computes
   - $t_i' = \begin{cases} g^{a^x \cdot b^{s_i}} \cdot h^{s_i'} & \text{if } c[i] = 0 \\ y_1^{b^{s_i}} \cdot h^{s_i'} & \text{otherwise.} \end{cases}$
   - $c' = H(e||y_1||a||b||g||h||x||t_1'|| \ldots ||t_l')$

   and verify $c \stackrel{!}{=} c'$

We will define the second part of the proof as

$$\pi_2 = (c, s_1, \ldots, s_l, s_1', \ldots, s_l')$$

Finally, we need to combine the two parts of the proof to get our whole proof:

$$\pi = (\pi_1, \pi_2)$$

A.2 Code Snippets

```
function updateDepositStack(StackEntityStruct.StackEntity memory _newEntity) public {
    // if the stack is empty just save the new entity
    if (depositStack.isEmpty()) {
      depositStack.push(_newEntity);
    } else {
      while(true) {

        // last entity of the stack
        StackEntityStruct.StackEntity memory lastEntity = depositStack.peek();

        // last entity of the stack and the new entity should have the same hash level to calculate the new value
        if (lastEntity.hashLevel == _newEntity.hashLevel) {

          // hash last element of queue with newest element and increment hashlevel
          bytes32 newValue = hashLeftRight(hasher, lastEntity.value, _newEntity.value);
          uint256 newHashLevel = _newEntity.hashLevel + 1;
          _newEntity = StackEntityStruct.StackEntity(newValue, newHashLevel);

          depositStack.pop();

          if (depositStack.isEmpty() || depositStack.peek().hashLevel != _newEntity.hashLevel) {
            depositStack.push(_newEntity);
            break;
          }
        } else {
          depositStack.push(_newEntity);
          break;
        }
      }
    }
}

function mpbDeposit(bytes32 _commitment) external payable nonReentrant {
  require(!commitments[_commitment], "The commitment has been submitted");
  StackEntityStruct.StackEntity memory newEntity = StackEntityStruct.StackEntity(_commitment,0);

  if(currentIndex % 2**subTreeLevel == 2**subTreeLevel-1){
    updateDepositStack(newEntity);
    mpbInsert(depositStack.pop());
    require(depositStack.isEmpty(), "not empty stack");

  } else {
    updateDepositStack(newEntity);
  }
  commitments[_commitment] = true;
  _processDeposit();

  emit Deposit(_commitment, currentIndex, block.timestamp);
  currentIndex++;
}
```

```
function mpbInsert(StackEntityStruct.StackEntity memory _newEntity) public {
    uint32 _nextIndex = nextIndex;
    require(_nextIndex != uint32(2)**levels, "Merkle tree is full. No more leaves
        can be added");
    bytes32 currentLevelHash;
    bytes32 left;
    bytes32 right;

    uint _pyramidNextIndex = pyramidNextIndex;
    uint pyramidCurrentIndex = _pyramidNextIndex;
    currentLevelHash = _newEntity.value;

    for (uint32 i = sub_tree_level; i < levels; i++) {
      if (pyramidCurrentIndex % 2 == 0) {
        left = currentLevelHash;
        right = zeros(i);
        filledSubtrees[i] = currentLevelHash;
      } else {
        left = filledSubtrees[i];
        right = currentLevelHash;
      }
      currentLevelHash = hashLeftRight(hasher, left, right);
      pyramidCurrentIndex /= 2;
    }
    uint32 newRootIndex = (currentRootIndex + 1) % ROOT_HISTORY_SIZE;
    currentRootIndex = newRootIndex;
    roots[newRootIndex] = currentLevelHash;
    pyramidNextIndex = _pyramidNextIndex + 1;
    nextIndex = _nextIndex + 1;
  }
```

Bibliography

[1] "Ganache." Available at: https://trufflesuite.com/ganache/.

[2] "Git hub forum zeth." Available at: https://github.com/firoorg/zeth/issues/2.

[3] "Jubjub." Available at: https://z.cash/technology/jubjub/.

[4] "solidity-bignumber." Available at: https://github.com/firoorg/solidity-BigNumber.

[5] "Tornado cash." Available at: https://tornado.cash/.

[6] "Us bans tornado cash." Available at: https://www.cnbc.com/2022/08/08/crypto-mixing-service-tornado-cash-blacklisted-by-treasury-department-.html.

[7] "zeth." Available at: https://github.com/clearmatics/zeth.

[8] M. Agrawal, N. Kayal, and N. Saxena, "Primes is in p," *Ann. of Math*, vol. 2, pp. 781–793, 2002.

[9] M. R. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen, "Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity," in *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I* (J. H. Cheon and T. Takagi, eds.), vol. 10031 of *Lecture Notes in Computer Science*, pp. 191–219, 2016.

[10] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S. W. Cocco, and J. Yellick, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018* (R. Oliveira, P. Felber, and Y. C. Hu, eds.), pp. 30:1–30:15, ACM, 2018.

[11] J. Baylina, K. Gurkan, R. Semenov, A. Pertsev, adria0, E. Ben-Reuven, arnaucube, E. S., and M. Bellés, "circomlib," 2020. Available at: https://github.com/tornadocash/circomlib#c372f14d324d57339c88451834bf2824e73bbdbc.

[12] J. Baylina, K. Gurkan, R. Semenov, A. Pertsev, adria0, E. Ben-Reuven, arnaucube, E. S., and M. Bellés, "snarkjs," 2020. Available at: https://github.com/tornadocash/snarkjs#869181cfaf7526fe8972073d31655493a04326d5.

[13] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pp. 459–474, IEEE Computer Society, 2014.

[14] S. Benabbas, R. Gennaro, and Y. Vahlis, "Verifiable delegation of computation over large datasets," in *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara,*

CA, USA, August 14-18, 2011. Proceedings (P. Rogaway, ed.), vol. 6841 of *Lecture Notes in Computer Science*, pp. 111–131, Springer, 2011.

[15] J. C. Benaloh and M. de Mare, "One-way accumulators: A decentralized alternative to digital sinatures (extended abstract)," in *Advances in Cryptology - EUROCRYPT '93, Workshop on the Theory and Application of of Cryptographic Techniques, Lofthus, Norway, May 23-27, 1993, Proceedings* (T. Helleseth, ed.), vol. 765 of *Lecture Notes in Computer Science*, pp. 274–285, Springer, 1993.

[16] D. Benarroch, M. Campanelli, D. Fiore, K. Gurkan, and D. Kolonelos, "Zero-knowledge proofs for set membership: Efficient, succinct, modular," in *Financial Cryptography and Data Security - 25th International Conference, FC 2021, Virtual Event, March 1-5, 2021, Revised Selected Papers, Part I* (N. Borisov and C. Díaz, eds.), vol. 12674 of *Lecture Notes in Computer Science*, pp. 393–414, Springer, 2021.

[17] J. B. Bernabé, J. L. Cánovas, J. L. H. Ramos, R. T. Moreno, and A. F. Skarmeta, "Privacy-preserving solutions for blockchain: Review and challenges," *IEEE Access*, vol. 7, pp. 164908–164940, 2019.

[18] A. Biryukov, D. Khovratovich, and I. Pustogarov, "Deanonymisation of clients in bitcoin P2P network," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014* (G. Ahn, M. Yung, and N. Li, eds.), pp. 15–29, ACM, 2014.

[19] M. Blum, P. Feldman, and S. Micali, "Non-interactive zero-knowledge and its applications," in *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali* (O. Goldreich, ed.), pp. 329–349, ACM, 2019.

[20] D. Boneh, B. Bünz, and B. Fisch, "Batching techniques for accumulators with applications to iops and stateless blockchains," in *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part I* (A. Boldyreva and D. Micciancio, eds.), vol. 11692 of *Lecture Notes in Computer Science*, pp. 561–586, Springer, 2019.

[21] D. Boneh and V. Shoup, "A graduate course in applied cryptography," 2020. Available at: `https://crypto.stanford.edu/~dabo/cryptobook/BonehShoup_0_4.pdf`.

[22] J. Bootle, A. Cerulli, P. Chaidos, E. Ghadafi, J. Groth, and C. Petit, "Short accountable ring signatures based on DDH," in *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part I* (G. Pernul, P. Y. A. Ryan, and E. R. Weippl, eds.), vol. 9326 of *Lecture Notes in Computer Science*, pp. 243–265, Springer, 2015.

[23] B. Bünz, S. Agrawal, M. Zamani, and D. Boneh, "Zether: Towards privacy in a smart contract world," in *Financial Cryptography and Data Security - 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers* (J. Bonneau and N. Heninger, eds.), vol. 12059 of *Lecture Notes in Computer Science*, pp. 423–443, Springer, 2020.

[24] J. Camenisch, *Group signature schemes and payment systems based on the discrete logarithm problem*. PhD thesis, ETH Zurich, Zürich, Switzerland, 1998.

[25] J. Camenisch and A. Lysyanskaya, "Dynamic accumulators and application to efficient revocation of anonymous credentials," in *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings* (M. Yung, ed.), vol. 2442 of *Lecture Notes in Computer Science*, pp. 61–76, Springer, 2002.

[26] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers," in *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology*

Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings (T. Rabin, ed.), vol. 6223 of *Lecture Notes in Computer Science*, pp. 465–482, Springer, 2010.

[27] J. Groth, "On the size of pairing-based non-interactive arguments," in *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II* (M. Fischlin and J. Coron, eds.), vol. 9666 of *Lecture Notes in Computer Science*, pp. 305–326, Springer, 2016.

[28] C. Kang, C. Lee, K. Ko, J. Woo, and J. W. Hong, "De-anonymization of the bitcoin network using address clustering," in *Blockchain and Trustworthy Systems - Second International Conference, BlockSys 2020, Dali, China, August 6-7, 2020, Revised Selected Papers* (Z. Zheng, H. Dai, X. Fu, and B. Chen, eds.), vol. 1267 of *Communications in Computer and Information Science*, pp. 489–501, Springer, 2020.

[29] A. E. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pp. 839–858, IEEE Computer Society, 2016.

[30] D. V. Le and A. Gervais, "AMR: autonomous coin mixer with privacy preserving reward distribution," in *AFT '21: 3rd ACM Conference on Advances in Financial Technologies, Arlington, Virginia, USA, September 26 - 28, 2021* (F. Baldimtsi and T. Roughgarden, eds.), pp. 142–155, ACM, 2021.

[31] J. Li, N. Li, and R. Xue, "Universal accumulators with efficient nonmembership proofs," in *Applied Cryptography and Network Security, 5th International Conference, ACNS 2007, Zhuhai, China, June 5-8, 2007, Proceedings* (J. Katz and M. Yung, eds.), vol. 4521 of *Lecture Notes in Computer Science*, pp. 253–269, Springer, 2007.

[32] M. D. Liskov, "Fermat primality test," in *Encyclopedia of Cryptography and Security* (H. C. A. van Tilborg, ed.), Springer, 2005.

[33] M. D. Liskov, "Miller-rabin probabilistic primality test," in *Encyclopedia of Cryptography and Security* (H. C. A. van Tilborg, ed.), Springer, 2005.

[34] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings* (C. Pomerance, ed.), vol. 293 of *Lecture Notes in Computer Science*, pp. 369–378, Springer, 1987.

[35] S. Micali, M. O. Rabin, and J. Kilian, "Zero-knowledge sets," in *44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA, Proceedings*, pp. 80–91, IEEE Computer Society, 2003.

[36] I. Miers, C. Garman, M. Green, and A. D. Rubin, "Zerocoin: Anonymous distributed e-cash from bitcoin," in *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pp. 397–411, IEEE Computer Society, 2013.

[37] L. Reyzin and S. Yakoubov, "Efficient asynchronous accumulators for distributed PKI," in *Security and Cryptography for Networks - 10th International Conference, SCN 2016, Amalfi, Italy, August 31 - September 2, 2016, Proceedings* (V. Zikas and R. D. Prisco, eds.), vol. 9841 of *Lecture Notes in Computer Science*, pp. 292–309, Springer, 2016.

[38] N. Szabo, "Formalizing and securing relationships on public networks," *First Monday*, vol. 2, no. 9, 1997.

[39] N. Szabo, "Formalizing and securing relationships on public networks," *First Monday*, vol. 2, no. 9, 1997.

[40] P. Todd, "Merkle mountain ranges," 2018. Available at: https://github.com/opentimestamps/opentimestamps-server/blob/master/doc/merkle-mountain-range.md.

[41] Z. Wang, S. Chaliasos, K. Qin, L. Zhou, L. Gao, P. Berrang, B. Livshits, and A. Gervais, "On how zero-knowledge proof blockchain mixers improve, and worsen user privacy," *CoRR*, vol. abs/2201.09035, 2022.

Erklärung

Erklärung gemäss Art. 30 RSL Phil.-nat. 18

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Bern 6.9.22

Ort/Datum

Unterschrift