EFFICIENT CRYPTOGRAPHIC CONSTRUCTIONS FOR RESOURCE-CONSTRAINED BLOCKCHAIN CLIENTS

by

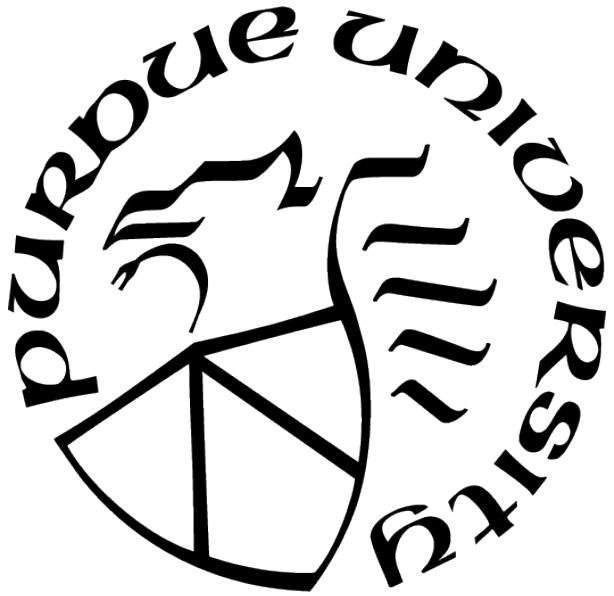
Duc Viet Le

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy



Department of Computer Science

West Lafayette, Indiana

August 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Aniket P. Kate, Co-Chair

School of Computer Science

Dr. Mikhail J. Atallah, Co-Chair

School of Computer Science

Dr. Jeremiah Blocki

School of Computer Science

Dr. Elena Grigorescu

School of Computer Science

Approved by:

Dr. Kihong Park

To my parents and my wife, Huyen Nguyen

ACKNOWLEDGMENTS

For months, I have been delaying writing this portion of the thesis. This is because there are so many people that I am indebted to for the success of this journey, and I cannot possibly list their contributions in just a few sentences. However, I will attempt to do my best and hope that no one is missed.

First and foremost, my most sincere thanks go to my two advisers, Professor Aniket Kate and Professor Mikhail Atallah. I am immensely grateful for the independence and mentorship they provided me during these five years. The two of them have taught me how to be a better researcher and a better person.

I am grateful to all my collaborators, whose contributions make this dissertation possible. I would like to thank my co-author and friend Pedro Moreno Sanchez for his time in Vienna. Pedro gave me invaluable advice and perspectives both in life and in research. Also, a special thanks to Arthur Gervais for offering me an internship at his company and giving me the freedom to choose research topics. Working with Arthur was a great pleasure.

I want to thank my committee members, Professor Jeremiah Blocki and Professor Elena Grigorescu, for agreeing to be a part of my dissertation committee. I was lucky enough to be a teaching assistant for Jeremiah's Cryptography class. I learned a great deal from his enthusiasm for teaching and research.

I am grateful to all labmates, present and past, for suffering me in the last five years. Special thanks to Sze Yiu and Adithya for joining me in so many dinners and patiently listening to all complaints about work and life.

Last but not least, I am indebted to my parents for their support and to my wife who has sacrificed a lot for this journey. I owe this thesis to them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The blockchain offers a decentralized way to provide security guarantees for financial transactions. However, this ability comes with the cost of storing a large (distributed) blockchain state and introducing additional computation and communication overhead to all participants. All these drawbacks raise a challenging scalability problem, especially for resource-constrained blockchain clients. On the other hand, some scaling solutions typically require resource-constrained clients to rely on other nodes with higher computational and storage capabilities. However, such scaling solutions often expose the data of the clients to risks of compromise of the more powerful nodes they rely on (e.g., accidental, malicious through a break-in, insider misbehavior, or malware infestation). This potential for leakage raises a privacy concern for these constrained clients, in addition to other scaling-related concerns. This dissertation proposes several cryptographic constructions and system designs enabling resource-constrained devices to participate in the blockchain network securely and efficiently.

Our first proposal concerns the storage facet for which we propose two add-on privacy designs to address the scaling issue of storing a large blockchain state. The first solution is an oblivious database framework, called T^3, that allows resource-constrained clients to obliviously fetch blockchain data from potential malicious full clients. The second solution focuses on the problem of using and storing additional private-by-design blockchains (e.g., Monero or ZCash) to achieve privacy. We propose an add-on tumbler design, called AMR, that offers privacy directly to clients of non-private blockchains such as Ethereum without the cost of storing and using different blockchain states.

Our second proposal addresses the communication facet with focus on payment channels as a solution to address the communication overhead between the constrained clients and the blockchain network. A payment channel enables transactions between arbitrary pairs of constrained clients with a minimal communication overhead with the blockchain network. However, in popular blockchains like Ethereum and Bitcoin, the payment data of such channels are exposed to the public, which is undesirable for financial applications. Thus, to hide transaction data, one can use blockchains that are private by design like Monero. However,

existing cryptographic primitives in Monero prevent the system from supporting any form of payment channels. Therefore, we present *Dual Linkable Spontaneous Anonymous Group Signature for Ad Hoc Groups (DLSAG)*, a linkable ring signature scheme that enables, for the first time, off-chain scalability solutions in Monero.

To address the computation facet, we address the computation overhead of the gossip protocol used in all popular blockchain protocols. For this purpose, we propose a signature primitive called *Flexible Signature*. In a flexible signature scheme, the verification algorithm quantifies the validity of a signature based on the computational effort performed by the verifier. Thus, the resource-constrained devices can partially verify the signatures in the blockchain transactions before relaying transactions to other peers. This primitive allows the resource-constrained devices to prevent spam transactions from flooding the blockchain network with overhead that is consistent with their resource constraints.

1. INTRODUCTION

In recent years, there has been an increased interest in the public blockchain, as it has enabled various financial applications without the need for a centralized trusted authority. However, the increasing adoption of blockchain raises several scalability issues. While the research community has proposed solutions to the scaling problem, those solutions are typically designed for high-end computational devices (i.e., personal laptops, workstations). Therefore, those solutions often prevent blockchain adoption for resource-constrained devices which cannot participate in the blockchain network meaningfully. On the other hand, existing solutions [77] dedicated to resource-constrained devices often expose the data of those devices to malicious nodes [68], thereby, raise another concern about privacy. In general, besides the privacy issue, three notable challenges hindering the adoption of blockchain for resource-constrained devices are the storage overhead, the communication overhead, and the computation overhead.

1.1 Challenges for Resource-constrained Devices in Permissionless Blockchains

In the following, we summarize three main challenges for resource-constrained devices in the permissionless Blockchain.

The Storage Overhead. It is well-known that the blockchain data (i.e., Bitcoin and Ethereum) has become too large to be stored by resource-constrained devices of any kind. To this end, the simplified payment verification (SPV) client has become a widely adopted solution to resolve the storage problem. In such a solution, SPV clients rely on other nodes to download and verify only the relevant part of the blockchain. However, such reliance by resource-constrained clients on other nodes implies an increased risk to privacy, as this solution clearly reveals clients' their transaction data and their transactions of interest. This information leakage is often undesirable for both business and personal entities because it can reveal sensitive information about clients (e.g., trade deals or medical bills) to outsiders (e.g., competitors or analytical companies). However, designing a privacy-preserving system offering oblivious access to resource-constrained blockchain clients is technically challenging. For instance, existing techniques proposed in research communities [105, 131] are not scalable

to handle a large number of requests from clients due to the limitation of the underlying cryptographic primitives.

On the other hand, to protect transaction data, some clients choose to use blockchains that are private by design, such as Monero [5] and ZCash [143]. However, this adoption of private blockchain leads to two significant problems. First, existing clients of smart-contract-enabled blockchains need to store additional blockchain data of those private blockchains and perform complicated swaps to exchange assets between two blockchains. Second, a complete switch to private-by-design blockchain prevents resource-constrained clients from using expressive applications (e.g., Decentralized Finance (DeFi) Applications [1, 41]) enabled by smart contracts. Thus, resource-constrained clients with a lack of storage capability face a dilemma between privacy and expressiveness. Therefore, it is natural to design an add-on privacy solution for existing clients of non-private blockchains. However, existing offered by the research community [108, 78] have several limitations. A solution based on ring signature [108] only offers a small anonymity set. Other solutions rely on a non-trusted third party [78, 152] to mix crypto asset; however, a non-trusted third party can always censor or prevent clients from mixing their assets (i.e., attack against availability).

The Communication Overhead. Opening communication and issuing many transactions to the blockchain peer-to-peer (P2P) network can be expensive in terms of communication overhead for resource-constrained clients, as every transaction needs to be broadcasted by the client and then validated by miners. Therefore, a natural direction to reduce such communication overhead is to employ payment channels or payment channel networks [124] to resource-constrained clients, a scalability solution already adopted in Bitcoin and Ethereum. In a payment channel, two resource-constrained clients can open a channel with each other by submitting one "opening" transaction agreed upon by both clients to the P2P network. After creating the channel, the clients can exchange transactions faster without constantly connecting to the P2P network. Finally, two clients can close the channel at any time by submitting another "closing" transaction to the P2P network. Thus, payment channels and payment channel networks significantly reduce the communication exchanged between the P2P network and resource-constrained clients.

However, in blockchains like Ethereum and Bitcoin, the data of payment channels is exposed to the public, which is undesirable for personal and business clients. Therefore, one may want to leverage private-by-design blockchains like Monero to build private payment channels in which the data of the channels are private. Still, building payment channels in Monero is non-trivial due to the cryptographic primitives used in Monero.

The Computation Overhead. Most of the blockchain systems often use a gossip protocol for broadcasting transactions among peers. In the gossip protocol, participating nodes need to verify the validity of incoming transactions via signature verification before relaying them to other nodes. This mechanism prevents malicious to flood the network with spam transactions. However, performing such a verification incurs a significant computation overhead on resource-constrained clients as they need to verify all incoming transactions. Thus, a cryptographic signature primitive that offers a trade-off between the computation and the error probability in verification can be useful to clients in the gossip protocol. Nevertheless, it is far from trivial to build such primitive as none of the prominent digital signature schemes (such as RSA and (EC)DSA) can offer such flexibility in the verification.

1.2 Contributions

This thesis takes a significant step towards addressing these major scaling challenges by proposing different cryptographic constructions, especially suited for resource-constrained devices to interact with the blockchain network securely and privately. This dissertation focuses on demonstrating the following statement:

> *Current blockchain designs introduce significant scaling challenges to resource-constrained clients in terms of storage, communication, and computation overheads. It is possible to build a secure and efficient system for resource-constrained clients to address those challenges without sacrificing clients' privacy.*

In the following, we describe our contributions in addressing these challenges to demonstrate the veracity of this statement.

Addressing Storage Overhead with Add-on Privacy Solutions. To address the storage facet, this dissertation proposes two add-on privacy solutions: T^3 and AMR. In par-

ticular, T^3 framework offers efficient oblivious accesses for resource-constrained devices when connecting to servers with a continuously changing database. T^3 can be used by blockchain resource-constrained clients to obliviously fetch the blockchain data from potentially malicious nodes. The other system, AMR, is a *privacy-preserving autonomous* mixer that allows resource-constrained clients to mix their crypto assets privately. AMR directly allows users of non-private blockchains (i.e., Ethereum) to maintain their transaction by obfuscating their transaction graph. Thus, AMR eliminates the need of switching to private blockchains like ZCash or Monero and avoids storing additional blockchain states of those private blockchains.

Addressing Communication Overhead with Private Payment Channels. to address communication overhead and privacy challenges, we present a *Dual Linkable Spontaneous Anonymous Group Signature for Ad Hoc Groups (DLSAG)*, a linkable ring signature scheme that enables for the first time off-chain scalability solutions in Monero such as payment channels and payment-channel networks. Such primitive will reduce the communication overhead required for the constrained devices when connecting in the blockchain peer to peer network while hiding the transaction data from being exposed to the public network.

Finally, we present a *Flexible Signature Framework*. A flexible signature framework offers a trade-off between the error probability in the verification procedure and the computation overhead. With application to the blockchain, we envision that the flexible signature framework will allow constrained devices to strengthen the blockchain network by helping verify signatures in blockchain transactions; therefore, it helps minimize computation overhead for constrained devices when participating in the blockchain network.

1.3 Outline of the Dissertation

This dissertation is organized into three parts. Part I focuses on how to address the storage overhead and contains Chapter 2 and Chapter 3. In Chapter 2, we propose the oblivious database framework, T^3, that allows constrained clients to obtain data from full clients obliviously. In Chapter 3, we present an *autonomous* mixer design, AMR, that directly offers privacy to clients on non-private blockchain like Ethereum without the need of using and storing state of private blockchains like ZCash or Monero. Part II focuses on how to

improve the communication overhead while preserving blockchain clients' privacy. This part consists of Chapter 4. Part III, which consists of Chapter 5, proposes a new signature primitive, called *Flexible Signature*, that helps reduce the computation overhead of resource-constrained blockchain clients. Finally, we summarize this dissertation in Chapter 6

Part I

Addressing Storage Overhead with Add-on Privacy Solutions

2. OBLIVIOUS DATABASE FRAMEWORK FOR SIMPLIFIED PAYMENT VEFIFICATION CLIENTS

Over the last few years, we have seen a great interest in the public blockchain. The Bitcoin blockchain offers a way to provide security for financial transactions. However, due to the vast adoption of Bitcoin, the size of its blockchain has become too large for resource-constrained devices such as personal laptops or mobile phones, raising performance and privacy concerns in the community. As of October 2018, the size of the unindexed Bitcoin blockchain was 230 GB.

To this end, Bitcoin's simplified payment verification (SPV) client has become a widely-adopted solution to resolve the storage problem for constrained devices. Satoshi Nakamoto [120] sketched the idea of SPV clients in the Bitcoin whitepaper, and in the Bitcoin improvement proposal 37 (BIP37) [77], Mike Hearn combined Nakamoto's idea with Bloom filters to standardize the design of Bitcoin SPV clients. This design has become a de facto standard for other SPV clients such as BitcoinJ [23] and Electrum [58].

The core of SPV clients consists of downloading and verifying the part of the blockchain that is relevant to the SPV client itself. In particular, the SPV client loads its addresses into a Bloom filter and sends the filter to a Bitcoin full client, that uses the filter sent by the client to identify and send back the block contains transactions that are relevant to the SPV client, along with Merkle proofs for those transactions.

However, the current SPV solution relying on Bloom filters raises security and privacy concerns to the SPV client when it communicates with potentially compromised nodes (whether accidentally or maliciously). Gervais et al. [69] showed that it is possible for a malicious node to learn several addresses of the client from the Bloom filter with high probability. Moreover, if the adversarial node can collect two filters issued by the same client, it can learn a considerable number of addresses owned by the client.

To provide a strong privacy guarantee for SPV clients, one needs a solution that can hide wallets/addresses queried by the SPV clients [79]. While such a system can be built using private information retrieval (PIR) primitives, the existing cryptographic PIR solutions [83, 131] are not scalable to handle millions of Bitcoin users. On the other hand, to gain more

efficiency, one can use ORAM and TEE to propose generic PIR systems [80, 60, 145]. However, as we will see later in this chapter, naively combining ORAM scheme as it is with TEE makes the practicality of those generic systems questionable when used in a large network like Bitcoin due to the lack of concurrency in ORAM as well as the limitation of TEE with restricted memory.

Our Contributions. This work aims not only to design a system that provides SPV clients with privacy-preserving access to the Bitcoin blockchain data but also to consider other practical aspects on how to scale such a system to handle client requests in a large-scale. Our contributions can be summarized as follows:

Firstly, we present a design for a system that can handle up to thousands of requests per minute from Bitcoin SPV clients based on a *restricted access* Oblivious Random-Access Memory (ORAM) and the trusted execution capabilities of TEE. In particular, one of the main contributions of our design is the optimization access in the prominent tree-based ORAM schemes that allow those ORAM schemes to support concurrent accesses which is essential for handling SPV clients' requests. In this design, the access privacy guarantee is still maintained because of our natural assumption that the rational Bitcoin SPV clients should only query for their particular transaction *once* before the arrival of a new Bitcoin block. Nevertheless, we later show that even when the SPV clients are irrational then the privacy for such clients is only compromised for a short period of time. The security guarantee of T^3 also relies on the trusted execution capabilities of TEE that allows SPV clients to perform ORAM operations securely and remotely. Our generic design works with other blockchains, any tree-based ORAM schemes [147, 149, 158], and any TEE with attestation capability.

Secondly, we implemented a prototype of T^3 and evaluated its performance to demonstrate the practicality of our approach. More specifically, we extracted the unspent transaction outputs set of Bitcoin in October 2018 and used it to measure the performance of the system when handling clients' requests. The implementation of T^3 also adopts standard techniques (i.e., oblivious operations using cmov [145, 3, 133]) to be secure against known side-channel attacks [95, 94, 161]. Moreover, the use of recursive ORAM constructions in T^3 makes the system much more suitable for TEE with restricted trusted memory like Intel

SGX. We then show that the running time of the ORAM read access decreases linearly with the number of the threads used. Our implementation is available at [151].

Finally, we conclude that putting natural restrictions on the access patterns on oblivious memory can lead to significant performance improvement and better ORAM design. While the applicability of T^3 in cryptocurrencies beyond Bitcoin is apparent, we believe our work will also motivate further research on oblivious memory with restricted access patterns.

2.1 Design Goals and Solution Overview

In this part, we define the system components, outline our security goals, and give an overview of how our system works.

2.1.1 System Components

There are three key components of this system: the Bitcoin network, a client, and an untrusted full node. The **Bitcoin Network** is a set of nodes that maintains the Bitcoin blockchain, and the network validates and relays the new Bitcoin block produced by miners. A **Client** is a Bitcoin Simplified Payment Verification node that remotely connects to the secure TEE on the untrusted full node to perform oblivious searches on the unspent transaction output (UTXO) set. The client is also able to connect to the Bitcoin Network to obtain other network metadata such as the latest Bitcoin block header. A **Full Node** is an untrusted entity made up of two components: an untrusted full node and several trusted TEEs (i.e., the *managing*, *reading*, and *writing* TEEs). Moreover, the untrusted full node stores three encrypted databases which are the *read-once* ORAM tree, the *original* ORAM tree, and the Bitcoin header chain. The untrusted full node hosts a potentially malicious Bitcoin client (e.g., bitcoind) that handles the communication with the Bitcoin Network.

2.1.2 Design Goals

The goal of our system is to leverage the trusted execution capabilities of a Trusted Execution Environment (TEE) with attestation to design a public Bitcoin full node that supports oblivious search and update on the current Bitcoin unspent transaction output

database. Our system aims to provide data confidentiality and privacy to Bitcoin SPV clients on a large scale by using standard encryption and Oblivious RAM techniques on the current set of unspent transaction outputs. The main goals that T^3 tries to achieve are:

Privacy. T^3 aims to provide privacy and confidentiality to SPV clients' requests. In particular, the system allows SPV clients to obliviously search its relevant transactions without revealing their addresses to potentially malicious providers by using TEE to encrypt the data and using ORAM schemes to eliminate known side channel leakages [133, 3, 80, 145].

Validity. The SPV client should be able to obtain valid information based on the provided addresses, and a malicious adversary should not able to tamper the Blockchain data with invalid transaction outputs.

Completeness. The system should provide clients with access to most of its relevant transactions to determine balance or to obtain essential information to form new transactions.

Efficiency. The system should be practical to deploy. More specifically, the system should be efficient enough to handle different concurrent SPV clients' requests without compromising the privacy of the clients.

2.1.3 Solution Overview

The idea of using ORAM schemes and trusted execution environments to construct database systems that support oblivious accesses has been investigated by the research community [80, 60, 145]. However, the efficiency and scalability of those systems are hampered by the lack of concurrency of traditional ORAM schemes [149, 158].

In this work, we design T^3 to overcome the limitations of efficiency and concurrency plaguing existing systems. Our design is motivated by the following observations. The first observation is that each ORAM access in a standard tree-based ORAM setting is a combination of two operations: a *read-path* operation and an *eviction* operation. By separating the effects of two operations into two different trees: a *read-once* ORAM tree and an *original* ORAM tree, one can use read-path operation on the *read-once* ORAM tree to handle clients' requests simultaneously while performing a non-blocking eviction operation on the *original* ORAM tree sequentially.

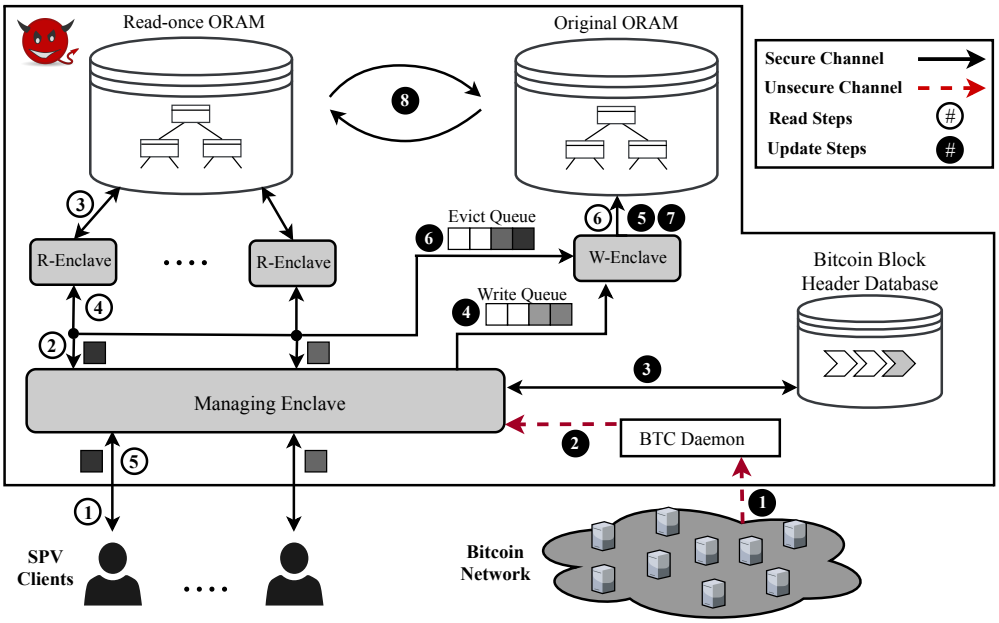


Figure 2.1. Overview of T^3 design.

The second observation is that the access privacy guarantee of this approach relies on the characteristic of the Bitcoin blockchain. In particular, the Bitcoin network generates new Bitcoin block on average of 10 minutes, and if we require T^3 to periodically synchronize these the two trees, then the privacy of clients' queries is preserved. Moreover, if we assume that upon receiving transactions belonged to its addresses, the rational client should not query same transactions *again* until the next block arrives, the proposed approach on the separation of *read-path* and *eviction* procedure not only does not affect the privacy guarantees of ORAM access but also allows T^3 to efficiently handle more clients' requests. More importantly, we argue that even when the SPV clients are irrational by submitting requests for the same transaction more than once, the privacy of those clients is only compromised for a short period (i.e., 10 minutes for the Bitcoin network) because T^3 will always synchronize the old instance of the *read-once* ORAM tree with the more updated instance of the *original* ORAM tree. With the intuition of T^3 described above, we outline the workflow of our design in Fig. 2.1:

Full node Initialization ❶-❽: Initially, the managing TEE will initialize a *writing* TEE that creates an empty ORAM tree. For each of Bitcoin block obtained from the network, the

managing TEE verifies the proof of work of the block before passing relevant update data to the *writing* TEE in order to populate the ORAM tree. With the current size of the Bitcoin blockchain, this operation may take several hours. However, once the TEEs catch up with the current state of the Bitcoin blockchain, we expect that the TEE only has to perform a batch of update accesses on the ORAM tree every 10 minutes. When the initialization is completed, the *managing* TEE creates two copies of the ORAM tree which are the *read-once* ORAM tree and the *original* ORAM tree.

Oblivious *read-once* Protocol ①-⑥: To obtain its unspent outputs, the client first performs the remote attestation to the *managing* TEE. The remote attestation mechanism allows the client to verify the correctness of program execution inside the TEE. More importantly, after a successful attestation, the client can use standard key exchange mechanism [55] to share a secret session key with the TEE to establish a secure connection with the *managing* TEE. Upon receiving client's connection requests, the *managing* TEE creates a *reading* TEE with its copies of the ORAM position map and the ORAM stash to handle client subsequent requests. Next, after having a secure channel, the client will send his Bitcoin addresses along with the proof of ownership of those addresses to the TEE. The *reading* TEE will use a mapping function to map Bitcoin addresses into the ORAM block identification number and performs *read-once* ORAM access on the ORAM tree. Those *read-once* ORAM access do *not* involve the eviction procedure which requires re-encrypting and remapping the ORAM block. The eviction procedure will be performed on the *original* ORAM tree by the *writing* TEE.

Oblivious Write Protocol ❶-❽: The T^3 requires to update the ORAM tree via batch of write accesses every 10 minutes on average. T^3 will rely on a standard Bitcoin client to handle the communication with the Bitcoin network to obtain blockchain data [1]. Thus, T^3 needs to verify the block relayed by a potentially malicious Bitcoin client before updating the ORAM tree. More specifically, in the design, T^3 stores a separate Bitcoin header chain to verify the proof of work and the validity of all transactions inside a Bitcoin block. After the verification, the *managing* TEE forms a batch of ORAM updates and delegates those updates to the *writing* TEE. Once those updates are finished, the *managing* TEE will queue

[1]↑This feature can be easily included in the future implementation of T^3.

up read requests from SPV clients to allow the *writing* TEE to finish the eviction requests from the *read* TEEs during the updating interval. As soon as the *writing* TEE finishes performing those eviction requests, the *managing* TEE updates the position map and *stash*, and makes the ORAM tree used by the *writing* TEE become the new ORAM tree used by *reading* TEE. At this point, the *reading* TEE can use the new tree instance to respond to clients' requests while the *writing* TEE performs the eviction procedure on another copy of the same ORAM tree.

2.2 Preliminaries and Threat Model

2.2.1 Trusted Execution Environment

The design of T^3 relies on a trusted execution environment (TEE) to prove the correctness of the computations. TEE is a trusted hardware that provides both confidentiality and integrity of computations as well as offer an authentication mechanism, known as *attestation*, for the client to verify computation correctness. In this work, we chose Intel SGX [43] to be the building block of our system. However, with minor modifications, the design of our system can be extended to any TEE with *attestation* capabilities such as Keystone-enclave [87] and Sanctum [44] as other trusted execution environments might not have the same strengths/weaknesses as Intel SGX.

Intel SGX is a set of hardware instructions introduced with the 6th Generation Intel Core processors. We use Intel SGX as a TEE for the execution of an ORAM controller on the untrusted full node. The relevant elements of intel SGX are as follows: **Enclave** is the trusted execution unit that is in a dedicated portion of the physical RAM called the enclave page cache (EPC). The SGX processor makes sure that all other software components on the system cannot access the enclave memory. Intel SGX supports both **local and remote attestation** mechanisms to allow remote parties or local enclaves to authenticate and verify if the program is correctly executed within an SGX context. More importantly, attestation protocols provide the authentication required for a key exchange protocol [43], i.e., after a successful attestation, the concerned parties can agree on a shared session key using Diffie-Hellman Key Exchange [55] and create a secure channel.

Limitations. Intel SGX comes with various limitations that have been uncovered by the academic community over the past few years. Some of these limitations are:

- **Side-Channel Attacks:** While Intel SGX provides security guarantees against direct memory attacks, it does not provide systematic protection mechanisms against side-channel attacks such as page table-based [161, 94], cache-based [31], and branch-prediction-based [95]. Through page table and cache attacks, a privileged attacker can observe cache-line-granular (i.e., 64B) memory access patterns from the enclave program. On the other hand, the branch-prediction attack can potentially leak all the control-flow taken by the enclave program.

- **Enclave Page Cache Limit:** The size of the Enclave Page Cache (EPC) is limited to around 96MB [8]. Although Intel SGX alleviates this limitation by supporting page-swapping between trusted memory region and untrusted memory region, this operation is expensive due to encryption and integrity verification [43, 8].

- **System Calls:** Intel SGX programs are restricted to ring-3 privileges and therefore rely on the untrusted OS for ring-0 operations such as file and network I/O. Various previous works try to solve this problem using library OSes [155] and/or other techniques [80].

Oblivious Operations Inside the Enclave. Several techniques [133, 80, 145, 123] have been introduced to mitigate side-channel attacks on the SGX. In this work, we built our system based on the implementations of both Zerotrace [145] and Obliviate [3]. Therefore, our system inherited standard secure operations from both of libraries. In particular, their implementations use an oblivious access wrapper by using the x86 instruction cmov as introduced by Raccoon [133]. Using cmov, the wrapper accesses every single byte of a memory object while reading or writing only the required bytes in memory. From the perspective of an attacker (which can only observe access-patterns), this is the same as reading or modifying every byte in memory. We refer readers to [145, 3, 133] for detailed descriptions of these oblivious operations.

2.2.2 Oblivious Random-Access Machine

Oblivious Random Access Machine (ORAM) was first introduced by Goldreich et al [70] for software protection against piracy. The core of ORAM is to hide the access patterns resulted from reading and writing accesses on encrypted data. The security of ORAM can be described as follows.

Definition 2.2.1. *[149] Let* $\vec{y} = (\mathsf{op_i}, \mathsf{bid_i}, \mathsf{data_i})_{i\in[n]}$ *denote a sequence of accesses where* $\mathsf{op_i} \in \{read, write\}$, $\mathsf{bid_i}$ *is the identifier, and* $\mathsf{data_i}$ *denotes the data being written. For an ORAM scheme* Σ*, let* $\mathsf{Access}_\Sigma(\vec{y})$ *denote a sequence of physical accesses pattern on encrypted data produced by* $\vec{y}$*. We say: (a) The scheme* Σ *is secure if for any two sequences of accesses* $\vec{x}$ *and* $\vec{y}$ *of the same length,* $\mathsf{Access}_\Sigma(\vec{x})$ *and* $\mathsf{Access}_\Sigma(\vec{y})$ *are computationally indistinguishable. (b) The scheme* Σ *is correct if it returns on input* $\vec{y}$ *data that is consistent with* $\vec{y}$ *with probability* $\geq 1 - \mathsf{negl}(|\vec{y}|)$ *i.e negligible in* $|\vec{y}|$

Tree-based ORAM schemes. One strategy of designing an ORAM scheme is to follow the tree paradigm proposed by Shi et al. [147] and Stefanov et al. [149]. In tree-based ORAM, the client encrypts their database into N different encrypted data blocks and obliviously stores those data blocks in a binary tree of height $\lceil \log_2(N) \rceil$. Each node in the tree is called a *bucket*, and each *bucket* can contain up to Z blocks. The client also maintains a *position map*, to indicate which path a data block resides on. Finally, the client needs to have a *stash* to store a path retrieved from the server.

Each access in both ORAM schemes requires two operations: a ReadPath operation and an Evict operation. Intuitively, ReadPath takes as input the ORAM block identifier, bid, accesses the position map, and retrieves the path that block bid resides onto the stash, S. After performing ORAM access (i.e., read/write) on the identified block, the block is assigned to a different path and pushed back to the tree via the Evict operation. In general, the Evict operation takes a stash and the assigned path as input, writes back blocks from stash to the assigned path, and updates the position map.

PathORAM/CircuitORAM Scheme. In this work, we consider two popular tree-based constructions of ORAM: PathORAM [149] and CircuitORAM [158]. While

PathORAM offers simple ReadPath and Evict operations, CircuitORAM offers a smaller circuit complexity for the Evict procedure. Thus, CircuitORAM is more efficient when implemented with Intel SGX. As noted in [145, 80], CircuitORAM can operate with $Z = 2$ compared to $Z = 4$ as in PathORAM; therefore, the server storage overhead is significantly reduced. Moreover, the size of *stash* in CircuitORAM is smaller compared to the size of *stash* in PathORAM; this allows a more efficient performance when scanning the stash as one needs to scan the whole path and stash to avoid side-channel leakage.

Recursive ORAM. In a non-recursive tree-based ORAM setting, the client has to store a position map of the size $O(N)$ bits. This approach, however, is not suitable for a resource-constrained client. Stefanov et. al [149] presented a technique that reduces the size of the position map to $O(1)$. The main idea of those constructions is to store a position map as another ORAM tree in the server, and the client only store the position map of the new ORAM tree. The client recursively stores the position map into another ORAM tree until the size of the position map is small enough to be saved on the client's storage. One main drawback of those constructions is the increased cost in the communication between a client and the server. In our setting, this cost can be safely ignored because the communication between client and server becomes the I/O access between TEE and the random access memory.

2.2.3 Blockchain

The Bitcoin blockchain is a distributed data structure maintained by a network of nodes. On average of 10 minutes, the network outputs a block which is a combination of transactions and a block header. Each block header contains relevant information about the Bitcoin block such as Merkle root, nonce, network difficulty. The Merkle root can be used to verify the membership of Bitcoin transactions, and the nonce and difficulty are used to check the proof of work. Each Bitcoin transaction contains a set of inputs and outputs where transaction inputs are unused outputs of previous transactions.

Unspent Transaction Output Database. In the Bitcoin network, the balance of a Bitcoin address is determined by the values of those outputs that have not been used in other

transactions. These outputs are called Unspent Transaction Outputs (UTXO). Moreover, in the implementation of common Bitcoin nodes such as Bitcoin core [20], Bitcoin nodes maintain a separate database that keeps track of all unspent transaction outputs and other metadata of the Bitcoin blockchain. Therefore, we realize that if a full node can securely update and maintain the integrity of the UTXO set via while provides SPV clients with oblivious accesses to the UTXO set, the privacy of the SPV client is preserved.

Bitcoin transaction types. In the Bitcoin, transactions are classified based on the structure of the input and output scripts. There are five types of standard script templates which are *Pay-to-Pubkey* (P2PK), *Pay-to-PubkeyHash* (P2PKH), *Pay-to-ScriptHash* (P2SH), *Multisig*, and *Nulldata*. Intuitively, scripting in Bitcoin provides a way to prove the ownership of the coins.

In this work, we only consider two types of transactions: *Pay-to-PubkeyHash* (P2PKH) transaction and *Pay-to-ScriptHash* (P2SH) transaction. According to [52, 116], those two types of transactions made up of 97-99% of the UTXO set. Also, one can assume that the *Pay-to-Pubkey-Hash* transaction is one variant of the *Pay-to-Script-Hash* transaction because both transaction types require the spender's knowledge of the preimage of the hash digest before being able to spend those outputs. For simplicity, from this point on, we assume that the only information needed to obtain the unspent output is the public key hash, *pkh*. All other transaction types such as *Multisig* and P2PK can be easily supported in the future.

Block creation interval. The block creation time in Bitcoin is the time that the network takes to generate a new block, and block creation time is specified to be 10 minutes on average by the network. We call the waiting period between the most recent block and a new block, *block creation interval*. In this work, we discretize time as *block creation intervals*.

Deterministic Wallet. In Bitcoin, a deterministic wallet [53] is a system that allows the creation of several public addresses on-fly from a single seed. The main idea of deterministic wallets is to generate an unlimited number of addresses for a client to help mitigate the risk of reusing addresses [2]. Thus, ideally, in Bitcoin, users are expected to create a new address for each person who is paying, and after receiving the coin, the address should never be used

again. Therefore, it is reasonable to expect that the number of unspent outputs for each address is one.

UTXO-based Blockchains. After the advent of Bitcoin, the blockchain community has developed different cryptocurrencies to address the shortcomings of Bitcoin. While the employed underlying cryptographic primitives are different, the transaction structure of those cryptocurrencies follows the similar design paradigm as in Bitcoin: Transactions are formed based on outputs of previous transactions, and the creation of transactions forms new unspent outputs, and the notion of balance in these cryptocurrencies is determined by the values of those unspent outputs. We called those UTXO-based cryptocurrencies. Few examples of UTXO-based currencies are Litecoin [100], Dash [50], and Zcash [143]. Thus, as the design will become apparent in later sections, we argue that the design of T^3 applies not only to Bitcoin but also to other UTXO-based blockchains.

2.2.4 Threat Model

We assume that SPV *clients* are honest and rational which means that before during the *block creation interval*, an SPV *client* should not request the full node for transaction outputs of the same public key hash more than once.

The underlying remote attestation service provided by TEE is assumed to be secure and trusted. The local attestation between enclaves is secure. The full node and its programs are assumed to be untrusted except for programs running within an enclave.

We assume that the adversary who controls the operating system can observe, inject, and modify encrypted messages sent by enclaves. The adversary also can observe memory access patterns of both trusted and untrusted memory. Also, the computation power of the adversary is assumed to be limited. During the *block creation interval*, the adversary should not have enough computation power to forge a new Bitcoin block that satisfies the current Bitcoin network difficulty. As the time of writing, the network difficulty [22] is around 6×10^9; therefore, the expected number of hashes to mine a Bitcoin block is roughly 2^{72}.

The full node's attacks on availability are out of scope. More specifically, denial of service (DoS) attacks by system admin and untrusted operating system are out of the scope.

Otherwise, such adversaries can prevent the enclaves from receiving new bitcoin block by shutting down the communication channel between the enclave and the Bitcoin network as the enclave has to rely on the untrusted OS to perform system calls such as file and network I/O. On the other hand, for DoS attacks from the client, we will outline possible DoS attacks and offer solutions to mitigate them in Section 2.5.

2.3 Proposed System

In this section, we describe how T^3 stores the UTXO set by exploring different mappings between the unspent transaction outputs and the ORAM blocks. Next, we demonstrate how Intel SGX can be considered as a trusted execution unit to access ORAM and perform read/write operations in an oblivious manner. Finally, we will describe how the system handles clients' requests during a write operation.

2.3.1 Storage Structure of the UTXO set

In this part, we show how T^3 stores the UTXO set into ORAM tree structures.

Bitcoin unspent transaction output mapping. In the design of T^3, the SPV clients only know his/her addresses (i.e., the public key hashes); therefore, to return the outputs belonging to the client's address, TEE needs to know the mapping between the address and the ORAM block identification.

We propose two secure mappings to store unspent outputs in the ORAM tree as naive mapping may lead to attacks on the system. Both approaches use standard pseudorandom function (PRF) along with a secret key generated by the enclave. The first approach is to map a single Bitcoin address into a single ORAM block, and the second approach is to map a Bitcoin address into multiple ORAM blocks. We will later explain the trade-off between these two approaches.

Single address into single ORAM block. In this design, during the initialization, we require the program inside the enclave to use a PRF to map the public key hash to ORAM block identification. The secret key of the PRF is generated inside the enclave; thus, the mapping is known only to the SGX. We define the mapping as follow:

- $\mathsf{bid} \leftarrow \text{OBlockMap}(pkh, k_b)$: the function takes as input a 20-bytes hash digest pkh and a secret key k_b, it outputs the block identification number $\mathsf{bid} \in \{0, \ldots, N-1\}$.

The PRF approach offers some flexibility when deciding the size of an ORAM blocks and the size of height of the ORAM tree. These two factors affect the size of the position map (resp. number of recursive levels) for non-recursive (resp. recursive) ORAM constructions. However, since the output domain of $\text{OBlockMap}(\cdot, \cdot)$ is limited to the size of the ORAM blocks, there will exist collisions. The following lemma gives us a loose upper bound on the number of addresses that should be stored inside an ORAM block.

Lemma 2.3.1. *(Addresses per ORAM block) Let m be the number of public key hashes, N be the number of ORAM blocks. If the OBlockMap() acts as a truly random function, then the maximum number of addresses in each ORAM block is smaller than $e \cdot m/N$ with a probability $1 - 1/N$.*

Proof. This is a standard max-load analysis. We refer readers to [48] for detailed analysis. We note that there exists a tighter bound, but we use $e \cdot m/N$ bounds to simplify the equation. □

The second approach of Figure 2.2 gives us a high-level overview of this approach.

Single address into many ORAM blocks. Mapping a single address into a single ORAM block incurs less work on the full node as it requires a single ORAM access for an address. However, if one wants to allow each address to have more than one output, using the first approach implies that the storage overhead will increase linearly. Thus, we need a different mapping without linear increasing in storage overhead. To fix this shortcoming,

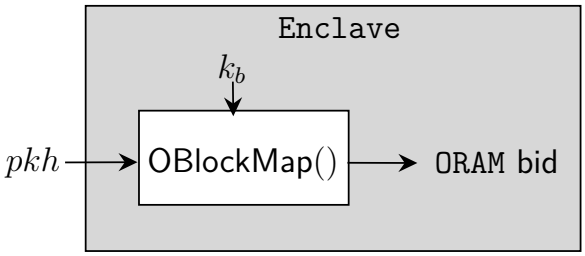


Figure 2.2. Single address into Single ORAM block.

the system needs to assign unspent outputs into ORAM block uniformly. One method is to allow a client to specify the number of ORAM accesses to obtain all its unspent outputs as long as the number of requests does not exceed certain threshold. We define the mapping as follows:

- $\{\mathsf{bid}_i\}_{i\in\{0,\ldots,\delta-1\}} \leftarrow \text{OBlockMap}(pkh, k_b, \delta)$: the function takes as input a 20-bytes hash digest pkh, a secret key k_b, and a number δ where the maximum value of δ is specified by the system. It outputs a set of block identification numbers $\{bid_i\}_{i\in\{0,\ldots,\delta-1\}} \subseteq \{0,\ldots,N-1\}$.

This approach also introduces some leakage as some addresses may contain more unspent outputs than others. Alternatively, the system can fix the value of δ ORAM accesses for all addresses with the expense of performance (i.e., one address incurs constant ORAM accesses). Similarly, the storage overhead of T^3 can be computed using the following claim:

Lemma 2.3.2. *(UTXO per ORAM block) Let m be the number of unspent outputs, N be the number of ORAM blocks. If the OBlockMap acts as a truly random function, then the maximum number of outputs in each ORAM block is smaller than $e \cdot m/N$ with probability at least $1 - 1/N$*

The proof is identical to proof of Lemma 2.3.1. Figure 2.3 offers an overview of both approaches.

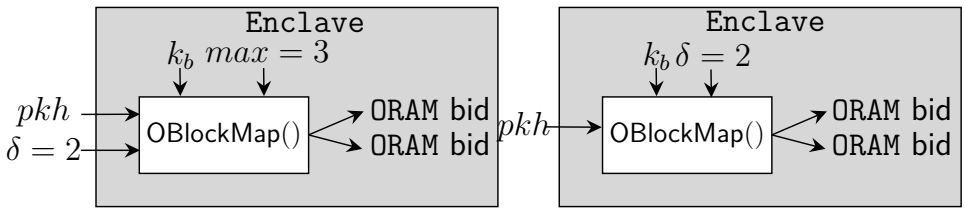


Figure 2.3. Single Address into One/Many ORAM block(s). One approach allows the SPV client to specify the number of ORAM accesses with a maximum threshold. The other approach maps single address into a constant number of ORAM access

Storage. In this system, we require the untrusted full node to store three separate databases which are the *read-once* ORAM tree, the *original* ORAM tree, and the blockheader

chain. In particular, ***Read-Once* ORAM Tree** serves as a dedicated storage to handle clients' requests. The structure of the tree is identical to the standard ORAM tree. ***Original* ORAM Tree** is where all standard ORAM eviction operations are performed. In this work, we also require the enclave to maintain the **Bitcoin Header Chain** to verify the proof of work of the bitcoin block sent by other bitcoin clients. The header chain is stored in the untrusted memory with an integrity check.

2.3.2 Oblivious Read and Write Protocols

In T^3, the SPV client is the party who invokes read accesses, and the Bitcoin network is the party who invokes write accesses. The TEE in the full node is the one that performs both of those accesses on behalf of the client and the Bitcoin network.

Full Node's System Components. Before explaining how oblivious read and write accesses work, we first start outlining the different components of our design. The full node is initialized with different enclaves: *Managing Enclave* $\mathcal{E}_m$ coordinates other enclaves and to handle requests from the clients. The *managing* enclave also handles the communication with other Bitcoin client or local Bitcoin client (bitcoind) via request procedure calls (RPC) to obtain Bitcoin blocks. Upon receiving the Bitcoin block, the *managing* enclave also verifies the integrity of the block using a separated header chain. *Reading Enclave* $\mathcal{E}_r$ is a dedicated enclave initialized by the *managing* enclave. It has a copy of ORAM position map and its own stash. The *reading* enclave operates on the *read-once* ORAM tree. Also, the *reading* enclave only performs ORAM ReadPath operations to obtain data while ORAM Eviction operations will be handled by the *writing* enclave. *Writing Enclave* $\mathcal{E}_w$ performs Eviction procedure for each read request and performs ORAM writing accesses when a new Bitcoin block arrives from the Bitcoin network.

Oblivious *read-once* Protocol. In this part, we describe how a remote client can perform a read access on the UTXO set. First, we denote K_b to be the block mapping key, bid to be the ORAM block identification. We let $(\mathsf{Enc}, \mathsf{Dec})$ denote an authenticated encryption scheme. We assume that the the full node has already been initialized with a *writing* enclave, $\mathcal{E}_w$ and a *managing* enclave, $\mathcal{E}_m$. The *managing* enclave has a similar copy

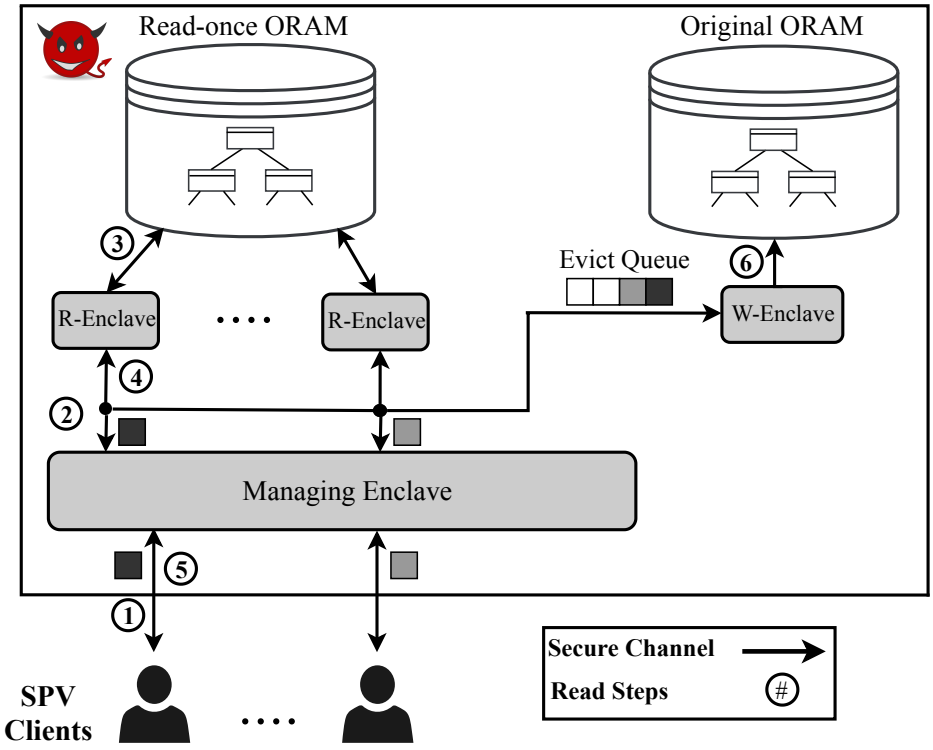


Figure 2.4. T^3 Oblivious Read Protocol.

of the position map as the map in the *writing* enclave. Figure 2.4 presents the oblivious read protocol of T^3. The oblivious read protocol can be described as follows:

1. **The client establishes a secure channel** [2] **with the *managing* enclave** ①: First, the client performs a remote attestation with the secure *managing* enclave, $\mathcal{E}_m$, and agrees on a session key, K_s. The client encrypts his address along with the proof of ownership of that address and sends the encrypted query to the full node to be passed to $\mathcal{E}_m$. For simplicity, we assume that the plaintext only contains a public key hash, pkh, that the client is interested in, and the proof of ownership of the pkh is ϕ, $C \leftarrow \mathsf{Enc}_{K_s}(pkh, \phi)$. Note that there are different ways to prove the ownership of public key hash/addresses. In Bitcoin, if the public key is never revealed before, the proof of ownership can simply be the public key (i.e., $\phi = pk$ such that $H(pk) = pkh$). Alternatively, the system can enforce a client to provide the signature and the public key to prove the ownership of the public key hash.

[2] ↑The standard instantiation of a secure channel is using SSL/TLS channel

2. **The *managing* enclave initializes a *reading* enclave** ② : after receiving a client's request, $\mathcal{E}_m$ initializes a dedicated *reading* enclave, $\mathcal{E}_r$ to handle the client's future requests. Also, we require that the enclaves authenticate each other, and the existence of a secure channel between enclaves. Moreover, the *reading* enclave has its copy of the position map, its own stash, the block mapping key K_b, and the agreed session key K_s.

3. **The *managing* enclave identifies and forwards ORAM Block ID to both *reading* and *writing* enclaves** ②: After decrypting the ciphertext $(pkh, \phi) \leftarrow \mathsf{Dec}_{K_s}(C)$, $\mathcal{E}_m$ verifies the proof ϕ and pkh, then uses $\textsc{OBlockMap}(\cdot,\cdot)$ [3] function to learn the ORAM block ID, $\mathsf{bid} \leftarrow \textsc{OBlockMap}(pkh, K_b)$ where K_b is the secret key generated by the enclave during initialization for mapping purposes. After obtaining the ORAM id, bid, the *managing* enclave forwards bid to the *writing* enclave for the eviction procedure, and forwards the (pkh, bid) to the *reading* enclave.

4. **The *reading* enclave performs *read-once* ORAM access on the *read-once* ORAM tree** ③: Based on the given bid, the *reading* enclave performs ORAM read only accesses on the ORAM tree to obtain the block. If the block contains the unspent output that belongs to the public key pkh, the *reading* enclave adds outputs into the response R. To mitigate the size leakage, the response R is padded with dummy data if there is no UTXO found.

5. **The *reading* enclave responds to the Client** ④-⑤ : The enclave encrypts the response, R, using the session key K_s then sends it to the client.

6. **The *writing* enclave performs the eviction procedure on the *original* ORAM tree** ⑥: After obtaining the bid from the *managing* enclave, the update enclave will perform a standard ORAM read accesses on the *original* ORAM tree. The goal of this procedure is to use the $\mathsf{Eviction}$ procedure inside standard ORAM operation to rerandomize the location of the actual block. No actual data is return in this step.

[3] ↑For simplicity, we assume that the one-to-one mapping is used here

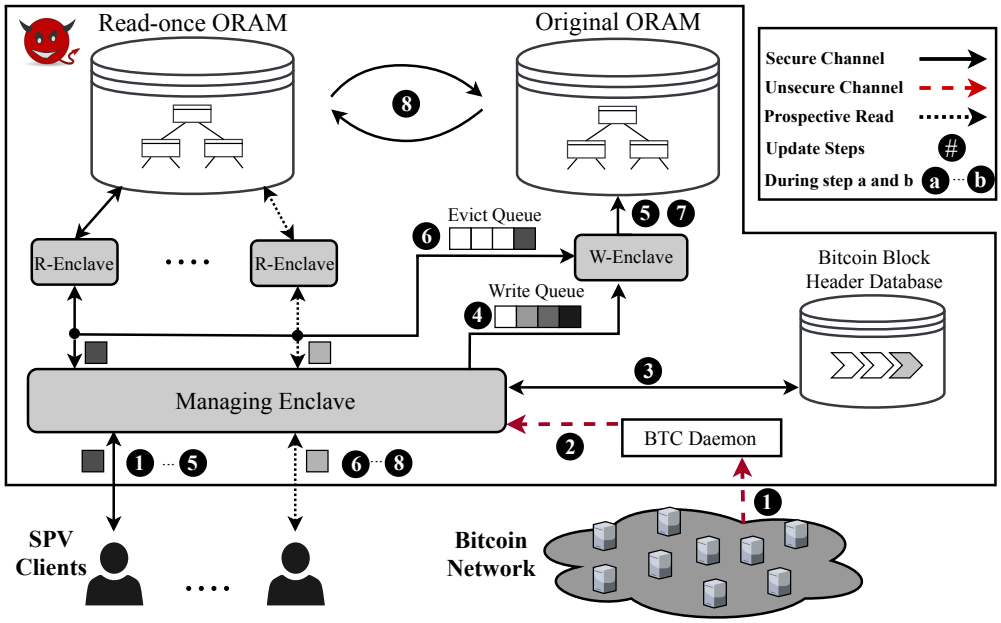


Figure 2.5. T^3 Oblivious Write Protocol.

Oblivious Write Protocol. We explain how T^3 handles oblivious write accesses while handling clients' requests as presented in Fig. 2.5:

1. **The *managing* enclave verifies a new Bitcoin block ❶-❸** : Once a bitcoin block arrives to the system from the Bitcoin network, the *managing* enclave $\mathcal{E}_m$ can obtain it from the Bitcoin client. The enclave needs to verify the integrity of the new block by computing the Merkle root and verifying the proof of work to make sure that the block has not been tampered by the untrusted OS. For the detail of these computations, we refer readers to [21]. Moreover, as discussed in Section 2.3.1, to verify a newly arrived block, the system is required to keep a separate block headers chain with integrity check in the untrusted memory. Once $\mathcal{E}_m$ verifies the bitcoin block, $\mathcal{E}_m$ starts pruning the transactions to obtain relevant information of the transactions' inputs and outputs. Then, $\mathcal{E}_m$ uses OBlockMap$(\cdot,\cdot)$ to find the ORAM block identification to queue up ORAM write requests to the *writing* enclave. During this process, the oblivious read protocol performs as normal on the *read-once* ORAM tree.

2. **The *managing* enclave sends write requests to the *writing* enclave ❹**: Once the pruning process completes, the $\mathcal{E}_m$ starts sending write requests based on data

extracted from the bitcoin block to the *writing* enclave, $\mathcal{E}_w$. On the other hand, for each eviction request resulted from SPV client's requests, $\mathcal{E}_m$ starts queuing up those eviction requests.

3. **The *writing* enclave performs write accesses on the *original* ORAM tree ❹-❺**: Upon receiving writing requests from $\mathcal{E}_m$, the $\mathcal{E}_w$ performs all writing requests in the writing queue on the *original* ORAM tree.

4. **The *writing* enclave finishes all eviction requests queued up on the *original* ORAM tree ❻-❼**: Once finished updating the tree, the $\mathcal{E}_w$ signals $\mathcal{E}_m$ to start queuing up clients' requests and performs all eviction requests incurred by SPV clients' read requests during update interval. Finally, when it finishes, it signals the $\mathcal{E}_m$ to update the *read-once* ORAM tree and make a copy of the position map.

5. **The *managing* enclave performs an update the *read-once* ORAM tree and the *original* ORAM tree and enclave metadata ❽**: In particular, $\mathcal{E}_m$ discards the current copy of the *read-once* ORAM tree, and makes 2 identical copies of the most updated *original* ORAM tree. One is used as *read-once* ORAM tree, and the other is used as *original* ORAM tree. Also, the new position map and new stash are updated for the *managing* enclave. Once this process is finished, $\mathcal{E}_m$ starts answering SPV clients' requests again.

Figure 2.5 gives us an overview of the oblivious write protocol.

2.4 Evaluation and Comparison

In this section, we describe our configuration, our experimental results, and the storage overhead of the system based on the analysis of the UTXO set on the Bitcoin blockchain. Moreover, we give a comparison between T^3 and the current existing SPV solution in term of performance and communication overhead. Finally, we address the capabilities of T^3 compared to other related works.

Table 2.1. Performance of two different types of PATH/CIRCUITORAM accesses on different block size.

		T^3 (**PathORAM**, $Z = 4$)		T^3 (**CircuitORAM**, $Z = 2$)	
N	Block Size	*read-once* Access	Standard ORAM Access	*read-once* Access	Standard ORAM Access
2^{20}	6,528 bytes (96 utxos)	16.34 ms	30.40 ms	2.13 ms	6.45 ms
2^{21}	3,264 bytes (48 utxos)	9.24 ms	16.58 ms	1.27 ms	3.76 ms
2^{22}	2,176 bytes (32 utxos)	7.56 ms	12.42 ms	1.05 ms	2.92 ms
2^{23}	1,088 bytes (16 utxos)	4.12 ms	7.78 ms	0.72 ms	2.09 ms
2^{24}	544 bytes (8 utxos)	2.43 ms	5.89 ms	0.64 ms	1.70 ms

2.4.1 Configuration

Software. We implemented our system with C++ using Intel SGX SDK v2.1.3. The implementation of the ORAM controller is built on top the Zerotrace [145] implementation. In order to handle the communication with the Bitcoin network, we have used `libjson-rpc-cpp` [84] framework to build C++ wrapper functions to communicate with the Bitcoin daemon (`bitcoind` [20]) from inside the enclave through JSON-RPC calls. For extracting the UTXO database, we used the `bitcoin-tool` implementation proposed in [52]. This allows us to save time during the initialization phase. Finally, we used `python-bitcoinlib` [130] to compare the performance of T^3 with the current existing SPV solution.

Database. To reduce the time of initializing both ORAM trees from the *genesis* block, we used `bitcoin-tool` implementation proposed in [52] to extract the Bitcoin UTXO set in February 2019. We have downloaded a snapshot of the Bitcoin blockchain including block 0 to 551,731, containing a total of 58,156,895 Unspent Transaction Outputs (UTXO). Figure 2.6 shows the distribution of the unspent transaction outputs per address. Despite the Bitcoin community's suggestion [2] against the address reuse, we find that more than 7% of the addresses have more than 2 UTXOs. However, to give one the benefit of doubt, we considered at most two UTXOs per wallet ID. This results in covering more than 92% of all the UTXOs per wallet ID. Also, as discussed in section 2.3, by using different mapping, one can cover more percentage of Bitcoin addresses.

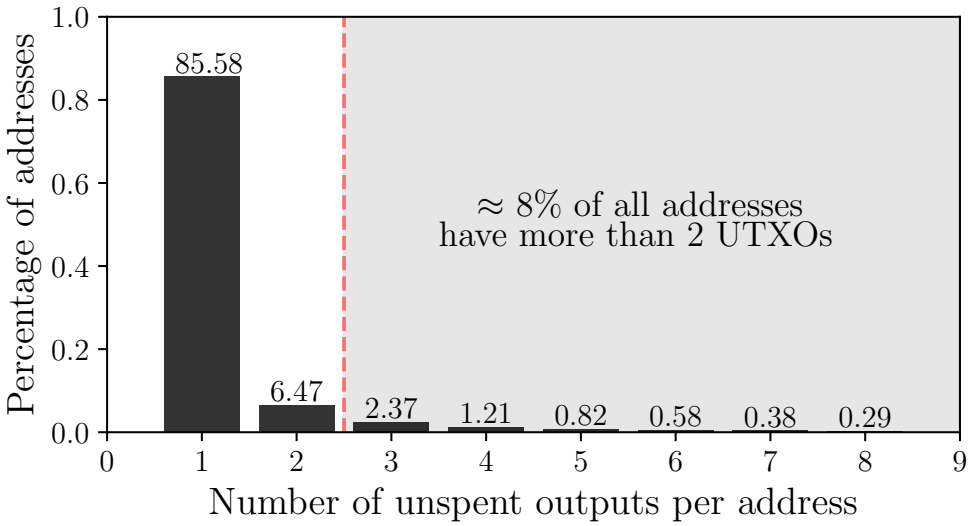


Figure 2.6. Number of transactions per wallet ID. By allowing each address can have up to 2 UTXO, T^3 can cover approximate 92% of the UTXO set.

Hardware. We evaluated the performance of T^3 on a desktop which is equipped with Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz, 128GB RAM. Since Intel(R) Xeon(R) silver 4116 is not SGX-enabled CPU, we obtain the performance results by running our implementation in the simulation mode. However, we expect to not have much of a performance difference when executing in the two different modes. More specifically, we have tested the performance of T^3 using a smaller ORAM tree in the hardware mode on a commodity desktop equipped with SGX-enabled Intel Core i7. Comparing the hardware and simulation mode results (i.e., simulation on the Intel Core i7 CPU), we see no noticeable difference in the running time of both *read-once* and standard ORAM accesses.

2.4.2 Experimental Results

We have implemented the proof of concept of T^3 using multiple threads. As reported in [76, 154], as long as the total amount of memory used by all threads does not exceed the EPC limit, the performance gain should be similar to the use of different enclaves. In this work, we implemented all functionalities in one single enclave, and we used multiple threads to concurrently accesses the ORAM trees.

Table 2.2. Performance gain of multiple-thread *read-once* access on Path ORAM and Circuit ORAM with $N = 2^{24}$ block size = 544 bytes.

Number of Threads	T^3 (PathORAM)	T^3 (CircuitORAM)
1	2.43 ms	0.64 ms
2	1.40 ms	0.58 ms
3	0.90 ms	0.43 ms
4	0.73 ms	0.35 ms

System parameters. We tested our system with both recursive PathORAM and recursive CircuitORAM using different tree size $N = 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}$. We allow each Bitcoin address to have up to 2 unspent transaction outputs, and we use the single address into single ORAM block mapping approach described in Section 2.3.1 to map addresses into ORAM block. Finally, we use Lemma 2.3.1 to determine the size of each ORAM block.

Performance of *read-once* and standard ORAM accesses. In T^3, the *reading* enclave performs *read-once* accesses to handle client's requests in an efficient manner. Table 2.1 presents an overall performance of a standard ORAM access as well as the performance of a *read-once* access for both CircuitORAM and PathORAM. For this experiment, we took the average running time of 10,000 accesses.

As shown in the results, ORAM constructions with smaller block sizes provide a better performance in both schemes. The reason is that oblivious operations like oblivious comparisons and cmov-based stash scan are more efficient because of a smaller size stash. Moreover, CircuitORAM gives a better performance compared to PathORAM, as it can operate on a smaller block compared to PathORAM; therefore, CircuitORAM requires a smaller stash size allowing faster oblivious executions.

Parallelization. Since there is no race condition in *read-once* accesses, the design of T^3 allows different threads to concurrently perform *read-once* accesses on the *read-once* ORAM tree. Compared to other oblivious system like Bite [145], T^3 is able to handle bursty client read requests concurrently while the eviction requests are distributed sequentially during the *block creation interval*. To measure this performance gain, we used multiple threads to access the *read-once* enclave and perform *read-once* access simultaneously on a tree of size $N = 2^{24}$

and ORAM block of size 544 bytes. Table 2.2 shows the performance of T^3 implemented using multiple threads for both CircuitORAM and PathORAM.

Comparison to Current SPV Solutions. We give a comparison in term of performance and communication overhead over several number of requests to the existing SPV client's solution and to BITE [105] Oblivious database.

1. *Performance*: Figure 2.7 gives us an overview of the performance of T^3 compared to the performance of the current existing SPV with Bloom filter solution and the performance of BITE Oblivious database. It shows the response latency from the client's perspective. In this comparison, a request for the SPV solution with Bloom filter solution means the time the full node takes to scan one Bitcoin block, and a request for T^3 and BITE means the time it takes to perform an ORAM access on the ORAM tree. For T^3, we used $N = 2^{24}$ and block of size 544 bytes for both PathORAM with $Z = 4$ and CircuitORAM with $Z = 2$. For BITE database, based on our understanding of their construction, we re-implemented BITE using non-recursive construction of PathORAM, and we used the same ORAM block of size 32kB which leads to the number of block is $N = 2^{17}$. Also, we also provide an additional construction of BITE which is implemented using recursive PathORAM and suggested parameters for T^3 where the tree is of size 2^{24} and block of size 544B. Figure 2.7 gives us the overall performance of three existing solutions.

The performance of T^3 is better than the performance of the SPV with Bloom filter solution because T^3 does not recompute the Merkle path again for each transaction as well as does not use Bloom filter to scan the block. Also, T^3 outperforms BITE oblivious database as the BITE system does not consider the use of recursive ORAM construction. Another reason is that the size of the ORAM block used in BITE is large; hence, the cost of oblivious operation like cmov-based stash scan becomes more expensive. Thus, we envision and realize an improved construction of BITE using recursive construction of PathORAM to demonstrate the practical impact of using recursive ORAM construction on TEE with restricted memories.

2. *Communication Overhead*: In term of communication between client and full node, T^3 offers much lower communication overhead compared to the existing solution for SPV

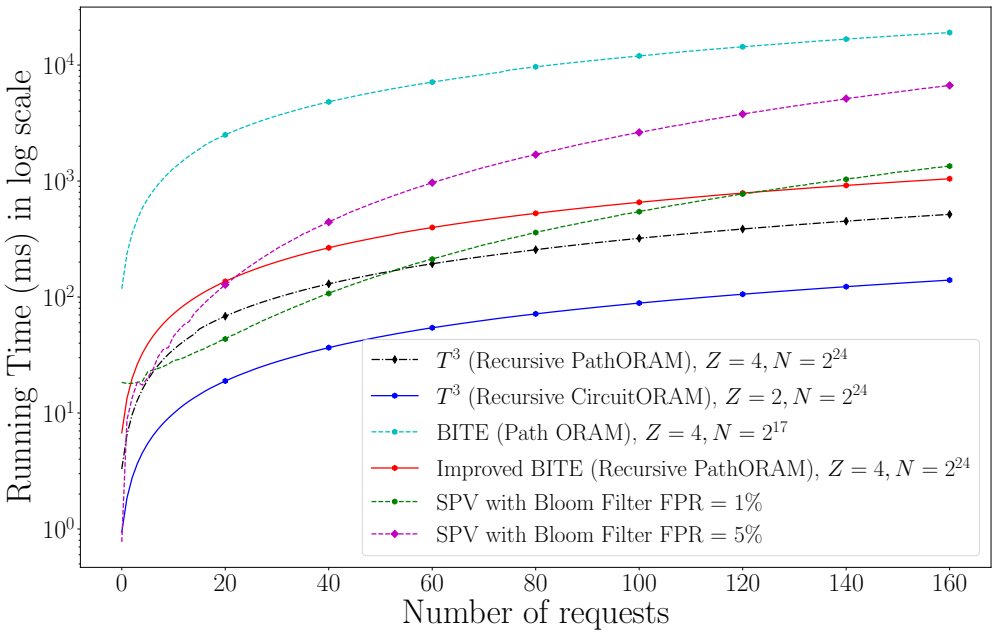


Figure 2.7. Performance of T^3 using PATH/CICRUIT ORAM with block of size 544B, the current SPV with Bloom filter, Original BITE oblivious database block of size 32kB, and improved BITE with block of size 544B. For the SPV client with Bloom filter, we used the false positive rate of 1% and 5%.

clients. T^3 does not need to provide the SPV clients with the Merkle proofs to its relevant transactions because all those proofs are validated by the Intel SGX before being added the ORAM tree. Also, due to the false positive rate used in the Bloom filter, the traditional full node will send additional irrelevant information to the SPV client. Figure 2.8 shows an overview of the communication cost of T^3 compared to the current solution. To give an estimation of the communication cost of the current SPV solution, we assumed that each request requires a separate Merkle proof. Moreover, we set the size of the transaction data is approximately $\mathsf{fpr} \cdot \mathsf{BlockSize}$ bytes where the fpr is the false positive rate and the $\mathsf{BlockSize}$ is the size of the Bitcoin block. To obtain estimation, we used block 551,731 that has the block size of 1,149 KB and contains 3,017 transactions. In practice, we would expect the Bitcoin blocks to have different sizes which results the communication cost to be different across blocks. Therefore, the results in Fig. 2.8 is only a pessimistic estimation on the communication overhead using the current SPV solution. We omit the comparison to the

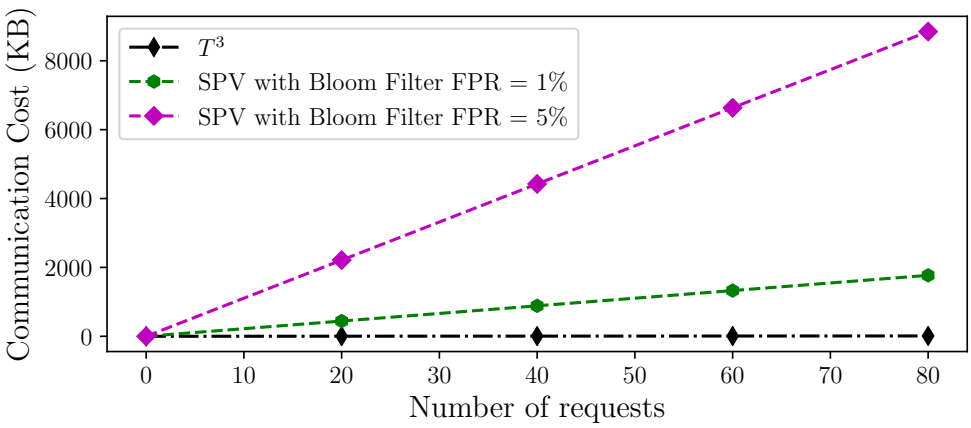


Figure 2.8. Communication cost of T^3 and the current SPV solution. Since both systems return the information of unspent outputs to the client, the communication overhead of BITE will be equal to the communication overhead of T^3.

communication overhead of BITE because both T^3 and BITE return a fixed amount a data to the SPV clients.

Storage Overhead. As noted in the previous section, using ORAM incurs a constant size blow up of the storage of the UTXOs (e.g., $\approx 4\times$ for CircuitORAM, 6-8$\times$ for PathORAM). In particular, for PathORAM with $Z = 4$, the storage cost of ORAM trees is about $\approx 51GB$, and for CircuitORAM with $Z = 2$, the storage cost of two ORAM tree is around $\approx 26GB$. For integrity protection, T^3 only requires the full node to store the Bitcoin header chain with integrity check which is approximately 44MB in the untrusted region.

2.4.3 Comparison with Other Oblivious Systems

We compare T^3 with Bite [105] Oblivious Database that also uses ORAM and TEE to provide a generic PIR system for Bitcoin client, ConcurORAM [38] that provides concurrency access to ORAM clients, Obliviate [3] that prevents leakage from file system accesses, and ZeroTrace that proposes an efficient generic oblivious memory access primitives. Section 2.4.3 compares those systems based on the capabilities of supporting concurrency access, enabling recursive construction, and preventing side-channel leakage.

Table 2.3. Comparison between T^3 and other oblivious systems.

System	Capabilities		
	Concurrency	Recursive Construction	Side-channel Protection
ConcurORAM [38]	✓	✗	- [a]
Obliviate [3]	✗	✗	✓
Zerotrace [145]	✗	✓	✓
Bite Oblivious Database [105]	✗	✗	✓
T^3 (This work)	✓	✓	✓

[a]↑ConcurORAM does not aim to provide side-channel protection for TEE. Hence, we omit this comparison.

For generic trusted hardware-based systems like Bite oblivious database and Obliviate, while providing protection against side-channel leakage, those systems do not consider the use of recursive ORAM construction to reduce the EPC memory usage. Hence, the performance of those systems will degrade once the database becomes too large. Other works that harnesses the use of recursive ORAM construction are Zerotrace; however, concurrency is not supported in the current version of Zerotrace. ConcurORAM is a recent ORAM construction that offers concurrency accesses from the clients; however, due to more optimized eviction strategy and complex synchronization schedule, the recursive construction of ConcurORAM introduces implementation challenges.

2.5 System Analysis

2.5.1 Security Claims

In order to prove the security properties of T^3's design, we put forth six claims, each of which represents the security of a major component of T^3 in term of privacy goal.

Claim 2.5.1. *The managing enclave does not leak user-related information to an attacker*

The managing enclave is responsible for three tasks — (a) converting wallet IDs to UTXOs, (b) creating and managing threads which will perform read operations on the *read-once* ORAM tree, and (c) handle the updates to be performed on the *original* ORAM tree. Firstly, the conversion of wallet IDs to their respective UTXOs is private since the channel between clients and the *managing* enclave is secured by the shared key during the remote

attestation process. When receiving addresses from a client, the *managing* enclave uses blockmapping function to map each address to a fixed number of ORAM blocks. This does not reveal information about the number of outputs belonging to an address. Secondly, each read thread performs the same operations irrespective of the wallet ID provided to it, i.e., each thread simply retrieves an ORAM block using ORAM accesses implemented with cmov-based oblivious executions. Lastly, the only thing revealed by the update process of T^3 is the number of blocks updated into the Write Tree. However, this is public information and T^3 does not try to hide it. Each update is performed using an ORAM access which ensures that the attacker is unaware of the final position of each block.

Claim 2.5.2. *The optimized read operations on read-once ORAM tree do not leak information.*

As explained in section 2.3, the *read-once* ORAM tree is accessed using an optimized read operation which chooses not to shuffle and write-back the retrieved path to the *read-once* ORAM tree. However, as suggested by the Bitcoin protocol and the analysis of the UTXO set shown in Section 2.4.1, most of the addresses are generated once only to receive new output from the sender. Thus, the read operations are secure as each path corresponding to a UTXO should only be accessed once during a read interval and will be shuffled before the next interval. On the other hand, the leakage happens only when the client queries the same address again; however, the client does not need to request again as there are no new transactions for the next block creation interval.

Claim 2.5.3. *The write operations performed on the original ORAM tree do not leak information.*

There are two specific operations performed on the *original* ORAM tree— (a) the UTXOs are updated based on the updated bitcoin block, and (b) the previously accessed ORAM blocks are shuffled. However, all of these updating accesses are standard ORAM operations implemented in a side-channel-resistant manners as previously done by [145, 3]. Therefore, all write operations reveal no information about a user's UTXO.

Claim 2.5.4. *The data fetched from the untrusted world to the TEE is correct.*

There are two major sources of data transferred from the untrusted to the trusted world — (a) the updated block fetched from the Bitcoin daemon after a fixed interval and (b) the ORAM tree blocks which are fetched from the untrusted world into the TEE. As mentioned, the enclave obtains Bitcoin blocks from outside the enclave. However, T^3 verifies the integrity of the Bitcoin block based on the proof of work and the header chain, and since the cost of producing a valid block is expensive, we argue that T^3 should be able to obtain valid block from the Bitcoin network. Also, T^3 maintains a Merkle Hash Tree (MHT) of the ORAM trees and therefore prevents malicious tampering by verifying all encrypted data fetched from the untrusted memory using the MHT.

Claim 2.5.5. *The multiple threads involved do not create synchronization issues.*

It is worth-noting that multiple threads are only involved while accessing the Read Tree of T^3. Thanks to the optimized read operation, T^3 does not run into synchronization bugs since there is no memory region that could be simultaneously written to by more than one thread. Each thread shares the position map but only reads from the position map. Each thread contains its own stash memory which is written to separately by each thread.

Claim 2.5.6. *The memory interactions within the enclave are side-channel-resistant.*

The design of T^3 incorporates different defenses against the side-channel threats [161, 94, 95] plaguing Intel SGX. In particular, we used ORAM operations to hide all data access patterns on the untrusted memory region, and we incorporated similar oblivious operation techniques introduced in [133, 3, 145] to prevent operations inside the enclave from leaking sensitive information. Finally, the implementation of T^3 is also secure against branch-prediction attacks since each individual operation (e.g., accessing Read Tree, updating Write Tree etc.) takes the same sequence of branches and therefore reveals no information to the attacker, from the accessed branches.

2.5.2 Other Goals Achieved by T^3

In this subsection, in addition to the **Privacy** goal describe in Section 2.5.1, we explain how T^3 achieves the other goals mentioned in Section 2.1.2.

Validity. Under the assumption that the adversary does not have enough computational power to form a new Bitcoin block, the system will only obtain valid transaction by verifying the Merkle root and the proof of work of the Bitcoin block.

Completeness. By offering different ways of mapping between Bitcoin addresses and ORAM block id, we can offer services to $92 - 96\%$ of all clients with some trade-off between storage overhead and performance.

Efficiency. Our contribution to efficiency is threefold. First, our system can handle bursty requests from client concurrently because of the two-tree design. Second, we minimize the downtime of the system by having the writing enclave performed updates on one tree and reading enclave handled clients' requests on the other tree. The full node's downtime now depends on the number of requests that the system receives when the writing enclave performs ORAM updates on the original ORAM tree. Finally, by enforcing clients to provide the proof of ownership of the address, we prevent other clients from querying addresses that do not belong to them; hence, we reduce the number of redundant requests from the clients.

2.5.3 Other attacks and Countermeasures

Denial of Service Attacks from Malicious Clients. While the design of T^3 is practical, a malicious client can still incur a large processing time on the full node by creating lots of addresses and sending large number of requests for those requests. One way to mitigate such attack is to apply fees on users of the service. Another approach to mitigate denial of service attack is to use a cuckoo filter [61] to load and delete unspent addresses from the UTXO set upon update. Upon receiving requests from client, the managing enclave can verify if the address matches the filter as well as the proof of ownership of that address before performing ORAM accesses.

Spectre and Related Attacks [99, 89]. T^3 can employ any TEE which is vulnerable to digital side-channels (i.e., access pattern-inference attacks such as page table, cache, branch prediction, etc.) but is secure against micro-architectural defects (i.e., reading memory contents directly from the TEE). Speculative execution attacks, which fall into the micro-architectural defects category, is a concern; however, Intel has recently released hardware

patches to address those. Therefore, T^3 can be effectively used alongside patched processors to provide SPV client protections against digital side-channel attacks.

2.6 Concluding Remarks

In this chapter, we developed a system design that supports an efficient oblivious search on unspent transaction outputs for Bitcoin SPV clients while securely maintains the state of the Bitcoin UTXO set via an oblivious update protocol. Our design leverages the TEE capabilities of Intel SGX to provide strong privacy and security guarantees to Bitcoin SPV client even with the presence of a potentially malicious full node. Moreover, by putting reasonable assumptions on the accessing frequency of the SPV clients, we present different optimizations in standard tree-based ORAM construction that offers both privacy and efficiency to the clients. We showed that the prototype of the system is much more efficient than the use of standard ORAM and TEE construction as it is. Also, our implementation shows one order of magnitude performance gain when combining recursive ORAM construction the current existing construction to stress the importance of using recursive ORAM construction in TEE with restricted memory.

Finally, while the applicability of T^3 in cryptocurrencies beyond Bitcoin is apparent, we believe our work will motivate further research on oblivious memory with the restricted access patterns and other complex blockchains (i.e., Ethereum) that maintain much bigger state than the UTXO state.

3. AUTONOMOUS ADD-ON PRIVACY COIN MIXER, AMR

More than a decade after the emergence of permissionless blockchains, such as Bitcoin, related work has thoroughly shown that the blockchain's pseudonymity is not offering its clients strong anonymity. Several works have therefore attempted to deanonymize clients, cluster addresses [6, 68], and build privacy solutions to protect the clients' privacy [114, 144, 5, 140, 139, 78]. Those existing privacy solutions can be categorized into two classes: *(i)* a fundamental blockchain redesign to natively offer better privacy to clients, and *(ii)* add-on privacy solutions that aim to offer privacy for clients of existing, non-privacy-preserving blockchains. However, the former class of solutions often requires existing clients of different blockchain to download and store additional states of private-by-design blockchains. This additional storage can be costly for existing resource-constrained clients of non-private blockchains if they want to be a part of both blockchains. On the other hand, a complete switch to private blockchain will prevent constrained clients from accessing decentralized financial applications enabled by smart-contract enabled blockchains.

Thus, this work focuses on add-on privacy solutions that mix cryptocurrency coins within an anonymity set. This solution directly allows existing clients of non-private blockchains to obtain privacy without the need of switching or storing additional blockchain state. However, one known problem of such mixers is that their provided privacy depends on the anonymity set size, i.e., on the protocol's number of active clients. Also, in those systems, to gain a certain degree of privacy, users often need to keep their digital assets locked in the system for a certain period before withdrawing. This locking period prevents users from performing any financial activities on those assets, i.e., there is an opportunity loss of investing the assets for a financial return.

Hence, this work's particular focus is to find new ways to incentivize existing clients of non-private to participate and keep their funds in the mixer. First, like to popular "DeFi farming" protocols [42], our system, called AMR, chooses to reward mixer participants by granting governance tokens when a client deposits coins for at least time t within the mixer. Naturally, the reward payout must remain privacy-preserving, i.e., a reward payment must be unlinkable to a deposit from the same client of the mixer. Clients can utilize the collected

tokens to govern AMR in a decentralized manner, without the need for an external server or centralized entity. Secondly, by leveraging existing popular lending platforms [1, 41, 162], AMR can allow clients to earn interest on their deposited funds. This approach makes AMR the first mixer design that generates financial interest on participants' funds. We hope that such a mixer attracts clients that are privacy-sensitive and interested in a token reward to maximize the anonymity set and client diversity within AMR.

We formalize the zk-SNARK-based AMR system, and implement the mixer in $1,013$ lines of Solidity code. A deposit costs $1.2m$ gas (31.95 USD), while a withdrawal costs $0.3m$ gas (9.12 USD), receiving a reward costs to $1.5m$ gas (41.07 USD) [1] [2]. These numbers support a real-world deployment, that could support over $66,000$ deposits per day given Ethereum's transaction throughput (assuming no withdrawals). The resulting anonymity set sizes, which can easily exceed $1,000$ while operating at constant system costs, offer stronger privacy than, e.g., the ring signature-based privacy solution [108], whose costs scale linearly with the size of the anonymity set and are hence practically capped at anonymity set sizes of 24 ($8m$ gas for withdrawing).

Our contributions can be summarized as follows.

- We formalize and present a practical zk-SNARK based mixer AMR, which breaks the linkability between deposited and withdrawn coins of a client on a smart contract enabled blockchain, and we provide a formal security and privacy analysis of the proposed system.

- To decentralize AMR's governance and incentivise clients to join the system, we invent a privacy-preserving reward scheme for its clients. We believe that in practice, an incentive scheme would attract more and a wider variety of clients to such privacy solution, and hence contribute to a better anonymity for all involved clients.

- We leverage popular existing lending platforms [1, 41] to propose the first autonomous decentralized on-chain mixer that allows users to earn interest on their deposited fund. This approach further incentivises users to keep their funds in the system.

[1]↑Estimated using Ethererum price of $380.4 in 08/25/2020 14:39 UTC.
[2]↑Using the gas price of 70 Gwei. 1 GWei is 1×10^{-9} Ether.

- We implement AMR and show that the system can be deployed and operated efficiently on a permissionless blockchain. A deposit into the system costs $1.2m$ (31.95 USD), a withdrawal costs $0.3m$ gas (9.12 USD) and collecting a reward costs $1.5m$ gas (41.07 USD) in transaction fees on the current Ethereum network. The anonymity set size of AMR could grow to up to 2^d [3], while operating at constant system costs once deployed (we applied a Merkle depth tree of $d = 30$ within this evaluation). Generating client-side zkSnark proofs costs 3.607 seconds respectively on commodity hardware.

3.1 Preliminaries

In this section, we define several building blocks for AMR.

3.1.1 Background on Smart Contract Blockchains and Lending Platforms

Ethereum Blockchain. The Ethereum blockchain acts as a distributed virtual machine that supports quasi-Turing-complete programs. The capability of executing highly expressive languages in those blockchains enables developers to create *smart contract*. The blockchain also keeps track of the state of every account [160], namely *externally-owned accounts (EoA)* controlled by a private key, and *contract account* own by contract's code. Transactions from EoA determine the state transitions of the virtual machine. Transactions are either used to transfer Ether or to trigger the execution of smart contract code. The costs of executing functions are expressed in terms of *gas* unit. In Ethereum, the transaction's sender is the party that pays for the cost of executing all contract operations triggered by that transaction. For a more thorough background on blockchains, we refer the interested reader to [27, 9].

Lending Platforms on Ethereum Blockchain. Smart-contract-enabled blockchains like Ethereum give rise to many other decentralized financial (Defi) applications. Defi applications allow parties to participate in the financial market without relying on any trusted third party while retaining full custody of their funds. Defi applications appear in different forms, such as decentralized exchanges, lending platforms, or derivatives. At the time

[3] $\uparrow d$ is the depth of the Merkle tree

of writing, the Defi space accumulates over 10bn dollars of digital assets, and hundreds of millions of dollars of assets are traded daily in those Defi platforms.

For this work, we focus on existing lending protocols [1, 41]. At its core, lending protocols let *borrowers* acquire digital assets with a specified interest rate by placing upfront collaterals into the system. Later, to retrieve the collaterals, *borrowers* need to pay back the borrowed funds along with an additional interest amount. Similarly, users also act as *lenders* by depositing digital assets into the protocol, and the deposited amount will generate interest until users redeem those assets. Finally, the interest rates for borrowing and lending are determined by the state of the lending platforms. In this work, we are only interested in the depositing and redeeming functionalities of lending platforms.

Definition 3.1.1. *A lending protocol, Σ, reserves the following actions:*

- $\mathsf{amt}_\Sigma \leftarrow \textsc{Deposit}(\mathsf{amt})$ *takes as input of* amt *of coins, and outputs a corresponding amount of* amt_Σ *tokens.* amt_Σ *tokens are minted upon deposits and sent to the depositor, and the value of* amt_Σ *increases over time.*

- $\mathsf{amt} + R \leftarrow \textsc{Redeem}(\mathsf{amt}_\Sigma)$ *takes as input* amt_Σ *tokens, and deposits* $\mathsf{amt} + R$ *to the function invoker. The interest amount R is determined by protocol Σ.*

This definition aims to capture a high-level overview of how the depositing and redeeming functionalities work in a lending platform. For a detailed constructions of each actions in these lending protocols, we refer interested readers to [1, 41, 162].

Governance Token and Yield Farming in Decentralized Finance (DeFi). Users of DeFi platforms are often awarded governance tokens for interacting or providing liquidity to DeFi platforms. These tokens can for instance be used for governance and value accrual/yield farming. Governance means that users can use their tokens to vote for changes in the contract during its lifetime. In term of value accrual, platforms [47, 41] allow users to lock their governance tokens in a pool to be eligible to obtain trading fees collected by the DeFi platform. This approach allows a fair distribution of protocol fees to users who take on the opportunity cost of holding the governance tokens. In this work, we adapt a similar technique of having a distribution pool to fairly distribute total accrued interest collected by the mixer to users.

3.1.2 Cryptographic Primitives

Notation. We denote by 1^λ the security parameter, by $\mathsf{negl}(\lambda)$ a negligible function in λ, and by $\mathsf{poly}(\lambda)$ a polynomial function in λ. We express by $(\mathsf{pk}, \mathsf{sk})$ a pair of public and secret keys. Moreover, we require that pk can always be efficiently derived from sk, and we denote $\textsc{extractPK}(\mathsf{sk}) = \mathsf{pk}$ to be the deterministic function to derive pk from sk. $k||r$ denotes concatenation of two binary string k and r. We denote $\mathbb{Z}_{\geq a}$ to denote the set of integers that are greater or equal a, $\{a, a+1, \dots\}$. We let PPT denote probabilistic polynomial time. We use $st[a, b, c \dots]$ to denote an instance of the statement where $a, b, c \dots$ have fixed and public values. We use a shaded area i, j, k to denote the private inputs in the statement $st[a, b, c; i, j, k]$.

Collision Resistant Hash Function. a family H of hash functions is collision resistant, iff for all PPT $\mathcal{A}$ given $h \xleftarrow{\$} H$, the probability that $\mathcal{A}$ finds x, x', such that $h(x) = h(x')$ is negligible. we refer to the cryptographic hash function h as a fixed function $h : \{0,1\}^* \to \{0,1\}^\lambda$. For the formal definitions of cryptographic hash function family, we refer reader to [137].

zk-SNARK. A zero-knowledge Succinct Non-interactive ARgument of Knowledge (zk-SNARK) can be considered as "succinct" NIZK for arithmetic circuit satisfiability. For a field $\mathbb{F}$, an arithmetic circuit C takes as inputs elements in $\mathbb{F}$ and outputs elements in $\mathbb{F}$. We use the similar definition from Sasson *et al.*'s Zerocash paper [144] to define arithmetic circuit satisfiability problem. An arithmetic circuit satisfiability problem of a circuit $C : \mathbb{F}^n \times \mathbb{F}^h \to \mathbb{F}^l$ is captured by relation $R_C = \{(st, \mathsf{wit}) \in \mathbb{F}^n \times \mathbb{F}^h : C(st, \mathsf{wit}) = 0^l\}$; the language is $\mathcal{L}_C = \{st \in \mathbb{F}^n \mid \exists\ \mathsf{wit} \in \mathbb{F}^l\ s.t\ C(st, \mathsf{wit}) = 0^l\}$.

Definition 3.1.2. *zk-SNARK for arithmetic circuit satisfiability is triple of efficient algorithms* $(\textsc{Setup}, \textsc{ZKProve}, \textsc{ZKVerify})$*:*

- $(\mathsf{ek}, \mathsf{vk}) \leftarrow \textsc{Setup}(1^\lambda, C)$ *takes as input the security parameter and the arithmetic circuit* C*, outputs a common reference string that contains the evaluation key* ek *later used by prover to generate proof, and the verification key* vk *later used by the verifier*

to verify the proof. The public parameters, pp*, is given implicitly to both proving and verifying algorithms.*

- $\pi \leftarrow \textsc{Prove}(\mathsf{ek}, st, \mathsf{wit})$ *takes as input the evaluation key* ek *and* $(st, \mathsf{wit}) \in R_C$*, outputs a proof* π *that* $(st, \mathsf{wit}) \in R_C$

- $0/1 \leftarrow \textsc{Verify}(\mathsf{vk}, \pi, st)$ *takes as input the verification key, the proof* π*, the statement* st*, outputs* 1 *if* π *is valid proof for* $st \in \mathcal{L}_C$*.*

In additional to *Correctness, Soundness,* and *Zero-knowledge* properties, a zk-SNARK requires two additional properties *Succinctness* and *Simulation Extractability.* We defer the definitions of these properties to [75].

Commitment Scheme. A commitment scheme allows a client to commit to chosen values while keeping those values hidden from others during the committing round, and later during the revealing round, client can decide to reveal the committed value.

Definition 3.1.3. *A commitment scheme* $\textsc{Com} = (P_{\textsc{Com}}, V_{\textsc{Com}})$ *consists of: A* committing *algorithm* $P_{\textsc{Com}}(m, r)$ *takes as input a message* m *and randomness* r*, and outputs the commitment value* c*. A* Reveal *algorithm,* $V_{\textsc{Com}}(c, m, r)$ *takes as input a message* m*, and the decommitment value* r *and a commitment* c*, and returns 1 iff* $c = P_{\textsc{Com}}(m, r)$*. Otherwise, returns 0.*

We use commitment schemes that achieve two properties: *binding* means that given commitment c, it is difficult to find a different pair of message and randomness whose commitment is c, and *hiding* means that given commitment c, it is hard to learn anything about the committed message m from c.

Authenticated Data Structure (ADS). An authenticated data structure can be used to compute a short digest of a set $X = \{x_1, \ldots, x_n\}$, so that later one can prove certain properties of X with respect to the digest. In this work, we are only interested in a data structure for set membership:

Definition 3.1.4. *An authenticated data structure for set membership* $\Pi = (\textsc{Init}, \textsc{Prove}, \textsc{Verify}, \textsc{Update})$ *is a tuple of four efficient algorithms:*

- $y \leftarrow \textsc{Init}(1^\lambda, X)$ *the initialization algorithm takes as input the security parameter and the set* $X = \{x_1, \ldots, x_n\}$ *where* $x_i \in \{0,1\}^*$*, output* $y \in \{0,1\}^\lambda$.
- $\pi \leftarrow \textsc{Prove}(i, x, X)$ *takes as input an element* $x \in \{0,1\}^*, 1 \leq i \leq n$*, and set* X*, outputs a proof that* $x = x_i \in X$.
- $0/1 \leftarrow \textsc{Verify}(i, x, y, \pi)$ *takes as input* $1 \leq i \leq n, x \in \{0,1\}^*, y \in \{0,1\}^\lambda$*, and proof* π*, output* 1 *iff* $x = x_i \in X$ *and* $y = \textsc{Init}(1^\lambda, X)$*. Otherwise, return 0.*
- $y' \leftarrow \textsc{Update}(i, x, X)$ *takes as input* $1 \leq i \leq n, x \in \{0,1\}^*$ *and set* X*, output* $y' = \textsc{Init}(1^\lambda, X')$ *where* X' *is obtained by replace* $x_i \in X$ *with* x.

We require the ADS to be *correct* and *secure*. We defer the formal definitions of these properties to Boneh and Shoup's book [25]. Typical examples of authenticated data structures are Merkle tree [112] or RSA Accumulators [96, 17].

3.2 System Overview

We proceed to define the system components, overview, goals and the threat model.

3.2.1 System Components

There are three components of this system: the client, the AMR smart contract, and onchain lending platforms. A **Client** interacts with the AMR smart contract through externally owned accounts. A client can either deposit coins, withdraw coins, or redeem a reward. The AMR **Contract** is the blockchain smart contract that holds deposits and handles withdrawals and reward redemptions. The contract keeps track of different data structures and parameters to verify the correctness and the integrity of transactions sent to the contract. The AMR **Pool** is a smart contract that takes the accrued interest from a lending platform and proportionally distributes the reward among clients who lock their governance tokens to the pool. The **Lending Platforms** (cf. Section 3.1.1) are smart contracts that allow users to deposit digital assets and earn interest based on those assets.

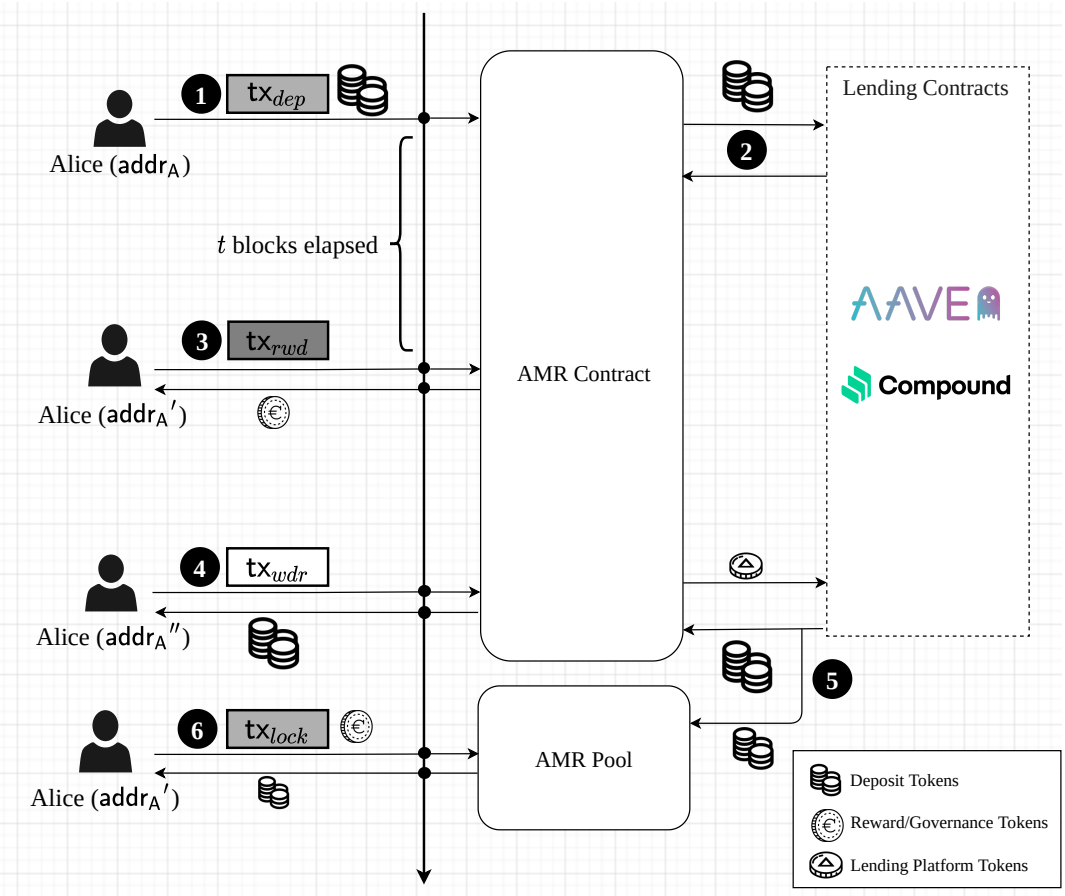


Figure 3.1. AMR System Overview.

3.2.2 Overview of AMR

Figure 3.1 outlines the overview of interactions in AMR.

Deposits. In AMR, clients deposit a fixed amount of coins into the system. The client forms a depositing transaction to deposit coins, then sends this transaction through the P2P Network ❶. Once the transaction is validated, miners record the transaction in a blockchain block. Each deposit transaction decreases the balance of the clients' address by a fixed amount of coins. In step ❷, upon receiving a valid deposit from the user, AMR deposits users fund into lending platforms to obtain an equivalent amount of tokens for future withdrawals.

Reward Redemptions. AMR allows clients to earn governance tokens as rewards based on certain conditions. In Figure 3.1, the requirement for a client to redeem a reward is to

keep the deposit inside the contract pool for t blocks. To obtain a reward, a client forms a redeeming transaction and forwards the transaction to the P2P Network ❸. The redeeming transaction includes a cryptographic proof certifying that the client has deposited coins at least t blocks in the past and that the coins remain in the AMR contract. Finally, miners validate the redeeming transaction using the current state of the AMR contract. Once the redeeming transaction gets validated, the transaction gets recorded to a blockchain block, and the network updates the state of the AMR contract.

Withdrawals. The client forms a withdrawing transaction to withdraw coins, then sends this transaction through the P2P Network ❹. The withdrawing transaction includes cryptographic proof certifying that the client has issued a depositing transaction in the past without revealing precisely which one the depositing transaction is. In step ❺, upon receiving valid withdrawing transactions from the client, the contract autonomously redeems the original deposit from lending platforms and the accrued interest. Finally, the contract deposits the redeemed amount into user's address and the accrued interest into a separate AMR pool.

Fair Interest Allocation. At any given time, clients can lock their governance tokens to the AMR pool ❻. AMR distributes the total accrued interest to addresses that lock their AMR governance tokens in AMR pool. This step is straightforward but offers a fair allocation of interest to users who contribute more to AMR's privacy set.

3.3 AMR System

In the following, we discuss various components of the AMR system and provide more details of how AMR operates. In the following algorithm descriptions, we use `tx.sender` to denote the address of the sender from which `tx` was sent.

Condition for Reward in AMR. The longer time the clients wait before withdrawing/redeeming, the more deposit transactions are issued to the AMR contract. Thus, as the number of deposit transaction (i.e., the anonymity set) increases, the harder it is to link a withdrawing/redeeming transaction with the original deposit transaction. In AMR, we incentivise clients by providing rewards to clients who can prove that the deposit funds

are not withdrawn before a certain time, measured in several blocks. The provided reward can for instance represent a governance token for a client to participate in the decentralized governance of AMR parameters.

3.3.1 AMR Contract Setup

The setup phase generates public parameters and data structures for the AMR contract and clients. All cryptographic parameters are generated for the contract. The contract is also initialized with different data structures to prevent clients from double-withdrawal and double-redemption. The deposit and reward amounts, amt and amt_{rwd}, are specified as a fixed deposit number of coins and a fixed reward amount of governance tokens. The condition for redeeming rewards, t_{con}, is also declared. A lending platform, Σ, (cf. Section 3.1.1) is determined during this setup phase. A pool, Γ_{AMR}, is deployed, and this pool periodically distributes the accrued interest to addresses that lock governance tokens.

We denote pp^h to be the state of the contract at block h. The state contains all data structures initialized during the setup phase. Moreover, this state is given implicitly to all clients' and contract's algorithms. Finally, the AMR contract is deployed during this phase.

3.3.2 AMR Client Algorithms

In our system, clients have access to the following algorithms to interact with the AMR smart contract. Also, all transactions are implicitly signed by the client using the private key of the Ethereum account that creates the transaction.

- $(\mathsf{wit}, \mathsf{tx}_{dep}) \leftarrow \textsc{CreateDepositTx}(\mathsf{sk}, \mathsf{amt})$ takes as input the private key sk and the amount, amt, coins specified in the setup phase, outputs a deposit transaction tx_{dep} and the secret note wit which is used as witness for creating future withdraw and reward transactions.

- $(\mathsf{wit}', \mathsf{tx}_{rwd}) \leftarrow \textsc{CreateRedeemTx}(\mathsf{sk}', \mathsf{wit})$ takes as input a private key sk' and the secret note wit, outputs a reward-redeeming transaction tx_{rwd} along with a new secret note, wit'.

- $\mathsf{tx}_{wdr} \leftarrow$ CreateWithdrawTx($\mathsf{sk}'', \mathsf{wit}$) takes as input a private key sk'' and the secret note wit, outputs a withdrawing transaction tx_{wdr}.

- $\mathtt{tx}_{lock} \leftarrow$ CreateLockTransaction($\mathsf{sk}', \gamma_{rwd}, t_{lock}$) takes as input an amount, γ_{rwd}, of governance tokens and an unlock value, t_{lock}, specifying how long, γ_{rwd}, will remain locked in the system, outputs a locking transaction, $\mathtt{tx}_{lock}$.

3.3.3 AMR Contract Algorithms

The AMR contract should accept the deposit of funds, handle withdrawals, and reward redemptions. Summarizing, the AMR contract should provide the following functionalities.

- $0/1 \leftarrow$ AcceptDeposit(tx_{dep}) takes as input the deposit transaction tx_{dep}. The AMR contract deposits amt into the lending platform Σ to obtain amt_{Σ}. Finally, the algorithm outputs 1 to denote a successful deposit, otherwise 0.

- $0/1 \leftarrow$ IssueWithdraw(tx_{wdr}) takes as input the withdraw transaction tx_{wdr}. The AMR contract uses amt_{Σ} to redeem $\mathsf{amt}+R$ from Σ. The algorithm outputs 1 to denote a successful withdraw and deposits $\mathsf{amt} + R$ into $\mathsf{tx}_{wdr}.\mathsf{sender}$. Otherwise, outputs 0.

- $0/1 \leftarrow$ IssueReward(tx_{rwd}) takes as input the reward transaction tx_{rwd} and the condition t_{con} specified during the setup algorithm, outputs 1 if tx_{rwd} satisfies the t_{con} for reward and deposit amt_{rwd} governance tokens as reward to $\mathsf{tx}_{rwd}.\mathsf{sender}$. Otherwise, output 0.

3.3.4 System Goals

In the following, we outline our system goals.

Correctness. Generally, AMR needs to ensure that clients should not be able to steal coins from the AMR contract or from other clients. Moreover, we design AMR such that clients can redeem a reward after they have deposited their coins into the AMR contract for certain period of time, as a reward system will incentivise clients to deposit more into the system while contributing to the size of the anonymity set.

AMR needs to provide the following guarantees: *(i)* It is infeasible for clients to issue n withdrawal transactions without issuing at least n deposit transactions into the AMR contract beforehand. *(ii)* It is infeasible for a client to issue a redeeming transaction without having any coins locked in the AMR contract. *(iii)* A valid redeeming transaction indicates that a client always has at least one deposit locked in the AMR contract for a specified duration.

Privacy. In addition to correctness, AMR needs to ensure the privacy to clients of the system. Considering an adversary that has access to the history of all depositing, withdrawing, and redeeming transactions sent to AMR contract, the system needs to ensure *(i)* the unlinkability between deposit and withdrawing transactions *(ii)* the unlinkability between deposit and redeeming transactions *(iii)* the unlinkability between withdrawing and redeeming transactions.

Availability. Like to the availability definition proposed by Meiklejohn and Mercer's Möbius system [108], AMR should ensure that *(i)* no one can prevent clients from using the mixer, and *(ii)* once the coins are deposited to the contract, no one can prevent clients from withdrawing their coins.

Frontrunning Resilience. Some transactions (i.e., deposit transactions) in AMR alter the state of the AMR contract, while other transactions (i.e. withdrawing/redeeming transactions) have to rely on the state of the contract to form the cryptographic proofs. Thus, if there are multiple concurrent deposit transactions issuing to the contract, some transactions will get invalidated by those transactions that modify the state of the contract. For example, in AMR, to withdraw or redeem a reward, a client Alice has to issue a withdrawal and a redemption transactions containing cryptographic proofs proving that Alice deposited a coin in the past. Alice generates those cryptographic proofs w.r.t all current deposit transactions issued to the AMR contract. However, if another client Bob tries to deposit coins into the AMR contract, and Bob's transaction gets mined before Alice withdrawing/redeeming transactions, the proofs included in Alice transactions are no longer valid (because the state used for her proofs is outdated). This is a *front-running* problem [35, 59].

Therefore, to ensure the usability of the system, the AMR contract should be resilient against *front-running* by both clients and miners.

3.3.5 Threat Models

We assume that the cryptographic primitives (cf. Section 3.1) are secure. We further assume that adversaries are computationally bounded and can only corrupt at most 1/3 of the consensus participants of the blockchain. Thus, we assume that an adversary cannot tamper with the execution of the AMR smart contract. We assume that clients can always read the blockchain state and write to the blockchain. Note that blockchain congestion might temporarily affect the *availability* property of AMR but does not impact the *correctness* and *privacy* properties. We assume that the adversary has the capabilities of a miner, i.e. can reorder transactions within a blockchain block, inject its own transactions before and after certain transactions. Also, we assume that the adversary can always read all transactions issued to the AMR contract, while the transactions are propagating on the P2P network, and afterwards when they are written to the blockchain. For a withdrawal and a redeem transaction, we assume that the client pays transaction fees either through a non-adversarial relayer (cf. Section 3.7), or the client possesses a blockchain address with funds that are not linkable to his deposit transaction. Finally, we assume that the underlying lending platforms used by AMR are secure.

3.4 Detailed zkSNARK-based System Construction

We now present a zk-SNARK-based construction of AMR.

3.4.1 Building Blocks

Hash Functions. $H_p : \{0,1\}^* \to \mathbb{F}$ is a preimage-resistant and collision-resistant hash function that maps binary string to an element in $\mathbb{F}$, $H_{2p} : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ be a collision-resistant hash function that maps two elements in $\mathbb{F}$ into an element in $\mathbb{F}$.

Deposit Commitments. A secure commitment scheme ($\mathrm{P}_{\mathrm{COM}}$, $\mathrm{V}_{\mathrm{COM}}$) can be constructed using a secure hash function, $H_p : \{0,1\}^* \to \mathbb{F}$, as follows: (1) $\mathrm{P}_{\mathrm{COM}}(m, r)$ returns $c = H_p(m||r)$, (2) $\mathrm{V}_{\mathrm{COM}}(c, m, r)$ verifies if $c \stackrel{?}{=} H_p(m||r)$.

In AMR, before depositing into the contract, a client samples randomnesses, k_{dep}, r and computes the deposit commitment: $\mathsf{cm} = H_p(k_{dep}||r)$ as a part of a deposit transaction.

Merkle Tree over Deposit Commitments, T_{dep}. The AMR contract maintains a Merkle tree, T_{dep}, over all commitments. a Merkle tree is an instance of an authenticated data structures for testing set membership [25] (cf. Section 5.1). The Merkle tree in the AMR contract is a complete binary tree and initialized with zero values at its leaves. As deposit transactions arrive, the AMR contract keeps track of the number of deposit transactions and updates the trees through the AcceptDeposit algorithm. A Merkle tree can be constructed using a collision-resistant hash function, H_{2p}.

We denote path_i the Merkle proof of cm_i. We denote the Merkle tree root at block h to be root^h_{dep}. We let $\mathsf{root}_{dep}.\mathsf{blockheight}$ to be the height of the blockchain block when root_{dep} gets updated. Figure 3.2 gives an illustrative example of the Merkle tree maintained by the AMR contract.

Withdrawal Proof. To withdraw coins from AMR, a client needs to prove three conditions: *(i)* the client knows the committed values of some existing commitments used to compute the tree root via zkSnark proof, *(ii)* the client did not withdraw in the past by passing a *fresh* nullifier value, *(iii)* the client knows the secret key used to issue the withdrawing transaction.

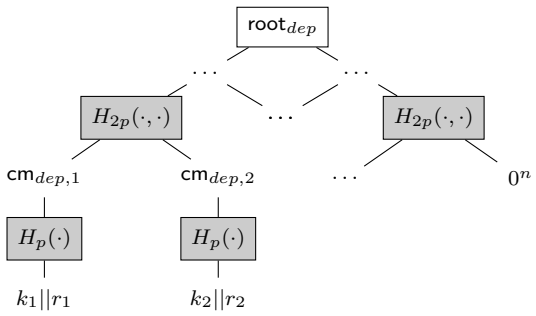


Figure 3.2. Illustrative example of the Merkle tree, T_{dep}. The tree keeps track of commitments from by clients' deposit transactions. The root of the tree, root_{dep} is used to verify the NIZK proofs from withdrawing and redeeming-reward transactions.

The last condition prevents network adversaries from stealing a valid proof by binding the public/private key to the zksnark proof. For a Merkle tree T with a root, root_{dep}, a client needs issue a proof proving the following relation:

$$
\begin{aligned}
R_{wdr} : \{&\mathsf{pk}, \mathsf{sn}, \mathsf{root}_{dep}; \mathsf{sk}, k_{dep}, r, \mathsf{path}_i : \\
&\mathsf{pk} = \textsc{extract}\mathrm{PK}(\mathsf{sk}) \wedge \mathsf{sn} = H_p(k_{dep}) \wedge \\
&\mathsf{cm} = H_p(k_{dep}||r) \wedge T.\textsc{Verify}(i, \mathsf{cm}, \mathsf{root}_{dep}, \mathsf{path}_i))\} \qquad (3.1) \\
\text{Where } &\mathsf{pk}, \mathsf{sn}, \mathsf{root}_{dep} \text{ are public values and} \\
&\mathsf{sk}, k_{dep}, r, \mathsf{path}_i \text{ are private values.}
\end{aligned}
$$

The nullifier value is used to ensure correctness by preventing clients from double-withdrawal.

Reward Proof. Intuitively, to prove that funds remained in the system for a certain time period, users can simply prove to the contract that they know some commitment their, cm, that is a member of an older Merkle root. To achieve such condition, the AMR contract always maintains an t_{con}-blocks-old Merkle root that serves as an anchor for clients to issue the reward proof. Like withdrawing, to redeem, clients need to nullify the old commitment, cm, by issuing a nullifier value, sn, and submit a new commitment, cm' to be eligible for future redeems and withdrawals. This requirement allows AMR to maintain system correctness and hide the link between reward-redeeming and withdrawing transactions.

In summary, to redeem coins from AMR, a client needs to prove that: *(i)* the client knows the committed value of some existing commitments used to compute the current reward Merkle tree root via a zkSnark proof, *(ii)* the client did not withdraw in the past by passing a *fresh* nullifier value sn, and *(iii)* Finally, the client needs to refresh its original deposit by submitting a new commitment to be eligible for future reward redemptions and withdrawals.

3.4.2 Contract Setup

Let $\mathbb{F}$ be the finite field used in AMR, during the AMR contract setup phase, the setup algorithm samples secure hash functions $H_p : \{0,1\}^* \to \mathbb{F}, H_{2p} : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ from secure collision-resistant hash families. The AMR contract is initialized with several parameters: amt for the fixed number of coins to be mixed, amt_{rwd} indicating the fixed amount of coins

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 $\mathsf{amt}_{rwd} \in \mathbb{Z}_{>0}$ to be a fixed reward amount
Choose $t_{con} \in \mathbb{Z}_{>0}$ to be condition for getting reward
Choose $d \in \mathbb{Z}_{>0}$, Let $X = \{x_1, \ldots, x_{2^d}\}$
    where $x_i = 0^\lambda$ for all $x_i \in X$
Choose Σ to be the lending platform
Deploy Γ_{AMR} to be the interest distribution pool
Initialize an empty tree $\mathsf{root}_{dep} = T.\textsc{Init}(1^\lambda, X)$,
Choose $k \in \mathbb{Z}_{>0}$, set $\mathsf{RootList}_{wdr,k}[i] = \mathsf{root}_{dep}$,
    for $1 \le i \le k$
Set $\mathsf{root}^{\mathsf{curr}}_{rwd} = \mathsf{root}^{\mathsf{next}}_{rwd} = \mathsf{root}_{dep}, \mathsf{index} = 1$
Construct C_{wdr} for relation described in Equation (3.1).
Let Π be the zk-SNARK instance.
    – Run $(\mathsf{ek}_{dep}, \mathsf{vk}_{dep}) \leftarrow \Pi.\textsc{Setup}(1^\lambda, C_{wdr})$
    Initialize: $\mathsf{DepositList} = \{\}, \mathsf{NullifierList} = \{\}$,
Deploy smart contract AMR with parameters :
$\mathsf{pp} = (\mathbb{F}, H_p, H_{2p}, \mathsf{amt}, \mathsf{amt}_{rwd}, t_{con}, \Sigma, \Gamma_{\mathsf{AMR}}$
    $T, \mathsf{index}, \mathsf{RootList}_{wdr,k}, \mathsf{root}^{\mathsf{curr}}_{rwd}, \mathsf{root}^{\mathsf{next}}_{rwd},$
    $(\mathsf{ek}_{dep}, \mathsf{vk}_{dep}), \mathsf{DepositList}, \mathsf{NullifierList})$

Figure 3.3. AMR Setup. The public parameters, pp, contains all information needed to interact with the AMR contract, and pp can be queried by any client.

to be rewarded, and t_{con} specifying the minimum number of blocks that clients need to wait before redeeming rewards.

Setting up Merkle Trees. Let T be the Merkle tree of depth d, the setup algorithm described in Section 3.3.1 initializes T with zero leaves and initializes $\mathsf{index} = 1$ to keep track of latest deposits. Also, the algorithm initializes two lists: $\mathsf{RootList}_{wdr,k}$ to be the list of k most recent roots of T. Finally, the contract keeps track of the current reward root, $\mathsf{root}^{\mathsf{curr}}_{rwd}$ that is used by clients to form reward proofs. and the next reward root $\mathsf{root}^{\mathsf{next}}_{rwd}$. Recall that t_{con} to be the minimum number of blocks that clients need to wait before redeeming a

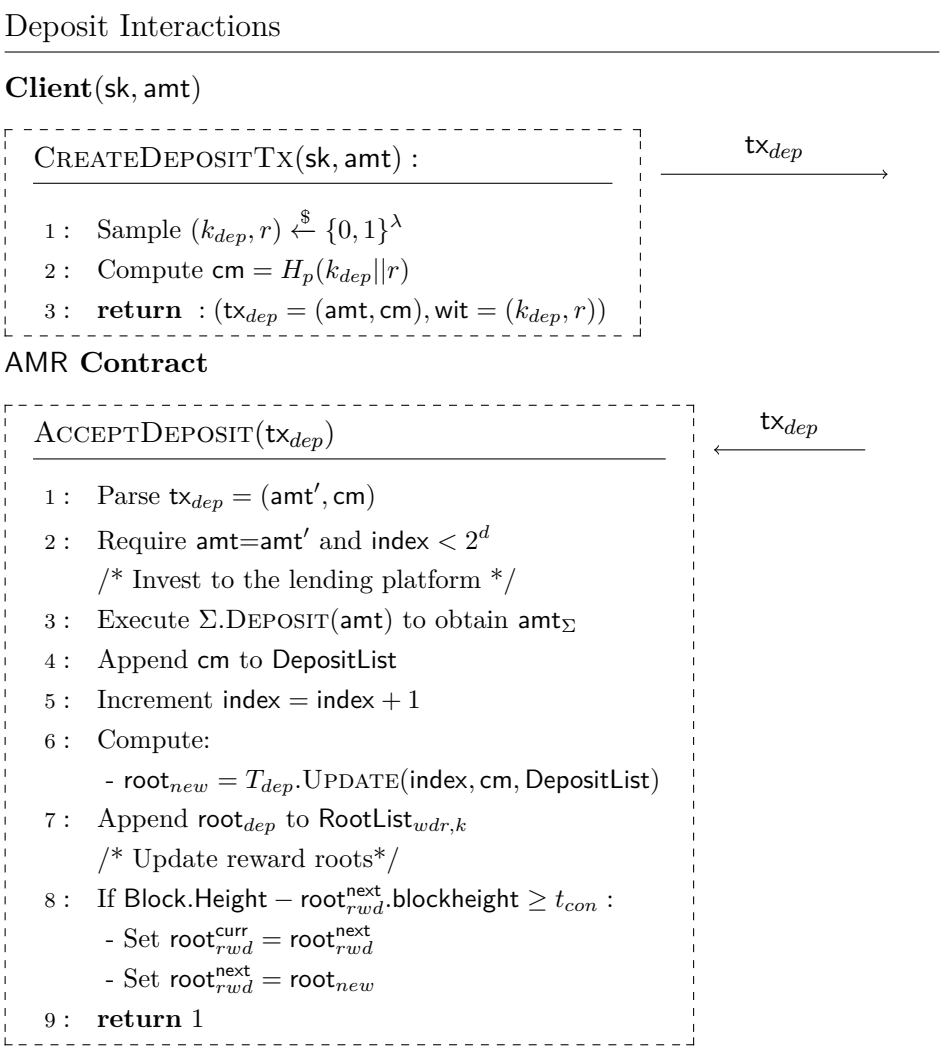


Figure 3.4. AMR's deposit interactions between the client (Client's CreateDepositTx algorithm) and AMR contract (AMR's AcceptDeposit algorithm). Transaction tx_{dep} is signed by sk. Block.Height denotes the block height of the block containing tx_{dep}

reward, we require: $\mathsf{root}^{\mathsf{next}}_{rwd}.\mathsf{blockheight} - \mathsf{root}^{\mathsf{curr}}_{rwd}.\mathsf{blockheight} \geq t_{con}$. This approach helps the AMR contract maintain t_{con}-blocks-old reward root without storing all other roots.

Setting up zk-SNARK parameters. Let Π be the zk-SNARK instance used in AMR, the setup algorithm Section 3.3.1 constructs circuit C_{wdr} capturing the relation described in Equation (3.1). Then, the setup algorithm runs Π.Setup on the circuit to obtain two keys, $(\mathsf{ek}_{dep}, \mathsf{vk}_{dep})$.

Reward-Redeeming Interactions

Client$(\mathsf{sk}', \mathsf{wit})$

$\textsc{CreateRedeemTx}(\mathsf{sk}', \mathsf{wit})$:

1 : Parse $\mathsf{wit} = (k_{dep}, r)$

2 : Sample $\mathsf{wit}' = (k'_{dep}, r') \xleftarrow{\$} \{0,1\}^\lambda$

3 : Obtain pp^h from the contract

4 : Compute :

$-\ \mathsf{sn}_{rwd} = H_p(k_{dep}), \mathsf{cm}_{old} = H_p(k_{dep}||r), \mathsf{cm}_{new} = H_p(k'_{dep}||r')$

5 : Get index i of cm_{old} from $\mathsf{DepositList}^h$

6 : Compute path_i^h such that:

- $T_{rwd}.\textsc{Verify}(i, \mathsf{cm}_{old}, \mathsf{root}_{rwd}^{\mathsf{curr}}, \mathsf{path}_i^h) = 1$

7 : Form $\mathsf{wit}_{rwd} = (\mathsf{sk}', k_{dep}, r, \mathsf{path}_i^h)$

8 : $\pi_{rwd} \leftarrow \Pi.\textsc{Prove}(\mathsf{ek}_{rwd}, st[\mathsf{pk}', \mathsf{sn}_{rwd}, \mathsf{root}_{rwd}^{\mathsf{curr}}], \mathsf{wit}_{rwd})$

9 : **return** $\mathsf{tx}_{rwd} = (\mathsf{sn}_{rwd}, \mathsf{root}_{rwd}^{\mathsf{curr}}, \pi_{rwd}, \mathsf{cm}_{new}), \mathsf{wit}' = (k'_{dep}, r')$

tx_{rwd} ⟶

AMR **Contract**

$\textsc{IssueReward}(\mathsf{tx}_{rwd})$:

⟵ tx_{rwd}

1 : Parse $\mathsf{tx}_{rwd} = (\mathsf{sn}_{rwd}, \mathsf{root}'_{dep}, \pi_{rwd}, \mathsf{cm}_{new})$

2 : Require:

- $\mathsf{root}_{rwd}^{\mathsf{curr}} = \mathsf{root}'_{dep}$
- $\mathsf{sn}_{rwd} \notin \mathsf{NullifierList}$
- $\Pi.\mathrm{ZK}\textsc{Verify}(\mathsf{vk}_{rwd}, \pi_{rwd}, st[\mathsf{msg.sender}, \mathsf{sn}_{rwd}, \mathsf{root}_{rwd}^{\mathsf{curr}}]) = 1$

3 : Append sn_{rwd} to $\mathsf{NullifierList}$

/* Refresh the old commitment */

4 : Append cm_{new} to $\mathsf{DepositList}$

5 : Increment $\mathsf{index} = \mathsf{index} + 1$

6 : Compute $\mathsf{root}_{new} = T_{dep}.\textsc{Update}(\mathsf{index}, \mathsf{cm}, \mathsf{DepositList})$

7 : Append root_{dep} to $\mathsf{RootList}_{wdr,k}$

/* Update reward roots*/

8 : If $\mathsf{Block.Height} - \mathsf{root}_{rwd}^{\mathsf{next}}.\mathsf{blockheight} \geq t_{con}$:

- Set $\mathsf{root}_{rwd}^{\mathsf{curr}} = \mathsf{root}_{rwd}^{\mathsf{next}}, \mathsf{root}_{rwd}^{\mathsf{next}} = \mathsf{root}_{new}$

Do $\mathsf{tx}_{rwd}.\mathsf{sender.transfer}(\mathsf{amt}_{rwd})$

9 : **return** 1

Figure 3.5. AMR's reward-redeeming interactions between the client (Client's CreateRedeemTx algorithm) and AMR contract (AMR's IssueReward algorithm).

Setting up commitments and nullifier lists. The AMR contract is initialized with two empty lists: a list, DepositList, that contains all cm included in depositing and reward-redeeming transactions, a list, NullifierList, that contains all unique identifiers (i.e. sn) appeared in withdrawing and reward-redeeming transactions. Figure 3.3 formally describes this setup algorithm.

3.4.3 Client Algorithms

These following algorithms specify how clients interact with the AMR smart contract.

Depositing. CreateDepositTx allows a client to deposit coins into the contract and outputs secret notes, wit, that can later be used to withdraw coins or obtain a reward.

Redeeming Reward. CreateRedeemTx allows clients with the secret note, wit, and the secret key sk to issue a proof, π_{rwd}, to prove to the AMR contract that that client has not withdrawn their deposited coins after certain number of block counts. In order to form such NIZK proof, the client obtains the current state of the contract to compute private inputs (i.e. Merkle path) for the zk-SNARK proof generation. Also, the proof generation requires the client to use an older root maintained by the contract as part of the computation. This approach allows a client to prove to the contract that the client's transaction was deposited before the root was computed. Along with the NIZK proof, a client will include the nullifier value as part of the transaction to prevent double-redemption from the AMR contract. Finally, the client includes a new commitment value, cm, in the reward-redeeming transaction to be eligible for future withdrawal and reward-redemption.

Withdrawing. CreateWithdrawTx allows a client with the secret note, wit, and secret key sk to issue proofs, π_{wdr}, to withdraw amt to the public key pk. In this step, AMR requires the client to issue a proof to verify that the client has deposited coins in the past along with a nullifier value sn to prove that those coins have not been withdrawn before and to prevent clients from withdrawing coins without having any coins deposited to the AMR contract.

Withdraw Interactions

Client(sk, wit)

CreateWithdrawTx(sk, wit) :

1 : Obtain pp^h from the contract
2 : Parse $\mathsf{wit} = (k_{dep}, r)$
3 : Compute $\mathsf{sn}_{wdr} = H_p(k_{dep}), \mathsf{cm} = H_p(k_{dep}||r)$
4 : Get index i of cm from $\mathsf{DepositList}^h$
5 : Choose $\mathsf{root}_{dep} \in \mathsf{RootList}_{wdr,k}$
6 : Compute $\mathsf{path}^h_{dep,i}$ such that
- T.Verify$(i, \mathsf{cm}, \mathsf{root}_{dep}, \mathsf{path}^h_{dep,i}) = 1$
7 : Form: $\mathsf{wit}_{dep} = (\mathsf{sk}, k_{dep}, r, \mathsf{path}^h_{dep,i})$
8 : $\pi_{wdr} \leftarrow \Pi$.Prove$(\mathsf{ek}_{dep}, st[\mathsf{pk}, \mathsf{sn}_{wdr}, \mathsf{root}_{dep}], \mathsf{wit}_{dep})$
9 : **return** $\mathsf{tx}_{wdr} = (\mathsf{sn}_{wdr}, \mathsf{root}_{dep}, \pi_{wdr})$

tx_{wdr} ⟶

AMR **Contract**

⟵ tx_{wdr}

IssueWithdraw(tx_{wdr}) :

1 : Parse $\mathsf{tx}_{wdr} = (\mathsf{sn}_{wdr}, \mathsf{root}_{dep}, \pi_{wdr})$
2 : Require
- $\mathsf{root}_{dep} \in \mathsf{RootList}_{wdr,k}$
- $\mathsf{sn}_{wdr} \notin \mathsf{NullifierList}$
- Π.ZKVerify$(\mathsf{vk}_{dep}, \pi_{wdr},$
$st[\mathsf{msg.sender}, \mathsf{sn}_{wdr}, \mathsf{sn}_{rwd}, \mathsf{root}_{dep}]) = 1$
3 : Let $\mathsf{balance}_\Sigma$ be the Σ token balance
4 : Let NoDepositsLeft be the number of deposits remained
/* Redeeming from the lending platform */
5 : Redeem $\mathsf{balance} + R$ by executing Σ.Redeem$(\mathsf{balance}_\Sigma)$
where $\mathsf{balance} = \mathsf{NoDepositsLeft} \cdot \mathsf{amt}$
6 : Append sn_{wdr} to NullifierList
/* Send the accrued interest to the distribution pool */
7 : Do Γ_{AMR}.transfer$(R/\mathsf{NoDepositsLeft})$
/* Send the original deposit to sender*/
8 : Do $\mathtt{tx}_{wdr}$.sender.transfer(amt)
/* Reinvest into the lending platform */
9 : Let $\mathsf{removedFund} = \mathsf{amt} + R/\mathsf{NoDepositsLeft}$
10 : Execute Σ.Deposit$(\mathsf{balance} + R - \mathsf{removedFund})$
11 : **return** 1

Figure 3.6. AMR's deposit interactions between the client (Client's CreateWithdrawTx algorithm) and AMR contract (AMR's IssueWithdraw algorithm).

3.4.4 Contract Algorithms

In this part, we formally define the contract algorithms.

Accepting Deposit. Upon receiving a deposit transaction from an externally owned account, the contract verifies the amount amt, updates the tree structure, recomputes the Merkle roots for the Merkle tree, and updates the $\mathsf{DepositList}$ list. Next, the AMR contract deposits amt into the lending platform Σ to retrieve amt_{Σ}. Finally, depending on the number of blocks mined, the contract always maintains a t_{con}-blocks-old root, so that clients use it to redeem rewards. Figure 3.4 formally describes this procedure.

Issuing Reward. Upon receiving reward transactions, the contract verifies that the proof, π_{rwd}, is valid with the $\mathsf{root}_{rwd}^{\mathsf{curr}}$, and the nullifier sn_{rwd} is not in the $\mathsf{NullifierList}$. Once the verification passes, the contract updates the $\mathsf{NullifierList}$ to prevent future double-redemption and double-withdrawal. Here, we note that $\mathsf{root}_{rwd}^{\mathsf{curr}}$ is the old state of the reward tree; therefore, being able to prove the membership of this root, one can prove that their deposit has not been withdrawn. Finally, the AMR contract updates the Merkle tree with the new commitment, cm_{new}. This update is similar to the deposit phase, and it helps the client to refresh their original commitment to be eligible for future redemption and withdrawals.

Issuing Withdraw. Upon receiving withdrawal transaction, the contract verifies the proof, π_{wdr} and verifies that the nullifier sn_{wdr} is not in the $\mathsf{NullifierList}$. To prevent future double-withdrawal, the AMR contract then appends sn_{wdr} to $\mathsf{NullifierList}$. Then, the AMR contract redeems all deposited funds from lending platform Σ along with the accrued interest R. Finally, the AMR deposits amt to the user's address and redeposits leftover funds into lending platforms. To avoid leakage in distributing accrued interest, AMR deposits the interest into a separate pool, Γ_{AMR}. Only users who hold the governance tokens obtained from rewards can later obtain this interest.

Figure 3.4, Figure 3.5, and Figure 3.6 formally describe the interactions between the clients and the smart contract in AMR.

Distributing Accrued Interest. We adapt the time-weight voting proposed by Curve [47] for this distribution step. In particular, in AMR, the pool, Γ_{AMR}, receives a portion of the total accrued interest upon each withdrawal. Clients need to lock governance

tokens to the pool to be eligible for redeeming this interest. The AMR pool periodically distributes the total accrued interest proportionally to clients based on their voting power.

Client's voting power is calculated based on their amount of governance tokens and how long they are willing to lock those tokens in the distribution pool. The pool requires two main functionalities: CreateLock and Claim. The pool is initialized with a value t_{max} denoting the maximum amount of blocks that client can lock their governance tokens.

- CreateLock($\texttt{tx}_{lock}$) takes as input the locking transaction, $\texttt{tx}_{lock}$, from the client. The locking transaction contains the governance tokens, γ_{rwd}, and t_{lock} specify the number of blocks that the client will lock γ_{rwd} in the pool. At any given point in time, the voting power of the client is $\gamma_{rwd} \cdot \frac{t}{t_{max}}$ where t is the time left to unlock $t \leq t_{lock}$.
- Claim() is a contract function that can be periodically triggered by the clients. When a client triggers this function, the pool calculates the client's current voting weight as $w = \gamma_{rwd} \cdot \frac{t}{t_{max}}$, and the total voting power of all users, W. Finally, the pool distributes the accrued interest proportionally to the client according their voting power and the total voting power, $\frac{w}{W}$.

The main goal of time-weighted voting is to distribute more reward not only to users who *contribute* more to AMR (i.e., having more governance tokens) but also to users who *commit* more to the system (i.e., locking their stakes for a longer period). Finally, for a more detailed description of this time-weight voting technique, we refer interested readers to [47].

3.5 System Analysis

In this section, we informally discuss how AMR achieves the security goals mentioned in Section 3.3.4. As mentioned in Section 3.3.5, the underlying cryptographic primitives (i.e., zk-SNARK, commitment scheme, hash functions) are assumed to be secure, and AMR's depositing and withdrawing functionalities can be thought as the shielding and de-shielding transactions in ZCash with a fixed denomination. Therefore, the security of AMR follows from the security of zk-SNARK-based applications like ZCash [144]. In particular, the malicious outsider will not be able to learn any information from the public data. However, adversaries

can still guess the pair-wise link between a withdrawing, a depositing, and reward-redeeming transactions. The probability of guessing correctly largely depends on the number of deposits, redemptions, and withdrawals issued to AMR. Thus, we need to understand this adversarial probability to quantify the privacy offered by AMR.

3.5.1 Privacy Metric

We let h be the height of the blockchain, we define: $\mathsf{AnomSet}^h$ be the set of commitments issued to the AMR contract until block height h by *honest* users, and the adversary does not know the preimages of those commitments. $\mathsf{NullifierSet}^h$ be the set of nullifiers appeared in either reward-redeeming or withdrawing transactions issued to the AMR until block height h by *honest* users. AnomSet and NullifierSet are always available to the adversary. We also assume that $|\mathsf{AnomSet}^h| - |\mathsf{NullifierSet}^h| > 0$ for all h.

We say cm originates sn, when k_{dep} is used to compute both cm in tx_{dep} and sn_{wdr} in tx_{wdr} or tx_{rwd}. We define:

- $\mathsf{cm} \overset{link}{\leftarrow} \mathsf{sn}$: if the value k used to compute $\mathsf{cm} = H_p(k||r) \in \mathsf{tx}_{dep}$ or $\in \mathsf{tx}_{rwd}$ is equal to the value k used to compute $\mathsf{sn} = H_p(k) \in \mathsf{tx}_{rwd}$ or $\in \mathsf{tx}_{wdr}$.

The reward linking advantage is the probability that an adversary can output the correct commitments that originates the nullifier value appeared in reward-redeeming transactions. We define that probability is as follow:

Definition 3.5.1. *(Reward Linking Advantage) Let $\mathcal{A}$ be the PPT adversary, tx_{rwd}^{h+1} be the only valid reward-redeeming transaction issued at block $h+1$ from an honest user. Let sn_{rwd}^{h+1} be the nullifier appeared in tx_{rwd}^{h+1} We define the adversarial advantage as follow:*

$$\mathsf{Adv}^h_{\mathcal{A},rwd} = \Pr\left[\mathcal{A}(\mathsf{tx}_{rwd}^{h+1}) \rightarrow \mathsf{cm} \in \mathsf{AnomSet}^h \ s.t. \ \mathsf{cm} \overset{link}{\leftarrow} \mathsf{sn}_{wdr}^{h+1}\right]$$

Similarly, the adversarial advantage in linking withdrawing transaction to other transactions is the probability that an adversary can guess correctly the commitment that originates the nullifier value appeared in withdrawing transaction.

Definition 3.5.2. *(Withdraw Linking Advantage) Let $\mathcal{A}$ be the PPT adversary, tx_{wdr}^{h+1} be the only valid withdrawing transaction issued at block $h+1$ from an honest user. Let sn_{wdr}^{h+1} be the nullifier appeared in tx_{wdr}^{h+1}. We define the adversarial advantage as follow:*

$$\mathsf{Adv}_{\mathcal{A},wdr}^{h} = \Pr\left[\mathcal{A}(\mathsf{tx}_{wdr}^{h+1}) \rightarrow \mathsf{cm} \in \mathsf{AnomSet}^{h} \ s.t. \ \mathsf{cm} \overset{link}{\leftarrow} \mathsf{sn}_{wdr}^{h+1}\right]$$

We assume that the deposit addresses are independent and unlinkable accounts for our privacy metric to hold. If the same entity deposits from different addresses, but a blockchain analysis allows to link those addresses, the anonymity set would only grow by at most 1 deposit.

3.5.2 Privacy Analysis

Systems without reward. In a vanilla AMR system that only supports depositing and withdrawing functionalities, a withdrawal transaction can be at the origin of any deposit transactions of honest users before the withdrawal transaction , under the assumption that all underlying cryptographic primitives are secure. The adversarial advantage in linking withdrawing transaction to the original deposit transaction is: $\mathsf{Adv}_{\mathcal{A},wdr}^{h} = 1/|\mathsf{AnomSet}^{h}| + \mathsf{negl}(\lambda)$ where $\mathsf{negl}(\lambda)$ is the adversarial advantage in breaking the underlying cryptographic primitive.

System with reward. Because AMR involves redeeming transactions, we need to analyze the adversarial advantages under different scenarios. In the following, we show the adversarial advantage in linking different transactions through the following claims.

Claim 3.5.1. *Assuming that all underlying cryptographic primitives are secure, the adversarial advantage in linking reward-redeeming transaction to other transactions as defined in Definition 3.5.1 is less than* $1/|\mathsf{AnomSet}^{h-t_{con}}| + \mathsf{negl}(\lambda)$

Sketch. AMR is parameterized with the value t_{con}, the number of blocks that a client needs to wait to be eligible for a reward. The adversary observes a redeeming transaction issued to the AMR contract after block height $h+1$ from an honest user. A valid redeeming transaction indicates to the adversary that the sender has issued commitment into the system at least $h-$

t_{con} blocks ago. Therefore, the probability that the adversary links the redeeming transaction to the correct commitment is hence: $\mathsf{Adv}^{h}_{\mathcal{A},rwd} \leq 1/|\mathsf{AnomSet}^{h-t_{con}}| + \mathsf{negl}(\lambda)$ where $\mathsf{negl}(\lambda)$ is the adversarial advantage in breaking the underlying cryptographic primitive.

Claim 3.5.2. *Assuming that all underlying cryptographic primitives are secure, the adversarial advantage in linking between withdrawing transaction to other transactions as defined in Definition 3.5.2 is* $1/|\mathsf{AnomSet}^{h}| + \mathsf{negl}(\lambda)$.

Sketch. Since we assume that the underlying cryptographic primitives are secure, the adversarial advantage in guessing correctly by breaking those primitives is negligible. Moreover, since each deposit and reward-redeeming transaction in AMR adds another leaf to the Merkle tree, the probability of guessing a correct leaf is equal to the number of Merkle leaves that are not controlled by the adversary. In another word, the probability is $1/(|\mathsf{AnomSet}^{h}|)$. Therefore, the adversarial advantage, $\mathsf{Adv}^{h}_{\mathcal{A},wdr} \leq 1/|\mathsf{AnomSet}^{h}| + \mathsf{negl}(\lambda)$ where $\mathsf{negl}(\lambda)$ is the adversarial advantage in breaking the underlying cryptographic primitive.

In summary, in AMR, given a reward-redeeming transaction, to guess the correct commitment, the adversary can reduce the size of the anonymity set by narrowing the search window to t_{con} blocks before the block containing the reward-redeeming transaction. On the other hand, given a withdrawing transaction, the adversary's advantage in guessing the correct commitment is still the same as the adversarial advantage in the system without reward, $1/|\mathsf{AnomSet}^{h}| + \mathsf{negl}(\lambda)$. The main reason is that, in AMR, beside each deposit, each reward-redeeming transaction also adds one additional commitment to the Anonymity Set, AnomSet. Therefore, AMR offers a bigger anonymity set than system without reward.

Privacy of the Accrued Interest Distribution. One naïve way to distribute the accrued interest is to split the total accrued interest equally among depositors. This approach reveals nothing about the original deposits. However, it introduces an unfair allocation of interest as users joining the system later receive the same amount of interest as users joining the system earlier.

In AMR, to achieve fairness in the accrued interest distribution, AMR allows users with governance tokens to lock their tokens in a separated pool (i.e., Γ_{AMR}). This pool receives a portion of the accrued interest from the AMR contract upon each withdrawal, and it

periodically distributes the total accrued interest to addresses that lock their governance token into the pool. The amount each address receives is based on the number of governance tokens and how long those tokens are locked. It's not difficult to see that AMR ensures the fairness in the accrued interest allocation because only users contributing more to the anonymity set of AMR, can redeem more governance tokens; therefore, they can obtain more accrued interest.

3.5.3 Other Goals Achieved By AMR

In addition to the privacy goal, we briefly explain how AMR achieves the other goals defined in Section 3.3.4.

Correctness. AMR satisfies correctness. If an adversary can provide a withdrawal transaction that verifies without depositing any coins into the system, there are two possible scenarios: First, the adversary can derive a new valid transaction for the current state of the contract (i.e. observing commitment list), or it intercepts a withdrawal transaction and replaces the recipient address with its address. However, in the first case, it implies that the adversary breaks the preimage-resistant security of the underlying hash function $H_p(\cdot)$, and the second case implies that the adversary breaks the security of the zk-SNARK instance.

Availability. We argue that AMR satisfies availability. Unlike existing centralized tumbler designs [78], the availability of the system relies on the fact that the tumbler has to stay online. Similar to Möbius [108], AMR is a smart contract that executes autonomously on the blockchain, so adversary cannot prevent clients from interacting (i.e., reading and writing) with the blockchain.

Front-Running Resilience. Recall that the AMR contract stores a list of k recent roots. To invalidate a withdrawal transaction, an adversary needs to "front-run" at least k deposit transactions before a withdrawal transaction. Thus, one can choose the value k to be sufficiently large so that the cost of attacking is too expensive for the adversary to carry out. More specifically, to invalidate a single deposit transaction, the amount of token an adversary needs to have are at least $k \times (\mathsf{amt} + \mathsf{fee}_{dep})$ where amt is the fixed denomination specified in Section 3.4.2 and fee_{dep} is the deposit fee. For example, if we set $k = 1000$,

Table 3.1. zk-SNARK Setup Cost

Tree Depth	# Constraints		Setup Time		Keys Size	
	C_{wdr}		t_{wdr}		$(\mathsf{ek}_{wdr}, \mathsf{vk}_{wdr} = 640B)$	
	Poseidon	MiMC	Poseidon	MiMC	Poseidon	MiMC
10	4,245	15,045	86.56s	246.99s	4.3MB	7.3MB
15	5,460	21,660	107.34s	377.44s	5.8MB	11.1MB
20	6,675	28,275	126.51s	465.27s	7.3MB	13.8MB
25	7,890	34,890	146.41s	642.04s	8.8MB	18.6MB
30	9,105	41,505	185.64s	729.03s	10.8MB	21.4MB

$\mathsf{amt} = 10$, assuming $\mathsf{fee}_{dep} = 0.02$, and let the token be *ether*, the adversary needs at least $k \times (\mathsf{amt} + \mathsf{fee}_{dep}) = 10,020$ *ethers* ($38m$ USD) to carry out the attack, and the adversary will lose at least 20 *ethers* (76,000 USD) in term of fee.

3.6 Evaluation

3.6.1 Parameters

Choice of cryptographic primitives. We use Groth's zkSNARK [75] as our instance of zk-SNARK due to its efficiency in term of proofs' size and verifier's computations. For cryptographic hash functions, we use a Pedersen hash function [113] for H_p and evaluate AMR using two different choices of hash functions for H_{2p}: the MiMC [4] and the Poseidon hash function [73]. Arithmetic circuits using MiMC and Poseidon hash yield a lower number of constraints and operations when compared to arithmetic circuits relying on other hash functions [85, 4] (i.e. SHA-256, Keccak). Moreover, both MiMC and Poseidon hash functions are not only designed specifically for SNARK applications, but also highly efficient for Ethereum smart contract applications in terms of gas costs. Finally, as discussed in Section 3.4.1, the commitment scheme and the Merkle tree can be directly instantiated using Pedersen and MiMC/Poseidon hash functions.

Software. For the arithmetic circuit construction, we use the `Circom` library [81] to construct the withdrawing circuit, C_{wdr} for the relation described in Equation (3.1). We use Groth's zk-SNARK proof system implemented by the `snarkjs` library [82] to develop the client's algorithms (cf. Section 3.3.2), and to perform the trusted setup for obtaining

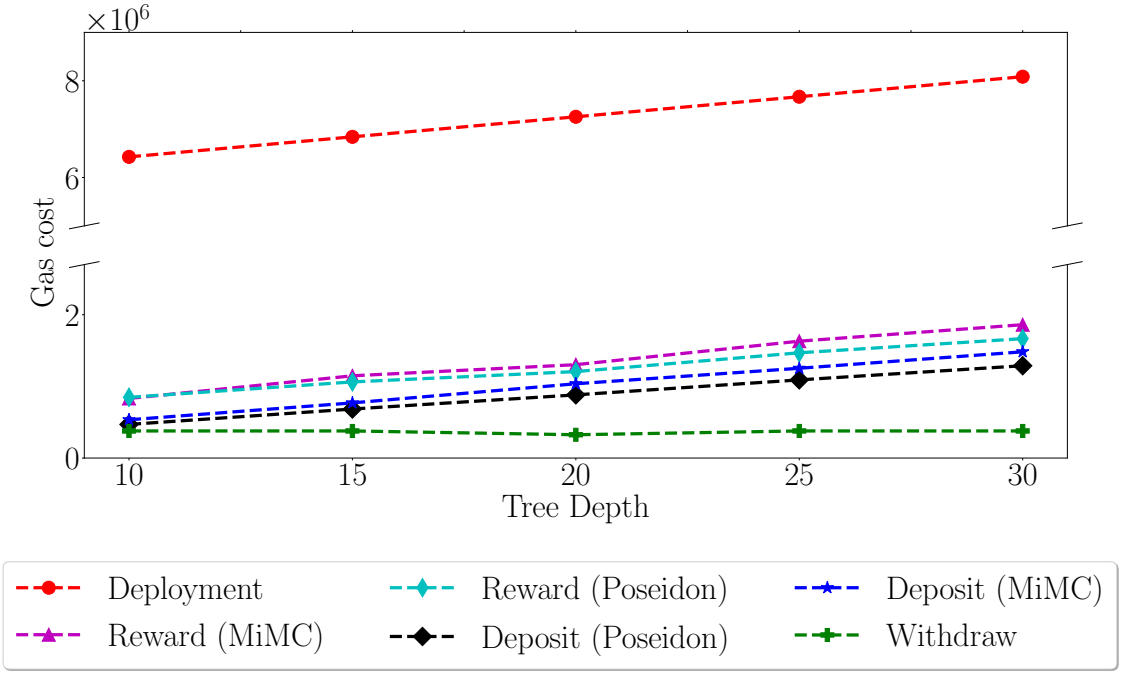


Figure 3.7. On-chain Costs of Deployments, Deposit, Withdrawal, and Reward Redemption for Different Tree Depths and Hash Functions.

the proving and evaluation keys for the AMR contract and clients. We deploy AMR to the Ethereum Kovan testnet [4] [5]. AMR contract consists of 1013 lines of Solidity code.

Hardware. We conducted our experiment on a commodity desktop machine, which is equipped with an Intel Core i5-7400 @3.800GHz CPU, 32GB RAM.

3.6.2 Performance

We measure the performance and the cost of AMR using the following tree depths $d = 10, 15, 20, 25, 30$.

zk-SNARK Setup. Table 3.1 presents an overall performance of the zk-SNARK setup for the withdraw circuit. For the MiMC hash function, for a tree of depth d, the withdraw circuit has $1,815 + 1,323 \times d$ constraints. For the Poseidon hash function, the withdraw circuit has $1815 + 243 \times h$ constraints.

[4]↑AMR's address: 0xdE992c4fBd0f39E5c0356e6365Bcfafa1e94970b
[5]↑A demo video AMR can be found at the following URL: https://youtu.be/-oAQlsRTF08

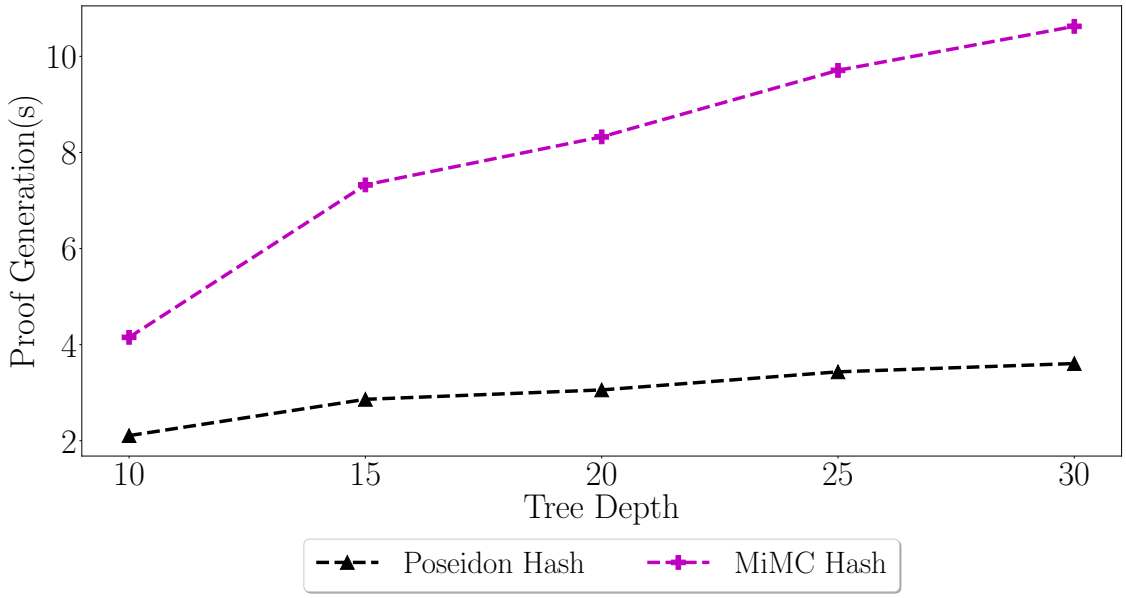


Figure 3.8. zkSnark Proof Generation Time for Poseidon and MiMC hash functions.

Onchain Costs. Figure 3.7 provides the overall costs of deployment, deposit, reward, and withdraw for different tree depths. The cost of deploying the contract is the most expensive operation, accounting from $\approx 6m$ gas for $h = 10$ to $\approx 8m$ gas for $h = 30$ for both the MiMC and Poseidon hash functions. However, we note that the deployment cost is a one-time cost which is amortized over the lifetime of the contract. The cost of the depositing transaction depends on the depth of the tree, which is approximately $43,000 + 51,000 \times h$ for the MiMC hash and approximately $43,000 + 41,000 \times h$ for the Poseidon hash function. The gas cost for verifying a withdrawing transaction is approximately $320,000$ for all tree depths and both choices of hash functions. The gas cost for a reward-redeeming transaction is equal to a total gas cost of a deposit and a withdraw as the AMR contract needs to verify the zkSnark proof as well as to update the Merkle tree.

zk-SNARK Proof Generation. As the Poseidon hash function generates less constraints for the arithmetic circuit, than the MiMC hash function (i.e. 243 vs 1323), we observe a reduction of 3× for the clients' proof generation time with an AMR system using the Poseidon hash function. Figure 3.8 presents the time for a client to generate the zkSnark proofs for different tree depths and hash functions.

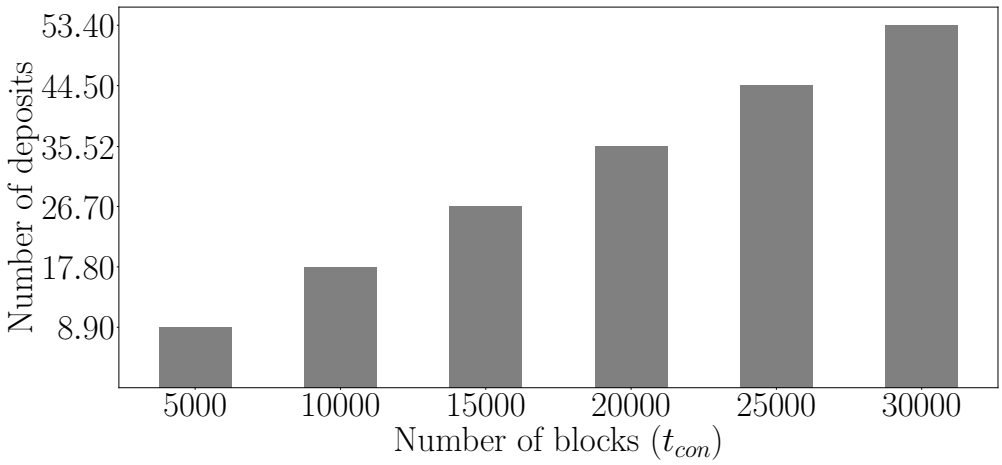


Figure 3.9. Average number of deposit transactions issued to the contract over the span of $5,000$, $10,000$, $15,000$, $20,000$, $25,000$, $30,000$ blocks.

Lending Platforms' Additional Costs. In additional to the cost of executing cryptographic functions in the AMR contract, we also need to consider the cost of other interactions with decentralized lending platforms such as Aave [1] or Compound [41]. These costs are the gas cost of depositing into and redeeming from lending platforms. We estimate the costs of these interactions using data from Etherscan [6] and Compound developer documentations [7]. Thus, depositing into these lending platforms takes approximately $0.3m$ gas (for both Aave and Compound), and redeeming from these platforms takes less than $0.2m$ gas for Aave and less than $0.1m$ gas for Compound. Therefore, depending on the choice of lending platforms, we would expect additional $0.3m$ gas for AMR's depositing function and additional $0.2m$ gas for AMR's withdrawing function.

3.6.3 Empirical Analysis on Tornado Cash

To become eligible for a reward payment in AMR, clients need to keep their deposit in the contract locked for a predefined period (i.e. t_{con} blocks). Thus, one needs to decide what the suitable value for t_{con} is (this value could be set by voting with the governance token).

[6]↑https://etherscan.io/
[7]↑https://compound.finance/docs#networks

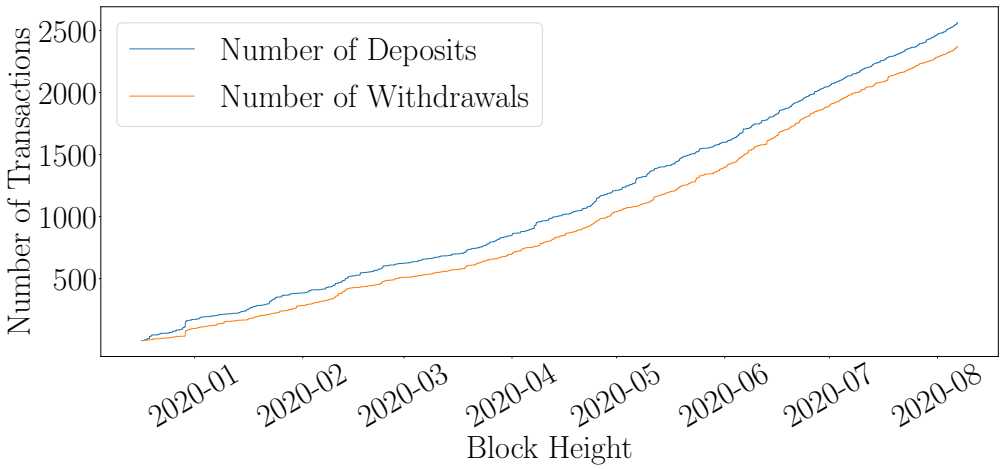


Figure 3.10. Number of deposits and withdrawals issued to the tornado cash 10 ETH pool.

We perform an empirical analysis on the tornado cash system [153] which is, to the best of our knowledge, the only zk-SNARK-based mixer deployed to the Ethereum main net. Tornado cash supports two operations: deposit and withdraw. We analyzed their 10.0 ETH denomination deposit pool [8] from block $9,161,895$ (25 December 2019) to block $10,726,597$ (25 August 2020) to understand how frequent clients deposit to the tornado cash system. This frequency allows us to derive an appropriate value for how long client should keep their funds in AMR contract to be eligible for a reward. For example, Figure 3.9 suggests that for the waiting period of $t_{con} = 30,000$ (approximately 4.5 days), we can expect an additional 52 deposit transactions issued to the contract intermittently, and the more deposit transactions reach the contract, the higher the anonymity set becomes.

Moreover, over the course of 8 months (Cf. Figure 3.10), we observe a total of $2,810$ deposit transactions, and $2,606$ withdrawing transactions on the tornado cash contract. We note that if the number of withdrawing transactions equals to the number of deposit transactions at any point in time, the size of the anonymity set is reduced to zero. Thus, in contrast to Tornado cash, the reward mechanism in AMR is used to incentivise clients to keep a deposit in the system to help maintain a healthy gap between the number of deposits and withdrawals.

[8]↑Address: 0x910Cbd523D972eb0a6f4cAe4618aD62622b39DbF

3.7 Discussion and Applications

Trusted Setup in zk-SNARK. As discussed in Section 3.1, a zk-SNARK requires a trusted setup to generate the evaluation and proving key for each circuit. While one can assume that there exists a trusted third party which helps run the setup, this trust assumption is typically not welcome by the blockchain community, because if such third party can maliciously generate the keys (or the common reference string), it can form a valid proof and steal the contract funds.

To remove the trusted third-party assumption, one can run a multi-party computation (MPC) setup where users can contribute a share to the trusted setup. Several works [28, 16, 29] proposed different protocols for such trusted setup, and they showed that as long as one participant is honest, the zk-SNARK instance will be secure. In particular, the Zcash team has performed such MPC setup for their protocol parameters in 2017 [115]. However, the MPC setup may need to be carried out independently for different circuits and related works [104, 37, 66, 39] have proposed several zk-SNARK constructions that utilizes a universal setup that can be used for *any* circuits with bounded size. These zk-SNARK constructions can be easily integrated into AMR in the future.

Transferring arbitrary denomination. The current version of AMR does not allow clients to transfer arbitrary number of coins privately among clients. To achieve such property, one either needs an out-of-band communication channel between a sender and a recipient to transfer secret notes, or the sender can spend more fees to store additional encrypted data onchain. Moreover, to prevent a sender from stealing coins from the recipient, one could use a similar commitment scheme and encryption as used in Zcash [143]; however, the use of these primitives will increase the cost of the onchain verification. Nevertheless, we will leave the transferring functionality of AMR for the future work.

Sender outsources transaction fee payment. Issuing a transaction requires the payment of fees, and clients should not use the same address for such payment, otherwise their addresses can be linked. In practice users can chose to use a relayer, who broadasts transactions and is paid from a fraction of the withdraw or reward transaction. The relayer can receive the corresponding client proof through a side channel.

Constant querying state. Most blockchain clients (e.g. MetaMask) outsource their blockchain information to centralized services such as Infura. Those centralized services are aware of, the clients' blockchain address(es), IP address as well of the fact that the client queried the AMR contract state. These services are therefore privacy critical, as they may be able to link different addresses from the same client. We hence recommend a privacy aware client to operate an independent validating full blockchain client or use network-level anonymity solutions such as Tor or Virtual Private Network (VPN) before connecting to these centralized services.

Decentralized Governance. Once deployed, AMR's system parameters, will likely need to be adjusted during its lifetime. One could choose an admin key to govern AMR, for the sake of decentralization, however, we believe that a decentralized approach would be beneficial. A governance token is hence the natural choice, whereby AMR can itself distribute those tokens to the clients participating in the protocol. We have identified the following parameters that should be governed: *(i)* new relayer addresses, *(ii)* condition for client reward, *(iii)* the amount of the reward. Once a new version of AMR is developed, the governance mechanism could vote to *(iv)* migrate deposits to a new contract with new features/bug fixes.

3.8 Related Work on add-on privacy solutions

Add-on Privacy Solutions for Smart Contract-enabled Blockchains. While we are not aware of any academic works that propose a zk-SNARK-based mixing system as ours, Tornado cash [153] appears to be the first system deployed in production which allows clients to deposit and withdraw fixed amount of coins. Our work is to the best of our knowledge the first academic work which presents such a system, formalizes the privacy and security properties, and importantly, adds a novel privacy preserving reward mechanism.

Meiklejohn *et al.* [108] propose an Ethereum-based tumbler called Möbius. The construction of Möbius relies on the linkable ring signature primitive and stealth address mechanism used in Monero [5] to hide the address of the true sender and the recipient. However, in Möbius, the size of the anonymity set is limited to the size of the ring, and the gas cost of

the withdrawing transaction increases linearly with the size of the ring. Thus, in term of privacy, AMR offers a bigger anonymity set over time while operating at constant system costs.

Bünz *et al.* proposed a private payment protocol for the Ethereum blockchain called Zether [35]. The core idea of Zether is to use Elgamal to encrypt the balances of clients. However, the cost of Zether transactions (i.e., $7.8m$ gas) is expensive for Ethereum, and Zether does not hide the receiver and recipient of a transaction. Diamond proposed Anonymous Zether [54] to address the later drawback, but the cost of Anonymous Zether is still expensive for blockchains such as Ethereum. For the maximum anonymity set of size 64 reported in the paper, the gas cost of a single transferring call in Anonymous Zether is $48.7m$ gas which is approximately 32 times the cost of an AMR deposit and 130 times the cost of an AMR withdrawal for $h = 30$ (The block gas limit in Ethereum is about $15m$ gas at the time of writing).

Rondelet and Zajac propose Zeth [138], which implements all functionalities of ZeroCash [144] as an Ethereum smart contract. While Zeth allows expressive functionalities, such as transferring arbitrary denomination of notes, it comes with the cost of using a bigger zk-SNARK circuit than in AMR. The choice of the SHA256 hash function in Zeth would result in approximately $59,281$ constraints in the arithmetic circuit, which is $50\times$ bigger than the constraints from using the MiMC hash (i.e., $1,323$ constraints) and $200\times$ bigger than the constraints from using the Poseidon hash (i.e., 243 constraints). While the authors of Zeth did not report any numbers on the proof generation time, we expect that the zk-SNARK proof generation time in Zeth is an order of magnitude larger than in AMR. The Zeth contract needs to store all encrypted notes from transferring function calls, and depending on the size of the transaction, this additional storage also incurs cost (storing a 32-byte data costs $20,000$ gas [160]). The authors of Zeth also report that an estimated cost of verifying a zk-SNARK proof is approximately $2m$ gas ($5\times$ the cost in AMR).

Other Tumbler Designs. The community proposes several centralised tumbler designs [26, 157, 78, 152]. The main essence of those designs relies on a centralised offchain server to mix users' funds, e.g., Tumblebit [78] and A2L [152]. Both require less trust in

the offchain server than solutions such as Mixcoin [26] and Blindcoin [157] by preventing the server from stealing funds from participants. However, centralised tumbling protocols cannot ensure the availability property, because the centralised system can always censor deposits from clients.

Existing decentralized tumbler designs, such as Coinshuffle [140, 141] and Coinjoin [106], address the availability problem by proposing protocols allowing participants to interact and form transactions that helps hide the sender from the recipient. However, the availability of participants and the interactivity among them can be difficult to enforce and may lead to privacy leaking side channels.

3.9 Concluding Remarks

Coin mixers allow alleviating to some degree the missing privacy properties of open and permissionless blockchains. Their operations are cost-intensive both from a transaction fee perspective and because "better" privacy is more expensive than "weaker" privacy when measuring privacy quality quantitatively with the anonymity set size.

In this work, we introduce a zk-SNARK-based coin mixer AMR. AMR is to our knowledge the first construction that allows to reward mixer participants which hold coins within the mixer for at least time t. Moreover, AMR allows users to earn interest on the deposited funds by leveraging popular DeFi lending platforms. This incentive mechanism should not only attract privacy-seeking users, but also participants that are interested in the underlying reward distribution. Therefore, we hope that such a system fundamentally broadens the diversity of the mixer user, improving the anonymity set quality for all involved users. Our implementation and evaluation shows that our mixer is practical by supporting anonymity set sizes beyond thousands of users.

Part II

Addressing Communication Overhead with Private Payment Channels

4. ADDRESSING COMMUNICATION OVERHEAD WITH PAYMENT CHANNELS IN MONERO

Bitcoin does not provide thorough privacy guarantees as largely demonstrated in the literature [109, 10, 7, 90, 134, 148]. In this state of affairs, Monero appeared in the cryptocurrency landscape with the distinguishing factor of adopting privacy by a design principle, combining for the first time *stealth address* [142], *linkable ring signatures* [101], *cryptographic commitments* [125] and *range proofs* [34]. As of the time of this writing, Monero has been regularly among the top 15 cryptocurrencies in market capitalization, has processed more than 6 million transactions since its creation [156], and is the most popular CryptoNote-style cryptocurrency [46]. Currently, the Monero blockchain processes around $4,000$ daily transactions and Monero coins are parts of a daily trade volume of more than 76M USD [40]. However, Monero leaves significant room for improvement. First, Monero suffers from *reduced expressiveness*: While cryptocurrencies like Bitcoin or Ethereum enable somewhat complex policies for spending coins (e.g., a coin can be governed by script-based rules), Monero only supports coins governed with (mostly a single) private key, reducing the functionality to simple transfer of coins with no policy associated with it.

Cryptocurrencies such as Bitcoin and Ethereum overcome this lack of expressiveness by adding script languages at the cost of fungibility [159] (i.e., transaction inputs/outputs can be easily distinguished by their script) and interoperability as those script languages are not compatible with each other. Thus, it is interesting to include new policies on spending Monero coins *cryptographically*, instead of including a scripting language that hampers fungibility and interoperability.

Second, Monero suffers from similar *scalability issues* as Bitcoin [45]: The permissionless nature of the Monero consensus algorithm limits the block rate to one block every two minutes on average. In fact, the scalability problem in Monero is more pressing. The crucial privacy goal in Monero relies on well-established cryptographic constructions to homogenize transactions: linkable ring signatures are used to obfuscate what public key corresponds to the signer of a transaction while commitment schemes and range proofs are leveraged to hide the exchanged amounts, ensure transaction validity and the expected coin supply. These key

design choices make Monero transactions require higher on-chain footprint than transactions in other cryptocurrencies. Although used only for less than five years, the Monero blockchain has currently a size of 59.37 GB and grows at around 635MB per month [117].

Given this trend, it would be interesting to enable payment channels and payment channel networks [103, 129, 132] in Monero, a scalability solution already adopted in Bitcoin and Ethereum where the transaction rate is no longer limited by the global consensus but rather by the latency among the two users involved in a given payment. However, this is far from trivial as current payment-channel networks are built upon script languages (e.g., hash-time lock contract) or digital signatures schemes such as ECDSA or Schnorr that are not available in Monero. Leveraging these techniques in Monero would hamper its fungibility.

In summary, the current state of affairs in Monero with respect to the reduced expressiveness, lack of interoperability, and severe scalability issues calls for a solution. Adopting solutions provided in other cryptocurrencies like Bitcoin and Ethereum is not seamlessly possible as they are not backwards compatible with Monero. Moreover, as aforementioned, the inclusion of a scripting language would hamper the fungibility and interoperability of Monero.

Our contributions. In this work, we present *Dual Linkable Spontaneous Anonymous Group Signature for Ad Hoc Groups (DLSAG)*, the linkable ring signature scheme for Monero that improves upon the lack expressiveness, interoperability, and scalability guarantees in Monero. In particular:

- **Expressiveness**. We formalize DLSAG (Section 4.2), a new linkable ring signature scheme that relies only on cryptographic tools already available in Monero and improves its expressiveness. In a bit more detail, DLSAG enables for the first time that Monero coins can be spent with one of two signing keys, depending on the relation between a time flag and the height of the current block in the Monero blockchain.

- **Scalability**. We describe how to leverage the DLSAG signatures to encode for the first time non-interactive refund transactions in Monero, where Alice can pay to Bob a certain amount of coins redeemable by Bob before a certain time in the future. After such time expires, the coins can be refunded to Alice. Refund transactions are

the building block that opens the door for the first time to scalability solutions based on payment channels for Monero (Section 4.5). In particular, we describe how to build uni-directional payment channels, payment-channel networks, off-chain conditional payments and atomic swaps.

- **Interoperability**. We further show that it is possible to combine the aforementioned payment channels protocols with the corresponding ones in other cryptocurrencies, making thereby Monero interoperable (Section 4.5).

- **Formal analysis**. We formally prove that DLSAG achieves the security and privacy goals of interest for linkable ring signatures, namely, unforgeability, signer ambiguity, and linkability as introduced in [101] (Section 4.2).

- **Implementation and adoption**. We have implemented DLSAG and evaluated its performance (Section 4.3) showing that it imposes a single bit more of communication overhead and smaller computation overhead as the current digital signature scheme in Monero, demonstrating thus its practicality. In fact, DLSAG is a new result that paves the way in practice towards an expressiveness and scalability solution urgently needed in Monero to improve its integration in the cryptocurrency landscape. DLSAG is actively being discussed within the Monero community as an option for adoption [121, 135] and it is compatible with other CryptoNote-style cryptocurrencies [46].

Comparison with related work. Poelstra introduced the notion of *Scriptless Scripts* [128] as a means of encoding somewhat limited smart contracts that no longer require the Bitcoin scripting language. Malavolta et al. [102] formalized this notion and extended it to support Schnorr and ECDSA digital signatures. In this work, we instantiate the notion of Scriptless Scripts to realize conditional payments compatible with DLSAG and the current Monero protocol. Bitcoin payment channels [124, 129, 51] have been presented in the literature as a scalability solution for the Bitcoin blockchain. Bitcoin payment channels have been then leveraged to build payment-channel networks in academia [74, 103, 88] and in industry [129, 132, 127]. However, none of these solutions are compatible with the current Monero. They rely on either Bitcoin script [103, 129, 132], ZCash script [74],

Ethereum contracts [88] or Schnorr signature scheme [127], none of which are available in Monero. Similarly, Bitcoin scripts have been leveraged to construct an atomic swap protocol [30]. We, instead, present a payment-channel network and atomic swap protocols that no longer require scripting language, and it is compatible with Monero. Goodell and Noether have proposed threshold signatures [72] for Monero whereas Libert et al. [97] proposed a logarithmic-size ring signature from the DDH assumption; although interesting, they do not address the expressiveness, interoperability and scalability issues considered in this work.

4.1 Background

Notation. We denote by $\mathbb{G}$ a cyclic group of prime order q and by g we denote a fixed generator of such group. We denote by $(\mathsf{pk}, \mathsf{sk})$ a pair of public and secret keys. We denote by $\vec{\mathsf{pk}}$ an array of public keys. We use letters A to Z to identify users in a protocol. We denote by $\mathtt{XMR}$ the Monero coins. Finally, we consider two hash functions: (i) $\mathsf{H_s}$ takes as input a bitstring and outputs a scalar (i.e., $\mathsf{H_s} : \{0,1\}^* \rightarrow \mathbb{Z}_q$); (ii) $\mathsf{H_p}$ takes as input a bitstring and outputs an element of $\mathbb{G}$ (i.e., $\mathsf{H_p} : \{0,1\}^* \rightarrow \mathbb{G}$).

Transactions. A Monero transaction [142] is divided in *inputs* and *outputs*. They are defined in terms of tuples of the form $(\mathsf{pk}, \textsc{Com}(\gamma), \Pi\text{-}amt)$ where pk denotes a fresh public key, $\textsc{Com}(\gamma)$ denotes a *cryptographic commitment* [125] to the amount γ and Π-*amt* denotes a *range proof* [34] that certifies that the committed amount is within a range $[0, 2^k]$ where k is a system parameter. In particular, each input consists of a set of such tuples while each output consists of a single tuple. The set of public keys included in an input is called a *ring*. Finally, the transaction includes a digital signature σ for each input.

In the illustrative example shown in Fig. 4.1, we assume that Alice has previously received 5 $\mathtt{XMR}$ in the public key $\mathsf{pk_A}$. We also assume that she wants to pay Bob 4 $\mathtt{XMR}$. For that, Alice first should get Bob's public key ($\mathsf{pk_B}$) and a fresh public key for herself ($\mathsf{pk_A}$) to keep the change amount. Second, Alice should choose a set of $n-1$ output tuples $\{(\mathsf{pk}_i, \textsc{Com}(v_i), \Pi\text{-}amt_i)\}$ already available in the Monero blockchain to complete the input. Finally, Alice should create a valid signature of the transaction content using the ring

Inputs:
[0] $\{(\mathsf{pk}_1, \textsc{Com}(v_1), \Pi\text{-}amt_1), \ldots, (\mathsf{pk}_{n-1}, \textsc{Com}(v_{n-1}), \Pi\text{-}amt_{n-1}),$ $(\mathsf{pk}_\mathsf{A}, \textsc{Com}(5), \Pi\text{-}amt_\mathsf{A})\}$
Outputs:
[0] pk_B, $\textsc{Com}(4)$, $\Pi\text{-}amt_\mathsf{B}$; [1] pk_A, $\textsc{Com}(1)$, $\Pi\text{-}amt_\mathsf{A}$
Authorizations:
[0] σ

Figure 4.1. Illustrative example of a (simplified) Monero transaction. Alice (pk_A) contributes 5 XMR to pay 4 XMR to Bob (pk_B) and get 1 XMR back (pk_A). Finally, the transaction is authorized with a ring signature σ from the input ring.

$(\mathsf{pk}_1, \ldots, \mathsf{pk}_{n-1}, \mathsf{pk}_\mathsf{A})$ and her private key sk_A. For that, she uses a linkable ring signature scheme.

4.1.1 Linkable Ring Signatures (LSAG)

The signature scheme used in Monero is an instantiation of the *Linkable Spontaneous Anonymous Group Signature for Ad Hoc Groups* (LSAG)[1] signature scheme [101]. We recall the definition of LSAG in Definition 4.1.1. Here, we explicitly add a generic definition of the linking algorithm which was briefly mentioned in [101].

Definition 4.1.1 (LSAG [101])**.** *An LSAG signature scheme is a tuple of algorithms (*KeyGen*,* Sign*,* Vrfy*,* Link*) defined as follows:*

- $(\mathsf{sk}, \mathsf{pk}) \leftarrow \textsc{KeyGen}(1^\lambda)$*: The* KeyGen *algorithm takes as input the security parameter* 1^λ *and outputs a pair of private key* sk *and public key* pk*.*
- $\sigma \leftarrow \textsc{Sign}(\mathsf{sk}, \vec{\mathsf{pk}}, m)$*: The* Sign *algorithm takes as input a private key* sk*, a list* $\vec{\mathsf{pk}}$ *of* n *public keys which includes the one corresponding to* sk*, a message* m *and outputs a signature* σ*.*

[1] Monero in fact uses a matrix version of LSAG (MLSAG) [122] to prove balance without revealing spent ring members. We describe here the simplest LSAG version but our constructions can be trivially extended to support matrix version.

- $b \leftarrow \textsc{Vrfy}(\vec{\mathsf{pk}}, m, \sigma)$: *The* $\textsc{Vrfy}$ *algorithm takes as a public key list* $\vec{\mathsf{pk}}$*, a message* m *and a signature* σ*, and returns* 1 *if* $\exists \mathsf{sk}, \mathsf{pk} \leftarrow \textsc{KeyGen}(1^\lambda)$ *s.t.* $\mathsf{pk} \in \vec{\mathsf{pk}}$ *and* $\sigma := \textsc{Sign}(\mathsf{sk}, \vec{\mathsf{pk}}, m)$*. Otherwise, it returns* 0*.*

- $b \leftarrow \textsc{Link}((\vec{\mathsf{pk}}_1, m_1, \sigma_1), (\vec{\mathsf{pk}}_2, m_2, \sigma_2))$: *The* $\textsc{Link}$ *algorithm takes as input two triples* $(\vec{\mathsf{pk}}_1, m_1, \sigma_1)$ *and* $(\vec{\mathsf{pk}}_2, m_2, \sigma_2)$*. The algorithm outputs 1 if* $\exists(\mathsf{sk}, \mathsf{pk}) \leftarrow \textsc{KeyGen}(1^\lambda)$ *s.t.* $\mathsf{pk} \in \vec{\mathsf{pk}}_1$*,* $\mathsf{pk} \in \vec{\mathsf{pk}}_2$*,* $\sigma_1 := \textsc{Sign}(\mathsf{sk}, \vec{\mathsf{pk}}_1, m_1)$ *and* $\sigma_2 := \textsc{Sign}(\mathsf{sk}, \vec{\mathsf{pk}}_2, m_2)$*. Otherwise, the algorithm outputs* 0*.*

Apart from the straightforward correctness definition, Liu et al. [101] define three security and privacy goals for a LSAG signature scheme. We present them here informally and defer their formal description to Section 4.2.2.

- **Unforgeability**: The adversary without access to the secret key should not be able to compute a valid signature σ on a message m.
- **Signer ambiguity**: Given a valid signature σ on a message m, the adversary should not be able to determine better than guessing what public key within the ring corresponds to the secret key used to create the signature.
- **Linkability**: Given two rings $\vec{\mathsf{pk}}_1$, $\vec{\mathsf{pk}}_2$, two valid signatures σ_1, σ_2 in two messages m_1, m_2, there should exist an efficient algorithm that faithfully determines if the same secret key has been used to create both signatures.

The current LSAG in Monero only supports transfer of coins authorized by a signature, reducing the expressiveness to payments. Adding a script language (as done in Bitcoin or Ethereum) would harm fungibility (i.e., transaction inputs/outputs can be easily distinguished by their script) and interoperability as those languages are not compatible with each other. Instead, in this work we aim to propose a signature scheme for Monero that cryptographically supports more expressive transaction authorization policies, without hampering the security and privacy guarantees of the current digital signature scheme.

Linkable Ring Signature in Monero. Fig. 4.2 shows the construction of LSAG originally used in the current Monero cryptocurrency.

Construction of LSAG in Monero [122]

- $(\mathsf{sk}, \mathsf{pk}) \leftarrow \textsc{KeyGen}(1^\lambda)$: Choose sk uniformly at random and set $\mathsf{pk} := g^{\mathsf{sk}}$. Output sk, pk.
- $\sigma \leftarrow \textsc{Sign}(\mathsf{sk}, \vec{\mathsf{pk}}, \mathtt{tx})$: Parse: $(\mathsf{pk}_1, \ldots, \mathsf{pk}_n) \leftarrow \vec{\mathsf{pk}}$. Sample $s_0, s_1, \ldots, s_{n-1}$ from $\mathbb{Z}_q$. Compute:

$$\mathcal{I} := \mathsf{H_p}(\mathsf{pk}_n)^{\mathsf{sk}}; L_0 := g^{s_0};\ R_0 = \mathsf{H_p}(\mathsf{pk}_n)^{s_0};$$
$$h_0 := \mathsf{H_s}(\mathtt{tx}||L_0||R_0).$$

  For $i \in \{1, \ldots, n-1\}$ compute the following series:

$$L_i := g^{s_i} \cdot \mathsf{pk}_i^{h_{i-1}};\ R_i := \mathsf{H_p}(\mathsf{pk}_i)^{s_i} \cdot \mathcal{I}^{h_{i-1}}$$
$$h_i := \mathsf{H_s}(\mathtt{tx}||L_i||R_i)$$

  Solve for s_0 such that: $\mathsf{H_s}(\mathtt{tx}||g^{s_0} \cdot \mathsf{pk}_n^{h_{n-1}}||\mathsf{H_p}(\mathsf{pk}_n)^{s_0} \cdot \mathcal{I}^{h_{n-1}}) = h_0$. For that, we get that $s_0 = s_0 - h_{n-1} \cdot \mathsf{sk}$. Return $\sigma = (s_0, s_1, \ldots, s_{n-1}, h_0, \mathcal{I})$
- $b \leftarrow \textsc{Vrfy}(\vec{\mathsf{pk}}, \mathtt{tx}, \sigma)$: Parse:

$$(s_0, s_1, \ldots, s_{n-1}, h_0, \mathcal{I}) \leftarrow \sigma,\ (\mathsf{pk}_1, \ldots, \mathsf{pk}_n) \leftarrow \vec{\mathsf{pk}}$$

  For $i \in \{1, \ldots, n\}$, compute the sequences:

$$L_i := g^{s_i} \cdot \mathsf{pk}_i^{h_{i-1}};\ R_i := \mathsf{H_p}(\mathsf{pk}_i)^{s_i} \cdot \mathcal{I}^{h_{i-1}}$$
$$h_i := \mathsf{H_s}(\mathtt{tx}||L_i||R_i)$$

  Return 1 if $h_0 = h_n$. Otherwise, return 0.
- $b \leftarrow \textsc{Link}((\vec{\mathsf{pk}}_1, \mathtt{tx}_1, \sigma_1), (\vec{\mathsf{pk}}_2, \mathtt{tx}_2, \sigma_2))$: If $(\textsc{Vrfy}(\vec{\mathsf{pk}}_1, \mathtt{tx}_1, \sigma_1) \wedge \textsc{Vrfy}(\vec{\mathsf{pk}}_2, \mathtt{tx}_2, \sigma_2)) = 0$, return 0. Else: parse $(s_0, s_1, \ldots, s_{n-1}, h_0, \mathcal{I}_1) \leftarrow \sigma_1$ and $(s_0, s_1, \ldots, s_{n-1}, h_0, \mathcal{I}_2) \leftarrow \sigma_2$. Return 1 if $\mathcal{I}_1 = \mathcal{I}_2$. Otherwise, return 0.

Figure 4.2. Construction of LSAG in Monero [122]. For ease of exposition, in the signing algorithm we assume that the secret key sk corresponds with the n-th public key pk_n. In practice, the position of true signer's public key is chosen uniformly random.

4.1.2 Preliminaries

In order to prove the security of DLSAG, we first need to introduce the following definitions and results.

Definition 4.1.2 (Forking algorithm [15])**.** *Let $\mathcal{A}$ be a PPT algorithm that takes as input some inp. Assume $\mathcal{A}$ has access to a random oracle $\mathcal{O}^{H_s}$ that outputs random element from*

$\mathbb{Z}_q$ and the query responses are temporally ordered by index $e_0, e_1, \ldots, e_{q_H-1}$. Define the forking algorithm associated with $\mathcal{A}$, denoted $\mathtt{F}_{\mathcal{A}}$, as the following algorithm:

1. *Take as input some inp, select random coins ρ for $\mathcal{A}$, and select q_H oracle query responses, $e_0, e_1, \ldots, e_{q_H-1} \xleftarrow{\$} \mathbb{Z}_q$.*
2. *Execute $\alpha \leftarrow \mathcal{A}(inp; \rho)$, responding to the i^{th} query to $\mathcal{O}^{H_s}$ made by $\mathcal{A}$ with the response e_i.*
3. *If $\alpha = \perp$ return $\perp$ and terminate. Otherwise, parse $(j, out) \leftarrow \alpha$.*
4. *Select new oracle query responses $e'_j, e'_{j+1}, \ldots, e'_{q_H-1} \xleftarrow{\$} \mathbb{Z}_q$.*
5. *Execute $\alpha' \leftarrow \mathcal{A}(inp; \rho)$, responding to the i^{th} query to $\mathcal{O}^{H_s}$ made by $\mathcal{A}$ with the response e_i when $i < j$ and e'_i otherwise.*
6. *If $\alpha' = \perp$, return $\perp$ and terminate. Otherwise, parse $(j', out') \leftarrow \alpha'$.*
7. *If $j = j'$ and $e_j \neq e'_j$, return (j, out, out'). Otherwise, return $\perp$.*

Lemma 4.1.1 (Generalized Forking Lemma [15])**.** *Let q_H be an integer, $\mathcal{A}$ be a randomized algorithm which takes as input some main input inp and $h_0, h_1, \ldots, h_{q_H-1} \in \mathbb{Z}_q$ and returns either a distinguished failure symbol $\perp$ or a pair (j, out), where $0 \leq j < q$ and out is some side output. The accepting probability of $\mathcal{A}$, denoted $acc(\mathcal{A})$, is defined as the probability that $\mathcal{A}$ does not output $\perp$ (where this probability is measured over the random selection of inp, $\{e_i\}_{i=0}^{q_H-1}$, and $\{e'_i\}_{i=j}^{q_H-1}$). Let $\mathcal{B}$ be the forking algorithm associated with $\mathcal{A}$ from Definition 4.1.2. Let $acc(\mathcal{B})$ be the probability (over the draw of inp and the random coins of $\mathcal{B}$) that $\mathcal{B}$ returns a non-$\perp$ output. Then*

$$acc(\mathcal{B}) \geq acc(\mathcal{A}) \left(\frac{acc(\mathcal{A})}{q_H} - \frac{1}{q} \right).$$

In particular, if $\mathcal{A}$ has non-negligible acceptance probability, then so does $\mathcal{B}$.

Definition 4.1.3 (One-More Discrete Logarithm Hardness [12])**.** *Let λ be a security parameter. Let N be natural number such that $1 \leq N < \mathsf{poly}\lambda$. Let $(\mathbb{G}, q, g) \leftarrow \textsc{Setup}(1^\lambda)$ be some*

group parameters. Let $\mathcal{O}^C$ be a corruption oracle. For any fixed N, these group parameters are said to satisfy the one-more discrete logarithm hardness (OMDL) assumption for N if any PPT algorithm $\mathcal{A}$ has at most a negligible probability of success in the following game.

1. *A sequence of $N+1$ independent and identically distributed observations of a uniform random variable on $\mathbb{G}$ are made, $S = \{\mathcal{H}_0, \ldots, \mathcal{H}_N\} \subseteq \mathbb{G}$. The group parameters $(\mathbb{G}, q, g)$ and the set S are sent to $\mathcal{A}$.*
2. *$\mathcal{A}$ is granted oracle access to $\mathcal{O}^C$.*
3. *$\mathcal{A}$ outputs an index $0 \leq i \leq N$ and a scalar $x \in \mathbb{Z}_q$.*

$\mathcal{A}$ succeeds if $g^x = \mathcal{H}_i$, the corruption oracle $\mathcal{O}^C$ is not queried with $\mathcal{H}_i$, and the corruption oracle $\mathcal{O}^C$ is queried at most N times.

Definition 4.1.4. *If $\mathcal{A}$ is an algorithm that runs in time at most t and succeeds at the one-more discrete logarithm game for some N with probability at least ϵ, then we say $\mathcal{A}$ is a (t, ϵ, N)-OMDL solver where ϵ is measured over the joint distribution of the random coins of $\mathcal{A}$ and the challenge group elements $\mathcal{H}_i$.*

Definition 4.1.5 (Decisional Diffie-Hellman Assumption)**.** *Let $(\mathbb{G}, q, g)$ be the group parameters. We say the Decisional Diffie-Helman Problem is hard relative to $\mathbb{G}$ if for all probabilistic polynomial time algorithms $\mathcal{M}$ there exist a negligible function $\epsilon(\cdot)$ such that*

$$\begin{aligned}\Pr\big[\mathcal{M}(\mathbb{G}, g, q, A, B, C) = b : (A, B, C) = (A_b, B_b, C_b) \\ \text{where } (A_0, B_0, C_0) = (g^{a_0}, g^{b_0}, g^{c_0}); \\ (A_1, B_1, C_1) = (g^{a_1}, g^{b_1}, g^{a_1 b_1})\big] \\ \leq \frac{1}{2} + \epsilon(\lambda)\end{aligned}$$

where a_i, b_i, c_i for $i \in \{0, 1\}$ are uniformly chosen from $\mathbb{Z}_q$.

4.2 Dual-Key LSAG (DLSAG)

In this section, we first describe DLSAG, our digital signature scheme for linkable ring signatures.

4.2.1 Key ideas and construction of DLSAG

Our approach builds upon a *tuple format* defined as $((\mathsf{pk}_{\mathsf{A},0}, \mathsf{pk}_{\mathsf{B},1}), \textsc{Com}(\gamma), \Pi\text{-}amt, t)$ and that enables to spend it to two different public keys (and potentially two different users) depending on a flag t. A dual-key tuple deviates from the current Monero tuple in two main points (highlighted in blue): (i) it contains two public keys instead of one to identify the two users that can possibly spend the output; and (ii) it includes an additional element t that denotes a switch (e.g., $\mathsf{pk}_{\mathsf{A},0}$ is used if t is smaller than the current block height in the Monero blockchain) between the public keys.

Dual-key tuple format enables the encoding of the logic for a refund transaction. In the sample tuple shown above, assume that t signals that $\mathsf{pk}_{\mathsf{A},0}$ must be used. Then Alice must choose a ring of the form $(\vec{\mathsf{pk}}_0, \vec{\mathsf{pk}}_1)$, containing $(\mathsf{pk}_{\mathsf{A},0}, \mathsf{pk}_{\mathsf{B},1})$ at some position, and sign with the secret key sk_{A}, that is, the secret key corresponding to the public key $\mathsf{pk}_{\mathsf{A},0}$. Conversely, if t signals that $\mathsf{pk}_{\mathsf{B},1}$ must be used, Bob can then sign with sk_{B} instead. Note that if a single user knows both sk_{A} and sk_{B}, such an user can always use a dual-key tuple independently of the value t.

The remaining step is to design a linkable ring signature scheme that supports this new tuple format. This, however, requires to address the following challenges.

Key-image mechanism. The ring signature scheme currently used in Monero achieves linkability by publishing the key-image constructed from the single public key. For instance, Alice produces a signature with sk_{A}; the signature will contain the key-image $\mathcal{I} = \mathsf{H}_{\mathsf{p}}(\mathsf{pk}_{\mathsf{A}})^{\mathsf{sk}_{\mathsf{A}}}$. If Alice signs again with sk_{A}, the same key-image would be computed and this can be detected. To mimic this behavior while handling the dual-key tuple format, the challenge is to define a single key-image that uniquely identifies a pair of public keys $(\mathsf{pk}_0, \mathsf{pk}_1)$ and yet can be computed knowing only one of the signing keys sk_b. Similar to the Diffie-Hellman key exchange mechanism [56], our approach redefines the key-image as $\mathcal{J} = g^{\mathsf{sk}_0 \cdot \mathsf{sk}_1}$, fulfilling thereby the expected requirements: (i) knowing sk_b suffices to compute $\mathcal{J} := \mathsf{pk}_{1-b}^{\mathsf{sk}_b}$; (ii) it uniquely identifies $(\mathsf{pk}_0, \mathsf{pk}_1)$ since $\mathsf{pk}_{1-b}^{\mathsf{sk}_b} = \mathsf{pk}_b^{\mathsf{sk}_{1-b}}$.

Hardening key-image linkability. The aforementioned key-image definition allows to link the pair of public keys $(\mathsf{pk}_0, \mathsf{pk}_1)$. However, it is crucial to make the key-image unique not

only to the pair of public keys but also to the output that contains them itself. Otherwise, one of the users could create another dual-key tuple with the same pair of public keys, create a signature with it (and thus a key-image), and effectively make the funds in the original tuple unspendable since in Monero every key-image is only allowed to appear once. That can be mitigated by introducing a random unique identifier, m, to each output, and this identifier can be included in the computation of the key-image without violating the security and privacy guarantees of the signature scheme. In Monero, such an unique identifier can be constructed by hashing the transaction that included the output and the output's position in the transaction. Thus, we may view the rings used in DLSAGs as consisting of unique triples, $(\mathsf{pk}_0, \mathsf{pk}_1, m)_{[1,n]}$, and we define the *dual key-image* to be $\mathcal{J} := g^{m_j \cdot \mathsf{sk}_{j,0} \cdot \mathsf{sk}_{j,1}}$, for some $j \in [1, n]$ corresponding to the position of the true signer in the ring.

The rest is to follow the idea of the Monero LSAG modified to support the new linkability tag. Figure 4.3 introduces the details of the DLSAG construction.

4.2.2 Security analysis

We use the existential unforgeability of ring signatures with respect to insider corruption introduced in [18]. Signer ambiguity and linkability properties are similar to those in LSAG [101], adapted to DLSAG syntax for readability.

Definition 4.2.1. *(**Existential unforgeability of ring signature with respect to insider corruption**) Let λ be a security parameter, let N, q_H, q_S, q_C be natural numbers such that $q_C \leq N \leq \mathsf{poly}(\lambda)$, $1 \leq q_H \leq \mathsf{poly}(\lambda)$, $1 \leq q_S \leq \mathsf{poly}(\lambda)$. Let $(\mathbb{G}, q, g)$ be some group parameters from a Dual LSAG signature scheme* (KeyGen, Sign, Verify, Link)*. Let $\mathcal{O}^C$ be a corruption oracle that can be queried up to q_C times which acts as a discrete logarithm oracle. Let $\mathcal{O}^S$ be a signature oracle that can be queried up to q_S times. Presume $\mathcal{O}^S$ takes as input some ring of public keys $\vec{\mathsf{pk}}$, message m, signing index ℓ, and parity bit b, and produces as output a valid signature. Let $\mathcal{O}^H$ be a random oracle that can be queried up to q_H times.*

Construction of DLSAG

- $(\mathsf{sk}, \mathsf{pk}) \leftarrow \textsc{KeyGen}(1^\lambda)$: Choose $\mathsf{sk}_0, \mathsf{sk}_1$ uniformly at random from $\mathbb{Z}_q$, m as a bitstring chosen uniformly at random from $\{0,1\}^n$. Set both $\mathsf{pk}_b := g^{\mathsf{sk}_b}$ for $b \in \{0,1\}$. Output $\mathsf{sk} = (\mathsf{sk}_0, \mathsf{sk}_1), \mathsf{pk} = (\mathsf{pk}_0, \mathsf{pk}_1, m)$.

- $\sigma \leftarrow \textsc{Sign}(\mathsf{sk}_b, \vec{\mathsf{pk}}, \mathtt{tx})$: Parse: $((\mathsf{pk}_{1,0}, \mathsf{pk}_{1,1}, m_1), \ldots, (\mathsf{pk}_{n,0}, \mathsf{pk}_{n,1}, m_n)) \leftarrow \vec{\mathsf{pk}}$. Sample $s_0, s_1, \ldots, s_{n-1}$ from $\mathbb{Z}_q$. Compute:

$$\mathcal{J} := \mathsf{pk}_{n,1-b}^{m_n \cdot \mathsf{sk}_b};\ L_0 := g^{s_0};\ R_0 := \mathsf{pk}_{n,1-b}^{s_0 \cdot m_n};\ h_0 := \mathsf{H_s}(\mathtt{tx}||L_0||R_0);$$

  Then, for $i \in \{1, \ldots, n-1\}$, compute the following sequences:

$$L_i := g^{s_i} \cdot \mathsf{pk}_{i,b}^{h_{i-1}};\ R_i := \mathsf{pk}_{i,1-b}^{s_i \cdot m_i} \cdot \mathcal{J}^{h_{i-1}};\ h_i := \mathsf{H_s}(\mathtt{tx}||L_i||R_i)$$

  Now, solve for s_0 such that $\mathsf{H_s}(\mathtt{tx}||g^{s_0} \cdot \mathsf{pk}_{n,b}^{h_{n-1}}||\mathsf{pk}_{n,1-b}^{s_0 \cdot m_n} \cdot \mathcal{J}^{h_{n-1}}) = h_0$. For that, we get $s_0 = s_0 - h_{n-1} \cdot \mathsf{sk}_b$. Return: $\sigma = (s_0, s_1, \ldots, s_{n-1}, h_0, \mathcal{J}, b)$.

- $b \leftarrow \textsc{Vrfy}(\vec{\mathsf{pk}}, \mathtt{tx}, \sigma)$: Parse

$$(s_0, s_1, \ldots, s_{n-1}, h_0, \mathcal{J}, b) \leftarrow \sigma;\ ((\mathsf{pk}_{1,0}, \mathsf{pk}_{1,1}, m_1), \ldots, (\mathsf{pk}_{n,0}, \mathsf{pk}_{n,1}, m_n)) \leftarrow \vec{\mathsf{pk}}$$

  For $i \in \{1, \ldots, n\}$, compute the sequences:

$$L_i := g^{s_i} \cdot \mathsf{pk}_{i,b}^{h_{i-1}};\ R_i := \mathsf{pk}_{i,1-b}^{s_i \cdot m_i} \cdot \mathcal{J}^{h_{i-1}};\ h_i := \mathsf{H_s}(\mathtt{tx}||L_i||R_i)$$

  Return 1 if $h_0 = h_n$. Otherwise, return 0.

- $b \leftarrow \textsc{Link}((\vec{\mathsf{pk}}_1, \mathtt{tx}_1, \sigma_1), (\vec{\mathsf{pk}}_2, \mathtt{tx}_2, \sigma_2))$: If $(\textsc{Vrfy}(\vec{\mathsf{pk}}_1, \mathtt{tx}_1, \sigma_1) \wedge \textsc{Vrfy}(\vec{\mathsf{pk}}_2, \mathtt{tx}_2, \sigma_2)) = 0$: return 0. Else, parse: $(s_0, s_1, \ldots, s_{n-1}, h_0, \mathcal{J}_1, b_1) \leftarrow \sigma_1$ and $(s_0, s_1, \ldots, s_{n-1}, h_0, \mathcal{J}_2, b_2) \leftarrow \sigma_2$. Return 1 if $\mathcal{J}_1 = \mathcal{J}_2$, and 0 otherwise.

Figure 4.3. Construction of DLSAG. For ease of exposition, we assume that the secret key sk_b corresponds with the public key $pk_{n,b}$. As noted before, the position of the true signer's public key is chosen uniformly random.

The Dual LSAG signature scheme is said to be existentially unforgeable with respect to insider corruption if any PPT algorithm $\mathcal{A}$ has at most a negligible probability of success in the following game.

1. *The challenger selects a set of N public keys from the Dual LSAG signature scheme key space $\vec{PK} \leftarrow \left\{(\mathsf{pk}_{1,0}, \mathsf{pk}_{1,1}, m_0), \cdots, (\mathsf{pk}_{N,0}, \mathsf{pk}_{N,1}, m_N)\right\}$ and sends this set to the player $\mathcal{A}$.*

2. *The player is granted access to oracles $\mathcal{O}^C$, $\mathcal{O}^S$, and $\mathcal{O}^H$.*

3. *The player outputs a message m, a ring of public keys $\vec{\mathsf{pk}} = \{(Y_{1,0}, Y_{1,1}, m'_1), (Y_{2,0}, Y_{2,1}, m'_2), \cdots, (Y_{R,0}, Y_{R,1}, m'_R)\} \subseteq \vec{PK}$ where $R \geq 1$ and a purported forgery (σ, b).*

The player $\mathcal{A}$ wins if $\textsc{Verify}(\vec{pk}, m, \sigma) = 1$ and the following additional success constraints are satisfied:

- *The keys in $\vec{\mathsf{pk}}$ are distinct and every key $(Y_{i,0}, Y_{i,1}, m'_i) \in \vec{\mathsf{pk}}$ satisfies $(Y_{i,0},\ Y_{i,1},\ m'_i) = (\mathsf{pk}_{j(i),0}, \mathsf{pk}_{j(i),1}, m_{j(i)}) \in \vec{PK}$ for some $j(i)$;*
- *$\mathcal{O}^C$ has not been queried with any $Y_{i,b}$ for any i;*
- *The purported forgery is not a complete copy of a query to $\mathcal{O}^S$ with its corresponding response.*

Definition 4.2.2. ***Existential unforgeability with respect to insider corruption [18]*** *For a fixed N, q_H, q_S, and q_C, if $\mathcal{A}$ is an algorithm that operates in the game defined Definition 4.2.1 in time at most t and succeeds at the above game with probability at least ϵ, we say $\mathcal{A}$ is a $(t, \epsilon, N, q_H, q_S, q_C)$-forger where ϵ is measured over the joint distribution of the random coins of $\mathcal{A}$ and the challenge set $\vec{PK}$.*

Definition 4.2.3 (DLSAG signer ambiguity [101])**.** *A DLSAG signature scheme with security parameter λ is signer ambiguous if for any PPT algorithm $\mathcal{A}$, on inputs any message m, any list $\vec{\mathsf{pk}}$ of n public key pairs, any valid signature σ on $\vec{\mathsf{pk}}$ and m generated by user π, such that $\mathsf{sk}_\pi \notin \mathcal{D}_t$ and any set of t private keys $\mathcal{D}_t := \{\mathsf{sk}_1, \ldots, \mathsf{sk}_t\}$ where $\{g^{\mathsf{sk}_1}, \ldots, g^{\mathsf{sk}_t}\} \subset \vec{\mathsf{pk}}_b$, $n - t \geq 2$ and b is extracted from σ. There exists a negligible function $\mathsf{negl}(\cdot)$ such that:*

$$\left| \Pr\left[\mathcal{A}(m, \vec{\mathsf{pk}}, \mathcal{D}_t, \sigma) = \pi\right] - \frac{1}{n-t} \right| \leq \mathsf{negl}(\lambda)$$

Definition 4.2.4 (DLSAG linkability)**.** *A DLSAG signature scheme is linkable if there exists a PPT algorithm $\textsc{Link}$ that takes as input two rings $\vec{\mathsf{pk}}_1, \vec{\mathsf{pk}}_2$, two messages $\mathsf{tx}_1, \mathsf{tx}_2$, their corresponding DLSAG signatures σ_1, σ_2 (with respective true signing indices π_1 and π_2 not*

provided to LINK*), and outputs either 0 or 1, such that there exists a negligible function* $\mathsf{negl}(\cdot)$ *with the property that:*

$$\Pr\left[\textsc{Link}((\vec{\mathsf{pk}}_1, \mathsf{tx}_1, \sigma_1), (\vec{\mathsf{pk}}_2, \mathsf{tx}_2, \sigma_2)) = 1 | (\mathsf{pk}_{\pi_1}, m_{\pi_1}) \neq (\mathsf{pk}_{\pi_2}, m_{\pi_2})\right] +$$
$$\Pr\left[\textsc{Link}(\vec{\mathsf{pk}}_1, \mathsf{tx}_1, \sigma_1), (\vec{\mathsf{pk}}_2, \mathsf{tx}_2, \sigma_2)) = 0 | (\mathsf{pk}_{\pi_1}, m_{\pi_1}) = (\mathsf{pk}_{\pi_2}, m_{\pi_2})\right] \leq \mathsf{negl}(\lambda)$$

Theorem 4.2.1 (DLSAG unforgeability)**.** *DLSAG signature scheme is existentially unforgeable against adaptive chosen-plaintext attack (cf. Definition 4.2.2) provided that the One-More Discrete Logarithm (OMDL) problem* [2] *is hard, under the random oracle model.*

Proof. We construct (t', ϵ', N')-OMDL solver $\mathcal{B}$ from a $(t, \epsilon, N, q_H, q_S, q_C)$-forger $\mathcal{A}$. $\mathcal{A}$ takes as input a set of N public keys from the signature scheme, has q_S oracle queries available to a signing oracle $\mathcal{O}^S$, has q_H oracle queries available to a random oracle $\mathcal{O}^{H_s}$, and has q_C oracle queries available to a corruption oracle $\mathcal{O}^C$. We wrap $\mathcal{A}$ in an algorithm $\mathcal{A}'$ with the same oracle access that is appropriate for use in the forking algorithm.

$\mathcal{B}$ takes as input a set of $N' + 1 = 2N$ group elements (the challenge points) and has up to N' queries available to a corruption oracle $\mathcal{O}^C$. $\mathcal{B}$ executes a forking algorithm $\mathtt{F}_{\mathcal{A}'}$ as a black box, passing the challenge points onto $\mathtt{F}_{\mathcal{A}'}$ as input, which in turn forks a black box execution of $\mathcal{A}'$ (the simple wrapper of $\mathcal{A}$) using the challenge points as input.

$\mathcal{B}$ answers corruption oracle queries made by $\mathtt{F}_{\mathcal{A}'}$ by querying $\mathcal{O}^C$ directly and passing along the result. $\mathtt{F}_{\mathcal{A}'}$ answers corruption oracle queries for $\mathcal{A}'$ by passing them along to $\mathcal{B}$. $\mathtt{F}_{\mathcal{A}'}$ simulates responses to random oracle queries to $\mathcal{O}^{H_s}$ (or signing oracle queries to $\mathcal{O}^S$, respectively) made by $\mathcal{A}'$ by flipping coins.

In a transcript resulting in a successful forgery, $\mathcal{A}'$ queries the random oracle during verification with all queries of the form

$$h_{i+1} \leftarrow \mathcal{O}^{H_s}(\mathtt{tx} \mid\mid g^{s_i} \cdot Y_{i,b}^{h_i} \mid\mid Y_{1,(1-b)}^{s_i} \cdot \mathcal{J}^{h_i}).$$

That is to say, $\mathcal{A}'$ does not guess h_{i+1} but actually queries the random oracle at least once in each transcript (except in transcripts that occur with negligible probability). To

[2] ↑The One-More Discrete Logarithm hardness assumption is defined in [12].

see why, note that if $\mathcal{A}$ does not make one of these queries, then $\mathcal{A}$ is selecting h_{i+1} at random by flipping coins and later discovering that h_{i+1} is precisely the image of some $(\mathtt{tx} \mid\mid g^{s_i} \cdot \mathsf{pk}_{i,b}^{h_i} \mid\mid \mathsf{pk}_{i,1-b}^{s_i} \cdot \mathcal{J}^{h_i})$ through the random oracle. This occurs with probability at most $1/q$ which is negligible.

In the transcript of $\mathcal{A}'$, queries made to the random oracle occur in linear order; denote the responses received by $\mathcal{A}$ as $e_0, e_1, e_2, \ldots$. Define the distinguished pair (j, i) to be the index of the oracle response e_j such that the oracle query $e_j = h_{i+1} = \mathcal{O}^{H_s}(\mathtt{tx} \mid\mid g^{s_i} \cdot Y_{i,b}^{h_i} \mid\mid Y_{1,(1-b)}^{s_i} \cdot \mathcal{J}^{h_i})$ corresponds to the first verification query made to the random oracle. *We refer to such a transcript as a (j, i)-forgery.*

Note that any algorithm executing $\mathcal{A}$ in a black box can inspect the transcript of $\mathcal{A}$ and extract the pair (j, i) in $O(q_H)$ time. Hence, if $\mathcal{A}$ takes time t, then the simple wrapper $\mathcal{A}'$ takes time $t + O(q_H)$. Note that the acceptance probabilities $acc(\mathcal{A}) = acc(\mathcal{A}')$, and $\mathcal{A}'$ can be used in the forking lemma. The algorithm $\mathtt{F}_{\mathcal{A}'}$ runs $\mathcal{A}'$ as a black box, selecting its random tape. $\mathtt{F}_{\mathcal{A}'}$ rewinds the transcript of $\mathcal{A}'$ while preserving the random tape and the oracle responses preceding the rewind point $e_0, e_1, \ldots, e_{j-1}$. The algorithm $\mathtt{F}_{\mathcal{A}'}$ responds with new random values $e'_j, e'_{j+1}, \ldots$ from that point forward. By the forking lemma, if $\mathcal{A}'$ has success probability $acc(\mathcal{A}) > \epsilon$, then $\mathtt{F}_{\mathcal{A}'}$ has success probability $acc(\mathcal{B}) > \epsilon \left(\frac{\epsilon}{q_h} - \frac{1}{q}\right)$. In particular, if ϵ is non-negligible, then so is $acc(\mathtt{F}_{\mathcal{A}'})$.

For timing, the forking algorithm associated with $\mathcal{A}'$ runs in twice the time of $\mathcal{A}'$ in addition to whatever additional time is required to simulate the oracle queries made by $\mathcal{A}'$. In particular, since $\mathcal{A}'$ runs in time $t + O(q_H)$, $\mathtt{F}_{\mathcal{A}}$ runs in time at most $2t + O(4q_H + 2q_S)$.

Now in both transcripts produced by $\mathtt{F}_{\mathcal{A}'}$, the first random oracle query relevant to the forgery is the j^{th} query, and in both transcripts, the inputs to this query are identical. However, in each transcript, the query responses are different. In the first transcript we have

$$e_j = \mathcal{O}^{H_s}(\mathtt{tx} \mid\mid L \mid\mid R)$$

and in the second transcript we have

$$e'_j = \mathcal{O}^{H_s}(\mathtt{tx} \mid\mid L \mid\mid R)$$

for some $e_j \neq e'_j$, and where the inputs to these queries are identical.

Since $\vec{pk}$ is included in $\mathtt{tx}$, the ring of public keys in the forgery is the same in each transcript. At this point in the transcript, the forger may not have decided which ring member this assignment is made to, i.e. may not have decided upon an index i or value s_i such that $L = g^{s_i} \cdot Y_{i,b}^{h_i}$, and $R = Y_{1,(1-b)}^{s_i} \cdot \mathcal{J}^{h_i}$. Certainly the forger cannot know the values of h_i except with negligible probability, either, since the index j was selected to be the first oracle query used in verification of the forgery.

In fact, since $e_j \neq e'_j$ and this is the first oracle query made, the probability that the subsequent signature challenges $\{h_i\}_i$ are identical in each transcript is negligible. Yet the forger has produced from the first transcript some s_i, h_i and from the second transcript some s'_i, h'_i such that $L = g^{s_i} \cdot Y_{i,b}^{h_i} = g^{s'_i} \cdot Y_{i,b}^{h'_i}$.

Any algorithm running the forking algorithm $\mathtt{F}_{\mathcal{A}'}$ as a black box learns the index i common to both transcripts, the signing data from each transcript s_i and s'_i and the challenges h_i, h'_i from those transcripts, and can compute the discrete logarithm

$$Y_{i,b} = g^{\frac{s'_i - s_i}{h_i - h'_i}}$$

in time that is $O(1)$ related to inverting scalars.

Hence, $\mathcal{B}$ takes $2N$ group elements as input, runs in time at most $2t + O(4q_H + 2q_S + 1)$, has acceptance probability at least $\epsilon\left(\frac{\epsilon}{q_h} - \frac{1}{q}\right)$, makes at most $q_C \leq 2N - 1 = N'$ corruption oracle queries, and yet successfully produces the discrete logarithm of at least one challenge point. □

Theorem 4.2.2 (DLSAG signer ambiguity)**.** *DLSAG achieves signer ambiguity according to Definition 4.2.3 provided that the Decisional Diffie-Hellman assumption (DDH) is hard, under the random oracle model.*

Proof. We will consider WLOG that the DLSAG is signed by the first public key of the key pair, i.e. the before-key. The case for the after key is completely analogous.

Let m be a message, and $0 \leq t \leq n - 2$. Let $\vec{pk}$ be a ring of n public keys pairs, of which t private keys are known that corresponding to some of the before-keys. Let σ be a DLSAG

of the message m, with the ring $\vec{pk}$ by a random public key of index π whose private key is not among the revealed ones.

Assume that there exists a non-negligible function $\epsilon(\cdot)$ and PPT $\mathcal{A}$ such that:

$$\Pr\left[\mathcal{A}(m, \vec{pk}, \sigma) = \pi\right] \geq \frac{1}{n-t} + \epsilon(\lambda)$$

We will use $\mathcal{A}$ to construct a PPT $\mathcal{M}$ that violates the DDH assumption with non-negligible advantage.

Indeed, without loss of generality, we provide our proof for $t = 0$. The proof for $t \neq 0$ can be carried out in the same manner.

Upon receiving the DDH triple (A, B, C), $\mathcal{A}$ picks $n-1$ public key pairs of which $\mathcal{A}$ knows corresponding private before-keys. Append (A, B) at the end to obtain an n-sized ring, $[(A_i, B_i)]_{i=1}^{n}$ Pick a random index π and swap the pair in that entry with (A, B), let that be $\vec{pk}$.

In order to generate a purported signature σ by that ring with the index π on the given message. We will toss coins to set the random oracle query responses and feed those results back to $\mathcal{A}$ when it queries the oracle for verification.

Specifically, we pick random values $s_1, \ldots, s_n, h_1, \ldots, h_n$, and define, for all $i \in \mathbb{Z}_n$ the oracle query responses as:

$$h_{i+1} := \mathsf{H_s}(m || g^{s_i} \cdot A_i^{h_i} || B_i^{s_i} \cdot C^{h_i}).$$

If $C = g^{ab}$, then the above will be a proper DLSAG signature with the given oracle; if not, then C is just a random point and shouldn't be more likely to be linked to A, B than any other pair in the ring by $\mathcal{A}$.

Since $\mathcal{A}$ is able to extract the true signer from the given key image with non-negligible advantage, we feed $\sigma = (h_1, s_1, \ldots, s_n, C)$ to it. We set $\mathcal{M}(A, B, C)$ to return 1 if $\mathcal{A}(m, \vec{pk}, \sigma) = \pi$, and return a coin toss otherwise. Computing $\mathcal{M}$'s advantage:

$$\begin{aligned}
\Pr\left[\mathcal{M}(A,B,C) = b | b = 1\right] &= \Pr\left[(\mathcal{M}(A,B,C) = b | b = 1) \wedge (\mathcal{A}(m, \vec{pk}, \sigma) = \pi)\right] \\
&\quad + \Pr\left[(\mathcal{M}(A,B,C) = b | b = 1) \wedge (\mathcal{A}(m, \vec{pk}, \sigma) \neq \pi)\right] \\
&\geq 1 \cdot \left(\frac{1}{n} + \epsilon(\lambda)\right) + \frac{1}{2}\left(1 - \frac{1}{n} - \epsilon(\lambda)\right) \\
&= \frac{1}{2} + \frac{1}{2n} + \frac{\epsilon(\lambda)}{2}
\end{aligned}$$

And

$$\begin{aligned}
\Pr\left[\mathcal{M}(A,B,C) = b | b = 0\right] &= \Pr\left[(\mathcal{M}(A,B,C) = b | b = 0) \wedge (\mathcal{A} = \pi)\right] \\
&\quad + \Pr\left[\mathcal{M}(A,B,C) = b | b = 0 \wedge \mathcal{A} \neq \pi\right] \\
&= 0\left(\frac{1}{n}\right) - \frac{1}{2}\left(1 - \frac{1}{n}\right) \\
&= \frac{1}{2} - \frac{1}{2n}
\end{aligned}$$

Combining the two equations, we get:

$$\begin{aligned}
&\Pr\left[\mathcal{M}(A,B,C) = b\right] = \\
&\quad \Pr[b = 1] \Pr\left[\mathcal{M}(A,B,C) = b | b = 1\right] \; + \Pr[b = 0] \Pr\left[\mathcal{M}(A,B,C) = b | b = 0\right] \\
&\quad \geq \frac{1}{2}\left(\frac{1}{2} + \frac{1}{2n} + \frac{\epsilon(\lambda)}{2}\right) + \frac{1}{2}\left(\frac{1}{2} - \frac{1}{2n}\right) = \frac{1}{2} + \frac{\epsilon(\lambda)}{4}
\end{aligned}$$

Since $\epsilon(\lambda)$ is non-negligible, so is $\epsilon(\lambda)/4$, which shows that $\mathcal{M}$ breaks the DDH assumption with non-negligible probability, as we wanted to show. □

Theorem 4.2.3 (DLSAG linkability)**.** *DLSAG scheme achieves linkability as defined in Definition 4.2.4 provided that the OMDL problem is hard, under the random oracle model.*

Proof. We will use the notation introduced in the previous proof. Notice that in the unforgeability proof, the discrete logarithm of $Y_{i,b}$ was extracted by comparing the two representations of the same point L. At that point, one could have also extracted the discrete logarithm of $\mathcal{J}$ with respect to the point $Y_{1,(1-b)}^{m_i}$ by comparing the two representations of the point R:

$$\mathcal{J} = Y_{1,(1-b)}^{\left(\frac{s_i' - s_i}{h_i - h_i'}\right) m_i}.$$

Moreover, those discrete logarithms are the same.

Now, if there existed a PPT adversary $\mathcal{A}$, having no prior knowledge of private keys in the b-bit component other than the private key of a certain $(\mathcal{Z}^{(0)}, \mathcal{Z}^{(1)}, m)$, that could produce a signature σ with a purported dual key image $\bar{\mathcal{J}}$, distinct from the honest key image $\mathcal{J}$.

Then we could fork $\mathcal{A}$ and extract a second signature σ' whose first verification query is the same as that of σ.

$$e_j = \mathcal{O}^{H_s}(\mathtt{tx} \mid\mid L \mid\mid R)$$

and in the second transcript we have

$$e_j' = \mathcal{O}^{H_s}(\mathtt{tx} \mid\mid L \mid\mid R)$$

for some $e_j \neq e_j'$. Writing the representations of those two points we get:

$$g^{s_i} \cdot Y_{i,b}^{h_i} = L = g^{s_i'} \cdot Y_{i,b}^{h_i'}, \text{ and}$$

$$Y_{1,(1-b)}^{s_i m_i} \cdot \bar{\mathcal{J}}^{h_i} = R = Y_{1,(1-b)}^{s_i' m_i} \cdot \bar{\mathcal{J}}^{h_i'}$$

There are two cases to consider: If $Y_{i,b} = \mathcal{Z}^{(b)}$, then, as observed at the beginning of this proof, we extract the discrete logarithm of $\bar{\mathcal{J}}$ and conclude that $\bar{\mathcal{J}} = \mathcal{J}$, a contradiction.

Otherwise, if $Y_{i,b} \neq \mathcal{Z}^{(b)}$, then, again as observed at the beginning, we extract the discrete logarithm of $Y_{i,b}$, thus solving the DLP for that point.

By the above corollary, all we are left to show is that $\Pr\left[\textsc{Link}(\mathtt{tx}_1, \sigma_1, \mathtt{tx}_2, \sigma_2) = 1 | (\mathsf{pk}_{\pi_1}, m_{\pi_1}) \neq (\mathsf{pk}_{\pi_2}, m_{\pi_2})\right]$ is negligible.

Since our LINK algorithm just compares the dual key images, this would require a PPT algorithm $\mathcal{A}$ to obtain two tuples of the form (A, B, m_1) and (C, D, m_2) such that they both have the same point as dual key image, $\mathcal{J} = g^{abm_1} = g^{cdm_2}$.

However, if the output containing the dual address (A, B) is created at the i_1 position of the output vector of transaction $\mathtt{tx}_1$, then $m_1 := \mathsf{H_s}(\mathtt{tx}_1, i_1)$. This means, by the ROM, that a and b have to be fixed before the value of m_1. Similarly, $m_2 := \mathsf{H_s}(\mathtt{tx}_2, i_2)$ can only be known after c and d are fixed.

Each side of the equation $g^{abm_1} = g^{cdm_2}$ therefore behaves as a random oracle, so the chance of they matching is negligible. This shows that our DLSAG scheme is linkable. □

Further Security and Privacy Analysis. We have analyzed the security and privacy of the digital signature scheme. Recent privacy studies on Monero [91, 119] show that composition of several transactions (and thus signatures) can lead to new threats and leakages. In particular, we observe that DLSAG allows an observer to track when the receiver spends his coin if the sender use the stealth address mechanism used in Monero to generate the one time address for the receiver. Such linkability issue can be mitigated if the receiver spends his coins as soon as he receives it.

4.3 Implementation and performance analysis

Implementation. We developed a prototypical C++ implementation [57] of DLSAG to demonstrate the feasibility of our DLSAG construction in comparison with the Monero LSAG. We have implemented DLSAG and LSAG using the same cryptographic library, `libsodium` [98], and cryptographic parameters (i.e. the ed25519 curve) as currently used in Monero.

Testbed. We condsignucted our experiments on a commodity desktop machine,sign which is equipped with Intel(R) Core(TM) i5-7400 CPU @ 3.00 GHz CPU, 12GB RAM. In these experiments, we focus on evaluating the overhead of DLSAG over LSAG in terms of computation time and signature size.

Computation Time. The results depicted in Table 4.1 show that the running time of DLSAG is practically the same as the running time of LSAG in both signing and verifying

Table 4.1. Running time (in milliseconds) of DLSAG and LSAG for different ring sizes

	LSAG		**DLSAG**	
Ring Size	Sign	Vrfy	Sign	Vrfy
5	1.929 ms	1.835 ms	1.771 ms	1.699 ms
10	3.863 ms	3.789 ms	3.665 ms	3.428 ms
15	5.873 ms	5.577 ms	5.625 ms	5.512 ms
20	8.045 ms	7.952 ms	7.516 ms	7.428 ms

algorithms. Thus, DLSAG could be included in Monero without incurring computation overhead. We estimate that the computation time for DLSAG is systematically a 7% smaller than that of LSAG. One of the main reasons is that in DLSAG, we eliminate the use of hash-to-point evaluations (e.g., as required in the old key-image mechanism). More specifically, for ring of size n, both DLSAG signing and verifying algorithms incur approximately $\approx 4n$ group operations and n hash-to-scalar evaluations while in LSAG, signing and verifying algorithms require additional n hash-to-point evaluations, which we see as the main factor for the differences in running time. Therefore, our evaluation shows that DLSAG does not impose any computation overhead in comparison to current LSAG. In fact, if adopted, DLSAG might even slightly improve the signature creation and verification times.

Signature Size. Here, we studied the overhead in terms of signature size, and thus indirectly the communication overhead imposed by DLSAG. We observed that in comparison to the LSAG signature, the signature of DLSAG has just one extra parity bit to indicate the position of the public key needed for verification (i.e., either pk_0 or pk_1). This short signature size can be achieved at the cost of higher tuple footprint. However, DLSAG enables off-chain payments and thus reducing the number of on-chain tuples required overall. In summary, this evaluation shows that DLSAG can be deployed in practice with almost no communication overhead and yet improves the scalability of Monero since it enables off-chain operations as we discuss later in this chapter.

4.4 DLSAG in Monero

Bootstrapping DLSAG in Monero. DLSAG can be seamlessly added into Monero. First, Monero regularly performs network upgrades for consensus rules and protocol improvements that allows for the integration of new functionality such as DLSAG. Second, it is possible to have transactions that mix LSAG with DLSAG. A mixed transaction will contain a LSAG signature for each single-key input and a DLSAG signature for each input in the dual-key format. In fact, both formats only differ in the number of public keys and the inclusion of an extra field (i.e. flag t). Thus, Monero operations and verifications on the commitment and range proofs remain compatible.

Fungibility. Different tuple formats coexisting on the blockchain may be detrimental to fungibility. For instance, miners might decide to stop mining certain transactions depending on the tuple format chosen. In order to mitigate that, we note that direct transfers using single-key tuples can easily be simulated by setting the two public keys of the dual-key tuples to belong to a single user. Thus, the fungibility of Monero may not be hampered with dual-key tuples only.

Backwards compatible timelock processing. Dual-key tuples contain a flag t in the clear. We envision that this flag is implemented in Monero as a block height, so that given a pair $(\mathsf{pk}_0, \mathsf{pk}_1)$, pk_0 can be used before block t is mined and pk_1 is used afterwards. Although it is unclear and an interesting future research work, it could be possible that the different t values leak enough information for an adversary to break privacy, in the spirit of Monero attacks shown in the recent literature [119, 91]. Given that, in this work we proactively propose an alternative timelock processing scheme that allows to have indistinguishable timeouts. This scheme, added as an extension to the dual-key tuple format and DLSAG signature scheme helps to maintain the fungibility of Monero. We note that this timelock processing could be of individual interest as timelocks are part of virtually all cryptocurrencies.

The core idea of the timelock processing scheme is as follows. Instead of including t in the clear, each output contains a Pedersen commitment to that value $\textsc{Com}(t, r_1)$, where r_1 is the mask value which is included along with a proof (Π-*time*) that t is in the range $[0, 2^k]$.

Inputs:
[0] $((\mathsf{pk}_{1,0}, \mathsf{pk}_{1,1}), \textsc{Com}(v_1), \Pi\text{-}amt_1, \textsc{Com}(t_1), \Pi\text{-}time_1), \ldots,$ $(\mathsf{pk}_{n-1,0}, \mathsf{pk}_{n-1,1}), \textsc{Com}(v_{n-1}), \Pi\text{-}amt_{n-1}, \textsc{Com}(t_{n-1}), \Pi\text{-}time_{n-1}),$ $((\mathsf{pk}_\mathsf{A}, \mathsf{pk}_\mathsf{A}), \textsc{Com}(10), \Pi\text{-}amt_\mathsf{A}, \textsc{Com}(t_\mathsf{A}), \Pi\text{-}time_\mathsf{A})$
Outputs:
[0] $(\mathsf{pk}_\mathsf{B}, \mathsf{pk}_\mathsf{A})$, $\textsc{Com}(10)$, $\Pi\text{-}amt_\mathsf{A}$, $\textsc{Com}(t_\mathsf{B})$, $\Pi\text{-}time_\mathsf{B}$
Authorizations:
[0] σ^0

Figure 4.4. A simplified Monero transaction using dual-key tuples and hidden timelocks.

Now, one can prove that t has expired as follows: pick t such that $t < t$. If T is a block height such that $t < T$, that would tell the miner that indeed $t < T$, and such a transaction will be mined only if the appropriate key is being used. In order to convince the miner that the relation $t < t$ holds, the signer picks a random mask r_2 and forms the Pedersen commitment $\textsc{Com}(t - t, r_2)$, and includes this commitment along with the value t, a range proof Π-*time* to prove that $t - t$ is in range $[0, 2^k]$ and other ring member information.

4.4.1 Putting all together

In this section, we use the illustrative example in Fig. 4.4 to revisit the processes of spending and verifying a transaction assuming that Monero includes dual-key tuples, supports DLSAG signature scheme and the timelock processing scheme.

Assume that Alice has previously received 10 `XMR` in the public key $(\mathsf{pk}_\mathsf{A}, \mathsf{pk}_\mathsf{A})$ (i.e., input [0]). Assume that she wants to pay Bob for a service worth 10 `XMR` with a certain timeout t_B. Thus, either Bob claims the 10 `XMR` before t_B or Alice gets them refunded at the address pk_A. For this, Alice can create the transaction shown in Fig. 4.4. After this transaction is added to the Monero blockchain, Bob can get his coins by spending the output [0]. In the following, we describe the generation of this transaction and how it can be verified by the interested party (e.g., miners).

Transaction Generation. Assume that Alice wants to spend coins held in $(\mathsf{pk}_\mathsf{A}, \mathsf{pk}_\mathsf{A})$. First, Alice invokes the SIGN algorithm for DLSAG on input $(\mathsf{sk}_\mathsf{A}, ((\mathsf{pk}_{1,0}, \mathsf{pk}_{1,1}), \ldots, (\mathsf{pk}_{n-1,0}, \mathsf{pk}_{n-1,1}), (\mathsf{pk}_\mathsf{A}, \mathsf{pk}_\mathsf{A}), \mathtt{tx})$, obtaining thereby a signature σ. Second, she has to use the timelock processing mechanism to prove that t_A has not expired. For that, she creates

the tuple $(\textsc{Com}(t_{\mathsf{A}}), t_{\mathsf{A}}, \textsc{Com}(t_{\mathsf{A}} - t_{\mathsf{A}}), \Pi\text{-}time_{\mathsf{A}})$ as mentioned above. Similar to the problem of publishing commitment of amounts, publishing $\textsc{Com}(t_{\mathsf{A}})$ would reveal what public key within the ring is being used, hindering thus signer ambiguity. Fortunately, we can adapt the approach in Monero to handle value commitments for $\textsc{Com}(t_{\mathsf{A}})$.

Transaction Validation. Every miner can validate the inclusion of Alice's transaction in a block at height T by checking whether $t_{\mathsf{A}} < T$. If so, he proceeds to verify the range proofs for the commitment values. Next, he verifies that the DLSAG signature is correct using the corresponding Vrfy algorithm. Finally, the miner checks that the dual ring signature is also correct using the Vrfy algorithm as defined in DLSAG. We remind that using the extension of DLSAG as defined in the full version [118], the miner would have to verify only one dual signature, using the DLSAG verification algorithm.

4.5 Applications in Monero Enabled by DLSAG

In this section we overview the applications that are released by the introduction of DLSAG in Monero.

4.5.1 Building blocks

Commitment Scheme. A commitment scheme $\textsc{Com} = (\mathrm{P}_{\textsc{Com}}, \mathrm{V}_{\textsc{Com}})$ consists of a commitment algorithm $\mathrm{P}_{\textsc{Com}}(m) \rightarrow (\mathsf{com}, \mathsf{decom})$ and a verification algorithm $\mathrm{V}_{\textsc{Com}}(\mathsf{com}, \mathsf{decom}, m) \rightarrow b \in \{0, 1\}$. The commiment scheme allows a prover to commit to a message m without revealing it, and the verficiation algorithm allows a verifiers to be able to verify that message m was committed using the revealed decommitment information decom.

Zero-knowledge proofs (ZKP). A ZKP system allows a prover to prove to a verifier the validity of a statement without revealing more information than the pure validity of the statement itself. In particular, a ZKP is composed by two algorithms (ZKProve, ZKVerify) defined as follows. First, the prove algorithm $\Pi \leftarrow \text{ZKProve}(st, w)$ takes as input a statement st and a witness w and returns a proof Π. The verification algorithm $\top, \bot \leftarrow \text{ZKVerify}(st, \Pi)$ takes as input a statement st and returns $\top$ if Π is a valid proof

for st. Otherwise, it returns $\perp$. We require a ZKP that fulfills the zero-knowledge, soundness and completeness properties [71].

In our constructions, we instantiate it with the sigma protocol [146], using the Fiat-Shamir heuristic to make it non-interactive [63]. For simplicity of notation, we denote by $\Pi(\{x\}, (X, g))$ a proof of the fact that $X = g^x$ where X and g are public and x is maintained private from the verifier. Moreover, we denote by $\Pi(\{x\}, (X, g) \wedge (X', g'))$ a proof of the fact that $X = g^x$ and $X' = g'^x$, where x is maintained private from the verifier and the rest of values are public.

2-of-2 DLSAG signatures. Assume that Alice and Bob want to jointly pay a receiver R for a service. We require that Alice and Bob jointly create a ring signature that spends γ $\mathtt{XMR}$ from a dual-key $(\mathsf{pk}_{\mathsf{AB},0}, \mathsf{pk}_{\mathsf{AB},1})$, distributing them as γ' to $(\mathsf{pk}_{\mathsf{R},0}, \mathsf{pk}_{\mathsf{R},1})$ and the remaining $\gamma - \gamma'$ back to themselves. For that, Alice and Bob execute $2\text{OF}2\text{RSSIGN}(\mathsf{pk}_{\mathsf{AB},b}, [\mathsf{sk}_{\mathsf{AB},b}]_{\mathsf{A}}, [\mathsf{sk}_{\mathsf{AB},b}]_{\mathsf{B}}, \mathtt{tx})$ protocol, as shown in Fig. 4.5. The $2\text{OF}2\text{RSSIGN}$ protocol largely resembles the SIGN algorithm from the DLSAG scheme. The main difference comes in the computation of $h_0 = \mathsf{H}_{\mathsf{s}}(\mathtt{tx}||g^r||\mathsf{pk}^{rm}_{\mathsf{AB},1-b})$ where he targets g^r and $\mathsf{pk}^{rm}_{\mathsf{AB},1-b}$, as well as their shared key-image $\mathcal{J}_{AB}$, have to be jointly constructed by Alice and Bob.

This protocol results in Alice and Bob obtaining their share of the signature $[\sigma]_{\mathsf{A}}$ and $[\sigma]_{\mathsf{B}}$ that they must combine to complete the final ring signature $\sigma := ([s_0]_{\mathsf{A}} + [s_0]_{\mathsf{B}}, s_1, \ldots s_{n-1}, h_0, (\mathcal{J}_{\mathsf{A}} \cdot \mathcal{J}_{\mathsf{B}}), b)$. Interestingly, Alice (and similarly Bob) can verify that $[\sigma]_{\mathsf{B}}$ is indeed a share of a valid signature σ by computing

$$g^{([s_0]_{\mathsf{A}}+[s_0]_{\mathsf{B}})} \stackrel{?}{=} \frac{(R_{\mathsf{A}} \cdot R_{\mathsf{B}})}{\mathsf{pk}^{h_{n-1}}_{\mathsf{AB},b}}, \text{ where } R_{\mathsf{A}} = g^{[s'_0]_{\mathsf{A}}} \text{ and } R_{\mathsf{B}} = g^{[s'_0]_{\mathsf{B}}}$$

4.5.2 Payment channels in Monero

Background. A *payment channel* enables several payments between two users without committing every single one of them to the blockchain. For this reason, *payment channels* are being widely developed as a scalability solution in cryptocurrencies such as Bitcoin [129]. However, the conceptual differences between Monero and Bitcoin hinder a seamless adop-

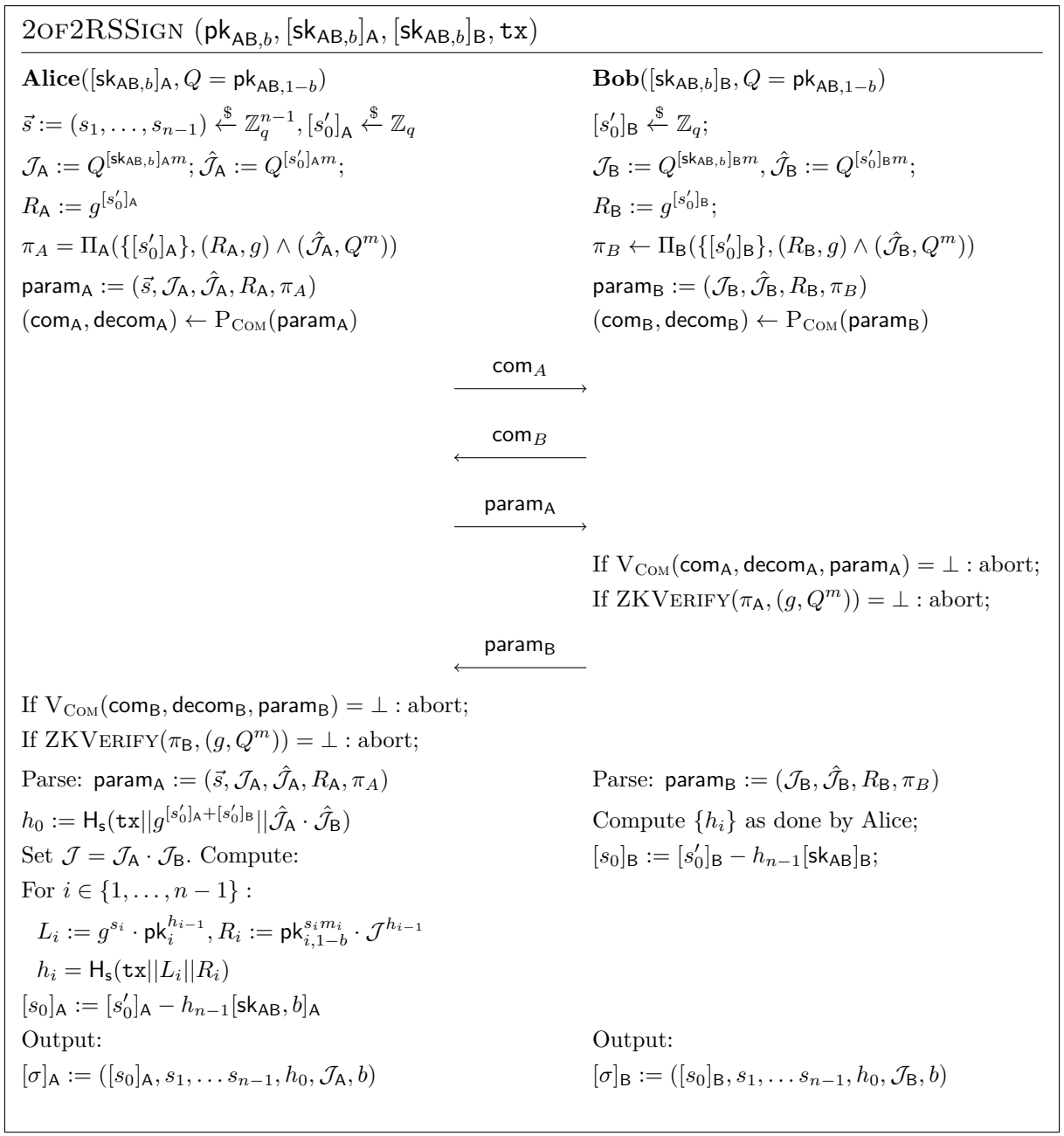


Figure 4.5. Description of the protocol 2OF2RSSIGN ($\mathsf{pk}_{\mathsf{AB},b}$, $[\mathsf{sk}_{\mathsf{AB},b}]_\mathsf{A}$, $[\mathsf{sk}_{\mathsf{AB},b}]_\mathsf{B}$, $\mathtt{tx}$), where pk_AB denotes a one-time address shared between Alice and Bob, $[\mathsf{sk}_{\mathsf{AB},b}]_\mathsf{A}$, $[\mathsf{sk}_{\mathsf{AB},b}]_\mathsf{B}$ denote the Alice and Bob shares of the private key for $\mathsf{pk}_{\mathsf{AB},b}$, and $\mathtt{tx}$ denotes the transaction to be signed.

tion of Bitcoin payment channels in Monero. We instead leverage the refund transactions described in this work.

The lifecycle of a payment channel between Alice and Bob consists of three steps. First, Alice and Bob must *open* a payment channel by including an on-chain transaction that transfers XMR from Alice into a public key $\mathsf{pk}_{\mathsf{AB}}$ whose private key $\mathsf{sk}_{\mathsf{AB}}$ is shared by Alice and Bob, that is, Alice holds $[\mathsf{sk}_{\mathsf{AB}}]_{\mathsf{A}}$ and Bob holds $[\mathsf{sk}_{\mathsf{AB}}]_{\mathsf{B}}$ such that $[\mathsf{sk}_{\mathsf{AB}}]_{\mathsf{A}} + [\mathsf{sk}_{\mathsf{AB}}]_{\mathsf{B}} = \mathsf{sk}_{\mathsf{AB}}$. Second, they perform *off-chain payments* by locally adjusting how many XMR each of them gets from the shared address. Finally, they must *close* the payment channel by submitting a second on-chain transaction that distributes the XMR from the shared address to Alice and Bob as defined by the last balance agreed off-chain. Thus, payment channels require only two on-chain transactions (open and close) but allow for many off-chain payments to take place during its life time. In the following, we show our design of payments channel using the building blocks explained in Section 4.5.1.

Open a payment channel. Assume that Alice holds γ XMR in a dual key $(\mathsf{pk}_{\mathsf{A},0}, \mathsf{pk}_{\mathsf{A},1})$ and she wants to create a payment channel with Bob. First, she transfers γ XMR to a dual key of the form $(\mathsf{pk}_{\mathsf{AB}}, \mathsf{pk}'_{\mathsf{A}})$ and sets the timeout to a desired block height t. This way, if Bob never manages to coordinate with Alice to spend from $\mathsf{pk}_{\mathsf{AB}}$, she will automatically regain control of her funds after that height, eliminating the need for a separate refund transaction. On the other hand, if Bob has received any off-chain transfers from $\mathsf{pk}_{\mathsf{AB}}$, he needs to be sure to put the final balance in a transaction on chain before the block with height t is published.

Off-chain payments. Assume that Alice wants to pay $\gamma' < \gamma$ XMR to Bob using the aforementioned payment channel. For that, Alice transfers γ' XMR from $(\mathsf{pk}_{\mathsf{AB}}, \mathsf{pk}_{\mathsf{A}})$ to a Bob's dual address $(\mathsf{pk}_{\mathsf{B},0}, \mathsf{pk}_{\mathsf{B},1})$ and the change $\gamma - \gamma'$ XMR back to an Alice's dual address $(\mathsf{pk}_{\mathsf{A}.0}, \mathsf{pk}_{\mathsf{A},1})$. As the XMR are being spent from the shared address $\mathsf{pk}_{\mathsf{AB}}$, the transaction must be signed by both users to be valid. The cornerstone of payment channels, however, is that only Alice signs otx and gives her share of the signature $[\sigma]_{\mathsf{A}}$ to Bob, who can in turn verify it. At this point, Bob publish the transaction and get the γ' XMR before the timelock expires. Instead, Bob locally stores otx and the corresponding signature $[\sigma]_{\mathsf{A}}$ until either Bob receives another off-chain payment for a value higher than γ' XMR or the channel is about to expire.

Close channel. The channel between Alice and Bob can be closed for two reasons. First, Bob does not wish to receive more off-chain payments from Alice. Then, assume that Bob got a pair $(\mathtt{tx}, [\sigma]_{\mathsf{A}})$, where $\mathtt{tx}$ is the last agreed balance. He can simply complete σ' with his own share $[\sigma']_{\mathsf{B}}$ and publish the transaction. Second, if the timelock included in the deposit transaction expires, and Alice regains control of the original γ $\mathtt{XMR}$ deposited.

4.5.3 Payment-Channel Network in Monero

Assume that Alice wants to perform an off-chain payment to Dave using a path of opened payment channels of the form Alice, Bob, Carol, Dave. Such a payment is performed in three phases. First, Dave creates a condition $(Y := g^y, Y^* := \mathsf{pk}^{ym}_{\mathsf{CD},1})$ and communicates the conditions (Y, Y^*) to Alice. Second, Alice creates a conditional payment to Bob under condition (Y, Y^*), who in turn creates a conditional payment to Carol under the same condition, and finally Carol creates the last conditional payment to Dave under condition (Y, Y^*). Finally, in the third phase, Dave reveals y to Carol to pull the coins from her, who in turn, reveals y to Bob and finally Bob to Alice.

We have to overcome a subtle but crucial challenge to make such construction fully compatible with Monero. The problem consists on that the same condition (Y, Y^*) cannot be used by every pair of users in the path: While g is the same for every user, each Y^*_i requires the value y (only known by Dave before the payment is settled) and the dual address $(\mathsf{pk}_{P_iP_{i+1}}, \mathsf{pk}_{P_i})$ that defines each of the payment channels (and therefore only known by the two users sharing the channel). To overcome that, we add an extra round of communication where each pair of users forward to the receiver of the payment their shared address' refund address multiplied by their output identifier (i.e., $\mathsf{pk}^{m_{AB}}_{\mathsf{A}}$ where pk_{A} is the refund address of the pair $(\mathsf{pk}_{\mathsf{AB}}, \mathsf{pk}_{\mathsf{A}})$). Upon reception of these values, the receiver computes the pair (Y, Y^*_i) for each user along with a zero-knowledge proof of the fact that both condition values are constructed as expected. Finally, the receiver sends these conditions and proofs back to each user in the payment path from the receiver to the sender.

Now, before setting the conditional payment, each user must validate the zero-knowledge proof produced by the receiver to ensure that the condition for the incoming payment is

built upon the same value y as the condition for the outgoing payment. It is important to note that soundness of the zero-knowledge scheme does not allow Dave to cheat on the proof and still be correctly validated by other users. Otherwise, it could be the situation that an intermediate user loses coins because his outgoing payment goes through but cannot use the same value y for unlocking the incoming payment.

4.6 Concluding remarks and outlook

We present DLSAG, a linkable ring signature scheme that serves as a building block to improve expressiveness, interoperability, and scalability in Monero. We have formally proven that DLSAG provides unforgeability, sender ambiguity, and linkability. We also evaluate the performance of DLSAG showing that DLSAG provides a single bit of communication overhead while slightly reducing the computation overhead when compared to current LSAG. Moreover, we contribute additional cryptographic schemes (e.g., timelock processing) to help to maintain the fungibility of Monero. DLSAG enables payment channels, payment channel networks, and atomic swaps for the first time in Monero. DLSAG is currently under consideration by Monero researchers as an option for adoption and it is also compatible with other CryptoNote-style cryptocurrencies [46].

Outlook. In the future, we identify the following future research directions:

- **Bi-directional payment channels:** In this work, we present a construction for uni-directional payment channels. An extension is thus the design and implementation of bi-directional payment channels. In particular, we find interesting to investigate if techniques in other scalability solutions, such as the Lightning Network, are compatible with our payment channels or what are the challenges otherwise.

- **Further expressiveness:** We envision that expressiveness of DLSAG could be expanded with threshold signatures similar to those of Thring [72] and key aggregation similar to that of [107]. A thorough investigation of these approaches constitutes a venue for future research.

- **Extend security and privacy models:** So far, security and privacy definitions for Monero focus on individual signatures. However, recent studies [119, 91] show that an adversary that considers several transactions (and thus several signatures) at a time, can create profiling information about the users. Thus, new security and privacy models are required to further characterized the security and privacy notions provided by the complete Monero cryptocurrency. Moreover, we plan to study the privacy guarantees provided by suggested extensions such as the timelock processing scheme.

- **Timelock offset analysis and mitigations:** To prove to the network that a certain timelock t has or has not expired, the signer publishes the timelock offset value t, which leaks information about the position of the real timelock t, which in turn leaks information about whether a certain ring is likely to represent the spend of an output that was controlled by two different parties, or just one. Coming up with heuristics to separate those two cases, on one hand; on the other hand, figuring out the correct timelock distributions to draw t from for transactions where it is not meaningfully being used should become interesting areas of research.

- **New privacy implications:** With the use of DLSAG and the new key image mechanism, we introduce a new privacy implication in the Monero blockchain. In particular, given two rings and their corresponding signatures, the sender can determine whether the two truly spent public keys belong to the same user (i.e., the two public keys where derived from the same stealth address with randomness provided by the sender herself). We refer to the full version [118] for the detailed description of the traceability method and practical countermeasures.

Part III

Improving Computation Overhead with Flexible Signature Framework

5. FLEXIBLE SIGNATURES

Traditional cryptographic primitives are not designed for uncertain settings with unpredictable resource constraints. Consider, for example prominent digital signature schemes (such as RSA and ECDSA), that allow a signer who has created a pair of private and public keys to sign messages so that any verifier can later verify the signature with the signer's public key. The verification algorithms of those signature schemes are deterministic and only return a binary answer for the validity of the signature (i.e., 0 or 1).

Such verification mechanisms may be unsatisfactory for an embedded module with unpredictable computing resources or time to perform the verification: if the module can only partially complete the verification process due to resource constraints or some *unplanned* real-time system interrupt, there are no partial validity guarantees available. Thus, the cost of verifying can be relatively expensive, especially for resource-constrained devices in the blockchain network as they need to verify blockchain transactions before relaying it to others.

This calls for a signature scheme that can quantify the validity of the signature based on the number of computations performed before the verification process was stopped. In particular, for a signature scheme instantiation with 128-bit security, we expect the verification process to be flexible enough to offer a validity (or confidence) level in the interval, $[0, 1]$, based on the resources available during the verification process. We observe that none of the previous existing signature schemes offers such a trade-off between the computation time/resource and the security level in a flexible manner.

Contribution. This chapter initiates the study of cryptographic primitives with flexible security guarantees that can be of tremendous interest to real-time systems. In particular, we investigate the notion of a flexible signature scheme that offers partial security for an unpredictably partial verification.

As the first step, based on the standard definition of digital signatures, we propose a new definition of a signature scheme with a flexible verification algorithm. Here, instead of returning a binary answer, the verification algorithm returns a value, $\alpha \in [0, 1] \cup \perp$ that quantifies the validity of the signature based on a number of computations performed.

Next, we provide a provably secure construction of the flexible signature scheme based on the Lamport-Diffie one-time signature construction [93] and the Merkle authentication tree [111]. The security of our signature relies on the difficulty of finding a ℓ-near-collision pair for a collision-resistant hash function. Through our analysis, we demonstrate that our construction still offers a high-security level against adaptive chosen message attacks despite performing fewer computations during verification. For example, a security level of 80 bits requires performing only around $\frac{2}{3}$rd of the total required hash computations for a Merkle tree of height 20.

Finally, we prototype our constructions in a resource-constrained environment by implementing those on a Raspberry Pi. We find that the performance of the proposed constructions is comparable to other prominent signature schemes in terms of running time while offering a flexible trade-off between the security level and the number of computations. Importantly, neither the security level nor the number of computations has to be pre-determined during verification.

Related Work. Fischlin [64] proposed a similar framework for progressively verifiable message authentication codes (MACs). In particular, the author presented two concrete constructions for progressively verifiable MACs that allow the verifier to spot errors or invalid tags after a reasonable number of computations. Also, the paper introduced the concept of detection probability to denote the probability that the verifier detects errors after verifying a certain number of blocks. In this work, we address the open problem of a progressively verifiable digital signature scheme, and we incorporate the detection probability concept into the security analysis of our schemes.

Bellare, Goldreich, and Goldwasser [14] introduced incremental signatures. Here, given a signature on a document, a signer can obtain a (new) signature on a similar document by partially updating the available signature. The incremental signature computation is more efficient than computing a signature from scratch and thus can offer some advantage to a resource-constrained signer. However, it provides no benefit for a resource-constrained verifier; the verifier still needs to perform a complete verification of the signature.

Signature scheme with batch verification [36, 13] is a cryptographic primitive that offers an efficient verifying property. Namely, after receiving multiple signatures from different sources, a verifier can efficiently verify the entire set of signatures at once. Batch verification signature scheme and flexible signature scheme are similar in that they offer an efficient and flexible verification mechanism. However, while the batch verification signature merely seeks to reduce the load on a busy server, the flexible signature focuses on a resource-constrained verifier who can tolerate a partial security guarantee from a signature.

Freitag et. al. [65] proposed the concept of signatures with randomized verification. Here, the verifying algorithm takes as input the public key along with some random coin to determine the validity of the signature. In those schemes, the attacker's advantage of forging a valid message-signature pair, (m^*, σ^*), is determined by the fraction of coins that accept (m^*, σ^*). Freitag et. al. constructed a signature scheme with randomized identity-based encryption (IBE) schemes using Naor's transformation and show that the security level of their signature scheme is fixed to the size of the underlying IBE scheme's identity space. While our work can be formally defined as a signature scheme with randomized verification, our scheme offers a more flexible verification in which the security level of the scheme can be efficiently computed based on the output of the verifying algorithm.

Finally, Fan, Garay, and Mohassel [62] proposed the concept of short and adjustable signatures. They offered three variants, namely setup adjustable, signing adjustable, and verification adjustable signatures offering different trade-offs between the length and the security of the signature. The first two variants allow the signer to adjust the length of the signature, while the last variant allows the verifier to shorten the signature during the verification phase. They presented three constructions for each variant based on indistinguishably obfuscation ($i\mathcal{O}$), and one concrete construction *only* for the setup-adjustable variant based on the BLS Signature Scheme [24]. Unfortunately, none of those constructions is suitable for constructing flexible signatures tolerating unpredictable interrupts.

λ	Security parameter
$[m]$	$\{1, \ldots, m\}$
$m_1 \| \| m_2$	Concatenation of strings m_1 and m_2
$(d_i)_{i \in [m]}$	Concatenation of m elements, $d_1 \| \| d_2 \| \| ... \| \| d_m$
$x \xleftarrow{\$} \mathcal{X}$	x is chosen uniformly at random from some set $\mathcal{X}$
$\Delta(x, y)$	Hamming distance between two binary strings x and y
$[\![r]\!]$	Optional parameter r in an algorithm definition

Figure 5.1. Notation

5.1 Preliminaries

Fig. 5.1 presents prominent notational conventions that we use throughout this work. Our constructions employ the following standard properties of cryptographic hash functions. We use $H : \mathcal{K} \times \mathcal{M} \to \{0,1\}^n$ to denote a family of hash functions that is parameterized by a key $k \in \mathcal{K}$ and message $m \in \mathcal{M}$ and outputs a binary string of length n. For this work, we consider two security properties for hash functions from [137], preimage resistance, collision resistance, and one weaker security notion from [92, 110], ℓ-near collision resistance.

Definition 5.1.1. *(**Preimage Resistance**) We call a family H of hash functions (t_{ow}, ϵ_{ow})-preimage resistant, if for any $\mathcal{A}$ that runs for at most t_{ow}, the adversary's advantage is:*

$$\mathsf{Adv}^{ow}_{H}(\mathcal{A}) = \Pr\left[\begin{matrix} k \xleftarrow{\$} \mathcal{K}, x \xleftarrow{\$} \mathcal{M} \\ y \leftarrow H(k,x), x \leftarrow \mathcal{A}(k,y) \end{matrix} : H(k,x) = y\right] \leq \epsilon_{ow}$$

Definition 5.1.2. *(**Collision Resistance**) We call a family H of hash functions (t_{cr}, ϵ_{cr})-collision resistant, if for any $\mathcal{A}$ that runs for at most t_{cr}, the adversary's advantage is:*

$$\mathsf{Adv}^{cr}_{H}(\mathcal{A}) = \Pr\left[\begin{matrix} k \xleftarrow{\$} \mathcal{K} \\ (x,x) \leftarrow \mathcal{A}(k) \end{matrix} : (x \neq x) \wedge (H(k,x) = H(k,x))\right] \leq \epsilon_{cr}$$

Definition 5.1.3. *(ℓ-**near-collision Resistance**) We call a family H of hash functions $(t_{\ell\text{-}ncr}, \epsilon_{\ell\text{-}ncr})$-$\ell$-near-collision resistant, if for any $\mathcal{A}$ that runs for at most $t_{\ell\text{-}ncr}$ and $0 \leq \ell \leq n$, the adversary's advantage is:*

$$\mathsf{Adv}^{ncr}_{H,\ell}(\mathcal{A}) = \Pr\begin{bmatrix} k \xleftarrow{\$} \mathcal{K}; \\ (x, x) \leftarrow \mathcal{A}(k, \ell) \end{bmatrix} : (x \neq x) \wedge (\Delta(H(k, x), H(k, x)) \leq \ell) \leq \epsilon_{\ell\text{-}ncr}$$

Generic Attacks. To find the preimage $t_{ow} = 2^q$ is required to achieve $\epsilon_{ow} = 1/2^{n-q}$ using exhaustive search. Due to the birthday paradox, however, only $t_{cr} = 2^{n/2}$ is required to find a collision with a success probability of $\epsilon_{cr} \approx 1/2$. Finally, Lamberger et. al. showed in [92] that at least $t_{\ell\text{-}ncr} = 2^{n/2}/\sqrt{\sum_{i=0}^{\ell} \binom{n}{i}}$ is required to find a ℓ-near-collision with a success probability of $\epsilon_{\ell\text{-}ncr} \approx 1/2$.

Unkeyed Hash Functions. In practice, the key for standard hash functions is public; therefore, from this point, we refer to the cryptographic hash function H as a fixed function $H : \mathcal{M} \rightarrow \{0, 1\}^n$.

5.2 Security Definition

In this section, we define our flexible signature scheme. We adopt the standard definition of a signature scheme [86] to the flexible security setting. An instance of an interrupted flexible signature verification is expected to return a validity value, α, in the range $[0, 1]$. To model the notion of runtime interruptions in the signature definition, we introduce the concept of an interruption oracle $\text{iOracle}_{\Sigma}(1^{\lambda})$ for signature scheme Σ and give the verification algorithm access to it. The interruption oracle outputs an interruption position r in the sequence of computation steps involved the verification algorithm. For simplicity, if we denote max to be the maximum number of computations needed (e.g. clock cycles, number of hash computations, or modular exponentiations) for a signature verification, then $\text{iOracle}_{\Sigma}(1^{\lambda})$ outputs a value $r \in \{0, \ldots, \mathsf{max}\}$. The specification of the interruption position varies depending on the choice of the signature scheme; e.g., in this work, we define the interruption position as the number of hash computations performed in the verification algorithm.

Definition 5.2.1. *A flexible signature scheme,* Σ*=(*GEN*,*SIGN*,*VER*), consists of three algorithms:*

- $\textsc{Gen}(1^\lambda)$ *is a probabilistic algorithm that takes a security parameter* 1^λ *as input and outputs a pair* (pk, sk) *of public key and secret key.*
- $\textsc{Sign}(sk, m)$ *is a probabilistic algorithm that takes a private key* sk *and a message* m *from a message space* $\mathcal{M}$ *as inputs and outputs a signature* σ *from signature space* $\mathcal{S}$*.*
- $\textsc{Ver}(pk, m, \sigma, [\![r]\!])$ *is a probabilistic algorithm that takes a public key* pk*, a message* m*, a signature* σ*, an optional interruption position* $r \in \{0, \ldots, \mathsf{max}\}$ *as inputs. If* r *is not provided, then the algorithm will query an interruption oracle,* $\textsc{iOracle}_\Sigma(1^\lambda)$ *to determine* $r \in \{0, \ldots, \mathsf{max}\}$*. The algorithm outputs a real value* $\alpha \in [0, 1] \cup \{\perp\}$ [1]*. The signature is invalid if* $\alpha = \perp$*.*

The following correctness condition must hold: For $\forall (pk, sk) \leftarrow \mathsf{Gen}(1^\lambda), \forall m \in \mathcal{M}, \forall r \in \{0, ..., \mathsf{max}\}$ *:* $\Pr[\textsc{Ver}(pk, m, \textsc{Sign}(sk, m), r) = \perp] = 0$*.*

Remark 1. *The interruption oracle only serves as a virtual party for definitional reasons. In practice, the verification algorithm does not receive the interruption position* r *as an input, and the algorithm continues to perform computations until it receives an interruption. To model runtime interruptions using the interruption oracle* $\textsc{iOracle}_\Sigma(1^\lambda)$*, in this work, we expect the flow of the verification algorithm to* not *be affected/biased by the* r *value offered by* $\textsc{iOracle}_\Sigma(1^\lambda)$ *at the beginning of the verification. Also, we note that depending on signature schemes, there can be more than one way to define the interruption position,* r *(e.g. clock cycles, number of hash computations, or modular exponentiations).*

Extracting Function. We assume that for a flexible signature scheme, there exists an efficient function, $\textsc{iExtract}_\Sigma(\cdot)$, that takes as input the validity of the signature α and outputs the interruption position r. Intuitively, for the case of an unexpected interruption, the verifier need not know when the verification algorithm is interrupted. However, based on the validity output α, the verifier should be able to use $\textsc{iExtract}_\Sigma(\cdot)$ to learn the

[1] $\alpha = 0$ means that no operations are performed in the verification algorithm.

interruption position, r. The definition of extracting function depends on the specification of the interruption position and signature scheme. We will define our $\text{iExtract}_\Sigma(\cdot)$ for each of our proposed constructions in Section 5.3 and Section 5.4.

Security of Flexible Signture Scheme. We present a corresponding definition to the existential unforgeability under adaptive chosen message attack (EUF-CMA) experiment in order to prove the security of our scheme. For a given flexible signature scheme $\Sigma = (\text{Gen}, \text{Sign}, \text{Ver})$ and $\alpha \in [0, 1]$, the attack experiment is defined as follows:

Experiment. $\text{FlexExp}_{\mathcal{A},\Sigma}(1^\lambda, \alpha)$:

1. The challenger $\mathcal{C}$ runs $\text{Gen}(1^\lambda)$ to obtain (pk, sk) and $\text{iExtract}_\Sigma(\alpha)$ to obtain position r. $\mathcal{C}$ sends (pk, r) to $\mathcal{A}$.

2. Attacker $\mathcal{A}$ queries $\mathcal{C}$ for signatures of its adaptively chosen messages. Let $Q_{\mathcal{A}}^{\text{Sign}(sk,\cdot)} = \{m_i\}_{i\in[q]}$ be the set of all messages that $\mathcal{A}$ queries $\mathcal{C}$ where the i^{th} query is a message $m_i \in \mathcal{M}$. After receiving m_i, $\mathcal{C}$ computes $\sigma_i \leftarrow \text{Sign}(sk, m_i)$, and sends σ_i to $\mathcal{A}$.

3. Eventually, $\mathcal{A}$ outputs a pair $(m^*, \sigma^*) \in \mathcal{M} \times \mathcal{S}$ [2], where message $m^* \notin Q_{\mathcal{A}}^{\text{Sign}(sk,\cdot)}$ and sends the pair to $\mathcal{C}$.

4. $\mathcal{C}$ computes $\alpha^* \leftarrow \text{Ver}(pk, m^*, \sigma^*, r)$. If $(\alpha^* \neq \perp)$ and $(\alpha^* \geq \alpha)$, the experiment returns 1; else, it returns 0.

Definition 5.2.2. *For the security parameter λ and $\alpha \in [0, 1]$, a flexible signature scheme Σ is $\left(t, \epsilon, q\right)$ existential unforgeable under adaptive chosen-message attack if for all efficient adversaries $\mathcal{A}$ that run for at most time t and query $\text{Sign}(sk, \cdot)$ at most q times, the success probability is:*

$$\mathsf{Adv}^{\mathsf{flex}}_{\mathcal{A},\Sigma}(n) = \Pr\left[\mathit{FlexExp}_{\mathcal{A},\Sigma}(1^\lambda, \alpha) = 1\right] \leq \epsilon$$

Here, t and ϵ are functions of α and λ, and $q = \mathsf{poly}(\lambda)$.

[2]↑The higher validity implies a higher interruption position. Hence, the best strategy for the adversary is to use the initial position defined by the challenger.

Flexible Lamport-Diffie One-time Signature

Given the security parameter λ, a preimage resistant hash function $F : \{0,1\}^n \to \{0,1\}^n$, a collision resistant hash function $G : \{0,1\}^* \to \{0,1\}^n$, the flexible Lamport-Diffie one-time signature scheme Σ_{fots} works as follows:

$\text{GEN}(1^\lambda)$: *for each* $i \in [n], b \in \{0,1\}$:

choose $sk_i[b] \xleftarrow{\$} \{0,1\}^n$, *set* $pk_j[b] = F(sk_i[b])$

output : $\mathsf{SK} = (sk_i[b])_{i\in[n],b\in\{0,1\}}$, $\mathsf{PK} = (pk_i[b]))_{i\in[n],b\in\{0,1\}}$

$\text{SIGN}(\mathsf{SK}, m)$: *compute* $d = G(m) = (d_i)_{i\in[n]}$, *parse* $\mathsf{SK} = (sk_i[b])_{i\in[n],b\in\{0,1\}}$

output : $\sigma = (sk_i[d_i])_{i\in[n]}$

$\text{VER}(\mathsf{PK}, m, \sigma, [\![r]\!])$: *if* r *is not provided: set* $r \leftarrow \text{IORACLE}(1^\lambda)$,

$k_F = 0$, $N = [n]$

compute $d = G(m) = (d_i)_{i\in[n]}$

write $\mathsf{PK} = (pk_i[b])_{i\in[n],b\in\{0,1\}}$, $\sigma = (\sigma_i)_{i\in[n]}$

while $(r > 0)$ *and* $(N \neq \emptyset)$:

choose $i \xleftarrow{\$} N$

if $F(\sigma_i) \neq pk_i(d_i)$, *return* $\alpha = \perp$

$N = N - \{i\}, k_F = k_F + 1, r = r - 1$

output : $\alpha = k_F/n$

Figure 5.2. Construction of the Flexible Lamport-Diffie One-time Signature

5.3 Flexible Lamport-Diffie One-time Signature

In this section, we present our concrete construction of the flexible one-time signature scheme. This construction is based on the Lamport-Diffie one time signature construction introduced in [93].

5.3.1 Construction

We show the concrete construction of the flexible Lamport-Diffie one-time signature in Fig. 5.2. Here, we use the same key generation and signing algorithms from the Lamport-Diffie signature and modify the verification algorithm.

Key Generation Algorithm. The key generation algorithm takes a parameter 1^λ as input, and generates a private key by choosing $2n$ bit strings each of length n uniformly at random from $\{0,1\}^n$, namely, $\mathsf{SK} = (sk_i[b])_{i\in[n],b\in\{0,1\}} \in \{0,1\}^{2n^2}$. The public key is obtained

by evaluating the preimage-resistant hash function on each of the private key's n bit string, such that $\mathsf{PK} = (pk_i[b])_{i\in[n],b\in\{0,1\}}$ where $pk_i[b] = F(sk_i[b])$ and $F(\cdot)$ is the preimage-resistant hash function.

Signing Algorithm. The signing algorithm takes as input the message m and the private key SK. First, it computes the digest of the message $d = G(m) = (d_i)_{i\in[n]}$ where $d_i \in \{0,1\}$ and $G(\cdot)$ is a collision-resistant hash function that outputs digests of length n. The signature is generated based on the digest d as $\sigma = (sk_i[d_i])_{i\in[n]}$.

Flexible Verification Algorithm. This algorithm takes as input a message m, a public key PK, a signature σ, and an optional interruption position $[\![r]\!]$ and outputs the validity of the signature α. In this construction, we model the interruption condition $r \in \{0, 1, \ldots, n\}$, as the number of hash $F(\cdot)$ computations performed during verification. As mentioned earlier in Section 5.2, to faithfully model the interruption process, the flow of the verification algorithm should not be biased by the r value in any intelligent manner. First, the verification algorithm will query the interruption oracle to determine the interruption position r. The algorithm then computes the digest of the message, $d = G(m) = (d_i)_{i\in[n]}$. Now, instead of sequentially verifying the signature bits like the verification in the standard scheme, the flexible verification algorithm randomly selects a position i of the signature and checks whether $F(\sigma_i[d_i]) = pk_i[d_i]$. If there is one invalid preimage, the verification aborts and returns $\alpha = \perp$. Otherwise, once the interruption condition is met or all positions are verified, the algorithm returns the validity as the fraction of the number of bits that passed the verification check over the length of the signature. In this Lamport-Diffie construction, given the validity α value output by the verification algorithm, the verifier simply computes the interruption position as follows: $\textsc{iExtract}_{\Sigma_{fots}}(\alpha) = \lfloor \alpha \cdot n \rfloor$

5.3.2 Security Analysis

In the flexible Lamport-Diffie one-time signature setting, as the verification algorithm does not perform verification at every position of the signature, the adversary can increase the probability of winning by outputting two messages whose hash digests are close. This is equivalent to finding an ℓ-near-collision pair where ℓ is determined by the adversary.

Theorem 5.3.1 offers the trade-off between computation time and success probability for the adversary.

Theorem 5.3.1. *Let F be (t_{ow}, ϵ_{ow}) preimage-resistant hash function, G be $(t_{\ell\text{-}ncr}, \epsilon_{\ell\text{-}ncr})$ ℓ-near-collision-resistant hash function, k_F, k_G be the number of times $F(\cdot), G(\cdot)$ evaluated in the verification respectively, d be the Hamming distance between two message digests output by $\mathcal{A}$, and $t_{gen}, t_{sign}, t_{ver}$ be the time it takes to generate keys, sign the message, and verify the signature respectively. With $1 \leq k_F \leq n$, $k_G = 1$, the flexible Lamport-Diffie one-time signature Σ_{fots} is $(t_{fots}, \epsilon_{fots}, 1)$ EUF-CMA where:*

$$\alpha = k_F/n$$

$$t_{fots} = \min\{t_{ow}, t_{\ell\text{-}ncr}\} - t_{sign} - t_{ver} - t_{gen} \text{ where } 0 \leq \ell \leq n - k_F$$

$$\epsilon_{fots} \leq \min\left\{1, 2 \cdot \max\left\{\prod_{i=0}^{k_F-1}\left(1 - \frac{d}{n-i}\right), 4n \cdot \epsilon_{ow}\right\}\right\} \text{ where } 0 \leq d \leq \ell$$

Proof. Let m be the message asked by $\mathcal{A}$ during the experiment $\mathsf{FlexExp}_{\Sigma,\mathcal{A}}(1^\lambda, \alpha)$, and (m^*, σ^*) be the forgery pair. We define the distance, $d = \Delta(G(m), G(m^*))$. We notice that for a pair (m, m^*) output by the adversary during the forgery experiment, if $\Delta(G(m), G(m^*)) > n - k_F$, then by pigeonhole principle, at least one of different positions will be checked. Therefore, in order to maximize the success probability, the adversary has to choose ℓ and find a ℓ-near-collision pair where the Hamming distance of $G(m)$ and $G(m^*)$ is less than ℓ where $\ell \leq (n - k_F)$. In order to output such near-collision pair, $\mathcal{A}$ requires at least $t = t_{\ell\text{-}ncr} = 2^{n/2}/\sqrt{\sum_{i=0}^{\ell}\binom{n}{i}}$. Also, on the other hand, $\mathcal{A}$ may win the forgery experiment by spending t_{ow} to break the underlying preimage resistant hash function. Thus, subtracting the running time of generating, signing, and verifying algorithms, we have: $t_{fots} = \min\{t_{ow}, t_{\ell\text{-}ncr}\} - t_{sign} - t_{gen} - t_{ver}$ where $0 \leq \ell \leq n - k_F$. For the success probability, we let Miss be the event that no different bit gets verified. Since d is the Hamming distance between 2 message digests, either none of those different positions were checked, or some of those positions passed the check (i.e. the preimage was found). Thus, we rewrite $\mathcal{A}$'s advantage for the forging experiment as follows: $\Pr[\mathsf{FlexExp}_{\mathcal{A},\Sigma}(1^\lambda, \alpha) = 1] \leq \Pr[\mathsf{Miss}] + \Pr[\mathsf{FlexExp}_{\mathcal{A},\Sigma}(1^\lambda, \alpha) = 1 \wedge \overline{\mathsf{Miss}}]$.

The event $(\mathsf{FlexExp}_{\mathcal{A},\Sigma}(1^\lambda,\alpha)=1 \wedge \overline{\mathsf{Miss}})$ implies that $\mathcal{A}$ wins the forgery experiment by providing a preimage of $F(\cdot)$. Therefore, we can use $\mathcal{A}$ to construct a preimage finder $\mathcal{B}$. The reduction is presented in [33]. One can show:

$$\Pr\left[\mathsf{FlexExp}_{\mathcal{A},\Sigma}(1^\lambda,\alpha)=1 \wedge \overline{\mathsf{Miss}}\right] \leq 4n \cdot \mathsf{Adv}^{\mathsf{pre}}_{\mathcal{B},\mathsf{F}}(n) = 4n \cdot \epsilon_{ow} \tag{5.1}$$

Finally, $\Pr[\mathsf{Miss}]$ implies the adversary can win the forging experiment if the challenger does not perform verification on the different bits. Since d is the number of different bits between two digests, the probability that the challenger does not perform verification on those positions is:

$$\Pr[\mathsf{Miss}] = \prod_{i=0}^{k_F-1} \frac{n-d-i}{n-i} = \prod_{i=0}^{k_F-1} \left(1-\frac{d}{n-i}\right) \tag{5.2}$$

From equations (5.1) and (5.2), we have:

$$\Pr\left[\mathsf{FlexExp}_{\mathcal{A},\Sigma}(1^\lambda,\alpha)=1\right] \leq \min\left\{1, 2\cdot\max\left\{\prod_{i=0}^{k_F-1}\left(1-\frac{d}{n-i}\right), 4n\cdot\epsilon_{ow}\right\}\right\}$$

which completes the proof. □

Security Level. Towards making the security of flexible Lamport-Diffie one-time signatures more comprehensible, we adapt the security level computation from [33]. For any (t,ϵ) signature scheme, we define the security of the scheme to be $\log_2(t/\epsilon)$. As, in the flexible setting, the value of the pair (t,ϵ) may vary as the adversary decides the Hamming distance ℓ, for each value of $k_F \in \{0,\ldots,n\}$, we compute the adversarial advantage for all values $0 \leq \ell \leq n-k_F$ and output the minimum value of $\log_2\left(t_{fots}/\epsilon_{fots}\right)$ as the security level of our scheme. A detailed security level analysis for the Lamport-Diffie one-time signature is available in Section 5.5.1.

5.4 Flexible Merkle Tree Signature

We use the Merkle authentication tree [111] to convert the flexible Lamport-Diffie one-time signature scheme into a flexible many-time signature scheme.

5.4.1 Construction

In the Merkle tree signature scheme, in addition to verifying the validity of the signature, the verifier uses the authentication nodes provided by the signer to check the authenticity of the one-time public key. We are interested in quantifying such values under an interruption. To achieve such a requirement, we require the signer to provide additional nodes in the authentication path.

Key Generation Algorithm. Our key generation remains the same as the one proposed in the original Merkle tree signature scheme [111]. For a tree of height h, the generation algorithm generates 2^h Lamport-Diffie one-time key pairs, $(\mathsf{PK_i}, \mathsf{SK_i})_{i\in[2^h]}$. The leaves of the tree are digests of one-time public keys, $H(\mathsf{PK}_i)$, where $H(\cdot)$ is a collision-resistant hash function. An inner node of the Merkle tree is the hash digest of the concatenation of its left and right children. Finally, the public key of the scheme is the root of the tree, and the secret key is the set of 2^h one-time secret keys.

Modified Signing Algorithm. In the original Merkle signature scheme, a signature consists of four parts: the signature state s, a one-time signature σ_s, a one-time public key PK_s and a set of authentication nodes $\mathsf{Auth_s} = (a_i)_{i\in[h]}$. The verifier can use PK_s to verify the validity of the σ_s and use nodes in $\mathsf{Auth_s}$ and state s to efficiently verify the authenticity of $\mathsf{PK_s}$. For our signing algorithm, along with authentication nodes in the old construction, we require the signer to send the nodes that complete the direct authentication path from the one-time public key to the root. We call this set of nodes complement authentication nodes, $\mathsf{Auth}^c_s = (a_i)_{i\in[h]}$. The reason for including additional authentication nodes is to allow the verifier to randomly verify any level of the tree. Moreover, with additional authentication nodes, verifier can verify different levels of the tree in parallel. Fig. 5.3 describes an example of the new requirement for a tree of height three. The modified signature now consists of five parts: a state s, a Lamport-Diffie one-time signature σ_s, a one-time public key PK_s, a set of authentication nodes Auth_s, and a set of complement authentication nodes Auth^c_s.

Flexible Verification Algorithm. With additional authentication nodes, the verification algorithm can verify the authenticity of the public key at arbitrary levels of the authentication tree as well as use the flexible verification described in Section 5.3 to partially

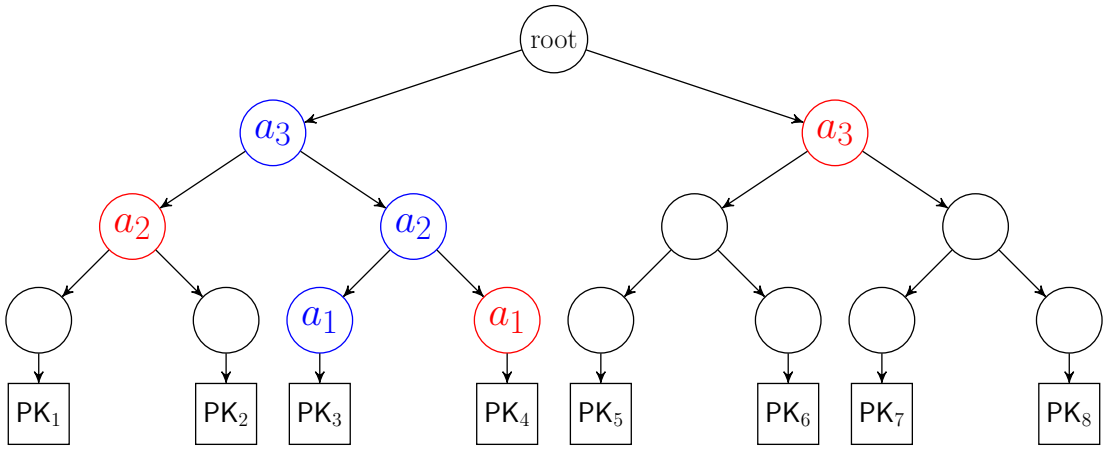


Figure 5.3. An example of new authentication nodes for PK_3 where $\mathsf{Auth}_3 = (a_1, a_2, a_3)$ is the set of authentication nodes in the original scheme and $\mathsf{Auth}_3^c = (a_1, a_2, a_3)$ is the set of additional authentication nodes

verify the validity of the one-time signature. In the end, the verification returns $\alpha = (\alpha_v, \alpha_a)$ that contains both the validity of the signature and the authenticity of the public key. In this construction, we define the interruption $r \in \{0, 1, \ldots, n + h + 1\}$, as the number of computations performed during the verification step.

In contrast to the verification performed in the one-time signature scheme, the security guarantee the verifier gains from the authenticity verification of the one-time public key only increases linearly as the number of computations performed on the authentication path increase: The adversary can always generate a new one-time key pair to sign the message that is not a part of one-time key pairs created by the generation algorithm. In the original Merkle scheme, such a key-pair will fail the authenticity check with overwhelming probability because the verifier can use the authentication nodes to compute and verify the root. However, in the flexible setting, the verifier may not be able to complete the authenticity verification, and there is a non-negligible probability that an invalid one-time public key will be used to verify the validity of the signature. Therefore, the verifier gains an exponential security guarantee about the validity of the one-time signature but only a linear guarantee about the authenticity of the public key as the number of computations increases.

To address this issue, the verification algorithm needs to balance the computations performed on the authentication path and the computations performed on the one-time signa-

Flexible Merkle Tree Signature Scheme

Given the security parameter λ, the tree height h, a preimage resistant hash function $F : \{0,1\}^n \to \{0,1\}^n$, a collision resistant hash function $H : \{0,1\}^* \to \{0,1\}^n$, $G : \{0,1\}^* \to \{0,1\}^n$, and a flexible Lamport-Diffie one-time signature scheme $\Sigma_{fots} = (\textsc{Gen}_{fots}, \textsc{Sign}_{fots}, \textsc{Ver}_{fots})$. The stateful flexible Merkle scheme Σ_{fms} works as follows:

$\textsc{Gen}(1^\lambda)$: *generate* 2^h *ots pairs* $\{(\mathsf{PK_i}, \mathsf{SK_i})\}_{i\in[2^h]}$ *using* $\textsc{Gen}_{fots}(1^\lambda)$

compute the inner nodes of the Merkle tree as follows:

$node_i[j] = H(node_{i-1}[2j-1]||node_{i-1}[2j])$

$2 \le i \le h+1, 1 \le j \le 2^{h+1-i}$

$node_1[i] = H(PK_i), 1 \le i \le 2^h$

output : $\mathsf{SK} = \{\mathsf{SK}_i\}_{i\in[2^h]}, \mathsf{PK} = root$ *(i.e.* $node_{h+1}[1]$*),* $s = 1$

$\textsc{Sign}(\mathsf{SK}, m, s)$: *compute* $\sigma_s = \textsc{Sign}_{fots}(\mathsf{SK}_s, m)$*, compute* $\mathsf{Auth_s} = (a_i)_{i\in[h]}$*, where*

$$a_i = \begin{cases} node_i[\lceil s/2^{i-1}\rceil + 1] & if\ \lceil s/2^{i-1}\rceil \equiv 1\ mod\ 2 \\ node_i[\lceil s/2^{i-1}\rceil - 1] & if\ \lceil s/2^{i-1}\rceil \equiv 0\ mod\ 2 \end{cases}$$

compute $\mathsf{Auth_s^c} = (a_i)_{i\in[h]}$*, where* $a_i = node_i[\lceil s/2^{i-1}\rceil]$

output : $\sigma = (s, \sigma_s, \mathsf{PK_s}, \mathsf{Auth_s}, \mathsf{Auth_s^c}), s = s+1$

$\textsc{Ver}(\mathsf{PK}, m, \sigma, [\![r]\!])$: *if* r *is not provided: set* $r \leftarrow \mathsf{iOracle}(1^\lambda)$,

set $N = [n], T = [h+1], k_F = 0, k_H = 0$

compute $G(m) = d = (d_i)_{i\in[n]}$ *parse* $(s, \sigma_{fots}, \mathsf{PK_{fots}}, \mathsf{Auth}, \mathsf{Auth^c}) \leftarrow \sigma$

write $\sigma_{ots} = (\sigma_i)_{i\in[n]}$*,* $\mathsf{PK_{fots}} = (pk_i[b])_{i\in[n],b\in\{0,1\}}$,

$\mathsf{Auth_s} = (a_i)_{i\in[h]}$*,* $\mathsf{Auth_s^c} = (a_i)_{i\in[h]}$

while $r > 0$ *and* $H \neq \emptyset$ *and* $N \neq \emptyset$ *do :*

if $1 - 1/2^{k_F/2} \le k_H/(h+1)$:

choose $i \xleftarrow{\$} N$*, if* $F(\sigma_i) \neq pk_i(d_i)$*, output :* $\alpha = \perp$

$N = N - \{i\}$*,* $k_F = k_F + 1$

else : *choose* $j \xleftarrow{\$} T$*, set* $a_{h+1} = \mathsf{PK}$

if $j = 1 \wedge a_1 \neq H(\mathsf{PK_s})$: *output :* $\alpha = \perp$

if $j > 1 \wedge a_j$ *is not a parent of* a_{j-1} *and* a_{j-1} : *output* $\alpha = \perp$.

$T = T - \{j\}, k_H = k_H + 1$

$r = r - 1$

output : $\alpha = (k_F/n, k_H/(h+1))$

Figure 5.4. The Flexible Merkle Signature Construction.

ture. We define the confidence for the validity of the one-time signature as $1-1/2^{k_F/2}$ and the confidence for authenticity of the one-time public key as $k_H/(h+1)$, where k_F is the number of computations performed on the one-time signature, k_H is the number of computations

performed on the one-time public key, and h is the height of the Merkle tree. To balance the number of computations, the verifier needs to maintain $1 - 1/2^{k_F/2} \approx k_H/(h+1)$. With the new signing and verifying algorithms described above, we present a detailed construction of the flexible Merkle signature scheme in Fig. 5.4. In this Merkle signature construction, given the validity $\alpha = (\alpha_v, \alpha_a)$ value output by the verification algorithm, the verifier can compute the interruption position as follow: $\text{iExtract}_{\Sigma_{fms}}(\alpha) = \lfloor \alpha_v n \rfloor + \lfloor \alpha_a (h+1) \rfloor$.

5.4.2 Security Analysis

Theorem 5.4.1 presents the trade-off between computation time and success probability for the adversary $\mathcal{A}$.

Theorem 5.4.1. *Let F be (t_{ow}, ϵ_{ow}) preimage-resistant hash function, G be $(t_{\ell\text{-}ncr}, \epsilon_{\ell\text{-}ncr})$ ℓ-near-collision-resistant hash function, H be (t_{cr}, ϵ_{cr}) collision-resistant hash function, k_F, k_G, k_H be the number of times $F(\cdot), G(\cdot), H(\cdot)$ performed respectively, d be the smallest Hamming distance between the forged message digest and other queried message digests, and $t_{gen}, t_{sign}, t_{ver}$ be the time it takes to generate keys, sign the message, and verify the signature respectively. With $1 \le k_F \le n$, $0 \le k_H \le h+1$, and $k_G = 1$, the flexible Merkle signature construction Σ_{fms} from flexible Lamport-Diffie one-time signature scheme is $(t_{fms}, \epsilon_{fms}, 2^h)$ EU-CMA, where*

$$\alpha = (k_F/n, k_H/(h+1))$$

$$t_{fms} = \begin{cases} \mathcal{O}(1) & \text{when } k_H < h+1, \\ \min\left\{t_{ow}, t_{\ell\text{-}ncr}, t_{cr}\right\} - 2^h \cdot t_{sign} - t_{ver} - t_{gen} & \text{where } 0 \le \ell \le n - k_F \end{cases}$$

$$\epsilon_{fms} \le \min\left\{1, 4 \cdot \max\left\{1 - \frac{k_H}{(h+1)}, 2^h \prod_{i=0}^{k_F-1}\left(1 - \frac{d}{n-i}\right), 2^{h+\log_2 4n} \cdot \epsilon_{ow}, \epsilon_{cr}\right\}\right\}$$
$$\text{where } 0 \le d \le \ell$$

Proof. Intuitively, if adversary $\mathcal{A}$ provides an invalid one-time public key, the verification must fail for at least one level of tree. Otherwise, $\mathcal{A}$ successfully finds a collision of H. However, in our scheme, since every level of the tree may not be verified, there is a possibility

that the forged level is not checked. We formalize the intuition as following; we let $\mathsf{InvalidOPK}$ be the event that $\mathcal{A}$ provides an invalid one-time public key. Consider the Merkle tree construction based on the one-time signature construction.

$$\begin{aligned}\Pr\left[\mathsf{FlexExp}_{\mathcal{A},\Sigma}(1^\lambda,\alpha)=1\right] &= \Pr\left[\mathsf{FlexExp}_{\mathcal{A},\Sigma}(1^\lambda,\alpha)=1 \wedge \mathsf{InvalidPK}\right] \\ &\quad + \Pr\left[\mathsf{FlexExp}_{\mathcal{A},\Sigma}(1^\lambda,\alpha)=1 \wedge \overline{\mathsf{InvalidPK}}\right]\end{aligned} \tag{5.3}$$

The $\mathsf{FlexExp}_{\mathcal{A},\Sigma}(1^\lambda,\alpha)=1 \wedge \mathsf{InvalidPK}$ implies that $\mathcal{A}$ provided an invalid one-time public key but won the forgery experiment. Thus, either the verifier failed to check a "bad" level of the tree or $\mathcal{A}$ found a collision of $H(\cdot)$. For a tree of height h, there are $h+1$ levels that one needs to verify for the complete authentication. Since k_H is the number of times $H(\cdot)$ is evaluated, using a union bound, we have:

$$\Pr\left[\mathsf{FlexExp}_{\mathcal{A},\Sigma}(1^\lambda,\alpha)=1 \wedge \mathsf{InvalidPK}\right] \leq 2\cdot\max\left\{1-\frac{k_H}{h+1},\epsilon_{cr}\right\} \tag{5.4}$$

If $\mathcal{A}$ found a collision of $H(\cdot)$, then we can construct a collision finder [33].

The event $\mathsf{FlexExp}_{\mathcal{A},\Sigma}(1^\lambda,\alpha)=1\wedge\overline{\mathsf{InvalidPK}}$ implies that $\mathcal{A}$ won the flexible forgery experiment for one-time signature scheme. Since we defined k_F to be the number of $F(\cdot)$ evaluated, the underlying flexible one-time signature is $(t_{fots},\epsilon_{fots},1)$. Therefore, using Theorem 1, we get:

$$\epsilon_{fots} \leq 2\cdot\max\left\{\prod_{i=0}^{k_F-1}\left(1-\frac{d}{n-i}\right),4n\cdot\epsilon_{ow}\right\} \text{ where } 0\leq d\leq \ell\leq n-k_F$$

Since there are 2^h instances of the flexible Lamport-Diffie one-time signature, it means that for $0\leq d\leq\ell\leq n-k_F$, $\mathcal{A}$ wins the forgery game with probability:

$$\begin{aligned}&\Pr\left[\mathsf{FlexExp}_{\mathcal{A},\Sigma}(1^\lambda,\alpha)=1 \ \wedge\ \overline{\mathsf{InvalidPK}}\right] \\ &\qquad\leq 2\cdot\max\left\{2^h\cdot\prod_{i=0}^{k_F-1}\left(1-\frac{d}{n-i}\right),2^{h+\log_2 4n}\cdot\epsilon_{ow}\right\}\end{aligned} \tag{5.5}$$

From equations (5.3), (5.4) and (5.5), for $0 \leq d \leq \ell \leq n - k_F$, we have:

$$\epsilon_{fms} \leq 4 \cdot \max \left\{ 1 - k_H/(h+1), 2^h \cdot \prod_{i=0}^{k_F-1} \left(1 - \frac{d}{n-i}\right), 2^{h+\log_2 4n} \cdot \epsilon_{ow}, \epsilon_{cr} \right\}$$

When $k_H < h + 1$, we simply let $t_{fms} = \mathcal{O}(1)$ because $\mathcal{A}$ will win the forgery experiment with probability $1 - k_H/(h+1)$. When $k_H = h + 1$, we have:

$$\epsilon_{fms} \leq 4 \cdot \max \left\{ 2^h \cdot \prod_{i=0}^{k_F-1} (1 - \frac{d}{n-i}), 2^{h+\log_2 4n} \cdot \epsilon_{ow}, \epsilon_{cr} \right\} \text{ where } 0 \leq d \leq \ell \leq n - k_F$$

and using [33, Theorem 5], we have $t_{fms} = \min\{t_{cr}, t_{fots}\} - 2^h \cdot t_{sign} - t_{ver} - t_{gen}$. Now, using Theorem 1, we get: $t_{fms} = \min\{t_{ow}, t_{\ell\text{-}ncr}, t_{cr}\} - 2^h \cdot t_{sign} - t_{ver} - t_{gen}$ where $0 \leq \ell \leq n - k$. This completes the proof. □

5.4.3 Other Signature Schemes

Over the last few years, several optimized versions of Merkle tree signature and one-time signature schemes have been proposed. This includes XMSS [32] and SPHINCS [19] for the tree signatures, and HORS [126], BIBA [136], HORST [19] and Winternitz [111] for one-time signatures. While the security analysis for each scheme may vary, we can use the same technique described above to transform those schemes into signature schemes with a flexible verification. In this work, we choose to use Lamport-Diffie One-time signatures in our construction for two reasons. First, the number of hash evaluations in Lamport-Diffie Signature verification is fixed for constant size messages, and this gives better and more precise security proofs. Second, Lamport-Diffie one-time signature has better performance in terms of the running time. Thus, according to our experiment and analysis, the Lamport-Diffie One-time signature scheme combined with Merkle Tree provides a better speed performance and more concrete security proofs.

We also investigate number-theoretic signature schemes and observe that the similar verification technique can be applied to the Fiat-Shamir Signature Scheme [63] as its signature is partitioned into different verifiable sets. However, compared to hash function evaluations, the computation of modular exponentiation is significantly more expensive and thus may

not be suitable for flexible security application environments. On the other hand, lattice-based signature schemes such as GPV signatures [67] can be an interesting candidate for a flexible signature construction. For GPV signatures, a public key is a matrix output by a trapdoor sampling algorithm, and a signature is output by a pre-image sampling algorithm. The signature verification is performed using a matrix and vector multiplication. The same randomized verification technique seems to be applicable here on different rows of the matrix. In the future, we plan to explore a flexible version of GPV signatures.

5.5 Evaluation, Performance Analysis, and Discussion

In this section, we evaluate the performance and the security level of the flexible Lamport-Diffie one-time signature and flexible Merkle signature schemes. For both schemes, the validity value α suggests the number of computations performed (i.e., k_H, k_F) during verification. Based on the value α, the verifier determines the security level achieved by the (interrupted) verification instance.

5.5.1 Security Level of Flexible Lamport-Diffie One-time Signature

The security level of a flexible Lamport-Diffie signature depends on the actual Hamming distance between two message digests output by the adversary and it can increase its advantage by spending more time to find a near-collision pair. However, it is unclear how to precisely measure the exact Hamming distance between those two digests. Therefore, we outline some possible assumptions in order to estimate precisely the value of $\Delta(G(m), G(m^*))$. Using the generic attack on finding near collision pair [92], we can assume that an adversary $\mathcal{A}$ who uses a generic birthday attack can always output a pair (m, m^*) such that $\Delta(G(m), G(m^*)) \leq \ell$ after spending $t_{\ell\text{-}ncr} = 2^{n/2}/\sqrt{\sum_{i=0}^{\ell} \binom{n}{i}}$. Second, for a fixed value ℓ, if the adversary finds a pair (m, m^*) such that $\Delta(G(m), G(m^*)) \leq \ell$, we let $d = \Delta(G(m), G(m^*))$ is equal to the expected value of $\Delta(G(m), G(m^*))$. The intuition behind the second assumption is that as we let the Hamming distance d decrease by 1, the probability that $\Delta(G(m), G(m^*)) = d$ decreases by factor of n; therefore, the actual value of d should be closer to ℓ than to 0.

We define the set $B_\ell(G(m)) = \{x \mid x \in \{0,1\}^n \wedge \Delta(x, G(m)) \leq \ell\}$. If $G(m)$ and $G(m^*)$ is a ℓ-near-collision pair, then $G(m^*) \in B_\ell(G(m))$. If $G(\cdot)$ behaves as an uniformly random function, then given ℓ, the expected value of $\Delta(G(m), G(m^*))$ is:

$$\mathbb{E}(\Delta(G(m), G(m^*))) = \sum_{j=0}^{\ell} j \cdot \frac{\binom{n}{j}}{|B_\ell(G(m))|} = \sum_{j=0}^{\ell} j \cdot \frac{\binom{n}{j}}{\sum_{i=0}^{\ell}\binom{n}{i}} \tag{5.6}$$

For the case of Lamport-Diffie one-time signature, we have $t_{gen} = 2n, t_{sign} = t_{ver} = n$. Combining Theorem 5.3.1 and equation 5.6, we have:

$$t_{fots} = \max\left\{1, \frac{2^{n/2}}{\sqrt{\sum_{i=0}^{\ell}\binom{n}{i}}} - 4 \cdot n\right\} \text{ for } \ell \leq n - k_F$$

$$\epsilon_{fots} \leq \min\left\{1, 2 \cdot \prod_{i=0}^{k_F-1}\left(1 - \frac{d}{n-i}\right)\right\} \text{ where } d = \mathbb{E}(\Delta(G(m), G(m^*)),$$
$$\text{given } \Delta(G(m), G(m^*)) \leq \ell$$

Finally, the adversary's advantage varies depending on the value of ℓ. Therefore, for a fixed value k_F, we compute the adversarial advantage all values $\ell \leq n - k_F$ and output the minimum value of $\log_2\left(t_{fots}/\epsilon_{fots}\right)$ as the security level of the scheme.

Fig. 5.5 gives the trade-off between the number of computations and the security level of the flexible Lamport-Diffie scheme. Compared to the original Lamport-Diffie scheme, our construction offers a reasonable security level despite a smaller number of computations. For example, while a complete verification requires 256 evaluations of $F(\cdot)$ to achieve the 128-bit security level, with only 128 evaluations of $F(\cdot)$, the scheme still offers around the 92-bit security level.

5.5.2 Security Level of Flexible Merkle Tree Signature

For the Merkle tree signature scheme, using the results from [49], [150], we have $t_{gen} = 2^h \cdot 2n + 2^{h+1} - 1, t_{ver} = n + h + 1, t_{sign} = (h+1) \cdot n$. There are two cases for the Merkle tree signature: (1) The authenticity check is complete, $k_H = h + 1$ and (2) The authenticity check is not complete, $k_H < h + 1$.

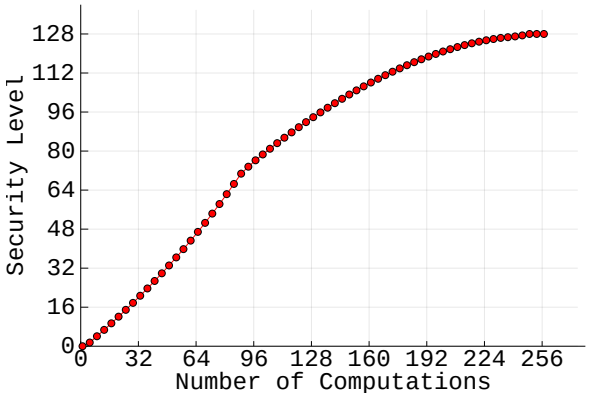


Figure 5.5. Security Level of Flexible Lamport-Diffie One-time Signature

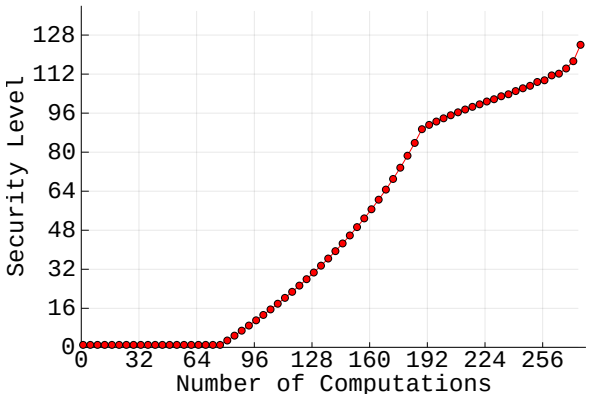


Figure 5.6. Security Level of Flexible Merkle Tree Signature

When $k_H < h+1$, the adversary's probability of winning is non-negligible, and the time it needs to spend on the attack is constant; therefore, when the authenticity check is not complete, we simply let: $t_{fms} = 1$, $\epsilon_{fms} = 1 - k_H/(h+1)$. When the authenticity verification is complete, $k_H = h+1$, using the equation described in Theorem 2, we obtain the following parameters for the flexible Merkle tree scheme:

$$t_{fms} = \max\left\{1, t_{\ell\text{-}ncr} - 2^{h+\log_2(h+1)n} - 2^{h\cdot\log_2 2n} - 2^{\log_2(n-h-1)}\right\} \text{ for } \ell \leq n - k_F$$

$$\epsilon_{fms} \leq \min\left\{1, 2^h \cdot \prod_{i=0}^{k_F-1}\left(1 - \frac{d}{n-i}\right)\right\} \text{ where } d = \mathbb{E}(\Delta(G(m), G(m^*)))$$

Using those formulas, we compute the security level of the flexible Merkle signature as $\log_2(t_{fms}/\epsilon_{fms})$. Fig. 5.6 shows the trade-off between the security level of the scheme and the number of computations of the flexible Merkle tree signature with $h = 20$. Notice that, for small number of computations, the security level of Merkle tree construction does not increase. The reason is that if the authenticity of the public key is not completely checked, the probability that the adversary wins the forgery experiment is always the fraction of the number of computations on the authentication path over the height of the tree, and the forging time remains constant. Moreover, for a tree of height h, there are 2^h instances of flexible Lamport-Diffie one-time signature. Therefore, if $F(\cdot)$ evaluated only for a small number of times, the cost of finding an ℓ-near-collision pair (for $\ell \leq n - k_F$) is cheap.

Table 5.1. Comparing flexible signature schemes performance for different levels of signature verification with other signature schemes.

	Signature Verification; Output Format: (**Timings, Security Level**)				
Percentage of Computations	20%	40%	60%	80%	100%
RSA 3072, $pk = 2^{16} + 1$	-	-	-	-	(1.43ms, 128)
DSA 2048	-	-	-	-	(4.93ms, 87)
EdDSA (Ed25519 curve)	-	-	-	-	(3.21ms, 128)
ECDSA (nistp256 curve)	-	-	-	-	(3.39ms, 128)
Lamport-Diffie OTS verification, $n = 256$	(0.16ms, 35)	(0.31ms, 79)	(0.43ms, 105)	(0.47ms, 121)	(0.54ms, 127)
Merkle signature verification, $n = 256, h = 20$	(0.85ms, 1)	(0.93ms, 19)	(1.00ms, 61)	(1.06ms, 99)	(1.23ms, 127)

The probability that such a pair passes the one-time verification step in one instance of 2^h instances of flexible Lamport-Diffie one-time signature is high. This leads to an undesirable security level during the first few computations.

5.5.3 Implementation and Performance

We have implemented prototypes of our proposed constructions in C, using the SHA-256 implementation of OpenSSL. We evaluated the performance of our proposed constructions on a Raspberry Pi 3, Model B equipped with 1GB RAM.

Table 5.1 gives the performance and security levels of the flexible verification algorithm of both schemes compared to other standard signature schemes (i.e., RSA, DSA, ECDSA, and EdDSA) based on the percentage of computations $p = 20\%, 40\%, 60\%$, 80%, and 100% for messages of size 256[3]. For other signature schemes, we obtain the performance of those schemes using the OpenSSL library. More specifically, for ECDSA, we used two standard curves: Ed25519 and nistp256. For the RSA signature scheme, we used the smallest recommended public key $2^{16} + 1$ for the verification algorithm. For the security levels of other signature schemes, we use the information from [11, 32]. As shown in Table 5.1, the performance of both flexible signature schemes is comparable to other standard schemes in terms of the verification running time. More importantly, both constructions offer an increasing security level at each step of the algorithm while other signature schemes can only provide

[3]↑We focus on the verification algorithm in this work. For the performance of signing, generation algorithms, and the size of the signature we refer readers to [32, 33].

such information at the end of the verification algorithm, and Table 5.1 demonstrates that in the form of (Timings, Security Level) pairs. Also, notice that as the number of verification computations increases, the Lamport-Diffie OTS gives a higher security level than the signing shorter hash digest approach which offers the security level that is equal to half of the length of the hash digest. The main reason is that the verification algorithm verifies the signature at random locations, and while the adversary may learn about the number of computations performed, the adversary does not know which indices of the signature get verified. Thus, the adversary has to decide how close the two digests should be to maximize his adversarial advantage. For the case of Merkle tree signatures, we do not see a huge improvement in the performance of the verification despite a smaller number of computations. This is because the computation of $H(\mathsf{PK}_{\mathsf{fots}})$ and $G(m)$ can be expensive, because of the use the Merkle-Damgård transformation in SHA2 hash family, as those computations requires more calls to the compression function depending on the input size. Nevertheless, for real-time environments, we expect messages to be smaller in size.

5.6 Concluding Remarks

We defined the concept of a signature scheme with a flexible verification algorithm. We presented two concrete constructions based on the Lamport-Diffie one-time signature scheme and the Merkle signature scheme and formally proved their security. We also implemented prototypes of our proposed constructions and showed that the running time performance of our proposed designs is comparable to other signature schemes in a resource-constrained environment. More importantly, compared to standard signature schemes with deterministic verification, our schemes allow the verifier to put different constraints on the verification algorithm in a spontaneous manner and still guarantee a reasonable security level. Our proposed signature scheme is one of the few cryptographic primitives that offers a trade-off between security and resources. Flexible signature can be highly useful for resource-constrained clients when validating and relaying transactions in blockchain networks.

6. SUMMARY

In this dissertation, we proposed several cryptographic constructions for resource-constrained blockchain clients. We envision that these constructions can be useful for resource-constrained devices to minimize the storage, the communication, and the computation overheads when participating in the permissionless blockchain. In particular, we proposed:

Minimizing Storage Overhead with Add-on Privacy Solutions We proposed an oblivious access framework called T^3 in Chapter 2. This framework offers efficient oblivious access to constrained devices when connecting to a potential compromised server with continuously changing database. This framework can be useful when building simplified payment verification solutions for blockchain light clients. In such scenario, the constrained devices that do not have the capability of storing the entire blockchain can outsource the blockchain state storage to other servers that run T^3, and the constrained devices can later query these servers for blockchain data obliviously. In Chapter 3, we presented an autonomous mixer design, that directly allows existing resource-constrained clients to mix their crypto assets without the need of using private blockchains; therefore, AMR helps constrained clients reduce the storage overhead by avoiding storing additional states of private blockchains.

Reducing Communication Overhead with Private Payment Channels In Chapter 4, we proposed a new ring signature primitive, DLSAG, that enables payment channels on the privacy-preserving blockchain, Monero, for the first time. Thus, with payment channels in Monero, resource-constrained clients can reduce the amount of data sent to the blockchain network and preserve the payment data of the channels at the same time.

Reducing Computation Overhead with Flexible Signature Framework Finally, in Chapter 5, we proposed a flexible signature framework. In a flexible signature, the verification algorithm assumes no resource restriction to be known in advance; therefore, the verification algorithm allows an efficient trade-off between the error probability in verification and the computation overhead. Hence, we believe that such a framework can be useful for the blockchain gossip protocol by allowing resource-constrained devices to help verify blockchain transactions with overhead that is consistent with their resource constraints.

6.1 Future Work

Beyond the work presented in this thesis, exploring the study of flexible security in both cryptographic primitives and secure system designs can be an exciting venue. For instance, we often see systems that offer fixed privacy and security guarantees, therefore, require a fixed and expensive amount of computations on users, but we seldomly see systems that flexibly give users a choice between computations and security. Thus, formalizing the trade-off between computations and security is an interesting concept.

Thus, it would be interesting to see how the flexible verification (c.f Chapter 5) can be extended on other cryptographic primitives, such as commitments, zero-knowledge proofs, or different classes of digital signatures (e.g., Lattice-based and code-based signatures). Moreover, reducing the size of the public key and the signature of flexible signature constructions can also be a venue that is worth exploring.

Moreover, T^3 (c.f Chapter 2) is a system that offers resource-constrained blockchain clients efficient access with a robust privacy guarantee. However, T^3 relies on the existence of TEEs. On the other hand, solutions [105, 131] relying on pure cryptographic techniques choose to sacrifice efficiency to offer a strong privacy guarantee. Thus, one question remained: is it possible to design an oblivious database framework that offers a flexible trade-off between efficiency and privacy without relying on the existence of TEEs?

REFERENCES

[1] *Aave: The Money Market Protocol*. https://aave.com/.

[2] *Address Reuse*. https://en.bitcoin.it/wiki/Address_reuse. Accessed in Dec 2019.

[3] Adil Ahmad et al. "OBLIVIATE: A Data Oblivious Filesystem for Intel SGX". In: *NDSS*. 2018.

[4] Martin Albrecht et al. "MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity". In: *Advances in Cryptology – ASIACRYPT 2016*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 191–219. ISBN: 978-3-662-53887-6.

[5] Kurt M. Alonso. *Zero to Monero: First Edition. A technical guide to a private digital currency; for beginners, amateurs, and experts*. https://web.getmonero.org/library/Zero-to-Monero-2-0-0.pdf.

[6] Elli Androulaki et al. "Evaluating user privacy in bitcoin". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2013, pp. 34–51.

[7] Elli Androulaki et al. "Evaluating User Privacy in Bitcoin". en. In: *FC*. 2013, pp. 34–51. ISBN: 978-3-642-39884-1.

[8] Sergei Arnautov et al. "SCONE: Secure Linux Containers with Intel SGX". In: *OSDI*. 2016.

[9] Shehar Bano et al. "SoK: Consensus in the age of blockchains". In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. 2019, pp. 183–198.

[10] Simon Barber et al. "Bitter to Better — How to Make Bitcoin a Better Currency". en. In: *FC*. 2012, pp. 399–414. ISBN: 978-3-642-32946-3.

[11] Elaine Barker. *Recommended for key management-Part 1: General*. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf.

[12] Bellare et al. "The One-More-RSA-Inversion Problems and the Security of Chaum's Blind Signature Scheme". In: *Journal of Cryptology* 16.3 (2003), pp. 185–215. ISSN: 1432-1378.

[13] Mihir Bellare, Juan A. Garay, and Tal Rabin. "Fast batch verification for modular exponentiation and digital signatures". In: *EUROCRYPT 1998*. 1998, pp. 236–250.

[14] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. "Incremental Cryptography: The Case of Hashing and Signing". In: *CRYPTO 1994*. 1994, pp. 216–233.

[15] Mihir Bellare and Gregory Neven. "Multi-signatures in the Plain public-Key Model and a General Forking Lemma". In: *CCS*. Alexandria, Virginia, USA, 2006, pp. 390–399. ISBN: 1-59593-518-5.

[16] E. Ben-Sasson et al. "Secure Sampling of Public Parameters for Succinct Zero Knowledge Proofs". In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 287–304.

[17] Josh Benaloh and Michael de Mare. "One-Way Accumulators: A Decentralized Alternative to Digital Signatures". In: *Advances in Cryptology — EUROCRYPT '93*. Ed. by Tor Helleseth. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 274–285. ISBN: 978-3-540-48285-7.

[18] Adam Bender, Jonathan Katz, and Ruggero Morselli. "Ring signatures: Stronger definitions, and constructions without random oracles". In: *Theory of Cryptography Conference*. Springer. 2006, pp. 60–79.

[19] Daniel J. Bernstein et al. "SPHINCS: Practical Stateless Hash-Based Signatures". In: *EUROCRYPT 2015*. 2015, pp. 368–397.

[20] *Bitcoin Core*. https://bitcoin.org/en/bitcoin-core/. Accessed in Dec 2019.

[21] *Bitcoin Developer Reference*. https://bitcoin.org/en/\developer-reference. Accessed in Dec 2019.

[22] *Bitcoin Difficulty and Network Hash Rate*. https://bitcoinwisdom.com/bitcoin/difficulty. Accessed in Nov 2019.

[23] *BitcoinJ*. https://bitcoinj.github.io/. Accessed in Dec 2019.

[24] Dan Boneh, Ben Lynn, and Hovav Shacham. "Short Signatures from the Weil Pairing". In: *Journal of Cryptology* 17.4 (2004), pp. 297–319.

[25] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography*. 2020.

[26] Joseph Bonneau et al. "Mixcoin: Anonymity for bitcoin with accountable mixes". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2014, pp. 486–504.

[27] Joseph Bonneau et al. "Sok: Research perspectives and challenges for bitcoin and cryptocurrencies". In: *Symposium on Security and Privacy*. IEEE. 2015, pp. 104–121.

[28] Sean Bowe, Ariel Gabizon, and Matthew D. Green. "A Multi-party Protocol for Constructing the Public Parameters of the Pinocchio zk-SNARK". In: *Financial Cryptography and Data Security*. Ed. by Aviv Zohar et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2019, pp. 64–77. ISBN: 978-3-662-58820-8.

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

[30] Sean Bowe and Daira Hopwood. *Hashed Time-Locked Contract transactions*. 2017. URL: https://github.com/bitcoin/bips/blob/master/bip-0199.mediawiki.

[31] Ferdinand Brasser et al. "Software Grand Exposure: SGX Cache Attacks Are Practical". In: *WOOT*. 2017. URL: https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser.

[32] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. "XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions". In: *PQCrypto 2011*. 2011, pp. 117–129.

[33] Johannes Buchmann, Erik Dahmen, and Michael Szydlo. "Hash-based Digital Signature Schemes". In: *PQCrypto 2009*. 2009, pp. 35–93.

[34] Benedikt Bünz et al. "Bulletproofs: Short Proofs for Confidential Transactions and More". In: *S&P*. 2018, pp. 315–334.

[35] Benedikt Bünz et al. "Zether: Towards Privacy in a Smart Contract World". In: *IACR Cryptol. ePrint Arch.* 2019 (2019), p. 191.

[36] Jan Camenisch, Susan Hohenberger, and Michael Østergaard Pedersen. "Batch Verification of Short Signatures". In: *EUROCRYPT 2007*. Ed. by Moni Naor. 2007, pp. 246–263.

[37] Matteo Campanelli, Dario Fiore, and Anaïs Querol. "LegoSNARK: Modular Design and Composition of Succinct Zero-Knowledge Proofs". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, 2075–2092. ISBN: 9781450367479. DOI: 10.1145/3319535.3339820. URL: https://doi.org/10.1145/3319535.3339820.

[38] Anrin Chakraborti and Radu Sion. "ConcurORAM: High-Throughput Stateless Parallel Multi-Client ORAM". In: *NDSS*. 2019. URL: https://www.ndss-symposium.org/ndss-paper/concuroram-high-throughput-stateless-parallel-multi-client-oram/.

[39] Alessandro Chiesa et al. “Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS”. In: *Advances in Cryptology – EUROCRYPT 2020*. Ed. by Anne Canteaut and Yuval Ishai. Cham: Springer International Publishing, 2020, pp. 738–768. ISBN: 978-3-030-45721-1.

[40] CoinMarketCap. *Bitcoin market capitalization*. Available at: https://coinmarketcap.com/currencies/bitcoin/. 2019.

[41] *Compound*. https://compound.finance/.

[42] *Compound*. Available at: https://compound.finance/.

[43] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. Cryptology ePrint Archive, Report 2016/086. https://eprint.iacr.org/2016/086. 2016.

[44] Victor Costan, Ilia Lebedev, and Srinivas Devadas. “Sanctum: Minimal Hardware Extensions for Strong Software Isolation”. In: *25th USENIX Security Symposium*. 2016. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan.

[45] Kyle Croman et al. “On Scaling Decentralized Blockchains”. en. In: *FC*. 2016, pp. 106–125. ISBN: 978-3-662-53357-4.

[46] *CryptoNote Currencies*. https://cryptonote.org/coins.

[47] *Curve DAO*. https://curve.fi/.

[48] Artur Czumaj. *Lecture notes on approximation and randomized algorithms*. http://www.ic.unicamp.br/~celio/peer2peer\/math/czumaj-balls-into-bins.pdf. Accessed in 2019.

[49] Erik Dahmen et al. “Digital Signatures Out of Second-Preimage Resistant Hash Functions”. In: *PQCrypto 2008*. 2008, pp. 109–123.

[50] *Dash*. https://www.dash.org/. Accessed in Dec 2019.

[51] Christian Decker and Roger Wattenhofer. “A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels”. In: *Stabilization, Safety, and Security of Distributed Systems SSS*. 2015, pp. 3–18.

[52] Sergi Delgado-Segura et al. “Analysis of the Bitcoin UTXO Set”. In: *BITCOIN*. 2018. DOI: 10.1007/978-3-662-58820-8_6. URL: https://doi.org/10.1007/978-3-662-58820-8_6.

[53] *Deterministic wallet*. https://en.bitcoin.it/wiki/Deterministic_wallet. Accessed in Dec 2019.

[54] Benjamin E. Diamond. *"Many-out-of-Many" Proofs with Applications to Anonymous Zether*. Cryptology ePrint Archive, Report 2020/293. https://eprint.iacr.org/2020/293. 2020.

[55] W. Diffie and M. Hellman. "New Directions in Cryptography". In: *IEEE Transactions on Information Theory* (1976). ISSN: 0018-9448. DOI: 10.1109/TIT.1976.1055638. URL: http://dx.doi.org/10.1109/TIT.1976.1055638.

[56] W. Diffie and M. Hellman. "New Directions in Cryptography". In: *IEEE Trans. Inf. Theor.* 22.6 (Sept. 2006), pp. 644–654. ISSN: 0018-9448.

[57] *DLSAG prototype numbers*. https://github.com/levduc/DLSAG-prototype-number. 2019.

[58] *Electrum Bitcoin Wallet*. https://electrum.org/. Accessed in Dec 2019.

[59] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. "SoK: Transparent Dishonesty: Front-Running Attacks on Blockchain". In: *Financial Cryptography and Data Security*. Ed. by Andrea Bracciali et al. Cham: Springer International Publishing, 2020, pp. 170–189. ISBN: 978-3-030-43725-1.

[60] Saba Eskandarian and Matei Zaharia. "ObliDB: Oblivious Query Processing for Secure Databases". In: *PVLDB* (2019). DOI: 10.14778/3364324.3364331. URL: http://www.vldb.org/pvldb/vol13/p169-eskandarian.pdf.

[61] Bin Fan et al. "Cuckoo Filter: Practically Better Than Bloom". In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*. CoNEXT '14. 2014. DOI: 10.1145/2674005.2674994. URL: http://doi.acm.org/10.1145/2674005.2674994.

[62] Xiong Fan, Juan Garay, and Payman Mohassel. *Short and Adjustable Signatures*. Cryptology ePrint Archive, Report 2016/549. 2016.

[63] Amos Fiat and Adi Shamir. "How To Prove Yourself: Practical Solutions to Identification and Signature Problems". In: *CRYPTO 1986*. 1987, pp. 186–194.

[64] Marc Fischlin. "Progressive Verification: The Case of Message Authentication". In: *Progress in Cryptology - INDOCRYPT 2003*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 416–429.

[65] Cody Freitag et al. “Signature Schemes with Randomized Verification”. In: *ACNS 2017*. 2017, pp. 373–389.

[66] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. Cryptology ePrint Archive, Report 2019/953. https://eprint.iacr.org/2019/953. 2019.

[67] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. “Trapdoors for Hard Lattices and New Cryptographic Constructions”. In: *STOC 2008*. 2008, pp. 197–206.

[68] Arthur Gervais et al. “On the privacy provisions of bloom filters in lightweight bitcoin clients”. In: *Computer Security Applications Conference*. 2014, pp. 326–335.

[69] Arthur Gervais et al. “On the Privacy Provisions of Bloom Filters in Lightweight Bitcoin Clients”. In: *ACSAC*. 2014. DOI: 10.1145/2664243.2664267. URL: http://doi.acm.org/10.1145/2664243.2664267.

[70] O. Goldreich. “Towards a Theory of Software Protection and Simulation by Oblivious RAMs”. In: *STOC*. 1987. DOI: 10.1145/28395.28416. URL: http://doi.acm.org/10.1145/28395.28416.

[71] Oded Goldreich, Silvio Micali, and Avi Wigderson. “Proofs That Yield Nothing But Their Validity Or All Languages in NP Have Zero-Knowledge Proof Systems”. In: *jacm* 38.3 (1991), pp. 691–729.

[72] Brandon Goodell and Sarang Noether. *Thring Signatures and their Applications to Spender-Ambiguous Digital Currencies*. Cryptology ePrint Archive, Report 2018/774. https://eprint.iacr.org/2018/774. 2018.

[73] Lorenzo Grassi et al. *Poseidon: A New Hash Function for Zero-Knowledge Proof Systems*. Cryptology ePrint Archive, Report 2019/458. https://eprint.iacr.org/2019/458. 2019.

[74] Matthew Green and Ian Miers. “Bolt: Anonymous Payment Channels for Decentralized Currencies”. In: *CCS*. 2017, pp. 473–489.

[75] Jens Groth. “On the Size of Pairing-Based Non-interactive Arguments”. In: *Advances in Cryptology – EUROCRYPT 2016*. Ed. by Marc Fischlin and Jean-Sébastien Coron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 305–326. ISBN: 978-3-662-49896-5.

[76] Danny Harnik et al. “Securing the Storage Data Path with SGX Enclaves”. In: *CoRR* abs/1806.10883 (2018). arXiv: 1806.10883. URL: http://arxiv.org/abs/1806.10883.

[77] Mike Hearn and Matt Corallo. *Connection Bloom filtering*. 2012. URL: https://github.com/\\bitcoin/bips/blob/master/bip-0037.mediawiki.

[78] Ethan Heilman et al. "Tumblebit: An untrusted bitcoin-compatible anonymous payment hub". In: *Network and Distributed System Security Symposium*. 2017.

[79] Ryan Henry, Amir Herzberg, and Aniket Kate. "Blockchain Access Privacy: Challenges and Directions". In: *IEEE Security & Privacy* 16.4 (2018), pp. 38–45.

[80] Thang Hoang et al. "Hardware-Supported ORAM in Effect: Practical Oblivious Search and Update on Very Large Dataset". In: *PoPETs*. 2019.

[81] Iden3. *Circom: Circuit compiler for zkSNARK*. https://github.com/iden3/snarkjs.

[82] Iden3. *Snarkjs: JavaScript and Pure Web Assembly implementation of zkSNARK schemes*. https://github.com/iden3/snarkjs.

[83] Angela Jäschke et al. "Short Paper: Industrial Feasibility of Private Information Retrieval". In: *SECRYPT*. 2017.

[84] *Json-roc-cpp*. https://github.com/cinemast/libjson-rpc-cpp. Accessed in Dec 2019.

[85] *Jubjub*. Available at: https://z.cash/technology/jubjub/.

[86] Jonathan Katz and Yehuda Lindell. "Introduction to Modern Cryptography". In: Chapman and Hall/CRC, 2007. ISBN: 1584885513.

[87] *Key Stone Project*. https://keystone-enclave.org/. Accessed in Dec 2019.

[88] Rami Khalil and Arthur Gervais. "Revive: Rebalancing Off-Blockchain Payment Networks". In: *CCS*. 2017, pp. 439–453.

[89] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *S&P*. 2019.

[90] Philip Koshy, Diana Koshy, and Patrick McDaniel. "An Analysis of Anonymity in Bitcoin Using P2P Network Traffic". en. In: *FC*. 2014, pp. 469–485. ISBN: 978-3-662-45472-5.

[91] Amrit Kumar et al. "A Traceability Analysis of Monero's Blockchain". In: *ESORICS*. 2017, pp. 153–173. ISBN: 978-3-319-66399-9.

[92] Mario Lamberger and Elmar Teufl. "Memoryless Near-Collisions, Revisited". In: *CoRR* (2012).

[93] Leslie Lamport. *Constructing Digital Signatures from a One Way Function*. Tech. rep. CSL-98. This paper was published by IEEE in the Proceedings of HICSS-43 in January, 2010. Microsoft Report, 1979.

[94] Jaehyuk Lee et al. "Hacking in Darkness: Return-oriented Programming against Secure Enclaves". In: *26th USENIX Security Symposium*. 2017.

[95] Sangho Lee et al. "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing". In: *26th USENIX Security Symposium*. 2017.

[96] Jiangtao Li, Ninghui Li, and Rui Xue. "Universal Accumulators with Efficient Nonmembership Proofs". In: *Applied Cryptography and Network Security*. Ed. by Jonathan Katz and Moti Yung. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 253–269. ISBN: 978-3-540-72738-5.

[97] Benoît Libert, Thomas Peters, and Chen Qian. "Logarithmic-Size Ring Signatures with Tight Security from the DDH Assumption". In: *Computer Security*. Ed. by Javier Lopez, Jianying Zhou, and Miguel Soriano. Cham: Springer International Publishing, 2018, pp. 288–308. ISBN: 978-3-319-98989-1.

[98] *Libsodium documentation*. https://libsodium.gitbook.io/doc/.

[99] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium*. 2018.

[100] *Litecoin*. https://litecoin.org/. Accessed in Dec 2019.

[101] Joseph K. Liu, Victor K. Wei, and Duncan S. Wong. "Linkable Spontaneous Anonymous Group Signature for Ad Hoc Groups". In: *Information Security and Privacy*. 2004, pp. 325–335.

[102] Giulio Malavolta et al. "Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability". en-US. In: *NDSS*. Jan. 2019.

[103] Giulio Malavolta et al. "Concurrency and Privacy with Payment-Channel Networks". In: *CCS*. 2017, pp. 455–471.

[104] Mary Maller et al. "Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updatable Structured Reference Strings". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, 2111–2128. ISBN: 9781450367479. DOI: 10.1145/3319535.3339817. URL: https://doi.org/10.1145/3319535.3339817.

[105] Sinisa Matetic et al. “BITE: Bitcoin Lightweight Client Privacy using Trusted Execution”. In: *28th USENIX Security Symposium*. 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/matetic.

[106] Greg Maxwell. “CoinJoin: Bitcoin privacy for the real world”. In: *Post on Bitcoin forum*. 2013.

[107] Gregory Maxwell et al. *Simple Schnorr Multi-Signatures with Applications to Bitcoin*. Cryptology ePrint Archive, Report 2018/068. https://eprint.iacr.org/2018/068. 2018.

[108] Sarah Meiklejohn and Rebekah Mercer. “Möbius: Trustless tumbling for transaction privacy”. In: *Proceedings on Privacy Enhancing Technologies* 2018.2 (2018), pp. 105–121.

[109] Sarah Meiklejohn et al. “A Fistful of Bitcoins: Characterizing Payments Among Men with No Names”. In: *IMC*. IMC ’13. 2013, pp. 127–140. ISBN: 978-1-4503-1953-9.

[110] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. 1st. USA: CRC Press, Inc., 1996. ISBN: 0849385237.

[111] Ralph C. Merkle. “A Certified Digital Signature”. In: *CRYPTO 1989*. 1990.

[112] Ralph C Merkle. “A digital signature based on a conventional encryption function”. In: *Conference on the theory and application of cryptographic techniques*. Springer. 1987, pp. 369–378.

[113] Silvio Micali, Michael Rabin, and Joe Kilian. “Zero-Knowledge Sets”. In: *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*. FOCS ’03. USA: IEEE Computer Society, 2003, p. 80. ISBN: 0769520405.

[114] Ian Miers et al. “Zerocoin: Anonymous distributed e-cash from bitcoin”. In: *Symposium on Security and Privacy*. 2013, pp. 397–411.

[115] Andrew Miller and Sean Bowe. *Zcash MPC Setup*. https://www.zfnd.org/blog/powers-of-tau/.

[116] Mohsen Minaei, Pedro Moreno-Sanchez, and Aniket Kate. *R3C3: Cryptographically secure Censorship Resistant Rendezvous using Cryptocurrencies*. Cryptology ePrint Archive, Report 2018/454. https://eprint.iacr.org/2018/454. 2018.

[117] *Monero monthly blockchain growth*. https://moneroblocks.info/stats/blockchain-growth.

[118] Pedro Moreno-Sanchez et al. *DLSAG: Non-Interactive Refund Transactions For Interoperable Payment Channels in Monero*. Cryptology ePrint Archive, Report 2019/595. https://eprint.iacr.org/2019/595. 2019.

[119] Malte Möser et al. "An Empirical Analysis of Traceability in the Monero Blockchain". In: *PETS* 2018.3 (2018), pp. 143 –163.

[120] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. URL: https://bitcoin.org/bitcoin.pdf.

[121] Sarang Noether and Brandon Goodel. *Dual linkable ring signatures*. https://ww.getmonero.org/resources/research-lab/pubs/MRL-0008.pdf.

[122] Shen Noether and Adam Mackenzie. "Ring Confidential Transactions". en. In: *Ledger* 1.0 (2016), pp. 1–18. ISSN: 2379-5980.

[123] Olga Ohrimenko et al. "Oblivious Multi-Party Machine Learning on Trusted Processors". In: *25th USENIX Security Symposium*. 2016.

[124] *Payment Channels*. URL: https://en.bitcoin.it/wiki/Payment_channels.

[125] Torben P. Pedersen. "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing". In: *CRYPTO*. 1991, pp. 129–140.

[126] Adrian Perrig. "The BiBa One-time Signature and Broadcast Authentication Protocol". In: *CCS 2001*. 2001, pp. 28–37.

[127] Andrew Poelstra. *Lightning in Scriptless Scripts*. 2017. URL: https://lists.launchpad.net/mimblewimble/msg00086.html.

[128] Andrew Poelstra. *Scriptless Scripts*. 2017. URL: https://download.wpsoftware.net/bitcoin/wizardry/mw-slides/2017-03-mit-bitcoin-expo/slides.pdf.

[129] Joseph Poon and Thaddeus Dryja. *The Bitcoin Lightning Network*. Whitepaper. 2016. URL: http://lightning.network/.

[130] *Python-bitcoinlib*. https://github.com/petertodd/python-bitcoinlib. Accessed in Dec 2019.

[131] K. Qin et al. "Applying Private Information Retrieval to Lightweight Bitcoin Clients". In: *2019 Crypto Valley Conference on Blockchain Technology (CVCBT)*. 2019. DOI: 10.1109/CVCBT.2019.00012.

[132] *Raiden Network*. URL: https://raiden.network/.

[133] Ashay Rane, Calvin Lin, and Mohit Tiwari. "Raccoon: Closing Digital Side-Channels through Obfuscated Execution". In: *24th USENIX Security Symposium*. 2015.

[134] Fergal Reid and Martin Harrigan. "An Analysis of Anonymity in the Bitcoin System". en. In: *Security and Privacy in Social Networks*. New York, NY, 2013, pp. 197–223. ISBN: 978-1-4614-4139-7.

[135] *Research meeting: 18 March 2019, 17:00 UTC*. https://github.com/monero-project/meta/issues/319.

[136] Leonid Reyzin and Natan Reyzin. "Better Than BiBa: Short One-Time Signatures with Fast Signing and Verifying". In: *ACISP 2002*. 2002, pp. 144–153.

[137] Phillip Rogaway and Thomas Shrimpton. "Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance". In: *FSE 2004*. 2004, pp. 371–388.

[138] Antoine Rondelet and Michal Zajac. *ZETH: On Integrating Zerocash on Ethereum*. 2019. arXiv: 1904.00905 [cs.CR].

[139] Tim Ruffing and Pedro Moreno-Sanchez. "Valueshuffle: Mixing confidential transactions for comprehensive transaction privacy in bitcoin". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2017, pp. 133–154.

[140] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. "Coinshuffle: Practical decentralized coin mixing for bitcoin". In: *European Symposium on Research in Computer Security*. Springer. 2014, pp. 345–364.

[141] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. "P2P Mixing and Unlinkable Bitcoin Transactions". In: *Network and Distributed System Security Symposium*. 2017.

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

[143] E. B. Sasson et al. "Zerocash: Decentralized Anonymous Payments from Bitcoin". In: *S&P*. 2014. DOI: 10.1109/SP.2014.36.

[144] Eli Ben Sasson et al. "Zerocash: Decentralized anonymous payments from bitcoin". In: *Symposium on Security and Privacy*. IEEE. 2014, pp. 459–474.

[145] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. "ZeroTrace : Oblivious Memory Primitives from Intel SGX". In: *NDSS*. 2018.

[146] C. P. Schnorr. “Efficient signature generation by smart cards”. In: *Journal of Cryptology* 4.3 (1991), pp. 161–174.

[147] Elaine Shi et al. “Oblivious RAM with O((logN)3) Worst-Case Cost”. In: *ASIACRYPT 2011*. 2011.

[148] Michele Spagnuolo, Federico Maggi, and Stefano Zanero. “BitIodine: Extracting Intelligence from the Bitcoin Network”. en. In: *FC*. 2014, pp. 457–468. ISBN: 978-3-662-45472-5.

[149] Emil Stefanov et al. “Path ORAM: An Extremely Simple Oblivious RAM Protocol”. In: *CCS*. 2013. DOI: 10.1145/2508859.2516660. URL: http://doi.acm.org/10.1145/2508859.2516660.

[150] Michael Szydlo. “Merkle Tree Traversal in Log Space and Time”. In: *EUROCRYPT 2004*. 2004, pp. 541–554.

[151] *T3 prototype implementation*. https://github.com/TEE-3/T3. 2019.

[152] Erkan Tairi, Pedro Moreno-Sanchez, and Matteo Maffei. *A2l: Anonymous atomic locks for scalability and interoperability in payment channel hubs*. Tech. rep. Cryptology ePrint Archive, Report 2019/589, 2019.

[153] *Tornado Cash*. Available at: https://tornado.cash/.

[154] Florian Tramer and Dan Boneh. “Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware”. In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=rJVorjCcKQ.

[155] Chia-Che Tsai et al. “Cooperation and Security Isolation of Library OSes for Multi-process Applications”. In: *EuroSys*. 2014. DOI: 10.1145/2592798.2592812. URL: http://doi.acm.org/10.1145/2592798.2592812.

[156] *Understanding the structure of Monero’s LMDB and how explore its contents using mdb_stat*. https://monero.stackexchange.com/questions/10919/understanding-the-structure-of-moneros-lmdb-and-how-explore-its-contents-using.

[157] Luke Valenta and Brendan Rowan. “Blindcoin: Blinded, accountable mixes for bitcoin”. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2015, pp. 112–126.

[158] Xiao Wang, Hubert Chan, and Elaine Shi. “Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound”. In: *CCS*. 2015. DOI: 10.1145/2810103.2813634. URL: http://doi.acm.org/10.1145/2810103.2813634.

[159] *What is Fungibility?* https://www.investopedia.com/terms/f/fungibility.asp.

[160] Gavin Wood. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* 151 (2014), pp. 1–32.

[161] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems". In: *S&P*. 2015.

[162] *Yearn Finance*. https://yearn.finance/.