MŰEGYETEM 1782

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

ZKP-based Audit for Blockchain Systems Managing Central Bank Digital Currency

Scientific Student Competition **Report**

Author:
Bertalan Zoltán Péter

Supervisor:
Dr. Imre Kocsis

2021-10-29

Acknowledgement

The research was supported within the framework of the Cooperation Agreement between the National Bank of Hungary and BME.

Abstract

Implementing a Central Bank Digital Currency (CBDC) has been a subject of research by major central banks around the world for years, and several prototypes have already been created in the process. Should a central bank allow handling of CBDC on a ledger which is not controlled by them, they may wish to specify certain conformance requirements, which are expected to be similar to those that exist for transactions performed in today's digital currencies. A key element of these requirements is to ensure that transactions only occur between parties whose identities have been confirmed.

A possible model to verify conformance in such a system is where the supervisory authority conducts audits on the transactions regularly, or whenever CBDC is removed from or transferred to the external ledger. However, modern distributed ledger systems may include information sensitive to the organizations and persons using the ledger. The central bank does not necessarily wish to access such information themselves, as the subject of the audit is usually the cash flow aspect of monetary traffic (in a nutshell, who gained access to the money in the process).

Zero-Knowledge Proof (ZKP) Systems – which have matured over the last years not only in the theoretical, but also in the technical sense – have the ability to prove the conformance of computations to certain criteria or the knowledge of information adhering to criteria, without revealing any more information in the process other than the computational ability or possession of the knowledge itself. In my work, I define a ZKP proof scheme which allows the operator of a ledger to prove the consistency of the ledger and show that transactions were only conducted between parties authorized by the central bank – all this without revealing details about the transactions (such as the exact participants, the direction of the transfer, or the transferred value). I design a prototype for this scheme in the context of a system where the issuance and management of a CBDC is done on a dedicated ledger and where digital currency can be created on consortial blockchain networks of organizations in exchange for bonding the respective amount of actual CBDC. Arbitrary transactions may be carried out with the digital currency on the external blockchain, and users may convert money covered by their balance back to CBDC on demand.

The defined scheme is not only applicable in the context of CBDC: it has other potential applications in the area of inter-enterprise cooperation.

Kivonat

A világ meghatározó jegybankjai évek óta vizsgálják a digitális jegybankpénz (CBDC) bevezetésének lehetőségeit, és számos prototípust létrehoztak már ennek megvalósítására. Amennyiben a jegybank lehetővé teszi a CBDC egy, nem a jegybank által felügyelt főkönyvön való kezelését, igényt tarthat bizonyos megfelelőségi kényszerek előírására, melyek várhatóan hasonlóak lesznek a digitális pénzzel végzett tranzakciókkal ma is támasztott követelményekhez. Ezek kulcsfontosságú eleme, hogy nagy értékű tranzakciók csak megfelelő identitás-ellenőrzésen átesett felek között történhessenek.

Ha a jegybankpénz kezelését egy, nem jegybank által karbantartott főkönyv végzi, akkor a megfelelőség ellenőrzésének egy lehetséges modellje, ha a főkönyvön végzett tranzakciókat a felügyeleti szerv időről időre – vagy akkor, amikor a főkönyvről digitális jegybankpénz ki- illetve oda átvezetésre kerül – auditálja. A modern elosztott főkönyvi rendszerek azonban olyan információkat is tartalmaznak, melyek a főkönyvet használó szervezetek, illetve személyek számára érzékeny természetűek, és amelyekhez – amennyiben ez lehetséges – a felügyeleti szerv sem kíván hozzáférni. Az audit tárgyát általában a forgalomnak csak a pénzmozgás aspektusa képezi.

A mára már nem csak elméleti, hanem műszaki értelemben is éretté vált tudásmentes bizonyítások (ZKP-k) lehetőséget teremtenek arra, hogy olyan módon bizonyítsuk számítások egy adott szempontrendszer szerinti megfelelőségét, illetve valamilyen követelményeket teljesítő információ ismeretét, hogy mindössze a számítási képességet vagy tudást bizonyítjuk, ennél több információt azonban nem fedünk fel. Dolgozatomban megadok egy olyan ZKP bizonyítási sémát, ami lehetővé teszi azt, hogy egy főkönyv karbantartója úgy igazolja azt, hogy a jegyzett tranzakciókban résztvevő felek a jegybank által jóváhagyottak és hogy a főkönyv maga konzisztens, hogy eközben nem fedi fel az egyes tranzakciók konkrétumait (mint a résztvevő felek, a pénzmozgás iránya, értéke). A sémára olyan rendszer kontextusában adok prototípust, ahol a CBDC kibocsátása és kezelése elsősorban egy dedikált főkönyvön történik, és amelyről jegybankpénz-lekötés ellenében vállalatok konzorciális blokklánc hálózatára lehet a CBDC-nek megfelelő digitális pénzt kibocsátani. A konzorciális hálózaton tetszőleges tranzakciók végezhetők a lekötött pénzzel, majd az egyes felhasználók egyenlegüknek megfelelően kezdeményezhetik a digitális pénz visszavezetését, azaz jegybankpénzzé való visszaváltását.

A megadott séma nem csak CBDC témakörben alkalmazható: egyéb potenciális alkalmazásai is vannak a vállalatközi együttműködés területén.

Contents

List of Figures

List of Tables

1 Introduction

Central Banks around the world are actively researching the possibilities of implementing their own Central Bank Digital Currency, with some institutions already testing prototypes and pilots, and others only in the research phase. Considering the popularity and success of cryptocurrencies and blockchains equipped with various smart contracts, we can safely to assume that once a production-grade CBDC emerges, there is soon going to be a real demand to make it usable on these blockchain networks, as the CBDC system most likely will not allow the installation of arbitrary smart contracts, if it supports any sort of programmability at all. There is already a solution known in the cryptocurrency world for the integration of a platform not capable of smart contract handling and another that is capable, in the form of platform bridging. Bridging could also be implemented to integrate CBDCs with consortial blockchain platforms.

However, such bridged CBDC would still be legal tender, and the CB would require obligatory audits on how the CBDC is used, to ensure, for example, that it does not get into the hands of disallowed clients or that no illegal activities are performed with it, such as participants orchestrating pyramid schemes. These audits may be ad-hoc, periodic, or both, and would most likely involve analyzing the blockchain's ledger in some sense.

Unfortunately, the straightforward arrangement of perusing the ledger contents is not a viable option, since the information in the ledger may be highly confidential, and access to the ledger would leak this information to the bank, who is coincidentally also not interested in these data.

In my work, I establish a simplified blockchain model and present a complete audit model and protocol which harnesses the power of Zero-Knowledge Proofs, a relatively new but already heavily utilized area of cryptography in digital currency ecosystems. While executing the protocol, the consortium proves the conformity of the blockchain state to five elementary criteria but with no need to reveal sensitive information. I have implemented this protocol as a prototype in *Zerojava*, a subset of the *Java* programming language that the *Zilch* ZKP framework can process (after a compilation step). Due to certain complications with the academic-grade framework, I was not able to fully implement a necessary primitive (the KECCAK-256 hash function) in *Zerojava*, which would have significantly increased the complexity of the computations. This component is by no means an indispensable part of the protocol however, since it is already designed for a theoretical, simplified blockchain architecture. I evaluated my implementation with a 'mocked' hash function, to get promising results: linear protocol execution time in the number of blocks (each containing a fixed number of two transactions) on an imaginary blockchain on which the audit was performed.

This paper is organized as follows: Chapter 2 gives a more in-depth explanation and review of CBDC today, with special attention to the bridging and auditability aspects. Then, Chapter 3 introduces a fascinating mathematical tool, the Zero-Knowledge Proof, which powers the heart of my audit protocol design. I go over the several types of proof systems, building up the concepts needed to understand zk-STARKs which *Zilch* internally uses. An example of a concrete, full implementation of a ZKP system, *Zilch* itself, is also overviewed at the end of this chapter.

The second part of the paper is the product of my practical work: in Chapter 4, I present a thorough formalization of my simplified blockchain model and subsequently define the audit model using these formalizations. From the model and the five constraints, I describe a complete audit algorithm and protocol in Chapter 5. Then, in Chapter 6, the prototype *Zilch* implementation of the protocol is examined. Finally, I conclude my results in Chapter 7 and write about the next steps and potential improvements of my solutions.

2 Central Bank Digital Currencies

Central Bank Digital Currency (CBDC) is a topic of avid research in today's financial technology sector. CBDC is a form of digital fiat money issued by Central Banks (CBs), just like physical banknotes that have existed for hundreds of years. Such instruments have been theorized and conceptually developed for decades, but it has only been in the recent years – and probably at least partially motivated by the success of cryptocurrencies – that CBs have begun to treat the introduction of production CBDCs as a mid- or short-term possibility. An ideal scenario retains the favourable properties of cash (such as anonymity and liquidity) while allowing for new potential features made possible by the digitalized nature of the instrument.

While the concept of CBDC is not new, the technology is still very much in a developing stage, and even the most mature systems have not been launched in a production setting. Around the world, most nations are merely doing research; some have come up with proof of concept implementations. Only a few implementations have reached a 'pilot' stage, for example, *Project Jasper* in Canada or *e-CNY* in China (Opare and Kim, 2020; Sun, 2021).

The following sections explore the pros and cons of CBDC, followed by a brief overview of the possible applications and the underlying technology. My work is closely related to the auditability aspect of digital currency and means to be applicable to future CBDC systems.

2.1 Benefits and drawbacks

Benefits One of the most apparent benefits of a CBDC is efficiency, especially in terms of transaction speed. Today's bank transactions are already fast, but a CBDC is capable of instantaneous transfers. Since the need for financial intermediaries is eliminated, transfers can take place directly from peer to peer, in real-time, with no processing or queueing required. The receiving party could immediately confirm payment. This also allows for reduced transaction costs, as payments are simpler and need fewer 'hops'. Additionally, since the processes are automated and fully digitalized, bank holidays no longer affect processing time (Koumbarakis and Dobrauz-Saldapenna, 2019).

Digitalized currency allows for useful new features. For example, so-called 'hash-locked' transactions allow a sending party to (provably) lock funds for a recipient with a secret key that only the sender knows. The receiver can verify the amount of locked funds but may not gain access to them unless they can provide the secret key set by the sender. The lock is a payment promise, which is completed as soon as the receiver performs the service requested by the sender (eg a certain product is shipped). Regular CB currency does not provide a simple, safe and efficient framework for such operations.

Another interesting aspect is the traceability of money by authorities. The digital nature of CBDC allows for sophisticated methods to track money. While traditional paper-based banknotes do have identifiers such as serial numbers, no one can tell where a given banknote is at any given time. One can consider this a great advantage of cash: it offers a high level of privacy regarding what an individual spends their money on. Depending on the design, a CBDC may not be untraceable, which can be seen as a privacy downgrade coming from cash, but also as an upgrade which might allow authorities to fight illegal activities more efficiently.

Risks and drawbacks For the average user, the simplest potential problem may appear to be fact that digital systems rely on electricity and the correct operation of networking services. In the event of a power outage, it is not possible to perform transactions or access one's CBDC balance, unlike traditional cash, which is available 24/7 and does not depend on practically anything (Koumbarakis and Dobrauz-Saldapenna, 2019).

2.2 Possible CBDC models

CBDC is similar to other forms of digital money in the sense that it must maintain a *ledger* of transactions or at least a registry of current account balances. How this ledger is accessed, structured, and maintained is a design choice. Figure 2.1 shows three possible retail CBDC architectures, as seen in a paper by Auer and Böhme (2020). In an *indirect* CBDC architecture, users do not directly claim CBDC through the CB but rather through intermediaries, denoted *CBDC Banks* in the figure. In a *direct* setup, there are no intermediaries involved. Finally, a *hybrid* arrangement is also possible, where intermediaries are involved in the process, but the legal claim of CBDC is still directly on the CB. In this case, intermediaries perform KYC-management and handle retail payments.

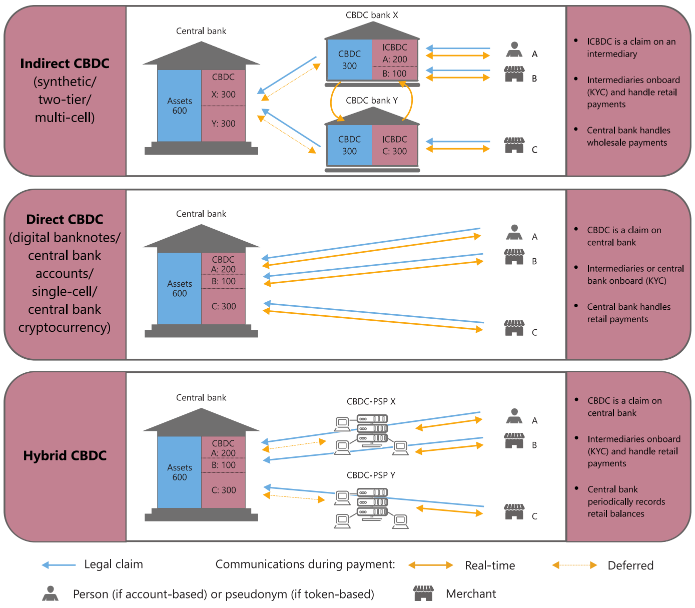

Indirect CBDC
(synthetic/
two-tier/
multi-cell)
Central bank
Assets
600
CBDC
X: 300
Y: 300
CBDC bank X
CBDC
300
ICBDC
A: 200
B: 100
CBDC bank Y
CBDC
300
ICBDC
C: 300
A
B
C
ICBDC is a claim on an intermediary
Intermediaries onboard (KYC) and handle retail payments
Central bank handles wholesale payments
Direct CBDC
(digital banknotes/
central bank
accounts/
single-cell/
central bank
cryptocurrency)
Central bank
Assets
600
CBDC
A: 200
B: 100
C: 300
A
B
C
CBDC is a claim on central bank
Intermediaries or central bank onboard (KYC)
Central bank handles retail payments
Hybrid CBDC
Central bank
Assets
600
CBDC
A: 200
B: 100
C: 300
CBDC-PSP X
CBDC-PSP Y
A
B
C
CBDC is a claim on central bank
Intermediaries onboard (KYC) and handle retail payments
Central bank periodically records retail balances
Legal claim
Communications during payment:
Real-time
Deferred
Person (if account-based) or pseudonym (if token-based)
Merchant

Figure 2.1. Potential CBDC architectures (Auer and Böhme, 2020)

2.2.1 Blockchains in CBDC systems

A blockchain is essentially a data structure consisting of a linked list of *blocks* that contain *records*. Each block (except for the first one, called a *genesis* block) has the cryptographic hash value of the previous block: this mechanism links blocks together. The chain is append-only: blocks are only added and never removed. Usually, the actual records for each block are stored separately in block bodies, and block headers only contain some reference to them (usually a Merkle-tree root).

These simple criteria define a structure suitable for securely storing information, such as financial ledgers. Security in this context refers to the fact that the history of the ledger cannot be changed – should one attempt to modify something in an existing block, the hashes become outdated.

Typically, blockchains are used as distributed databases, where each node on a peer-to-peer network maintains its own chain and the chains are kept in sync using some protocol. The protocol is referred to as a *consensus* protocol because it also defines how the system should handle nodes claiming a different state of the ledger. Basically, the majority of nodes claiming the same ledger state win.

I should mention that blockchains are one realization of a concept known as Distributed Ledger Technology (DLT), which is an umbrella term for systems where a digital ledger (ie a list of transactions) is maintained in a distributed manner. While there are other implementations, most of DLTs are blockchains. All blockchains are considered DLT, but not all DLT implementations rely on blockchains. In common terminology, we use the term DLT to describe non-cryptocurrency related systems, but for the purposes of this document, I will always mean a blockchain by DLT.

An essential property of blockchains is that ledger history can never be altered by design, a property ensured by the consensus protocol. This is especially important in cryptocurrencies, where the nodes are not necessarily trusted by one another, but the properties and protocols of the system make sure that malicious activity (ie an alteration of ledger history) can only be successful if the majority of nodes go rogue.

Cryptocurrencies widely use blockchains, but CBDC implementations will likely call for a centralized ledger solution, not a distributed one. That said, blockchain systems are still relevant in the context of CBDC when we consider the possibility of 'transferring' CBDC to an external blockchain network where it may be used similarly to cryptocurrencies – a concept introduced in the next section.

2.3 Bridged CBDC

By itself, CBDC provides nothing more than a ledger capable of recording transactions, which is basically observed as a payment service by end-users. It does not automatically make the *programmability* of digital money possible, which would mean being able to regulate the terms of CBDC usage or of the various processes which can be performed on CBDC (including payments, deposits, etc) by defining programs usually referred to as *smart contracts*.

Regardless of the chosen architecture, proposed and researched CBDC designs do not allow for smart contracts to be freely installed to the system, contrary to common practice when it comes to cryptocurrencies, where smart contracts are ubiquitous. There is a long-standing and robust demand for smart contracts capable of handling legal tender (ie CBDC). Hence it is safe to assume that after a CBDC is launched, there will soon be a need to make it possible to use the currency on platforms (blockchains) that support smart contracts. This integration will have to go deeper than simply using the CBDC service as a *payment leg*.

There already exist adopted integration schemes for platforms not supporting smart contracts and platforms that do support them, called *bridging*. Bridging two blockchain networks involves providing a mechanism that locks or destroys some units[1] (usually coins or tokens) on one chain and then creates corresponding units of the same value on the other chain. The effect of this procedure is that some funds are made unavailable on the source side, and new funds appear on the destination side. The destruction of units is often called 'burning' and the creation 'minting'. In the simplest of implementations, this is done by *wrapping* coins and a bijective mapping of the unit on the source side to the unit on the destionation side. After bridging to the receiving blockchain (also called a *side chain*), the newly created assets can be freely used. This conversion is bidirectional: when one decides to get the original units belonging to the bridged ones back, the inverse of the bridging process occurs: units are destroyed on the side chain, and the original (*backing*) units are relinquished to the user.

The available bridging technology today is rather diverse: there are centralized solutions such as *Hyperledger Cactus*, which have been designed primarily to support intercompany ledger integration, but there are also decentralized implementations, like *Chainlink*.

Such technology could also facilitate the introduction of liquid CBDC to consortial blockchain networks (which also potentially support smart contracts) through bridging. Figure 2.2 shows a simple architectural overview of a setup where a consortial blockchain is attached as a side chain to the CB through an intermediary service. In practical implementations, the side-chain could be an *Ethereum* network powered by *Hyperledger Fabric* for example.

[1] I refer to the monetary assets used within a blockchain by the term 'unit'. Practically, units may be coins, tokens, or really anything on the blockchain.

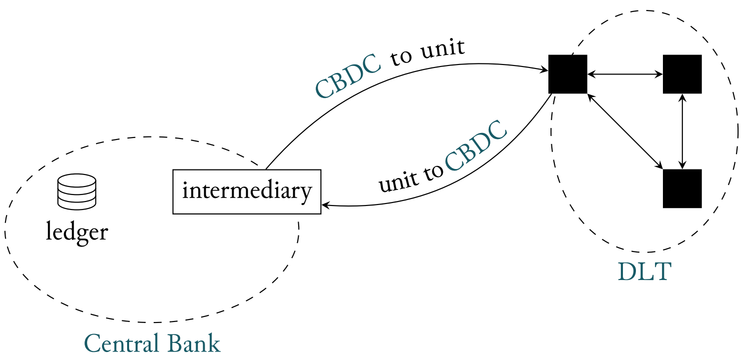


Figure 2.2. CBDC bridging

It is important to note that the bank is not necessarily interested in the internal structure of this network and its participants. Once CBDC 'leaves' the ledger and appears as units on the consortial network, it becomes currency handled however digital currency is handled on the consortial network. In practice, however, the central bank probably does define particular constraints and requirements regarding to the transactions on the consortial blockchain with the bridged CBDC units.

2.3.1 Auditability and accountability of bridged CBDC

In a real-life scenario, the usage of bridged CBDC units will most likely not be unregulated to prevent fraud. The Central Bank could define and enforce rules regarding transactions with bridged CBDC ranging from simple constraints requiring all parties to be whitelisted (or non-blacklisted) to more complex ones, like ones disallowing playing pyramid schemes, for example.

The problem is that transactions on these consortial networks may be confidential. The number and frequency of transactions, the sending and receiving parties and the transferred amount itself can all leak sensitive information about the operation of the organization, which they probably want to keep secret from the bank. On the other hand, the bank is also not interested in these data: all it wants is to ensure that the criteria for transactions with bridged CBDC are met, nothing more.

The question is: can the CB enforce these requirements and conduct periodic or on-demand audits *without* gaining access to sensitive information in the process? In this paper, I focus on an audit model and protocol based on cryptographical primitive called a Zero-Knowledge Proof, introduced in the next chapter. This essentially allows the bank to verify – or, from another perspective, the consortium to *prove* – that the requirements are satisfied without seeing into any of the transactions, or even their number.

3 Zero-Knowledge Proofs (ZKPs)

Zero-Knowledge Proofs (ZKPs) are cryptographical constructions that allow a Prover to prove the truth of a statement to a Verifier in such a way that no information is revealed in the process other than whether the statement is true. Even though the notion of ZKPs has been known since 1989, this area of mathematics and cryptography is still very new, actively researched, and not too well known by most.

3.1 Introduction to ZKPs

The classic textbook example for this concept is Ali Baba's cave. The story takes place in a ring-shaped cave with a single entrance and a magic door at the opposite side that can only be opened by a secret word. Two characters, Peggy and Victor enter the cave. Peggy knows the secret word to open the door and wishes to prove this to Victor. However, she does not want to share the secret with Victor, only to prove to him that she knows it.

There is a way for Peggy to prove knowledge of the secret to Victor in zero-knowledge: ie without revealing the secret itself. The protocol to accomplish this is as follows: Victor stays outside the cave, and Peggy enters. She may choose whether to walk the circular path in a clockwise or counter-clockwise direction, ie starting on the left or the right. A few moments later, Victor also enters the cave. Oblivious to whichever path Peggy may have taken, he calls out a direction from where he wants to see Peggy come back. If Peggy knows the secret, she can always meet Victor coming from the direction he requested: either by turning back or opening the magic door and going the other way around.

One iteration of this protocol allows Peggy to fool Victor with a 50 % chance: if she happens to choose the same way at first as the one Victor decides, she can return to the entrance without actually having to cross the door. Therefore the protocol is repeated several times. With each iteration, the chances of Peggy lying and not knowing the secret are halved. When repeated enough times, Victor can be assured that Peggy indeed knows the secret. After ten executions of this protocol, for example, there is statistically only a minuscule $2^{-10} = 0.0009\,\%$ chance that Peggy is cheating.

This tale is an example of an Interactive Zero-Knowledge Protocol (IZKP). The two parties – **P**eggy being the **P**rover and **V**ictor the **V**erifier – engage in an interactive exchange of messages during which the Verifier can ascertain that the Prover indeed possesses the knowledge they claim.

Several other types of Zero-Knowledge Proof systems exist, which are summarized in the following sections.

3.2 Definitions and basic properties

The concept of Zero-Knowledge Proofs originates from The Knowledge Complexity of Interactive Proof Systems by Goldwasser et al. (1989). In their paper, one of the first steps is the introduction of *interactive Turing Machines*: these are Turing Machines equipped with five tapes:

- a read-only input tape
- a work tape (for processing)
- a random tape (an infinite source of randomness)
- a read-only communication tape (the other TM's write-only communication tape)
- a write-only communication tape (the other TM's read-only communication tape).

Two of these TMs make the participants of an *interactive protocol*, where the two machines engage in a turn-based communication sequence. *Interactive Proof Systems* are defined as interactive protocols with some additional constraints on the probability of the machines halting and accepting.

Then, *zero-knowledge* interactive protocols are defined based on *computational indistinguishability*: in essence, the TM playing the role of the Verifier cannot distinguish the distribution of data on all of its tapes compared to what it could compute solely from its input tape. In other words, the Verifier does not learn new information during the protocol other than the proof itself.

ZKPs have a few basic but essential properties, and it is only pertinent that I introduce these as the definitions of the following few terms. A ZKP system must satisfy all of the following three properties (Mohr, 2007).

Definition 3.1: completeness

For all true statements being proven, there exists a proof that the Verifier will accept.

Definition 3.2: soundness

For all false statements attempted to be proven, there does not exist any proof that the Verifier would accept.

Definition 3.3: zero-knowledge

The defining property of zero-knowledge proof systems: the Verifier does not learn any additional information from the proof or the protocol other than the fact that the statement is indeed true.

Strictly speaking, completeness and soundness are properties of interactive proof systems. The third, zero-knowledge property, is what makes an interactive proof system an interactive ZKP system.

We now define some additional notions related to ZKP systems, which will be referred to later in this chapter (Mouris and Tsoutsos, 2021).

Definition 3.4: universal Turing Machine

A Universal Turing Machine (UTM) is a TM that is able to simulate an arbitrary Turing Machine for computation.

Definition 3.5: transparency

A proof system is *transparent* if it does not require a trusted setup; ie there is no need for a trusted entity to distribute keys in a setup phase.

Definition 3.6: post-quantum resilient ZKP system

A ZKP system where the cryptographic primitives used are not susceptible to attacks by quantum computers.

3.2.1 ZKPs for NP problems

Let us begin with a quick revision of two fundamental complexity classes. The standard definition of the complexity class P is that it is the set of problems that can be solved by a *deterministic* TM in *polynomial* time. The class NP denotes the set of problems that can be solved by a *nondeterministic* TM in *polynomial time*. Intuitively, $\mathsf{P} \subseteq \mathsf{NP}$. Whether $\mathsf{P} \overset{?}{=} \mathsf{NP}$ remains unknown, but most assume $\mathsf{P} \neq \mathsf{NP}$.

Remember, we have defined IZKPs in terms of communicating TMs. Turing Machines in general normally accept or reject *languages*. It turns out that for every language $L \in \mathsf{NP}$, there exists a IZKPs for L.

Theorem 3.1: Every NP language has an ZKP (Goldreich et al., 1991)

$$\forall L \in \mathsf{NP} : \exists \text{ Zero-Knowledge Proof system}$$

The proof of this theorem is out of scope for this paper, but it should be added that what Goldreich et al. show is that using a *bit-commitment scheme*, a ZKP system can be constructed for all languages in NP.

3.3 The many types of ZKPs

Over the the past few decades, there was tremendous progress in ZKP related research, which led to the definitions of newer and newer types of proof and argument systems. This I gives an overview of the most prominent steps building up to the framework that I used to implement a prototype of the audit protocol I designed.

3.3.1 Interactive Zero-Knowledge Protocol systems

This subsection is merely included for completeness, as we have already introduced the definition of IZKPs, and have even seen the textbook example of Ali Baba's magical cave: a simple interactive Zero-Knowledge protocol.

A more concrete and recognized example of an IZKP is the Feige–Fiat–Shamir identification scheme, a relatively simple algorithm where a Prover proves the knowledge of some secret to a Verifier without exposing the secret itself. This particular protocol relies on modular arithmetic. The protocol can be seen on Protocol 3.1.

3.3.2 Non-Interactive Zero-Knowledge Proof systems

So far, we have only discussed ZK protocols where proving happens synchronously, ie the Prover and the Verifier must interact with each other at the same time for the Prover to convince the Verifier. This is clearly somewhat of a limitation: it may not be convenient or even possible for both parties to be present. One might want a self-contained solution, like a little 'box' that the Prover hands over to the Verifier. Everything is contained within the box that is needed for the Verifier to be convinced; the Verifier stays in zero-knowledge, of course. Not only does the Verifier not require the presence of the Prover, they are also free to open the box at any time of their choosing.

Manuel et al. explain this new problem as a story of two mathematicians. One of them travels around the world, and whenever they discover the proof to a new theorem, they would like to share this discovery with their friend *Zero-Knowledge*. The mathematicians cannot communicate synchronously, but the traveller can send postcards to his friend.

Provided the Prover and Verifier agree on a short shared randomness (a string) beforehand, such non-interactive proofs are possible. For some problems (such as `3SAT`), the shared string is not even necessary.

NIZKPs are used by some privacy-focused cryptocurrencies, for example Zerocoin, which will be briefly examined in Chapter 5.

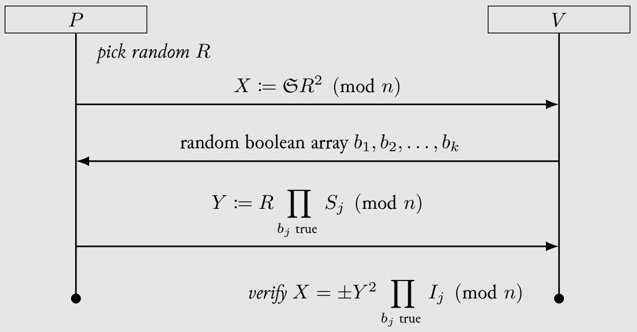
Protocol 3.1: Feige-Fiat-Shamir Identification Scheme (Feige et al., 1988)

P proves to V that they know a secret value S without letting V learn the value.

Setup Let $\mathfrak{S}$ be a function that returns $+1$ or -1, with an even distribution, like a coin flip. Let n be a value known by both parties (distributed by a third party): a *Blum integer*, ie $n = p \times q$ where p and q are both primes of the form $4t + 3$ ($t \in \mathbb{Z}$).

1. P chooses $S_1, S_2, \ldots, S_k \in \mathbb{Z}_n$
2. $\forall i \in k : P$ chooses $I_k := \mathfrak{S} S_k^{-2} \pmod{n}$
3. P publishes $I_1, I_2, \ldots, I_k$

Prover-Verifier protocol



P
V
pick random R
$X := \mathfrak{S}R^2 \pmod{n}$
random boolean array $b_1, b_2, \ldots, b_k$
$Y := R \prod_{b_j \text{ true}} S_j \pmod{n}$
verify $X = \pm Y^2 \prod_{b_j \text{ true}} I_j \pmod{n}$

3.3.3 Proofs and arguments of knowledge

At this point, the terminology starts to become a little confusing and not wholly standardized[1] What we have been discussing are Zero-Knowledge Proofs. There is another concept, called a *zero-knowledge proof of knowledge*, which refers to a protocol not only allowing the Prover to prove that a statement is true but also that they know a *witness* to the truthfulness of the statement.

Finally, there are also zero-knowledge *arguments* of knowledge, which are different from proofs of knowledge, because an argument may not even be true – but the invalid proofs (of these invalid statements) are computationally infeasible to find (Thaler, 2021).

We must introduce these concepts because currently, systems and protocols relying on zero-knowledge constructions do not use (N)IZKPs, but instead build on zk-SNARKs, zk-STARKs, etc – which will soon be introduced –, and these are all *arguments of knowledge* (hence the *-ARK* ending).

3.3.4 Succinct arguments of knowledge

Succinctness refers to the property that communication complexity and verifier time are *polylogarithmic* in computation size. Succinct arguments and proofs are preferred due to their efficiency (Bootle et al., 2020).

At the time of writing this paper, the available, lately published literature always strives for succinctness of proofs or arguments of knowledge.

[1]One can sometimes find *arguments* of knowledge and *proofs* of knowledge used interchangeably in literature.

Zero-Knowledge Succinct Non-interactive ARguments of Knowledge Putting together all of the concepts introduced above, we finally arrive at zk-SNARKs. In recent years, there has been a lot of progress in the research of zk-SNARKs, leading to the birth of the first efficient protocol: *Pinocchio*. A few years later, *Groth16* came along which, generates smaller proofs compared to *Pinocchio* (commonly referred to as [PHGR13]). These are already proving systems used in software (Parno et al., 2013; Groth, 2016).

Zero-Knowledge Succinct Transparent ARguments of Knowledge zk-STARKs are alternatives to zk-SNARKs which, most importantly, eliminate the need for a trusted setup and can be verified exponentially faster than zk-SNARKs (Ben-Sasson et al., 2018).

LegoSNARK *LegoSNARK* is a recent work that provides a framework to integrate the several specialized *proof gadgets* available, realizing that from an application perspective, several types of computations may be required, calling for different proof gadgets. There exist more universal solutions, but they come at the cost of performance. *LegoSNARK* provides a solution to the interoperability problems of these so-called proof gadgets (basically a zk-SNARKs reusable as a building block), as well as a 'lifting tool' that can upgrade or convert existing zk-SNARKs to CP-SNARKs, which are *commit-and-prove* versions that can then be linked using the *LegoSNARK* framework (Campanelli et al., 2019).

This technology can be considered state of the art at the time of writing this article. The offerings of *LegoSNARK* could perhaps be used to create a better, more efficient implementation of the audit protocol in Chapter 5.

3.3.5 Computations on von Neumann architectures as ZKPs

As we will see in Section 3.4, ZKPs usually operate on Arithmetic Circuits, which are rather inconvenient to work with. For implementing the audit algorithm outlined in Chapter 5 directly as a ZK protocol, a universal description language is required. Thankfully, it is possible to generate NIZKPs for arbitrary computations done on a von Neumann architecture, as shown by Ben-Sasson et al., by converting the program to ACs.

Hence, it was sufficient to implement the algorithm in a programming language that has a compiler or a framework available to facilitate the conversion. My choice was *Zilch*, a ZKP framework that we will examine later in this chapter.

3.3.6 ZKPs for set membership

Proving set membership is a crucial component when designing zero-knowledge based bridged CBDC audit protocols. As I have mentioned in Chapter 2, a simple requirement to verify during an audit is that all parties who ever took part in any transactions are allowed (2). This quickly leads to one looking into set membership proofs. To keep the identities of the transacting parties secret, we would also like to make these proofs zero-knowledge

At the early stages of my research, I concentrated on this 'whitelistedness' requirement only and therefore thought that the required protocol would exclusively rely on a simple Zero-Knowledge Proof of set membership. Later I realized that the satisfaction of several other requirements must also be proven, and thus, a specialized ZKP such as one for set membership is not enough, but rather, a generic, universal description language is required. Nonetheless, I include some of my findings regarding ZKPs of set membership, mainly because part of the audit protocol could, in theory, utilize these constructions.

Proof of set membership The formal statement of the problem is proving $x \in S$ (set S may be public) without exposing the value of x. For example, let x be a client of a bank and S be the set of the citizens of the imaginary country of Berryland. Employing ZKP, the bank can to prove $x \in S$ to a regulatory body without exposing the identity of its client.

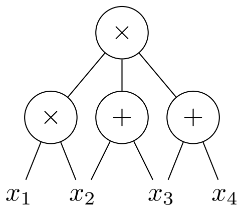


Figure 3.1. A simple example of an Arithmetic Circuit

One way to efficiently prove set membership is by using *accumulators*. We define the system as an $(Acc, Prove, Verify)$ triple, where

- $A \leftarrow Acc(S)$ accumulates the elements of set S into a short value A;
- $\pi \leftarrow Prove(S, x)$ generates a set membership proof π of $x \in S$;
- $Verify(A, x, \pi)$ verifies proof π, in the knowledge of A.

Two concrete realizations of such systems are *Merkle-trees* and RSA-based accumulators. In a binary Merkle-tree, the collision-resistant hash values of set elements form the leaves and each intermediary node is the hash of the concatenation of its children. This way, the tree's root acts as an accumulator: it is a short value that contains information about the entire tree, ie the entire set. To produce proof π, we need to collect all the sibling nodes' values encountered when traversing the tree from the leaf until the root.

Giving a proof that is zero-knowledge is a bit more involved and will not be included in this paper, but we can show that it is certainly possible if we recall Theorem 3.1 and consider that $x \in S$ is clearly an NP statement: given x, we can, of course check, in polynomial time, that x is an element of the otherwise public set S. *Zcash*, a privacy-protecting cryptocurrency built over *Zerocash* (which is also mentioned in Chapter 5) actually employs such set membership proofs (Benarroch et al., 2019).

3.4 Practical Implementations

The computational model of most ZKP systems uses Arithmetic Circuits (ACs) to express instructions (Mouris and Tsoutsos, 2021). Arithmetic circuits are defined as DAGs over a field and a set of variables. Figure 3.1 shows an example of an AC that expresses $(x_1 \times x_2) \times (x_2 + x_3) \times (x_3 + x_4)$.

Another way to understand these expressions is to select the desired result value (say 100 for the last example) and treat the tree as a constraint. Values x_i must have such values that the expression evaluates to 100.

Theoretically, all computations can be expressed by ACs. Given these circuits, the subject of proof of the ZKP in the protocol becomes 'I know the values for all x_i inputs and an output result value r which satisfy this AC.' In our simple example, this means that the Prover knows x_1, x_2, x_3, x_4, and a value r such that $(x_1 \times x_2) \times (x_2 + x_3) \times (x_3 + x_4) = r$ holds. For example, $x_1 = x_4 = 1$, $x_2 = x_3 = 4$, and $r = 100$ satisfies.

The Prover *commits* to a configuration of values (all inputs and the output), which can be thought of as putting the individual values into solid boxes, which the Verifier cannot see through. The commitment function has to be *additively* and *multiplicatively homomorphic*. Formally, if the commitment function is c, the following must hold:

$$x + y = c(x) + c(y)$$
$$x \times y = c(x) \times c(y)$$

This means that showing the equality of the commitment corresponding to the left hand side of the polynomial statement and the commitment of the right-hand side can be considered proof that the one supplying these commitments indeed knows values that satisfy the equation.

What I have described above is not an accurate explanation of ZKPs on ACs; in reality, the process is more complicated and often involves an interactive protocol between the Prover and the Verifier. However, actual protocols do rely on commitments. Complete background on ZKPs of ACs is not in the scope of this paper, but it is important to note that ACs are the low-level constructions that most ZKPs systems use.

As one might imagine, describing complicated programs as ACs is quite inconvenient and error-prone. Thankfully, some ZKP systems allow developers to write higher-level code, which is then compiled to ACs under the hood later. For these systems, the step-by-step process of getting to a proof protocol from lines of code is very similar to Figure 3.3, but does not necessarily involve the intermediate steps of converting to an assembly language or a set of execution traces. Internally, source code is translated into Arithmetic Circuits, which can then be turned into actual proofs.

Many systems, such as *PLONK*, *Hyrax*, or even the better known *Bulletproofs*, do not provide this feature and require users to define ACs manually. Some systems allow for higher-level procedural languages, such as *Pinocchio* or *Buffet*. *ZoKrates* is another better-known framework that allows specifying programs in a declarative way. However, none of these implementations allow for universal programming (Mouris and Tsoutsos, 2021).

For example, *ZoKrates* provides a *Javascript*-like language, with a `for` loop construction, but with *compile-time bounds*. It is impossible to write a loop that is executed n times, where n depends on the input to the program.[2]

For reference, see Table 3.1 for a tabular comparison of some ZKP systems that I have considered. The table has been created by selectively including rows and columns from the comparison table of Mouris and Tsoutsos (2021).

ZKP system	protocol	universal	post-quantum resilient	ease of programming[a]
PLONK	zk-SNARK	●	○	▬□▯
Hyrax	zk-SNARK	●	○	▬□▯
Pinocchio	zk-SNARK	○	○	▬■▯
Buffet	zk-SNARK	○	○	▬■▯
ZoKrates	zk-SNARK	○	○	▬■▯
Bulletproofs	zk-ShNARK[b]	●	○	▬□▯
Zilch	zk-STARK	●	●	▬■▮

[a] ▬□▯ : ACs; ▬■▯ : procedural; ▬■▮ : OO
[b] *Bulletproofs* is not considered a zk-SNARK because it is not *succinct* (it has linear verification time); *Sh* stands for *short* instead of *succinct*.

Table 3.1. Comparison of some ZKP systems (Mouris and Tsoutsos, 2021)

To implement a prototype for the audit model introduced in Chapter 4, I have used a lesser-known ZKP

[2] This is actually because *ZoKrates* `for` loops are actually just syntactic sugar for statements repeated a fixed number of times.

system that is truly universal and which comes with a compiler able to generate assembler-like instructions from a *Java*-like language: *Zilch*.

3.4.1 Zilch

Zilch is a very recent (2021) ZKP framework designed by Mouris and Tsoutsos. In comparison to other ZKP systems, *Zilch* is fully transparent, universal, post-quantum resilient and comes with a compiler for a language that is a subset of *Java* and allows for not only procedural but even object-oriented programming (Mouris and Tsoutsos, 2021).

Since this is the ZKP system that enabled me to prototype the audit protocol outlined in Chapter 5, I elaborate on its design and capabilities as the closure of this chapter.

Zerojava

One of the strongest arguments in favour of *Zilch* is its front-end programming language, named *Zerojava*. As the name suggests, the syntax and abilities of this language can be compared to that of *Java*. For example, the following control-flow keywords are recognized:

- `if`, `else` for conditional branches
- `while` for loops (of dynamic length)

Additionally, *Zerojava* supports OO programming and thus recognizes

- `class Foo { public int foo(int x) { ... } }` as a class with a method `foo` that takes a single integer parameter `x`;
- `Foo F; F = new Foo();` as an instantiation of the class;
- `int x; x = F.foo(42);` as a method call.

Theoretically, even single inheritance is possible, but I have not used this feature in my implementations. *Zerojava* supports the following types:

- `int`: a signed integer (whose size is determined by what parameters *Zilch* was compiled with)
- `int[]`: an array of `int`s
  - `a.length` can be used to get the length of array `a`
  - indexable by `int`s; eg `a[0]]` is the first array element
- `boolean`: a boolean type, like in *Java*

According to the documentation, the operators in Table 3.2 are supported. Finally, there are a few built-in functions that implement Prover functionality, read input and facilitate debugging, which are listed in Table 3.3.

In the Zilch Prototype chapter, we will see some examples of *Zerojava* code. For now, it is enough to note that *Zerojava* supports a variety of operators and keywords known from imperative languages, which makes it very convenient to use.

Under the hood, the *Zerojava* compiler uses the *JavaCC* library to parse the input code, then converts it to *zMIPS* instructions in a number of steps: first, to an intermediate format called *Spiglet*, then another intermediate format called *Kanga*, and from that to the actual *zMIPS* code that can be executed by the *Zilch* engine (Mouris and Tsoutsos, 2021).

operator(s)	meaning
`=`	assignment
`+`, `-`, `*`, `/`	addition, subtraction, multiplication, and division
`+=`, `-=`, `*=`, `/=`	assigning versions of `+`, `-`, `*`, and `/`
`%`, `%=`	modulo operator and its assigning version
`++`, `--`	unary increment and decrement operators
`&`, `|`, `^`	bitwise `AND`, `OR`, and `XOR` operators
`&=`, `|=`, `^=`	assigning versions of `&`, `|`, and `^`
`<<`, `>>`	binary shift left and right
`=<<`, `=>>`	assigning binary shift left and right
`!`, `&&`, `||`	logical negation (unary), `AND`, and `OR` operators
`==`, `!=`, `<`, `<=`, `>`, `>=`	common relational operators

Table 3.2. Operators available in *Zerojava* (Mouris and Tsoutsos, 2021)

function signature(s)	description
`Prover.answer(int)`	reply with an answer as a Prover
`int PublicTape.read()`, `int PrivateTape.read()`	read a value from the public and private input tapes
`int PublicTape.seek(int n)`, `int PrivateTape.seek(int n)`	read the `n`-th value from the public and private input tapes

Table 3.3. Built-in functions available in *Zerojava* (Mouris and Tsoutsos, 2021)

Design

In the computational model of *Zilch*, the Prover is a TM equipped with a private and a public read-only input tape. This is where input data can be supplied, and the `PrivateTape.*` and `PublicTape.*` *Zerojava* functions can be used to read from the tapes sequentially. During execution, the tapes' contents are read from simple text files, one value per line.

Instead of using ACs directly, *Zilch* executes instructions by an abstract MIPS-architecture processor. It has its own instruction set called *zMIPS*, which is similar to the original MIPS instruction set, with the addition of some special instructions such as `PUBREAD` or `SECREAD`, which read from the public and private input tapes, respectively, or `ANSWER`, which answers with a value in a register as a Prover and halts the machine.

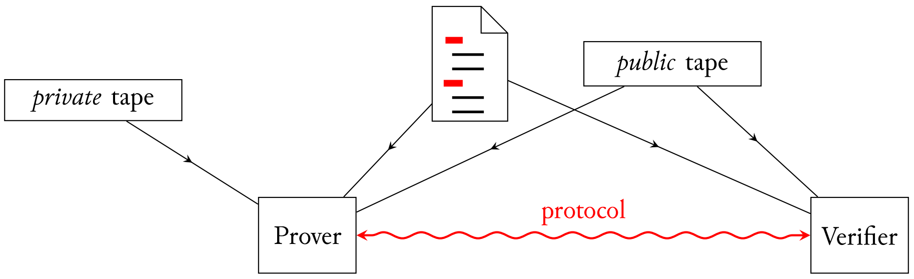


Figure 3.2. Overview of the interaction of a *Zilch* Prover and Verifier (Mouris and Tsoutsos, 2021)

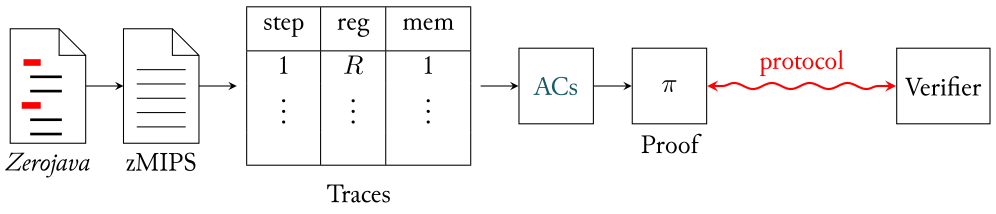


Figure 3.3. The compilation process of *Zilch* (Mouris and Tsoutsos, 2021)

Given *zMIPS* instructions and a (potentially empty) private and public input tape, *Zilch* runs both a Prover and a Verifier entity to execute an IZKP, during which the satisfaction constraints defined by the *zMIPS* code is proven (or disproven) (Mouris and Tsoutsos, 2021).

Implementation

Zilch consists of a front-end and a back-end component. The front-end is basically the *Zerojava* language and the provided compiler, which can produce *zMIPS* code from *Zerojava* source. The back-end is what is able to execute the ZKP protocols.

Given the *zMIPS* instructions of a computation, *Zilch* creates a *transcript*, which represents an execution trace (in other words, a sequence of states) of the program. The transcript is encoded in a table, where each row stands for the execution of an instruction, and the columns are for the values of the registers (of the abstract *zMIPS* processor). In addition to these execution traces, a set of polynomials are also generated, which encode what (memory) constraints must be satisfied at each execution step. Having the transcript and the constraints, *Zilch* employs the zk-STARK protocol, during which the Prover convinces the Verifier that the transcript is indeed the result of an execution of the computation (the *zMIPS* code, which is known by both parties) and of the constraints' integrity. The above description is simplified since this paper does not deal with the low-level details of how ZKPs work. I feel that it is pertinent to point out that the *Zilch* system relies on zk-STARKs, using an external library (Mouris and Tsoutsos, 2021).

Performance

The framework is implemented in native C++ code. According to the measurements of the authors (which were performed in a much more powerful environment then the measurements in this work), the prover execution time of *Zilch* is higher than in other ZKP frameworks, due to the fact that *Zilch* is not programmed by ACs. In general, Prover and Verifier execution time is $O(T \text{polylog} T)$ where T is the number of instructions. Communication between the two parties causes some additional overhead (Mouris and Tsoutsos, 2021).

In my experience during prototyping, in a suboptimal environment (with one processor core and $\sim$5 GB memory), a more involved *Zilch* program with several loops and function calls can take long minutes to execute.

4 A Blockchain Audit Model for Bridged CBDC

I define the audit model in two steps. First, I introduce the model of a simplified blockchain with mathematical formalizations. Based on this model, I then define the audit requirements formally.

4.1 Blockchain Model

To rigorously define an audit model, we first need to establish what we mean by the blockchain to which CBDC is bridged. Real-life implementations can be complicated: block headers can contain a lot of metadata, such as nonces and timestamps, and transactions themselves may include additional information such as gas-related values or any arbitrary data embedded in the transaction. Things get even harder to handle when we consider smart contracts.

I simplify this model and define a minimal blockchain and transaction model, with only the most essential and easy to understand elements of actual implementations. I then formalize this construction and express what we mathematically must prove during an audit.

4.1.1 Notation and terminology

In the following mathematical expressions, I will rely on the following operators, functions, and notation:

- $x := y$ means 'let x by equal to y' (an assignment)
- Hash is an arbitrary collision-resistant hash function
- $+\!\!+$ is the concatenation operator (meaning: concatenate the numerical (binary) representations of the values on either side)
- $|S|$ denotes the number of elements in the set S
- The creation of the blockchain is referred to as *genesis*, which involves choosing some parameters explained in Figure 4.1.2

4.1.2 Blockchain

The blockchain is an infinitely growing sequence of blocks, where each block consists of a block header (denoted by B) and a block body (denoted by T). The header contains data that makes this construction a blockchain, and the body is simply a sequence of transactions organized as a *Merkle-tree* which belong to the block. Figure 4.1 is a visual representation of a segment of the block sequence. Let us allow indexing of block headers and bodies in the following way:

- $B_i :=$ the header of the i-th block on the chain (ie the block at height i)
- $T_i :=$ the transactions of the i-th block on the chain (ie the body of B_i)

Thus, the entire blockchain $\mathfrak{B}$ at a given height (ie block count) h can be expressed as a sequence of ordered pairs of B_i and T_i:

$$\mathfrak{B} = ((B_0, T_0), \ldots, (B_h, T_h))$$

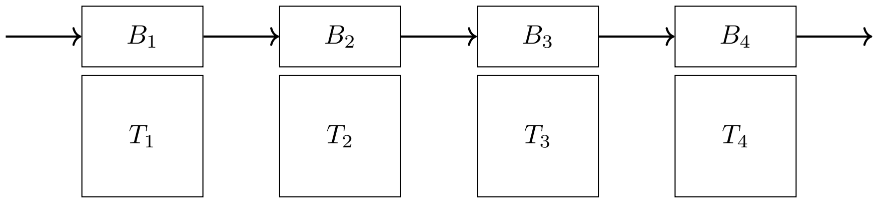
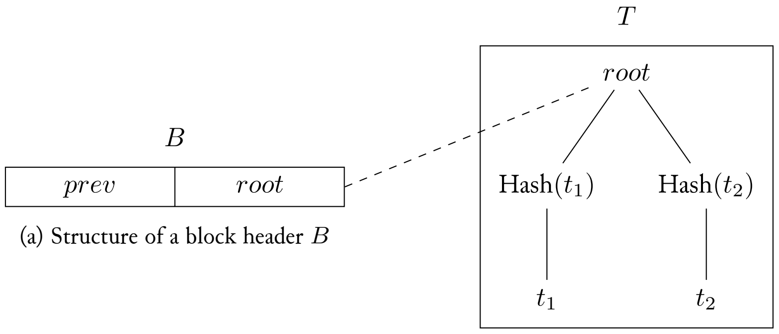


Figure 4.1. The block sequence of the blockchain model

Blocks

Each block consists of a header and a body. The header is an ordered pair of the *hash* of the header of the previous block ($prev$) and the root of the Merkle-tree that contains all the transactions in the block ($root$). The body is basically the sequence of transactions stored in a *Merkle-tree*, whose root is $root$. Figure 4.2 illustrates the individual block headers and bodies. There is one special block, called the *genesis* block, which is the first block in the blockchain. Since no blocks precede this block, its $prev$ value is nil. Below I formally define exactly what all of these variables mean:

- $B := (prev, root)$ (a block header)
- $\text{Hash}(B = (prev, root)) := \text{Hash}(prev) \mathbin{+\!\!+} \text{Hash}(root)$
- $prev :=$ the hash of the previous block's header, ie $prev_i := \text{Hash}(B_{i-1})$
- $root$: the root of the Merkle-tree containing the elements in the respective block body T



(a) Structure of a block header B

(b) Structure of a block body T

Figure 4.2. Visualization of block headers and bodies in the blockchain model

I denote the set of all blocks in the blockchain $\mathfrak{B}$ by $\mathbb{B}_{\mathfrak{B}}$.

Transactions

Transactions (denoted by t) are ordered triples formed by their *source* s, their *receiver* r and the *amount* a of funds transferred.

- $t := (s, r, a)$
- $s :=$ transaction source (an arbitrary identifier)
- $r :=$ transaction destination/receiver (an arbitrary identifier)
- $a :=$ amount transferred from s to d, $a \in \mathbb{N}$ (a natural number)

To further simplify the model, let us assume that each block contains exactly two transactions:

$$\forall (B, T) \in \mathfrak{B} : T = (t_1, t_2)$$
$$\forall (B = (root, prev), T = (t_1, t_2)) \in \mathfrak{B} : root = \text{Hash}\left(\text{Hash}(t_1) +\!\!+ \text{Hash}(t_2)\right)$$

The total set of transactions in a blockchain $\mathfrak{B}$ – denoted by $\mathbb{T}_{\mathfrak{B}}$ – are understood as the sequence of all transactions in all blocks starting from the genesis block:

$$\mathbb{T}_{\mathfrak{B}} := \{t \in T : (B, T) \in \mathfrak{B}\}$$

I also define $\mathcal{S}(t)$, $\mathcal{R}(t)$, and $\mathcal{A}(t)$ to denote the sender, receiver, and amount of a transaction t respectively.

$$\begin{aligned}
\mathcal{S} : \mathbb{T} \to \mathbb{A} \text{ with } \mathcal{S}(t) &= \text{ the sender } s \text{ of transaction } t = (s, r, a) \\
\mathcal{R} : \mathbb{T} \to \mathbb{A} \text{ with } \mathcal{R}(t) &= \text{ the receiver } r \text{ of transaction } t = (s, r, a) \\
\mathcal{A} : \mathbb{T} \to \mathbb{A} \text{ with } \mathcal{A}(t) &= \text{ the amount } a \text{ involved in transaction } t = (s, r, a)
\end{aligned}$$

Accounts

Accounts are abstract objects with only one property: their identifier (usually an integer). They represent the sources and targets of transactions. I assume that the number of accounts in the system is *predetermined* and *fixed*. There is always at least one account, called the *genesis account*, with identifier 0. In a sensible system, there must be at least one additional account; otherwise, the only transactions allowed are the genesis account looping funds back to itself.

Every account has its own *balance*, which is a non-negative integer. When the blockchain is created, the genesis account's balance becomes the *total balance* in the blockchain. In other words, there is a fixed amount of units ('money') in the system at all times.

Strictly speaking, the balance is not a property of an account or even the blockchain since it can be derived for any account from the *genesis* account's initial balance and the transactions that took place since genesis.

I denote the set of all accounts in a blockchain $\mathfrak{B}$ by $\mathbb{A}_{\mathfrak{B}}$.

Genesis

Genesis refers to the creation of a blockchain. The following parameters are involved when creating a blockchain $\mathfrak{B}$:

- a total balance $\$_{\mathfrak{B}}$
- the final set of accounts $\mathbb{A}_{\mathfrak{B}}$

4.2 Requirement formalization

In this section, we define the requirements blockchain state is expected to conform to, first in plain English, then using the formalizations introduced in the previous section. This will make it possible to describe the audit protocol algorithmically in Chapter 5.

4.2.1 Plain text requirements

Given my simple model, I have collected five simple requirements which must be verifiable in an audit. We want to ensure that for all transactions in any given block...

(1) the sender has sufficient balance to spend;

(2) the receiver is allowed to receive funds (ie is whitelisted);

(3) the balance of the receiver after the transaction equals their balance before the transaction *plus* the transferred funds;

(4) the balance of the sender after the transaction equals their balance before the transaction *minus* the transferred funds;

(5) the hash of the block header (found in the next block's header) is indeed the hash of the block's header concatenated with the root of the Merkle-tree that contains the transactions in the block.

All of these requirements (with perhaps the exception of (2)), are very elementary and should always be satisfied by a blockchain's state. The addition of (2) aims to represent a constraint beyond basic blockchain consistency that a CB may wish to enforce in a bridged CBDC scenario.

4.2.2 Formalized requirements

Let t^i_j denote the jth transaction in the ith block B_i (in our simplified model, $j \in \{1, 2\}$). For better understanding, here are the first few blocks of the blockchain, formalized.

$$\begin{aligned}
&B_0 = (nil, r_0)\\
&\quad : r_0 = \text{Hash}\left(\text{Hash}(j^0_1) \mathbin{+\!\!+} \text{Hash}(j^0_2)\right)\\
&B_1 = (\text{Hash}(B_0), r_1)\\
&\quad : \text{Hash}(B_0) = \text{Hash}\left(nil \mathbin{+\!\!+} \text{Hash}(r_0)\right), r_1 = \text{Hash}\left(\text{Hash}(j^1_1) \mathbin{+\!\!+} \text{Hash}(j^1_2)\right)\\
&B_2 = (\text{Hash}(B_1), r_2)\\
&\quad : \text{Hash}(B_1) = \text{Hash}\left(\text{Hash}(B_0) \mathbin{+\!\!+} \text{Hash}(r_1)\right), r_2 = \text{Hash}\left(\text{Hash}(j^2_1) \mathbin{+\!\!+} \text{Hash}(j^2_2)\right)\\
&\quad \vdots\\
&B_i = (\text{Hash}(B_{i-1}), r_i) \qquad i > 0\\
&\quad : \text{Hash}(B_{i-1}) = \text{Hash}\left(\text{Hash}(B_{i-2}) \mathbin{+\!\!+} \text{Hash}(r_{i-1})\right), r_i = \text{Hash}\left(\text{Hash}(j^i_1) \mathbin{+\!\!+} \text{Hash}(j^i_2)\right)
\end{aligned}$$

Now I also define a function $\mathcal{B}$ which we can later use to formalize requirements involving account balances.

$$\mathcal{B} : \mathbb{A} \times \mathbb{T} \to \mathbb{N} \text{ with } \mathcal{B}(u, t^j_i) = \text{ the balance of account } u \textit{ after} \text{ transaction } t^j_i$$

The balance can be expressed formally:

$$\mathcal{B}(u, t_j^i) = \mathcal{B}\left(u, \mathcal{P}(t_j^i)\right) + \begin{cases} +\mathcal{A}(t) & : \mathcal{R}(t) = u \\ -\mathcal{A}(t) & : \mathcal{S}(t) = u \\ 0 & : \text{ otherwise} \end{cases}$$

$\mathcal{P}(t)$ is the transaction immediately before t:

$$\mathcal{P}(t_j^i) = \begin{cases} t_{j-1}^i & : j > 1 \\ t_N^{i-1} & : \text{ otherwise} \end{cases}$$

where N is the (constant) number of transactions in each block (defined to be constantly 2 for simplicty).

In this recursive definition, u's balance after t_j^i equals their balance *before* t_j^i with the checked transaction's amount added or subtracted (or ignored), depending on the role of u in the transaction. I should also mention that $\forall u \in \mathbb{A} \setminus \{0\} : \mathcal{B}(u, nil) = 0$.

Let us eliminate the recursion for better usability in the implementation phase:

$$\mathcal{B}(u, t_j^i) = \sum_{n=0}^{i-1} \left(\sum_{t \in T_n : \mathcal{R}(t) = u} \mathcal{A}(t) - \sum_{t \in T_n : \mathcal{S}(t) = u} \mathcal{A}(t) \right) + \sum_{n=1}^{j} [\mathcal{R}(t_n^i) = u]\, \mathcal{A}(t_n^i) - \sum_{n=1}^{j} [\mathcal{S}(t_n^i) = u]\, \mathcal{A}(t_n^i)$$

where $[P]$ is an *Iverson-bracket* expression, ie $[P] = \begin{cases} 1 & : P \text{ is true} \\ 0 & : \text{ otherwise} \end{cases}$.

Now we can finally formalize the requirements in Table 4.1.

requirement	formalization
(1)	$\forall t_i^j \in \mathbb{T} : \mathcal{B}\left(s, \mathcal{P}(t_j^i)\right) \geq a$
(2)	$\forall t_i^j \in \mathbb{T} : \mathcal{S}(t_i^j) \in WhiteList \quad WhiteList \subseteq \mathbb{A}$
(3)	$\forall t_i^j \in \mathbb{T} : \mathcal{B}(r, t_j^i) = \mathcal{B}\left(s, \mathcal{P}(t_j^i)\right) + a$
(4)	$\forall t_i^j \in \mathbb{T} : \mathcal{B}(s, t_j^i) = \mathcal{B}\left(s, \mathcal{P}(t_j^i)\right) - a$
(5)	$\forall B_i = (prev_i, root_i), B_{i-1} = (prev_{i-1}, root_{i-1}), i > 0 \in \mathbb{B}$
	$: prev_i = \text{Hash}(B_{i-1}) = \text{Hash}(prev_{i-1} + root_{i-1})$

Table 4.1. Formalized requirements

4.3 Audit Model

The CB may order an audit at regular intervals, triggered by a CBDC bridging event in either direction, or at any other time. Audits can be performed with no need for human interaction, fully automatically.

When requested to perform an audit, one of the nodes on the consortial networks marks the current blockchain state. The subject of the audit is going to be the marked state.

From this point, there are two possibilities, depending on whether audits are designed to be interactive. In the case of an interactive protocol, the node engages in a synchronous exchange of messages with the CB, during which the satisfaction of the requirements by the marked blockchain state is proven. For a non-interactive protocol, the only difference is that there is no message exchange: the node is expected to respond to the CB with some audit data within a time frame, and these data are sufficient for the CB to be convinced of the satisfaction of the requirements. For example, disregarding our aims for the protocol to be zero-knowledge for a moment, the node could simply send the serialized blockchain data at the time of marking to the CB, which can then simply verify it itself. What we really seek though is some way for the blockchain state to be kept secret by the consortium, but retaining the ability to prove the requirements' satisfaction. The audit flow is visualized in Figure 4.3.

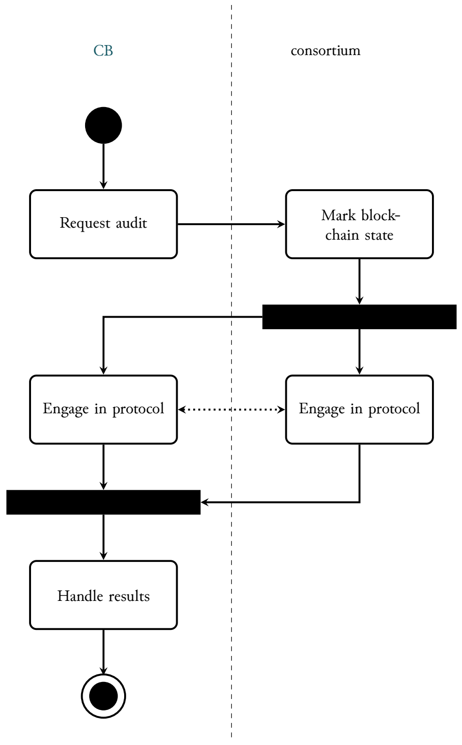


Figure 4.3. Informal activity diagram of an interactive audit

4.3.1 Commitment to the blockchain

One problem left to solve is how the state which is the subject of the audit is committed to the real blockchain state. Otherwise, the node on the consortial network could prove the correctness of an arbitrary state that has nothing to do with what has actually happened on the blockchain.

CB-controlled node

One simple solution to the commitment problem is for the CB to require their own blockchain node on the consortial network. This of course partially defeats the need for the kind of audit outlined in this paper. The CB's node could execute whatever audit logic the CB wants, then report back to the bank on a secure channel.

The problem is that this method, again, provides no privacy. The CB sees everything that happens on the blockchain of the consortium.

There is one way to improve on this solution: the CB could maintain a so-called *lightweight* or *header* node, which does not have access to block bodies (ie transactions), only headers. In our model described in Section 4.1, the header only contains the cryptographic hash value of the previous block's header and the root of the Merkle-tree which stores the transactions in the block, which is also a hash value. Thus, no information is leaked by the block headers themselves, except for their number, but this shortcoming can be mitigated by allowing for variable-length blocks.

In such a setup, an export of the marked state would be committed to the blockchain state in the sense that it contains the hashes of all blocks (except perhaps the last one), which are also recomputable for each block from the transactions.

Figure 4.4 shows a schematic diagram for this version.

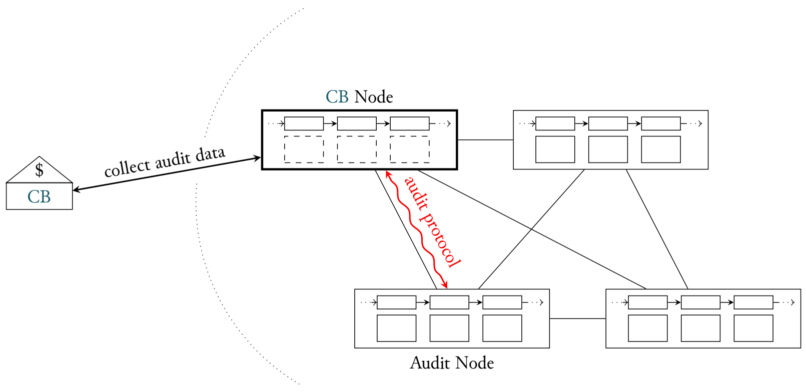


Figure 4.4. Audit model with a CB-controlled blockchain node

Without a CB blockchain node

Actually, if the marked state is committed to the blockchain, there is no need for the CB to maintain a node on the consortial network. Assuming the CB has knowledge of the genesis block's hash value, since the audited data contains all the block header hashes (which can be recalculated for verification), it should be possible to trace back the entire blockchain history to the genesis block. This version is illustrated in Figure 4.5.

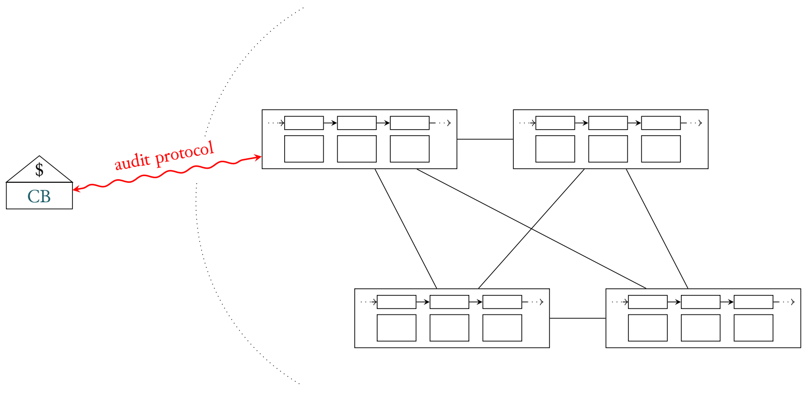


Figure 4.5. Audit model without a CB-controlled blockchain node

5 A ZKP-based Audit Protocol

In the following chapter, we briefly explore what privacy concerns must be considered with regards to audits, and how some cryptocurrencies provide solutions to these problems. In the second half of the chapter, we proceed to define an audit protocol in pseudocode, which will have to be implemented within a ZKP framework.

5.1 Privacy Concerns During Audits

As we briefly discussed in Subsection 2.3.1, a 'classic' audit (eg. where a representative of the CB is given access to the ledger contents and can directly verify that they conform to the requirements) may not be appealing for the consortia or companies with side chains due to privacy reasons. It leaks potentially confidential information to the CB, while even they do not want to access that information.

5.1.1 Privacy on Blockchain Networks

Blockchain technology capable of providing pseudonymous or anonymous accounts can more-or-less protect the identity of individual users, but normally, the transactions themselves are transparent. Everybody can see entries such as $(u_1, u_{20}, 200)$ (see the formalization introduced in Section 4.1) in the ledger.

In public networks such as *Bitcoin*, users (accounts) are identified by the public part of an asymmetric key pair, which results in the pseudonymous nature of *Bitcoin* addresses. One client may generate several identities by generating key pairs.

Experience indicates that pseudonimity is not sufficient to fully protect the privacy of users. A common solution to the problem is the usage of *Bitcoin laundries*, which are aptly named after how they *mix* bitcoins into a pool from whence they can be spent without allowing for tracing the spent coins back to their original owner. The concept is quite simple: one sends bitcoins to the laundry, they get 'clean' coins back, which are disconnected from the user. However, laundries, being third-party services operating over *Bitcoin*, require *trust*, since there is no guarantee a laundry will actually transfer the deserved 'clean' coins to the users who sent their bitcoins into the laundry.

There are extensions to *Bitcoin* which aim to improve on the privacy aspect and make *Bitcoin* more like cash, such as *Zerocoin* and *Zerocash*. Both operate over the *Bitcoin* network and employ various cryptographic technologies, including ZKPs to make coins untraceable. The operation of *Zerocoin* is briefly explained in the next subsection (without striving for accuracy), to give the reader an idea of how privacy-preserving cryptocurrencies can be implemented. In my research, I have studied these technologies in an effort to then understand the extensions to these systems which enable for audits.

Zerocoin is able to hide transaction sources. *Zerocash* also addresses the problem of hiding the amount in the transactions as well as hiding other metadata which *Zerocoin* does not handle, and introduces the notion of Decentralized Anonymous Payment schemes (DAPs) (Ben-sasson et al., 2014).

Given that we seek privacy-preserving audit possibilities on blockchain networks (which just happen to handle bridged CBDC), it is only natural to look into existing concepts in the blockchain world. The problems handled by these solutions are different, however: they aim to ensure auditability on a blockchain network where there are privacy-preserving measures in place, such as *Zerocash*. Naganuma et al. (2017) for example define a protocol to allow authorities to still trace zerocoins by embedding additional information in the transactions (the core idea is to add audit information encrypted by the public key of an auditor).

Initially, it seemed that the auditability problems of bridged CBDC would require similar solutions, but in the end, I used a different approach: the blockchain is arbitrary in terms of privacy features, but the audit protocol ensures that the proving of the satisfaction of the requirements is done in zero knowledge.

5.1.2 An intuitive explanation of *Zerocoin*

Let us say I want to mint a new zerocoin c. I generate a random serial number S and *commit* to it, using a secret r that only I know. For example, one can think of this commitment like putting S into a locked box that only opens with the key r, which only I have. I proceed to put this locked box (commitment) into a publicly accessible area. I also put bitcoins of the same value as c into a publicly accessible cauldron (a container).

Now I would like to spend my zerocoin. To do this, I need to know the secret r that was used to mint the coin – luckily, I was the one who chose it, so I know. Cue ZKPs: I generate a proof π of the statement 'I know a commitment (box) c and a secret key r such that r opens c to S. I publish this proof π together with the serial number S *in disguise*; for example, I post these two to some website anonymously.

Other users verify my proof π as well as the fact that S has not yet been seen (meaning the coin has not been spent yet). Provided my proof was accepted, I am allowed to take c's worth of bitcoins out from the cauldron.

Notice that I can take *any* bitcoins up to the value of c when I spend my zerocoin. By giving away my secret r to another user and providing them with the serial number S, I essentially payed them the coin: note, howhewer, that I *could* spend the coin before the other user, therefore this minimal system is not enough to accomodate for paying other users in zerocoins. One can see that *Zerocoin* operates in a similar manner to *Bitcoin* laundries, but it does more (Miers et al., 2013).

5.2 A complete audit protocol

Now that we have established our blockchain and audit models (see Chapter 4), we can define an exact audit protocol which accounts for privacy as well, using the power of ZKPs.

The protocol outlined in Algorithm 1 assumes direct communication between the CB and a blockchain node; in other words, it follows the model on Figure 4.5. A very similar protocol can be applied to the scenario where a CB-controlled node is present on the blockchain.

Let us start with an arbitrary state of the side-blockchain and an incoming audit request from the CB. The current state is marked: the audit is performed on the data that existed at the time of the audit request. In the meantime, the blockchain can still be operational, blocks can be appended, smart contracts can be executed, etc. From this point, the required steps are as follows:

1. If required, we convert the blockchain state into a format that can be fed into the audit algorithm. In this process, we trim all unnecessary information and essentially generate a *view* of blockchain which is basically just a sequence of transactions.

2. We open a communications channel between the CB and a node on the blockchain to convey audit information.

3. The two parties engage in an Interactive Zero-Knowledge Protocol, during which the algorithm outlined in Algorithm 1 is executed as a computation and successful termination signifies a successful audit. This algorithm must be previously agreed upon by both parties and it is itself public. The concrete arguments of the algorithm are only known by the blockchain node.

4. Whether successful or not, the CB takes note of the event.

Practically, communication can be done over for example TCP with TLS. It is up to the Central Bank to decide how the audit's results are handled: they may require a more in-depth audit to give the consortium a chance to come clear about the situation at the cost of exposing their private data or they may simply record the violation.

Algorithm 1 Blockchain playback verification algorithm

```
Require:
  $\$$ = the total balance
  $H$ = an array of block headers, where each header $h$ has a $prev$ and a $root$ denoted $h.prev$ and $h.root$
  $T$ = an array of block bodies, where each body $b$ consists of an array of *two* transactions $t_1$ and $t_2$, and
  each transaction $t$ has an $s$, a $r$, and an $a$ value, denoted by $t.s$, $t.r$, and $t.a$ respectively
  $W$ = a set of whitelisted recipient accounts
  Hash is an arbitrary hash function
  $+\!\!+$ is the concatenation operator
  function AUDIT($\$, H, T, W$)                        ▷ $\top$ if state conforms to requirements, $\bot$ otherwise
      $B_0 \leftarrow \$$                                          ▷ account 0's initial balance is the total
      $\mathcal{H} \leftarrow nil$                                                      ▷ last block's hash
      for $i \leftarrow 0 \mathinner{..} |H| - 1$ do
          if $\mathcal{H} \neq nil \wedge H_i.prev \neq \mathcal{H}$ then                                    ▷ (5)
              return $\bot$
          end if
          for all $t \in T_i$ do
              if $B_{t.s} < t.a$ then                                                      ▷ (1)
                  return $\bot$
              end if
              if $B_{t.r} \notin W$ then                                                   ▷ (2)
                  return $\bot$
              end if
              $B_{t.r} \leftarrow B_{t.r} + t.a$                                                  ▷ (3)
              $B_{t.s} \leftarrow B_{t.s} - t.a$                                                  ▷ (4)
          end for
          $\mathcal{H} \leftarrow$ HASHBODY($T_i$)
      end for
      return $\top$
  end function

  function HASHBODY($T$)
      $h \leftarrow$ HASHTRANSACTION($T_0$)
      for $j \leftarrow 1 \mathinner{..} |T| - 1$ do
          $h \leftarrow h +\!\!+$ HASHTRANSACTION($T_j$)
      end for
      return $h$
  end function

  function HASHTRANSACTION($t$)
      return Hash($t.s +\!\!+ t.r +\!\!+ t.a$)
  end function
```

6 Zilch Prototype

With the entire protocol available as pseudocode, it is a trivial task to write the necessary code in *Zerojava*, especially once one gets accustomed to *Zerojava*'s quirks.

6.0.1 Inputs and Outputs

First, I define how the blockchain state is converted to an input processable by Zilch. As mentioned in Subsection 3.4.1, *Zilch* gets its inputs on (virtual) *tapes*, like a Turing Machine. Based on our blockchain model established in Section 4.1, and given the format of a *Zilch* tape, it is relatively straightforward to invent a simple encoding. *Zilch* knows integers and booleans. For our purposes, a long enough sequence of integers should be sufficient to describe the entire blockchain state.

Examples of private and public tapes with commented lines can be seen on Listing 6.1 and Listing 6.2 respectively. The public tape contains the whitelist contents.

Listing 6.1: Private input tape example

```
1000              # $ (total balance)
2                 # |𝔹|
4                 # |𝕋|
3                 # |𝔸|

2                 # |{t : t ∈ B_0}|
0                 # prev_0 (ignored; always considered nil)
0                 # 𝒮(t^0_1)
1                 # ℛ(t^0_1)
100               # 𝒜(t^0_1)
0                 # 𝒮(t^0_2)
2                 # ℛ(t^0_2)
50                # 𝒜(t^0_2)

2                 # |{t : t ∈ B_1}|
12345678          # prev_1
1                 # 𝒮(t^1_1)
2                 # ℛ(t^1_1)
50                # 𝒜(t^1_1)
2                 # 𝒮(t^1_2)
1                 # ℛ(t^1_2)
25                # 𝒜(t^1_2)
```

Listing 6.2: Public input tape example

```
2                 # # of blocks to verify (counting from & including last)
2                 # |Whitelist| (let Whitelist be a sequence now)
1                 # Whitelist_0
2                 # Whitelist_1
```

The example private tape encodes the following blockchain state:

$$\begin{aligned}\mathfrak{B} = ((&\overbrace{B_0 = (nil, root_0)}^{\text{Hash}(B_0)=12345678}, T_0 = (t_1^0 = (0,1,100), t_2^0 = (0,2,50))),\\ &(B_1 = (1234578, root_1), T_1 = (t_1^1 = (1,2,50), t_2^1 = (2,1,25))))\end{aligned}$$

Given these inputs, it is not hard to algorithmize the audit protocol in *Zerojava*. The entire resulting source code has been appended to this paper in Listing A.1 in a highly readable form.

6.0.2 Notes on the hashing function

In Chapter 4 and also in Algorithm 1, Hash is defined as an arbitrary hash function. In a real-life scenario, this function is of course the one used by the blockchain system. Since *Ethereum*, a very popular blockchain platform uses KECCAK-256 for hashing, I decided to implement that myself in *Zerojava*, based on the publicly available reference paper and simple C implementation.

This task has proved much more difficult than anticipated: not only does *Zilch* not provide any variable types other than integers, integer arrays, and booleans, but it also seems to suffer from implementation bugs and is very hard to debug, since the only feedback one can get is `System.out.println` outputs during execution, and even those are only capable of printing out integer values.

Over the course of several days, I successfully reimplemented the function using *almost* only *Zerojava*'s limited language. I did this after the realization that since *Zerojava* is basically a subset of *Java*, I am able to develop the algorithm just like developing any *Java* program, paying attention not to use any features unavaiable in *Zerojava*. However, the algorithm relies heavily on (1) 64-bit unsigned integers (`uint64_t` in C) (2) single unsigned bytes (`uint8_t` in C). In *Java*, I was able to use `long`s and `byte`s (with the involvement of some additional bitwise operations), but it is still not trivial to translate this code into *Zerojava*.

Unfortunately, due to some bugs in *Zilch* that I was not able to resolve so far, I could not use a real KECCAK-256 implementation in my demo. Instead I 'mocked' hashing by simply using $x \mod p$ with a large p value to ensure collision resistance ($p = 65\,000$). For reference, however, I decided to include my working implementation as an appendix to this paper, at Listing A.2. This code has been tested as a *Java* program, but is not usable in *Zerojava* in this form, due to the usage of types such as `byte`s and some operators unknown to *Zerojava*. 'Porting' the code is not the real challenge, but getting *Zilch* to execute it seems to be very problematic.

6.1 Challenges and Limitations

As mentioned, *Zilch* seems to have implementation bugs. For short, simple programs, this usually does not cause problems (even though some instructions are simply broken, so much that even the unit tests that are shipped with *Zilch* do not execute), but they become more and more apparent for larger programs such as the one I implemented. Not to mention my KECCAK-256 implementation, which itself is magnitudes more complex than the audit code.

One of the problems I discovered during development is that the 'bitwise shift right' (>>) operator causes overflows and therefore segmentation faults when *Zilch* is compiled with 64-bit registers. *Zilch* ships with unit tests, which can be built into a binary `zilch-tests`. There is one unit test that ensures the correct behaviour of the shift right instruction, but it fails, when register length is set to 64. After several hours of debugging, I localized the error to be in *Zilch*'s algebra library, where at one point, data is written to an array that does not fit into the reserved memory anymore. I successfully (and rather inelegantly) solved this problem by increasing the value of a previously defined macro `ExtensionSize` from 128 to a 4096, which got rid of the array overflow.

Zilch also exhibited behaviour that can only be labelled *strange* while executing larger programs with several nested loops. Based on the information I could gather from debug `System.out.println` statements, *Zilch* unexpectedly jumps out of the current iteration and starts it over, again and again. I cannot tell whether it is only what *Zilch* prints on the output, or if the iterator variable really does not change value due to this, causing an infinite loop.

Compiling *Zerojava* is also not as painless as one would hope. Several operators that are available according to the specification do not actually work, such as the `XOR` assignment operator `^=`. *Zerojava* also requires excessive parenthetization: for example, one may not write `foo[i] ^ bar[i]`, only as `(foo[i]) ^ (bar[i])`. The error messages are rather misleading, but one can eventually guess what might be the problem, given at least the line number of the offending statement.

Unfortunately, even perfectly valid and parseable *Zerojava* code does not always compile well. The *Zerojava* compiler employs a visitor design pattern when processing the parsed input, but there are several variations of a mistake in the code: on a line where a single instruction is about to be appended to a `StringBuilder`, a function is called, which has the side effect of appending data to the same `StringBuilder`, causing mangled instructions where an instruction gets wedged in the middle of another. During compilation, the *Zerojava* compiler translates the code between several formats: this particular problem happens at the *Kanga* to *Spiglet* conversion step. Thankfully, this bug was relatively easy to fix: one simply needs to refactor the problematic line such that the function that is called inside the arguments of the `StringBuilder` append call is called *before*, on a separate line, then save the return value to a variable and pass that as an argument to the `StringBuilder` appending.

Zilch is a very powerful tool, but it can be a little confusing at times, difficult to handle. I have contacted the developer for more information, who has responded to my queries promptly, and gave me some advice that I was not yet able to try out. These problems delayed the successful implementation of the KECCAK-256 hash function in *Zerojava*, but it should be noted that evidently, the implementation should be possible, provided that the bugs mentioned are either worked around or eliminated.

6.2 Operation

The *Zerojava* compiler can be used to generate the *zMIPS* instructions for *Zilch*. Then, executing the *Zilch* binary with the right command-line arguments starts a simulated Prover–Verifier interaction during which the program runs and *Zilch* prints out the results to the standard output. In addition to the result, the timesteps that were required are also printed. Listing 6.3 shows the output of a demo execution.

Listing 6.3: *Zilch*'s output in a demo execution

```
$ zilch --asm Audit.zmips
----------------------------------------------------------
| Results of /home/bp99/zkp/audit/Audit.zmips            |
----------------------------------------------------------
| Answer (decimal)   = 1                                 |
| 32-bit answer      = 00000000000000000000000000000001 |
| Timesteps required = 2244                              |
----------------------------------------------------------
```

Based on the algorithm I am using, I expected $O(n)$ time results in n, the blockchain's length on the private tape. I have evaluated my implementation for inputs as little as 6 blocks up to inputs where there were 100 blocks on the private tape. The number of transactions in each block remained constant (two). Figure 6.1 shows a very clear linear relationship between the input size and the timesteps required to prove conformity. Timesteps are *Zilch*'s own units: they represent the execution of one instruction.

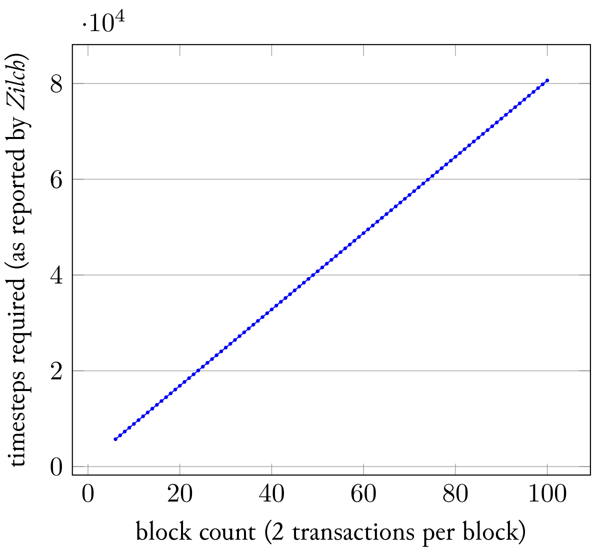


Figure 6.1. Evaluation results: number of timesteps required plotted against input size

7 Conclusion and Further Work

I have successfully implemented the audit protocol of my own design in *Zerojava*, and evidence suggests that the implementation is correct and the performance of the program is adequate (achieving linear execution time in input size). I have also done most of the work required to implement KECCAK-256 in *Zerojava* and therefore as a `zMIPS` assembler code for *Zilch*, but a careful review of my conversion of some *Java* features to *Zilch*-only features is in order. Other than that, *Zilch* or the *Zerojava* compiler definitely contains bugs, which hindered development; I have contacted the author and we are presently discussing the problems I reported.

As a continuation of my work, I plan to iteratively relax the limitations of my blockchain model, starting with, for example, the fixed number of transactions per block. An exciting challenge is how bridged CBDC that is processed by smart contracts can be handled. Ideally, I would like to be able to apply my protocol to real life blockchain systems, presumably with an additional step of taking a restricted view of the blockchain state (only collecting data that are necessary in an audit).

Looking further, the audit protocol outlined in this paper is not specialized to CBDC systems whatsoever. The prospects of applying the methodology in the broader area of intercompany integrations is certainly worth looking into.

I must also add that even though *Zilch*, or in fact its front end *Zerojava* enabled quick prototyping, especially given my inexperience in the area, it seems that it may not be the best tool of choice for the complexity of the problem – the implementation of the hash functions used by cryptocurrencies today already seemed too much for either the framework, or at least the compiler to handle. Also, an different implementation might employ non-interactive proofs instead of an interactive protocol like the one I designed, at the potential cost of performance. There are indubitably many possibilities to explore, even when solely considering the choice of cryptographic toolkit.

Acronyms

AC	Arithmetic Circuit 2, 12, 13, 14, 17, 18
CB	Central Bank 2, 3, 4, 5, 6, 7, 22, 23, 24, 25, 26, 27
CBDC	Central Bank Digital Currency 1, 2, 3, 4, 5, 6, 7, 12, 19, 22, 23, 26, 33
CP-SNARK	Commit and Prove Succinct Non-interactive ARgument of Knowledge 12
DAG	Directed Acyclic Graph 13
DAP	Decentralized Anonymous Payment scheme 26
DLT	Distributed Ledger Technology 5, 6, 7
IZKP	Interactive Zero-Knowledge Protocol 8, 10, 17, 27
KYC	Know Your Customer 5
MIPS	Microprocessor without Interlocked Pipelined Stages 17
NIZKP	Non-Interactive Zero-Knowledge Proof 10, 12
NP	Nondeterministic Polynomial time 10
OO	Object-Oriented 14, 15
RSA	Rivest, Shamir, Adleman 13
TCP	Transmission Control Protocol 27
TLS	Transport Layer Security 27
TM	Turing Machine 8, 9, 10, 17, 29
UTM	Universal Turing Machine 9
ZK	Zero-Knowledge 10, 12
ZKP	Zero-Knowledge Proof 1, 2, 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 26, 27
zk-ShNARK	Zero-Knowledge Short Non-interactive ARgument of Knowledge 14
zk-SNARK	Zero-Knowledge Succinct Non-interactive ARgument of Knowledge 11, 12, 14
zk-STARK	Zero-Knowledge Succinct Transparent ARgument of Knowledge 3, 11, 12, 14, 17, 18

Bibliography

Auer, R. A. and Böhme, R. (2020). The technology of retail central bank digital currency. *Monetary Economics: Central Banks - Policies & Impacts eJournal*.

Ben-Sasson, E., Bentov, I., Horesh, Y., and Riabzev, M. (2018). Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046. `https://ia.cr/2018/046`.

Ben-sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., and Virza, M. (2014). Zerocash: Decentralized anonymous payments from bitcoin. pages 459–474.

Ben-Sasson, E., Chiesa, A., Tromer, E., and Virza, M. (2014). Succinct non-interactive zero knowledge for a von neumann architecture. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 781–796, USA. USENIX Association.

Benarroch, D., Campanelli, M., Fiore, D., and Kolonelos, D. (2019). Zero-knowledge proofs for set membership: Efficient, succinct, modular. In *IACR Cryptol. ePrint Arch.*

Bootle, J., Chiesa, A., and Liu, S. (2020). Zero-knowledge IOPs with linear-time prover and polylogarithmic-time verifier. Cryptology ePrint Archive, Report 2020/1527. `https://ia.cr/2020/1527`.

Campanelli, M., Fiore, D., and Querol, A. (2019). Legosnark: Modular design and composition of succinct zero-knowledge proofs. Cryptology ePrint Archive, Report 2019/142. `https://ia.cr/2019/142`.

Feige, U., Fiat, A., and Shamir, A. (1988). Zero-knowledge proofs of identity. *Journal of Cryptology*, 1:77–94.

Goldreich, O., Micali, S., and Wigderson, A. (1991). Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. *J. ACM*, 38:691–729.

Goldwasser, S., Micali, S., and Rackoff, C. (1989). The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18:186–208.

Groth, J. (2016). On the size of pairing-based non-interactive arguments. Cryptology ePrint Archive, Report 2016/260. `https://ia.cr/2016/260`.

Koumbarakis, A. and Dobrauz-Saldapenna, G. (2019). Central bank digital currency: Benefits and drawbacks. page 9.

Manuel, B., Alfredo, D. S., Silvio, M., and Giuseppe, P. (1991). Noninteractive zero-knowledge. *SIAM J. Comput.*, 20(6):1084–1118.

Miers, I., Garman, C., Green, M., and Rubin, A. D. (2013). Zerocoin: Anonymous distributed e-cash from bitcoin. In *2013 IEEE Symposium on Security and Privacy*, pages 397–411.

Mohr, A. (2007). A survey of zero-knowledge proofs with applications to cryptography.

Mouris, D. and Tsoutsos, N. G. (2021). Zilch: A framework for deploying transparent zero-knowledge proofs. *IEEE Transactions on Information Forensics and Security*, 16:3269–3284.

Naganuma, K., Yoshino, M., Sato, H., and Suzuki, T. (2017). Auditable zerocoin. In *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, pages 59–63.

Opare, E. and Kim, K. (2020). A compendium of practices for central bank digital currencies for multinational financial infrastructures. *IEEE Access*, page 1.

Parno, B., Howell, J., Gentry, C., and Raykova, M. (2013). Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252.

Sun, J. (2021). Reflections on the latest e-cny pilot test in china. *Journal of Asia-Pacific and European Business*, 1(01).

Thaler, J. (2021). Proofs, arguments, and zero-knowledge. unpublished, 'made publicly available in conjunction with the Fall 2020 offering of COSC 544 at Georgetown University'.

Appendices

A Complete Source Code Listings

Listing A.1: Complete implementation of Algorithm 1 in *Zerojava*

```
class Audit {

        public static void main(String[] _args)
        {
                Hash    H;                      // hash calculator
                int[]   wl;                     // whitelist
                int[]   txSs, txRs, txAs;       // tx Senders,
                                                // Receivers, Amounts
                int[]   bal;                    // balances
                int[]   hash;                   // block header hashes
                int[]   txc;                    // tx count per block
                int     total;                  // total balance
                int     blkcnt, txcnt, ucnt;    // #blocks, #txs,
                                                // #accounts
                int     wllen;                  // length of whitelist
                int     lastn,                  // how many txs to
                                                // verify
                // the following are helper variables
                int     lasthash, curhash;
                int     i, j, k, l;
                int     m, z;
                int     he, ha;
                int     s, r, a;
                int     bals, balr;
                int     erri, errj, errt;
                int     nonval, end;
                boolean ok, err;


                /* PrivateTape format expected as defined */
                total   = PrivateTape.read();
                blkcnt  = PrivateTape.read();
                txcnt   = PrivateTape.read();
                ucnt    = PrivateTape.read();

                lastn   = PublicTape.read();
                wllen   = PublicTape.read();

                txSs    = new int[txcnt];
                txRs    = new int[txcnt];
                txAs    = new int[txcnt];

                // indices are account IDs, values are the balances
                bal     = new int[ucnt];

                // indices are block indexes,
                //   values are the previous block header's hashes
                hash    = new int[blkcnt];
                //   values are the number of txs in the block
                txc     = new int[blkcnt];

                wl      = new int[wllen];

                H       = new Hash();

                nonval  = txcnt - lastn;        // ignored txs
                end     = txcnt;


                /* Read whitelist */
                i = 0;
                while (i < wllen) {
                        wl[i] = PublicTape.read();
                        i++;
                }

                /* Read txs */
                i = 0;
                while (i < blkcnt) {
                        txc[i]  = PrivateTape.read();
                        hash[i] = PrivateTape.read();

                        m = txc[i];
                        j = 0;
                        while (j < m) {
                                // offset
                                l = (i > 0) ? txc[(i - 1)] : 0;
                                k = (i * l) + j;

                                txSs[k] = PrivateTape.read();
                                txRs[k] = PrivateTape.read();
                                txAs[k] = PrivateTape.read();

                                j++;
                        }

                        i++;
                }


                /* Set inital balances */
                bal[0] = total;
                i = 1;
                while (i < ucnt) {
                        bal[i] = 0;
                        i++;
                }

                /* Non-validating playback */
                lasthash = 0; curhash = 0;
                i = 0; j = 0; k = 0;
                while (k < nonval) {
                        s = txSs[k]; r = txRs[k]; a = txAs[k];
                        bals = bal[s]; balr = bal[r];

                        curhash += ((s + r) + a);

                        bal[s]  = bals - a;
                        bal[r]  = balr + a;

                        j++;
                        if (j == (txc[i])) {
```

```
                i++;
                j = 0;
                lasthash = curhash;
                curhash = 0;
            }

            k++;
        }
        // (i, j) is now the last (blk, tx) idx seen

        /* Validating playback */
        err = false; erri = -1; errj = -1; errt = -1;
        while (k < end) {
            r = txRs[k]; s = txSs[k]; a = txAs[k];
            bals = bal[s]; balr = bal[r];

            curhash += ((r + s) + a);

            bal[s]  = bals - a; // (3)
            bal[r]  = balr + a; // (4)

            // unless an error has already been found...
            if (!err) {
                ok = true;

                // verify last hash if at start of block
                // (5)
                if (j == 0) {
                    he = hash[i];
                    ha = H.hash(lasthash);
                    if (ha != he) {
                        ok = false;
                        errt = 5;
                    }
                }

                // whitelisted receiver (2)
                if (ok) {
                    ok = false;
                    z = 0;
                    while (z < wllen) {
                        if (r == (wl[z]))
                            ok = true;
                        z++;
                    }
                    if (!ok)
                        errt = 2;
                }

                // sufficient balance (1)
                if (ok) {
                    ok = (bals >= a);
                    if (!ok)
                        errt = 1;
                }

                // check if OK
                if (!ok) {
                    err = true;
                    erri = i;
```

```
                                        errj = j;
                                }
                        }

                        j++;
                        if (j == (txc[i])) {
                                i++;
                                j = 0;
                                lasthash = curhash;
                                curhash = 0;
                        }

                        k++;
                }

                /* Print errors */
                if (err) {
                        System.out.println(erri);
                        System.out.println(errj);
                        // errt mapping:
                        // -1 : no error (should never be printed)
                        //  1 : insufficient sender balance
                        //  2 : receiver not allowed
                        //  5 : bad prev hash
                        System.out.println(errt);
                }

                Prover.answer((err) ? 1 : 0);
        }

}


/* Mock Hash class instead of KECCAK256 */
class Hash {

        // for [(0,1,100),(0,2,50)] --> 153
        public int hash(int k)
        {
                return k % 65000;
        }

}
```

Listing A.2: Complete implementation of KECCAK in *Zerojava*

```
/*
 * ZeroJava reimplementation of the reference KECCAK
 * implementation on github.com/XKCP/XKCP
 * (Standalone/CompactFIPS202/C/Keccak-readable-and-compact.c)
 */
class KECCAK {

        /*
         * ------------------------------------------------------------
         * The following are helper methods used to get around the
         * limitations of Zerojava.
         * ------------------------------------------------------------
         */

        /* Bitwise negation as a method call: just like `~x' in C */
        public long not(long x)
        {
                // int -1 represents an all-1-bit value
                // (11...1 in binary)
                return (x ^ -1);
        }

        /*
         * Convert an array of 8 bytes to a 64-bit integer
         * (long, LITTLE ENDIAN byte order)
         */
        public long bytes2long(byte[] a)
        {
                long result;
                int i;

                result = 0;
                i = 7;
                while (i >= 0) {
                        result = ((result << 8) | ((a[i])) & 0xFF);
                        i--;
                }

                return result;
        }

        /*
         * Convert a 64-bit integer (long) to an array of 8 bytes
         * (LITTLE ENDIAN byte order)
         */
        public byte[] long2bytes(long x)
        {
                byte[] result;
                int i;

                result = new byte[8];
                i = 7;
                while (i >= 0) {
                        result[i] = (byte) ((x >> (i*8)) & 255);
                        i--;
                }

                return result;
        }


        /*
         * -------------------------------------------------------------
         * The following methods implement macros and operators from the
         * reference implementation.
         * -------------------------------------------------------------
         */

        /* The `MIN' macro from the reference implementation */
        public long min(long a, long b)
        {
                return (a < b) ? a : b;
        }

        /* The `ROL64' macro from the reference implementation */
        //
        // [A B C D E ... P Q R S]
        //  ^
        //  +---------------------+
        //                        v
        // ==> [B C D ... P Q R S A]
        // (moves off bits)
        public long rol64(long a, long off)
        {
                return ((a << off) ^ (a >>> (64 - off)));
        }

        /*
         * The indexing macro named `i' from the reference
         * implementation
         */
        public int i(int x, int y)
        {
                return (x + (5 * y));
        }

        /*
         * The `readLane' macro from the reference implementation.
         * The state is passed as a parameter as well.
         */
        public long readLane(byte[] state, int x, int y)
        {
                byte[] lane;
                int off;
                int i;

                lane = new byte[8];
                off = this.i(x, y);

                i = 0;
                while (i < 8) {
                        lane[i] = state[(8*off) + i];
                        i++;
                }

                return bytes2long(lane);
        }

        /* The `writeLane' macro from the reference implementation. */
        public long writeLane(byte[] state, int x, int y, long lane)
        {
                byte[] lanebytes;
                int off;
                int i;

                lanebytes = long2bytes(lane);
                off = this.i(x, y);

                i = 0;
                while (i < 8) {
                        state[(8*off) + i] = lanebytes[i];
                        i++;
                }

                return 0; // ignored
        }

        /* The `XORLane' macro from the reference implementation. */
        public long xorLane(byte[] state, int x, int y, long lane)
        {
                byte[] lanebytes;
                int off;
                int i;

                lanebytes = long2bytes(lane);
                off = this.i(x, y);

                i = 0;
                while (i < 8) {
                        state[(8*off) + i]
                            = (byte) ((state[(8*off) + i])
                            ^ (lanebytes[i]));
                        i++;
                }

                return 0; // ignored
        }


        /*
         * -------------------------------------------------------------
         * The following methods are reimplementations of the reference
         * implementation's functions
         * -------------------------------------------------------------
         */

        /*
         * Function that computes the linear feedback shift register
         * (LFSR) used to define the round constants
         * (see [Keccak Reference, Section 1.2]).
         *
         * -- from the reference implementation's LFSR86540 function
         *
         * Instead of modifying the value using a polonger, this method
         * returns the value that the LFSR should be set to, which is
         * the responsibility of the caller.
         * The reference implementation already returns a boolean value
         * though, (LFSR & 0x01 != 0).
         *
         * To return both values, this method actually returns an array
         * of two elements: the first one being 0 or 1 based on the
         * value of the boolean expression (1 if true of course) and the
         * second being the new value of LFSR which should be assigned
         * by the caller.
         */
        public byte[] lfsr86540(byte lfsr)
        {
                // result: [boolean(byte) return, byte new lfsrstate]
                byte[]        result;

                result = new byte[2];
                result[0] = (byte) (((lfsr & 0x01) != 0) ? 1 : 0);

```

```
		if ((lfsr & 128) != 0) // 0x80 = 128
			// x^8 + x^6 + x^5 + x^4 + 1 --> 0x71 = 113
			result[1] = (byte) ((lfsr << 1) ^ 113);
		else
			result[1] = (byte) (lfsr << 1);

		return result;
	}

	/*
	 * Function that computes the Keccak-f[1600] permutation of the
	 * given state.
	 *
	 * -- from the reference implementation's
	 *    KeccakF1600_StatePermute function
	 *
	 * Since Zerojava does not have pointers, the state is passed as
	 * an array and the method returns the new permuted state array
	 * which should be assigned to the state variable -- a
	 * responsibility of the caller.
	 */
	public long f1600statePermute(byte[] state)
	{
		// vars used in the reference impl as well
		long		round;			// round counter
		int		x, y, t, j;		// loop vars
		long		r, Y;			// helper vars for the rho and
						// pi steps
		// temporary vars needed to overcome Zerojava
		// limitations
		boolean		b;
		long		_;			// always ignored
		// -- theta step
		long[]		C;
		long		D;
		// -- rho and pi steps
		long		current, pitemp;
		// -- chi step
		long[]		chitemp;
		// -- iota step
		long		bitpos;			// `bitPosition'
		byte		lfsrstate;		// `LFSRstate'
		byte[]		lfsrresult;

		lfsrstate = 1; // 0x01

		round = 0;
		C = new long[5];
		while (round < 24) {
			/*
			 * -------- theta step --------
			 * (see [Keccak Reference, Section 2.3.2])
			 */

			// Compute the parity of the columns
			x = 0;
			while (x < 5) {
				C[x] = this.readLane(state, x, 0);
				y = 1;
				while (y < 5) {
```

```
                                        C[x] = ((C[x])
                                             ^ (this.readLane(state,
                                                              x, y)));
                                        y++;
                                }
                                x++;
                        }
                        x = 0;
                        while (x < 5) {
                                // Compute the theta effect for a given
                                // column
                                D = ((C[((x + 4) % 5)])
                                  ^ (this.rol64(C[((x + 1) % 5)],
                                     1)));
                                // Add the theta effect to the whole
                                // column
                                y = 0;
                                while (y < 5) {
                                        _ = this.xorLane(state, x, y,
                                                         D);
                                        y++;
                                }
                                x++;
                        }


                        /*
                         * -------- rho and pi steps --------
                         * (see
                         * [Keccak Reference, Section 2.3.3 and 2.3.4])
                         */
                        // Start at coordinates (1 0)
                        x = 1; y = 0;
                        current = this.readLane(state, x, y);
                        t = 0;
                        // Iterate over
                        // ((0 1)(2 3))^t * (1 0)
                        // for 0 <= t <= 23
                        while (t < 24) {
                                // Compute the rotation constant
                                // r = (t+1)(t+2)/2
                                r = ((((t + 1) * (t + 2)) / 2) % 64);
                                // Compute ((0 1)(2 3)) * (x y)
                                Y = (((2 * x) + (3 * y)) % 5);
                                x = y;
                                y = (int) Y;
                                // Swap current and state(x, y),
                                // and rotate
                                pitemp  = this.readLane(state, x, y);
                                _ = this.writeLane(state, x, y,
                                                   this.rol64(current,
                                                              r));
                                current = pitemp;

                                t++;
                        }


                        /*
                         * -------- chi step --------
                         * (see [Keccak Reference, Section 2.3.1])
                         */
                        y = 0;
                        chitemp = new long[5];
                        while (y < 5) {
                                // Take a copy of the plane
                                x = 0;
                                while (x < 5) {
                                        chitemp[x]
                                            = this.readLane(state,
                                                            x, y);
                                        x++;
                                }

                                x = 0;
                                while (x < 5) {
                                        _ = this.writeLane(state, x, y,
                                                           ((chitemp[x])
                                           ^ ((this.not(
                                                chitemp[((x + 1) % 5)]))
                                           & (chitemp[((x + 2) % 5)]))));
                                        x++;
                                }

                                y++;
                        }


                        /*
                         * -------- iota step --------
                         * (see [Keccak Reference, Section 2.3.5])
                         */
                        j = 0;
                        while (j < 7) {
                                bitpos = ((1 << j) - 1);
                                lfsrresult = this.lfsr86540(lfsrstate);
                                b = (lfsrresult[0] == 1);
                                lfsrstate = lfsrresult[1];
                                if (b)
                                        _ = this.xorLane(state, 0, 0,
                                                         (1L << bitpos));
                                j++;
                        }

                        round++;
                }

                return 0; // ignored
        }


        /*
         * Function to compute the Keccak[r, c] sponge function over a
         * given input.
         *
         * Parameters:
         *         r                The value of the rate r.
         *         c                The value of the capacity c.
         *         input                The input message.
         *         delimSuffix        Bits that will be automatically appended
         *                         to the end of the input message, as in
         *                         domain separation.
         *                         This is a byte containing from 0 to 7
         *                         bits.
         *                         These n bits must be in the least
         *                         significant bit positions and must be
         *                         delimited witha  bit at position n
         *                         (counting from 0=LSB to 7=MSB) and
         *                         followed by bits 0 from position n+1 to
         *                         position 7.
         *                         Some examples:
         *                         - If no bits are to be appended, then
         *                           delimSuffix must be 0x01.
         *                         - If the 2-bit sequence 0,1 is to be
         *                           appended (as for SHA3-*), delimSuffix
         *                           must be 0x06.
         *                         - If the 4-bit sequence 1,1,1,1 is to be
         *                           appended (as for SHAKE*), delimSuffix
         *                           must be 0x1f.
         *                         - If the 7-bit sequence 1,1,0,1,0,0,0 is
         *                           to be absorbed, delimSuffix must be
         *                           0x8b.
         *         outLen                The number of output bytes desired.
         *
         * One must have r+c=1600 and the rate a multiple of 8 bits in
         * this implementation.
         *
         * -- from the reference implementation
         */
        public byte[] keccak(long r, long c, byte[] input,
                             byte delimSuffix, long outLen)
        {
                // vars used by the reference implementation
                byte[]        state;
                long        rbyte, blksiz;        // `rateInBytes' and `blockSize'
                int        i;                 // a loop variable
                // vars used to overcome Zerojava limitations
                byte[]        out;                // output as a byte array
                int        j, l;                 // additional loop variables
                long        _;

                // initialize some variables
                rbyte = r / 8;
                blksiz = 0;

                // verify parameters are acceptable
                if (((r + c) != 1600) || ((r % 8) != 0))
                        System.exit(8);

                // Initialize the state
                state = new byte[200];
                j = 0;
                while (j < 200) {
                        state[j] = 0;
                        j++;
                }

                // Absorb all the input blocks
                l = input.length;
                while (l > 0) {
                        blksiz = this.min(l, r);
                        i = 0;
                        while (i < blksiz) {
                                state[i] = (byte) ((state[i])
                                         ^ (input[i]));
                                i++;
                        }
                        l -= blksiz;

                        if (blksiz == r) {
                                _ = this.f1600statePermute(state);
                                blksiz = 0;
                        }

                        l--;
                }

                // Do the padding and switch to the squeezing phase
                // Absorb the last few bits and add the first bit of
                // padding (which coincides with the delimiter in
                // delimSuffix)
                state[(int) blksiz]
                    = (byte) ((state[(int) blksiz]) ^ (delimSuffix));
                // If the first bit of padding is at position r-1, we
                // need a whole new block for the second bit of padding
                // /instead of 0x80, 128 is used/
                if (((delimSuffix & 128) != 0) && (blksiz == (r - 1)))
                        _ = this.f1600statePermute(state);
                // Add the second bit of padding
                state[(int) (rbyte - 1)]
                    = (byte) ((state[(int) (rbyte - 1)]) ^ 128);
                // Switch to the squeezing phase
                _ = this.f1600statePermute(state);

                // Squeeze out all output blocks
                // first, we must initialize the output array
                out = new byte[(int) outLen];
                i = 0;
                while (i < outLen) {
                        out[i] = 0;
                        i++;
                }
                // now for the actual squeezing
                j = 0;
                while (outLen > 0) {
                        blksiz = this.min(outLen, r);

                        i = 0;
                        while (i < blksiz) {
                                out[(j + i)] = state[i];
                                i++;
                        }

                        outLen -= blksiz;

                        if (outLen > 0)
                                _ = this.f1600statePermute(state);
                }

                return out;
```

```
            }

}
```