南 开 大 学

本科生毕业论文（设计）

中文题目：区块链压缩技术探究——bitcoin 区块链压缩以及以太坊 zk-Rollup

外文题目：Reasearch on Technologies of Bitcoin Data Compression -- Bitcoin Blockchain Compression and zk-Rollup of Ethereum

学　　号：1811391
姓　　名：刘衍辰
年　　级：2018 级
专　　业：信息安全
系　　别：信息安全
学　　院：网络空间安全学院
指导教师：苏明

完成日期：2023 年 5 月

关于南开大学本科生毕业论文（设计）的声明

本人郑重声明：所呈交的学位论文，是本人在指导教师指导下，进行研究工作所取得的成果。除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人创作的、已公开发表或没有公开发表的作品内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。本学位论文原创性声明的法律责任由本人承担。

学位论文作者签名：

2023 年　05 月　05 日

本人声明：该学位论文是本人指导学生完成的研究成果，已经审阅过论文的全部内容，并能够保证题目、关键词、摘要部分中英文内容的一致性和准确性。

学位论文指导教师签名：

2023 年　05 月　05 日

摘　要

区块链技术已经成为近年来备受关注的新兴技术，但是由于其基于分布式记账和去中心化的特点，导致其数据存储和传输成本相对较高。为了解决这一问题，本文在已有的方法基础上进行了实践操作并进行优化，使得可以有效地压缩区块链中的数据，并且不会影响其安全性和去中心化特性。

该方法可以将区块链中的交易数据进行有效压缩，并且不会引入任何安全隐患。具体来说，该算法利用了区块链中数据的冗余性，通过去除重复数据和使用更紧凑的编码方式来实现数据的压缩。此外，该算法还能够根据数据的特性和访问模式进行自适应压缩，以进一步提高压缩效率

同时作为基于区块链技术所搭建的智能合约平台，本文采用zk-SNARK中的rollup技术，抽象现有的账本模型并加以SNARK零知识证明的方法，可以有效地压缩以太坊区块链中的数据，并且不会影响其安全性和智能合约的执行效率。

关键词：区块链；比特币存储压缩；zk-SNARK；以太坊rollup

Abstract

Blockchain technology has become an emerging technology that has attracted much attention in recent years, but due to its characteristics based on distributed accounting and decentralization, its data storage and transmission costs are relatively high. In order to solve this problem, this paper conducts practical operations and optimizes the existing methods so that the data in the blockchain can be effectively compressed without affecting its security and decentralization characteristics.

This method can effectively compress the transaction data in the blockchain without introducing any security risks. Specifically, the algorithm takes advantage of the redundancy of data in the blockchain to achieve data compression by removing duplicate data and using a more compact encoding. In addition, the algorithm can also perform adaptive compression according to data characteristics and access patterns to further improve compression efficiency

At the same time, as a smart contract platform based on blockchain technology, this paper adopts the rollup technology in zk-SNARK, abstracts the existing account book model and adds SNARK zero-knowledge proof method, which can effectively compress the data in the Ethereum blockchain. And it will not affect its security and the execution efficiency of smart contracts.

Key words:Blockchain, Bitcoin storage compression, zk-SNARKs, rollup in Ethereum.

目 录

第一章 绪论

第一节 研究背景及研究意义

随着数字经济的发展和互联网应用的广泛普及，人们对于安全、透明、去中心化的信息管理方式的需求逐渐增加。区块链技术的出现，正是为了满足这一需求而应运而生。

区块链技术最初是由比特币的发明人中本聪提出的，旨在解决数字货币的去中心化和可信交易问题。随着时间的推移，区块链技术逐渐扩展到了金融、供应链、电子票据等领域，并且其应用范围还在不断拓展。目前，区块链技术已经成为全球范围内备受关注的新兴技术，并且在政府、企业、金融机构等领域得到了广泛的应用和推广。[1]

然而，虽然区块链技术具有诸多优势，如去中心化、安全、可追溯等，但其本身也存在一些问题和挑战，例如性能瓶颈、隐私保护等。因此，如何进一步完善区块链技术，提高其性能和应用范围，成为当前研究的热点和难点之一。在这种背景下，本文旨在探讨区块链技术的发展现状和未来趋势，分析其优缺点和面临的挑战，以及介绍近年来涌现的新型区块链技术和应用案例。通过对区块链技术的深入探讨和研究，为推动区块链技术的发展和应用提供有益的参考和启示。

为此，近年来涌现了一系列区块链存储压缩技术。这些技术通过各种方式，包括数据去重、压缩算法、分布式存储等[2]，来降低区块链数据的存储和传输成本，并且能够提高区块链的可扩展性和性能。例如，一些研究者提出了基于数据去重的压缩技术，利用区块链中数据的重复性，通过去除重复数据和使用更紧凑的编码方式来实现数据的压缩。还有一些研究者利用分布式存储技术，将区块链数据分散存储在多个节点上，从而实现更高的存储效率和数据可用性。[3]

尽管这些技术在一定程度上提高了区块链的可扩展性和性能，但是仍然存在一些问题和挑战，例如压缩算法的复杂度、数据安全和隐私保护等。因此，如何进一步研究和发展区块链存储压缩技术，并且在实际应用中发挥其优势和价值，仍然是当前研究和实践中的重要议题。

第二节 论文研究内容

由上文可知，数据去重、数据结构的优化可以作为区块链数据本地存储中的重点关注的点，因此本文主要分为两个大部分，一是基于比特币的已花销的交易记录的删减，二是基于以太坊账户模型存储结构的优化和零知识证明的部署，以保证该结构正确、有效：

1）比特币已花销交易的删减

本文对于比特币本地存储数据主要采用Python脚本进行操作，由于Bitcoin-qt同步下来的数据为unreadable的二进制.dat形式，因此先对其进行了编译转换为可读的txt文本形式，在熟知了每个.dat文件所存储的信息、每一个block所包含的信息以及每一条交易所存储的数据之后，计算交易的哈希值保留作为比对对象，最后转换回二进制的.dat形式并与原存储文件进行大小比对。由于本地存储大小有限，只下载同步了7个.dat文件进行实操，每个.dat文件中存储6000-8000个block，每个block中包含若干笔交易。

2）以太坊用户模型rollup以及zk-SNARK的应用

以太坊区别于初代blockchain技术应用的Bitcoin所采用的UTXO模型，交易双方的账户名（或称地址）成为了关键因素，每一笔交易会影响一个账户下的余额。因此以太坊的主链并不需要存储大量的交易信息而最好保证每一个账户的信息准确无误，并且需要经得起验证——即实现去中心化。本文同样主要采用python脚本从区块数据中提取账户信息（address）构建一棵用户状态的Merkle Tree，新交易只会影响新用户Merkle Tree的根节点和叶子结点。因此主链上就不必存储冗余的交易信息以及交易所打包形成的区块，转而使用更紧致的账户信息和基于libsnark生成的merkle tree零知识证明，可以极大程度上降低主链上负载的压力。

第三节 论文组织结构

本文从结构上主要分为五个部分，每个部分的主要内容如图 1 所示

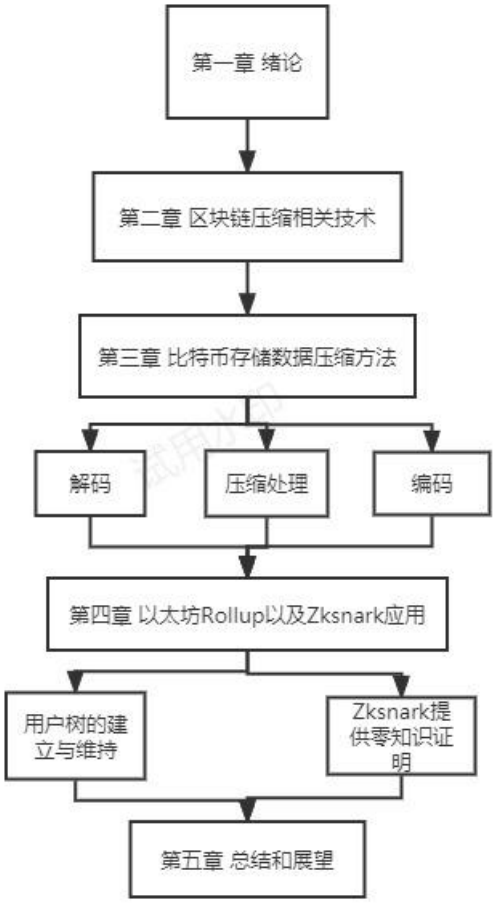


图1 论文结构图

第一章是绪论，主要介绍了区块链发展的研究背景与研究意义，对相关定义和概念进行了阐述，描述介绍了本文的研究内容和论文结构图。

第二章是对区块链压缩、实现减少开销的相关技术的说明，介绍了本文所采用的比特币存储压缩方案和以太坊 Rollup 方案的实现，以及最后需要使用 zk-SNARK 提供零知识证明的方案设计，描述了技术发展的背景和依据的特点，为后续实验的操作提供了理论基础。

第三章是对基于比特币 UTXO 机制的区块链系统压缩所实现需要进行的实验内容，本文通过同步数据，通过学习数据编码规则解析编码，并对需要的数据进行了保留筛选，最后以原始编码方案进行文件数据的写回。通过比对前后数据内存的大小和权衡中间过渡文件大小对本文所实现的工作进行效果评估。

第四章是对以太坊 Rollup 技术进行了实现，通过对区块数据的抽象化，回归传统的基于账户模型的方案，并借鉴比特币中 MerkleTree 的使用办法，构建用户状态模型，通过维护 32Bytes 的根节点来保持整个以太坊服务的稳定运行。并且由于类似于传统的中心化银行机制，需要对内部的数据提供证明使得外部用户在不知道秘密的情况下进行验证，一定程度上保持了去中心化的区块链设计准则。本文通过对 libsnark 库文件的学习和使用，能够生成对于小规模交易的处理和证明。除此之外，对于交易的有效性和对用户状态产生影响的正确性也需要通过 zk-SNARK 提供证明，本文在已有技术的基础上能够生成实现一笔交易的完整证明。

第五章对全文所涉及的技术和方法进行了总结，并指出了本文采用方法仍需改进的地方，对未来研究进行了展望。

第二章 区块链存储压缩相关技术

第一节 比特币数据压缩相关技术

比特币是最早的区块链加密货币之一，其交易数据一直是区块链存储和传输的主要负担之一。为了降低比特币的存储和传输成本，许多研究者提出了各种类型的比特币压缩技术。

这些压缩技术通常基于对比特币交易数据的分析和处理，利用数据的重复性、冗余性和统计特征，从而减少数据的存储和传输量。例如，一些研究者提出了基于哈希函数和字典编码的压缩技术，通过哈希函数将相似的交易数据映射到同一个字典项中，从而减少数据的存储量。还有一些研究者提出了基于压缩算法的技术，如 Lempel-Ziv-Welch（LZW）算法和 Huffman 编码等[4]，可以对比特币交易数据进行压缩，从而减少传输和存储的成本。

除了压缩技术，一些研究者还提出了基于分片和分区的技术，通过将比特币交易数据分割成多个部分，从而减少每个节点需要存储和传输的数据量。这些技术不仅可以提高比特币的可扩展性，还可以提高整个比特币网络的性能和效率。总体而言，比特币压缩技术是区块链存储压缩技术的一个重要分支，具有重要的理论意义和实际价值。未来还需要进一步的研究和探索，以发现更加高效和可靠的比特币压缩技术，从而进一步降低比特币的存储和传输成本。

第二节 以太坊 Rollup 技术发展现状

Rollup 技术是一种 Layer 2 扩容解决方案，可以将大量的以太坊交易数据压缩为一个单独的交易，并将其提交到以太坊主链上，从而大幅度提高以太坊的交易吞吐量和处理能力。目前主要有两种类型的 Rollup 技术：Optimistic Rollup 和 ZK Rollup。[5]

Optimistic Rollup 是一种基于乐观执行的 Rollup 技术，它假设所有交易都是有效的，然后在提交到主链之前进行验证。这种技术可以减少在主链上的交易确认时间和成本，但需要一定的安全性保证。

ZK Rollup 则是一种基于零知识证明的 Rollup 技术，它可以提供更高的安全性保障。ZK Rollup 利用零知识证明来验证提交到主链上的交易数据的正确性，从而减少对主链的依赖，并提供更高的隐私保护。

目前，以太坊 Rollup 技术正在得到广泛的关注和研究。许多开发者和团队正在积极探索和研究 Rollup 技术的实现和应用，以提高以太坊的交易吞吐量和处理能力。同时，也有一些挑战和限制，例如安全性、合约互操作性和用户体验等方面的问题，需要进一步的研究和解决。总体而言，以太坊 Rollup 技术是以太坊扩容的重要解决方案之一，具有重要的理论意义和实际应用价值。

第三节 零知识证明 zk-SNARK 技术发展简介

Zk-SNARK （ Zero-Knowledge Succinct Non-Interactive Argument of Knowledge）是一种零知识证明技术，它可以在不泄露任何信息的情况下证明某个陈述是真实的。它是由计算机科学家 Eli Ben-Sasson 等人在 2014 年提出的[6]，是实现区块链上隐私保护的关键技术之一。

Zk-SNARK 技术通过使用复杂的数学算法来实现零知识证明，该算法基于双线性对和椭圆曲线密码学，可用于验证特定的计算，而无需直接公开其输入和输出。Zk-SNARK 证明非常紧凑，可以压缩大量的计算数据成为一个极小的证明，并且证明的验证可以快速完成，这使得 zk-SNARK 技术在区块链中具有广泛的应用前景。

在区块链中，zk-SNARK 技术可以用于保护用户的隐私，例如匿名交易和身份验证。此外，zk-SNARK 技术也可以用于加密计算和扩容，例如 ZK Rollup 扩容方案，可以通过 zk-SNARK 技术将大量的交易数据压缩成一个小的证明，从而在区块链上实现高吞吐量的交易处理。

尽管 zk-SNARK 技术具有许多潜在的应用场景，但它也有一些挑战和限制，例如高计算成本、对安全性和隐私性的严格要求以及可扩展性等问题。然而，随着技术的不断发展和优化，zk-SNARK 技术将成为未来区块链隐私保护和扩容的关键技术之一。

第四节 本章小结

本章主要进行对区块链压缩技术发展以及现状进行了说明，首先介绍了各种方法的提出思想，为后文中实验操作提供了理论模型；同时也介绍了需要后续需要使用的工具，能够在已有的成果上继续完成、完善实验，使得压缩率以及安全性得到提升，为本文提供方法的参考和借鉴。

第三章 比特币存储数据压缩方法

第一节 研究内容及方法路线

3.1.1 比特币数据结构概述

比特币采用了一种称为UTXO（未使用交易输出）模型的数据结构，它与传统的账户模型有所不同。

在UTXO模型中，比特币的每一笔交易都会生成一组UTXO，表示未被使用的比特币输出。当一个比特币地址收到比特币时，这些比特币将被作为新的UTXO存储在比特币网络中。当用户想要发送比特币时，他们必须选择合适数量的UTXO，并将它们作为输入添加到交易中。在交易完成后，这些UTXO将会被标记为已使用，而新的UTXO将会生成并存储在比特币网络中。

比特币的每个区块都包含一个区块头和一组交易。区块头包含了区块的元信息，例如区块的版本号、前一个区块的哈希值、交易的Merkle树根等。而交易则包含了UTXO的生成和使用信息。比特币的区块链是由一系列区块按照顺序连接而成的，每个区块都包含了前一个区块的哈希值，这种链式结构保证了区块链的不可篡改性。

本地同步获取的数据以二进制形式存储。文件以.dat为后缀，需要进一步细致拆分内容生成可读的文件形式，以便后续操作。

3.1.2 编译存储文件的研究方法路线

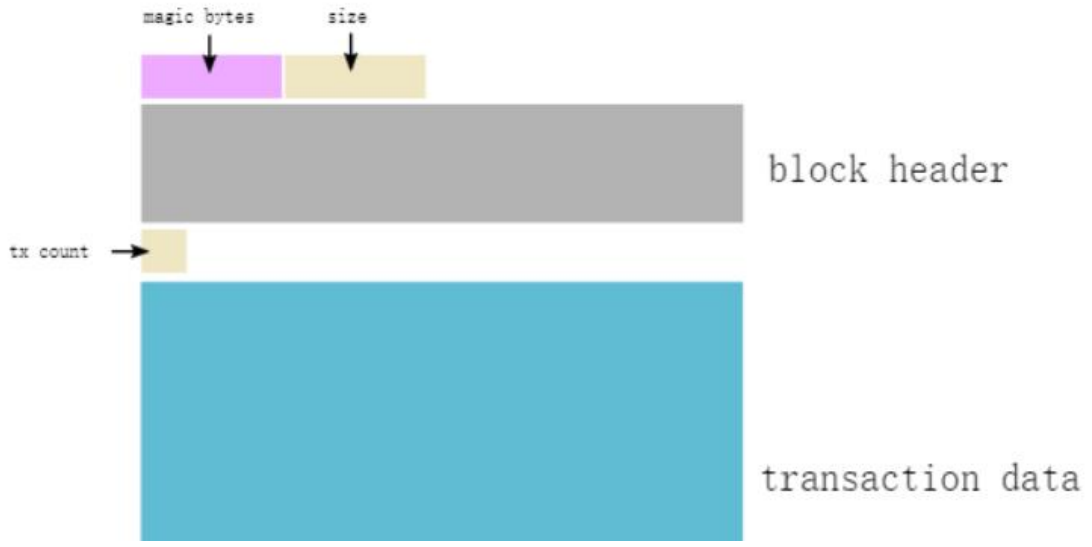


图 2　　比特币区块存储结构

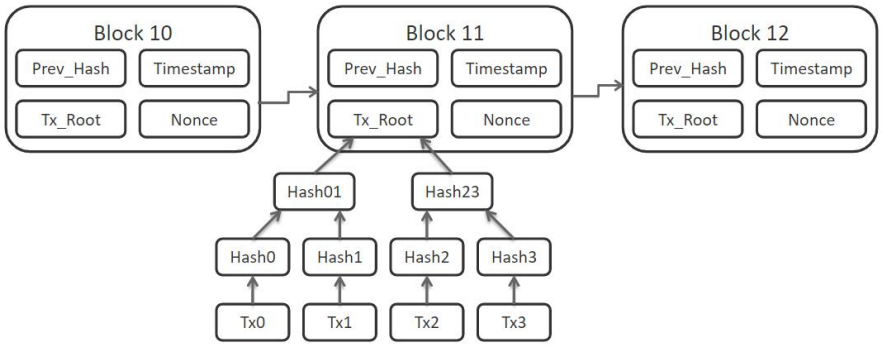


图3　区块链数据结构

本地存储的dat文件通常包括blkxxxxx.dat和revxxxxx.dat等文件，需要具体了解其含义通过编写python脚本进行readable文件形式的生成。

由于在dat文件中包含几千个区块，一个区块分为区块头和区块体，区块体中又包含若干笔交易，而这些数据都是紧密的挨在一起的，因此需要明确每一个byte所包含的具体含义。可以初步将数据分割为五个部分。

1) 魔术字（Magic Byte）：作为区块与区块之间的分割，占4个bytes，通常为”f9beb4d9”。

2) 接下来区块的数据大小：通常为4个bytes，说明该区块的内存大小，采用大端存储的方式，例如0x1d010000，读取时需要先转换0x0000011d，计算出10进制形式285，再加上之前所提到的魔术字8个字节，说明某个区块占293个bytes。

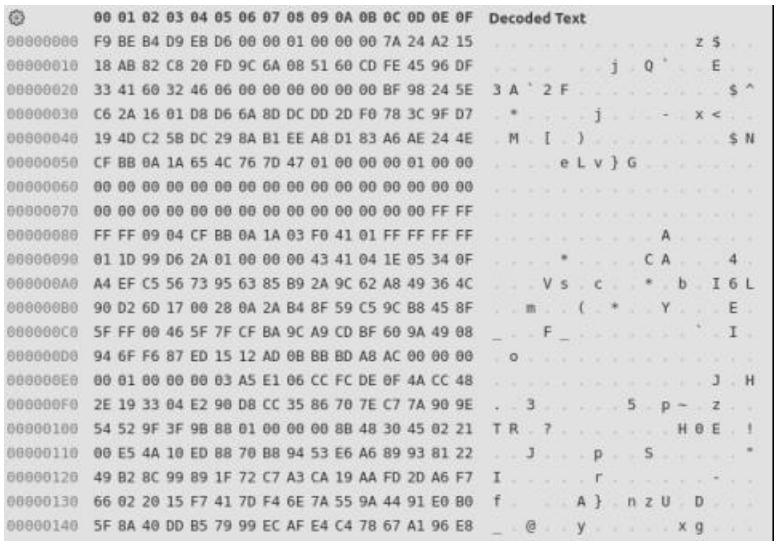
3) 区块头数据：固定80个bytes，其中分为4bytes的版本号，32bytes的上一个区块的哈希值，32bytes的Merkle Tree的根哈希，4bytes的时间戳记录区块产生的近似时间，4bytes的目标难度，4bytes的nonce值。

4) 交易数：该区块内所包含的交易个数。如果最开始的两个bytes小于253则该两位为交易个数，如果等于253则接下来的3位表示交易个数，如果等于254则接下来的5位，如果等于255则为接下来的9位，并且所有格式均为大端存储。

5) 交易数据：包含具体的交易数据，其中分为铸币交易coinbase transaction和普通交易。虽然作用不同但结构大致分为：交易版本号：4个字节；输入交易数量：1-9个字节（变长整数）；输入交易：每个输入交易包含了多个字段；引用的交易哈希：32个字节；引用的输出序号：4个字节；输入脚本长度：1-9个字节（变长整数）；输入脚本：长度不定，最大为10,000字节；序列号：4个字节；输出交易数量：1-9个字节（变长整数）；输出交易：每个输出交易包含了多个

字段；输出比特币数量：8个字节（64位整数）；输出脚本长度：1-9个字节（变长整数）；输出脚本：长度不定，最大为10,000字节；锁定时间：4个字节；最后4个字节是一个区块的校验码，用于确保文件没有损坏。

3.1.3 编译结果展示



	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	Decoded Text
00000000	F9 BE B4 D9 EB D6 00 00 01 00 00 00 7A 24 A2 15	. . . . . z $ . .
00000010	18 AB 82 C8 20 FD 9C 6A 08 51 60 CD FE 45 96 DF	. . . . j Q ` . E . .
00000020	33 41 60 32 46 06 00 00 00 00 00 00 BF 98 24 5E	3 A ` 2 F . . . . . $ ^
00000030	C6 2A 16 01 D8 D6 6A 8D DC DD 2D F0 78 3C 9F D7	. * . . . j . . - x < . .
00000040	19 4D C2 5B DC 29 8A B1 EE AB D1 83 A6 AE 24 4E	. M . [.) . . . . . $ N
00000050	CF BB 0A 1A 65 4C 76 7D 47 01 00 00 00 01 00 00	. . . . e L v } G . . . .
00000060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . . .
00000070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 FF FF	. . . . .
00000080	FF FF 09 04 CF BB 0A 1A 03 F0 41 01 FF FF FF FF	. . . . . A . . . .
00000090	01 1D 99 D6 2A 01 00 00 00 43 41 04 1E 05 34 0F	. . . . * . . . C A . . . 4 .
000000A0	A4 EF C5 56 73 95 63 85 B9 2A 9C 62 A8 49 36 4C	. . . V s . c . . * . b . I 6 L
000000B0	90 D2 6D 17 00 28 0A 2A B4 8F 59 C5 9C B8 45 8F	. . m . . (. * . . Y . . . E .
000000C0	5F FF 00 46 5F 7F CF BA 9C A9 CD BF 60 9A 49 08	_ . . F _ . . . . . ` . I .
000000D0	94 6F F6 87 ED 15 12 AD 0B BB BD A8 AC 00 00 00	. o . . . .
000000E0	00 01 00 00 00 03 A5 E1 06 CC FC DE 0F 4A CC 48	. . . . J . H
000000F0	2E 19 33 04 E2 90 D8 CC 35 86 70 7E C7 7A 90 9E	. . 3 . . . . 5 . p ~ . z . .
00000100	54 52 9F 3F 9B 88 01 00 00 00 8B 48 30 45 02 21	T R . ? . . . . . H 0 E . !
00000110	00 E5 4A 10 ED 88 70 B8 94 53 E6 A6 89 93 81 22	. . J . . p . . S . . . . "
00000120	49 B2 8C 99 89 1F 72 C7 A3 CA 19 AA FD 2D A6 F7	I . . . . . r . . . . - . .
00000130	66 02 20 15 F7 41 7D F4 6E 7A 55 9A 44 91 E0 80	f . . . . A } . n z U . D . .
00000140	5F 8A 40 DD B5 79 99 EC AF E4 C4 78 67 A1 96 E8	_ . @ . . y . . . . x g . . .

图 4　　编译前.dat 二进制数据

```
Magic number = D9B4BEF9
Block size = 0000D6EB
SHA256 hash of the current block hash = 0000000000000473A323BF4F0E08C76DE878B578B7FBD78B3D5C8EC34333C8D5
Version number = 00000001
SHA256 hash of the previous block hash = 00000000000006463260413 3DF9645FECD6051086A9CFD20C882AB1815A2247A
MerkleRoot hash = 83D1A8EEB18A29DC5BC24D19D79F3C78F02DDDDC8D6AD6D801162AC65E2498BF
Time stamp = 4E24AEA6
Difficulty = 1A0ABBCF
Random number = 7D764C65
Transactions count = 71

TX version number = 00000001
Inputs count = 01
TX from hash = 0000000000000000000000000000000000000000000000000000000000000000
N output = FFFFFFFF
Input script = 04CFBB0A1A03F04101
Sequence number = FFFFFFFF
Outputs count = 1
Value = 000000012AD6991D
Output script = 41041E05340FA4EFC55673956385B92A9C62A849364C90D26D1700280A2AB48F59C59CB8458F5FFF00465F7FCFBA9CA9CDBF609A4908946FF687ED1512AD0BBBBDA8AC
Lock time = 00000000
TX hash = D009E339AACA762A142F5BEF6599B9080BE2EA5186F93E8070E594F13C1CFB45

TX version number = 00000001
Inputs count = 03
TX from hash = 889B3F9F52549E907AC77E708635CCD890E20433192E48CC4A0FDEFCCC06E1A5
N output = 00000001
Input script = 483045022100E54A10ED8870B89453E6A6899381224 9B28C99891F72C7A3CA19AAFD2DA6F766022015F7417DF46E7A559A4491E0B05F8A40DDB57999ECAFE4C47867A196E8DCCDC50141045B47E46AD486668774C0
Sequence number = FFFFFFFF
TX from hash = A53017C626729DE3FDB280FA6D35D20F0FDA5ACEBFE4463CAAB484AE1C0888BC
N output = 00000001
Input script = 483045022100B63616B83A7B24EA02AA65DBAEB1C695A157F6BDFC998B08B4004633166ED7DD022074A0F0AC46444E6392E979678FD43A75D6C0137C9BA91F1DFEBEABF59D336C1801410448AFB37529CE0A942AF9
Sequence number = FFFFFFFF
TX from hash = F82271FBB5DA9EAA6516B91693A47E5C44CD89EB79E77847FA992D08539A4CC8
N output = 00000000
Input script = 47304402201F4555342129ABFDA0D5F94E20EB607308C5CAFF46C99901999D37915A45178102204BED5D288DC907CE6CC6963F8FC2C757AEBC10F8017139F63001AAE794AAB4890141046628824FFA455427771875
Sequence number = FFFFFFFF
Outputs count = 2
Value = 0000000000895440
Output script = 76A91478668255C8D3295E77143541116D8AA7963211AC88AC
Value = 0000000053724E00
Output script = 76A9144B31F6A26F1D88F3715996AE3AAA38D99255019088AC
Lock time = 00000000
TX hash = B2C57EA0828E061DDBCDF590828C19EE427CA5DEE061AC9BEF434140E8B2EFBD
```

图 5 编译后包含区块链可读内容的.txt 文件

经过了解了上述的存储方式，通过python脚本中的os和hashlib库对文件进行拆分解读，写入对应的txt文件中，可以大大增强其可读性，为后续筛选压缩做铺垫。

第二节 比特币数据压缩

3.2.1 数据压缩依据

由于比特币采用了UTXO的交易模型，通过交易哈希和解锁脚本规定金额的所属权在UTXO模型中，每个未花费的交易输出都被表示为一个包含金额和加密公钥的元组（tuple）。比特币网络维护一个UTXO数据库，其中存储了所有未花费的交易输出。当一个交易被验证后，它的输出就被添加到UTXO数据库中，成为新的未花费的交易输出。在之后的交易中，这个未花费的交易输出可以被引用（即被花费），但一旦被花费后，它就会被从UTXO数据库中移除。在UTXO模型中，每个未花费的交易输出都被表示为一个包含金额和加密公钥的元组（tuple）。比特币网络维护一个UTXO数据库，其中存储了所有未花费的交易输出。当一个交易被验证后，它的输出就被添加到UTXO数据库中，成为新的未花费的交易输出。在之后的交易中，这个未花费的交易输出可以被引用（即被花费），但一旦被花费后，它就会被从UTXO数据库中移除。[7]

同理对于已经发生过的历史交易，存储的时候可以替换原有的交易数据（金额、加锁解锁脚本、时间戳等），并使用该交易的哈希值进行替代，可以一定程度上减少本地文件存储的压力。如若需要某笔交易具体的信息可以在本地获取哈希之后通过bitcoin explorer等商用比特币服务器上查询。

3.2.2 实现代码解析

文件select_hash.py通过遍历所有可读的txt文件将所有的交易哈希进行记录，并在哈希之后附带该笔交易的输出个数便于之后判断该笔交易是否被消耗完。将结果写入hash_library.txt。

记录交易哈希并附带输出数的代码片段：

```
line = f.readline()
while line :
   if (line.startswith('Outputs count')):
            count = line.lstrip("Outputs count =")
             temp_line = temp_line + count
     if (line.startswith('TX from hash')):
          line = line.lstrip("TX from hash = ")
          H.write(line)
```

再通过遍历所有可读txt将消耗掉的交易哈希进行存储，因为UTXO模型金额来自上一笔交易以及索引，因此可以通过From hash来记录该笔交易的金额来自于哪个上一笔交易，等于被花费。

最后通过将所有hash_library中的哈希存入字典，遍历被花费的文件，每次对hash_library中的输出数进行递减，如果为0则意味着已经消耗完全，可以删除。

对交易哈希进行筛选保留的代码片段：

```
 while line :
        dict[line[0:64]] = line[65:].strip()
        line=L.readline()

line =S.readline()
while line :
        line = line.strip()
        if line.strip() in dict.keys():
                dict[line] = int(dict[line]) - 1
        line=S.readline()
```

第三节 写回原始二进制文件

3.3.1 实现思路

由于比特币原始数据已经整合的非常紧凑，固定大小的数据已经被写死，可变大小的数据只用两个字节便说明了其数据的规模和长度。因此当转换成可读txt文件时会造成很大程度上的冗余，并且各种标签也会增加文件的大小。为了更直观的对比压缩效果，需要将筛选过的交易拆分成dat文件中存储的形式进行比对。并且，由于原始区块中并没有存储每一笔交易的哈希值，而是存储具体信息，每次读取都需要实时进行计算，这对于本文所展示的实验造成一定程度上时间的消耗。

3.3.2 实现代码解析

首先为了验证该方法的可行性以及代码的准确度，将带有指示符的 txt 文件修改为更接近原始数据但是采用 UTF-8 形式编码的 txt 文件，用一对比数据的准确性，并且由于数据容量较大，本章仅以一个文件的操作进行展示。

对可读文件操作，进行检查的代码片段：

```
if tmpHex in line:
        resList.append(Tx_info)
    else:
        Tx_info = ''
    resList.append(''); tmpHex = ''; RawTX = '';Tx_info = '';
```

将结果写入 compressd.txt 进行比对，若某笔交易的哈希后附跟的输出数为 0，则说明该交易已经被消耗完全，将该哈希值进行保留，具体的内容细节进行删除。而如果其不为 0，则对交易的具体内容进行完全的保留。

比对后可以转为二进制形式，此时需要注意数据的存储形式，将计算得出的可读的小端形式转回大端形式进行存储。而若某笔交易被花掉，则在两笔交易之间加入 4 个 bytes 的分隔符 FFFFFFFF 进行分割。若读取到某笔交易的前八位为 FFFFFFFF 则说明之后的 32bytes 存储的为交易哈希，以及该交易已经被花销。否则正常进行解析即可。

实现插入分隔符写回二进制文件的代码片段：

```
L = len(RawTX)
    if tmpHex in line:
        RawTX = int(RawTX, 16)
        resList.append(RawTX.to_bytes(L//2,'big'))
    else:
        resList.append(Partion)
        resList.append(int(tmpHex,16).to_bytes(len(tmpHex)//2,'little'))
    tmpHex = ''; RawTX = '';
```

目前为止，一笔交易信息的最开始 4 个 bytes 为版本号，通常为 00000001（大端存储 01000000），因此 FFFFFFFF 的分隔符可以有效区别 spent 和 valid 的交易信息。

第四节 实验综述

3.4.1 实验工具

主要使用python脚本对存储的本地文件进行编译，其中调用了os库、hashlib库对数据进行读取和操作。编码、解码的过程需要严格遵循比特币本地数据.dat的格式进行读取、分析，并生成多个中间状态从而达到压缩前后文件大小的比对。

3.4.2 实验结果及分析

实验结果采用 dat 文件压缩率（P_{dat}）、txt 文件压缩率（P_{txt}）和中间文件大小之间的对比作为评价指标：

$$P_{dat} = \frac{\text{原始Dat文件大小}}{\text{新dat文件大小}} \tag{1}$$

$$P_{txt} = \frac{\text{解码后的txt文件大小}}{\text{去掉spent交易后的txt文件大小}} \tag{2}$$

所有文件的内存大小都列在下表进行对比，由于在压缩的过程中，每一笔交易的哈希值需要单独读取并进行计算，也就是说在二进制 dat 文件中并没有存储，因此直接进行二进制的压缩需要花费很长的时间，一个 dat 文件大约需要 20 分钟进行运行，后续可以通过多线程、算法的提升进行升级，但本章仅关注于文件本身的内存大小以及压缩后文件与源文件之间的大小比率，因此只压缩一个文件进行示例对比。

表 1 文件名、所含内容、文件大小

Blk00003.dat	一个原始二进制文件	134.2M
Blk00003.txt	一个编译后可读原始区块文件	368.4M
Compressed.dat	一个压缩后的二进制文件	43.1M
Hash_library.txt	记录所有的交易哈希和 output 数	172M
Compressed.txt	由 Blk00003.txt 去除 spent 后的中间文件	69.7M
Retain.txt	截止七个文件时仍未被完全花费的交易哈希	38.8M
Spent_output_hash.txt	所有已经被花费掉的交易哈希	308.9M

表8记录了原始文件、中间所用的文件、压缩后文件的大小，可以通过简单地比对得出压缩的结果。对于二进制文件而言，去掉spent的交易信息而使用Partition字符”FFFFFFFF”和该交易的小端哈希值进行替代后，根据（1）式计算压缩效率P_{dat}≈32.1%，去掉了70%左右的不必要信息；并且对于可读的txt文件（或许会在本地存储，以便使用者进行分析，因为二进制的文件无法直接使用），根据（2）式计算得出压缩率达到P_{txt}≈18.9%，由于编码问题以及压缩之后去掉了一些索引例如每项数据之前的提示，所以压缩率更高。其次可以看出，所有交易只计算哈希值就占用了172M的空间，而且本次实验所选取的区块数据为前8个blk*.dat文件，截止日期2012年7月，也就是说本次实验所使用的数据为比特币创立之后五年内的数据，从交易数的角度来看，压缩率$P_{transaction}$≈22.55%，对于立足于transaction本身的比特币所采用的UTXO机制来说，时刻更新交易状态会更好的提升区块链的性能和节省空间。

第五节 本章小结

本章具体描述了对比特币区块链结构的本地数据存储进行压缩的具体过程和对于二进制存储文件的解读过程。数据量相较于庞大的比特币系统来说不值一提但对于个人设备是一个挑战，这更加凸显出本文所做工作的重要性，如果需要更多的人参与到共识机制的建立中就需要降低其运行、维护的成本，否则又会回到由中心化的核心机构所维护的传统银行模式。实验结果初步来看符合 A.Sforzin，Matteo Maso 等人在《On the Storage Overhead of Proof-of-Work Blockchains》中所得出的结论[8]，即 spent 的交易在现有的区块链系统中占用了大约 70%—80%的资源，符合本文得出的二进制文件压缩率 30%。

第四章 以太坊 Rollup 以及 Zk-SNARK 的应用

第一节 研究内容及方法路线

4.1.1 以太坊 Rollup 的模型研究内容概述

相较比特币以交易为余额凭证的UTXO模型，以太坊采用传统的账户模型体系，个人的address和private key需要额外关注，并且账户中会有余额的产生，而非每一笔交易都需要付款方将所有的金额全部输出。因此对于交易信息的保留与取舍就会一定程度地影响以太坊中的存储空间占用问题。因此产生了Rollup技术，通过维护一棵新MerkleTree的方式，对主链上的存储进行分层，Layer1和Layer2，从而实现主链上空间的释放。[9]

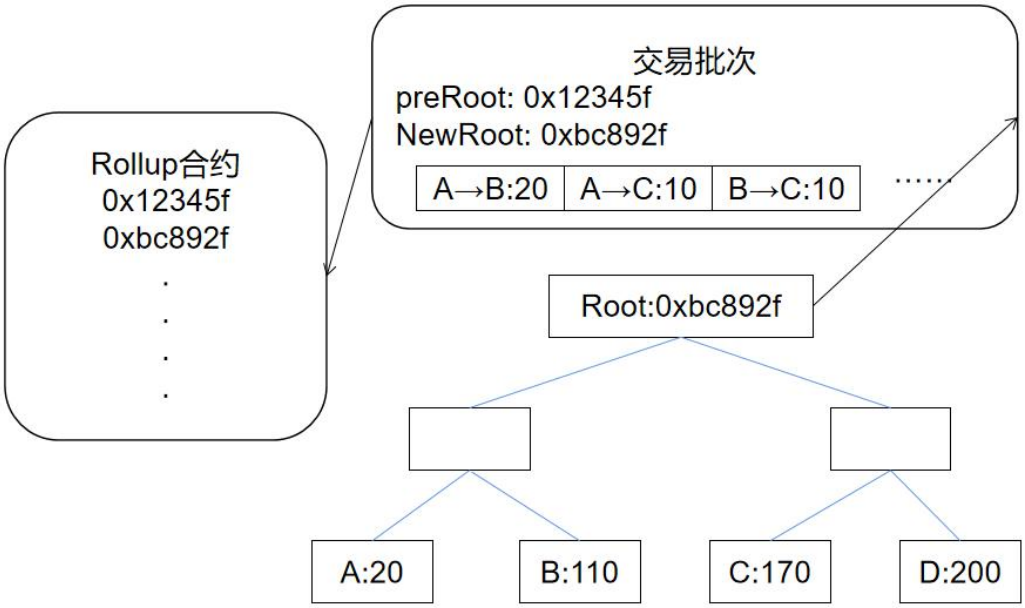


图6 Rollup模型示意图

4.1.2 Rollup 的研究方法路线

本章的研究目标是实现对于若干区块中交易的Rollup，构建用户状态MerkleTree，并对其进行维护。由于以太坊采用传统账户机制，因此一笔交易中需要关注的点即发送方（from），接收方（to），以及交易金额（value）。

以太坊区块数据内容：

{"blockHash":"0x29b29068575d939ed1168dce20a0e3538e00b21a779c9458f2be12d4aacf8736",
"blockNumber":"0x10278fa",
"from":"0x758e8229dd38cf11fa9e7c0d5f790b4ca16b3b16",

"gas":"0x25956",
"gasPrice":"0x7263a5854",
"maxFeePerGas":"0xcf68960f4",
"maxPriorityFeePerGas":"0x0",
"hash":"0x589710668735006888c0ec7fde99985e0a67b5325172a632a2609026d3dc005d",
"input":"0x2b8e6e2a000000000000000000000000000000000000000000000000000000000000000000000000000001
800000000000000000000000000000000000000000000000000000000000000000000000000000200000000000000000000
0000000000000000000000000000000000000000000135a3c5dfc81495e500000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000100000000000
000000000000000000000000000000000000000000000000000000000010278fa0000000000000000000000000000000
000000000000000000000000000000000000000064255a33000000000000000000000000000000000000000000000000
000000000000000000000000000000010000000000000000000000000000000000000000000000000000000000000000000
00000002600000000000000000000000000000000000000000000000000000000000000000000000000000000002800000
000000000000000000000000000000000000000000000000000000000000000000000000000000300000000000000000000
0000c02aaa39b223fe8d0a0e5c4f27ead9083c756cc2c0009c4000000000000000000b29f360726b26
05415abeeceeb8916a5e1aad064000000000000000000000000efc996ce8341cd36c55412b51df5bb
ca429a7617000000000000000000000000000000000000000000000000000000000000000000000000200000
00000000000000000000000000000000000000000000000000000003bf0e0f9b10c6de000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000",
"nonce":"0x4ab5",
"to":"0x000000000005af2ddc1a93a03e9b7014064d3b8d",
"transactionIndex":"0x0",
"value":"0x0",
"type":"0x2",
"accessList":[],
"chainId":"0x1",
"v":"0x1",
"r":"0xb6b3641a102de22f803636c0d5eb13a175aceecfea7d244be124458807ac08a",
"s":"0x510890032487df71d9ce54084818a832ffada1fb6d660afc63fae7a1e4750aad"}

以上为从以太坊第三方服务器QuickNode中调用API函数”GetBlockByNumber”得到的一个区块中的一笔交易具体信息。本章将以一个区块中的所有交易构建用户状态MerkleTree并赋予每个账户初始金额100以太币，并之后对于一笔新交易的到来生成基于libsnark的零知识证明。

4.1.3 代码实现及解析

同样，对于本地存储文件的操作通过编写python脚本实现。创建字典Users保存用户的address，并且对于新地址用户赋予初始值10^{20}wei（1eth=10^{18}wei）。

之后对用户余额和交易金额进行检查，因为如果转账发生需要余额多余转账金额。待所有交易信息传输完成后采用MD5哈希算法将字典中保存的（Address:Deposit）键值对，即用户信息生成32字节的状态哈希值。采用循环Sha256计算，最后生成64字节的Merkle Root。

生成 MerkleRoot 的代码片段

```
Merkle_root = ""
while Merkle_root == "":
    temp_list = []
    temp_str = ""
    if len(User_list)%2 != 0:
        User_list.append(User_list[len(User_list)-1])
    i = 0
    while i   < len(User_list):
        temp_str = User_list[i]+User_list[i+1]
        temp_list.append(sha256(temp_str))
        i = i+2
    User_list = temp_list
    if (len(User_list)==1):
        Merkle_root = User_list[0]
    else:
        continue
print(Merkle_root)
```

第二节 用户状态树维持

4.2.1 对新交易的处理

新交易的到来会触发检查：

1. 交易的金额需小于发起方的余额，包括汽油费的估算。

2. 发起方的签名有效，并且 Nonce 值正确。

3. 发起方减少的金额和接收方增加的金额数值一致。

需要满足上述三点，才能够初步满足一笔交易的有效性，开始对于接收方发送发叶子结点的存储数据进行修改，并生成新 Merkle 状态树以及产生对于该交易的证明。

4.2.2 对用户状态树的修改

如若一笔交易被验证为有效的、合法的，主链上公开的 MerkleTree 会因为一笔交易改变两个叶子结点的哈希值从而整个树进行修改变动。然而证明过程无法兼顾许多叶子节点同时发生变动所产生的结果，也就是说一笔交易可能且只能改变两个叶子结点的状态值和哈希值。所以对一笔交易来说，会先修改发送方的余额，记录下修改的叶子结点的索引（index1）以及金额变动数 δ_1，再修改接收方的金额，计算出新 MerkleRoot 和接收方叶子结点的索引（index2）以及金额变动数 δ_2，便于之后对节点和金额变动生成证明使用。

第三节 通过 Zk-SNARK 实现非交互证明

4.3.1 实现思路

传统的零知识证明使用交互式协议，在证明者和验证者之间进行多轮通信，以使验证者能够逐步确信语句的真实性。

相比传统的零知识证明，zk-SNARK（零知识可扩展非交互性证明）是一种更高效的零知识证明协议。Zk-SNARK 允许证明者生成一个包含证明的短证书，并将其发送给验证者，而不需要进行多轮通信。这使得 zk-SNARK 在某些应用中具有更高的效率和性能。

4.3.2 libsnark 工具的使用

鉴于本文只是通过zk-SNARK实现证明来达到压缩空间占用的目的，而并非需要针对证明进行优化和解析，因此本文采用libsnark直接生成证明并在该库中实现对于证明的验证。libsnark是一个流行的用于实现zk-SNARK的C++库。下面是使用libsnark来实现zk-SNARK生成证明的一般步骤：[10]

1. 定义一个回路（circuit），该回路可以表示你要证明的语句或程序的计算过程。

2. 通过Gadget将回路映射为一组约束系统（constraint system）。

3. 生成一对公私钥对（keypair），其中公钥将用于验证证明，私钥将用于生成证明。

4. 将计算输入传递给生成证明的程序，生成证明并输出。

5. 验证程序使用公钥和输入来验证证明，如果验证成功，则证明通过。

其中对于libsnark使用最重要的理解是将所有问题转换成求解多项式的问题，通过将多项式“铺平”的方法，进行例如不需要知道某个方程解的情况下验证该方程是正确的。虽然一笔交易会生成三个证明，但是主链上再不需要存储区块数据、交易数据等无限增长的数据，只需要通过维持一棵MerkleTree保存其Root值，64Bytes和一笔交易所需要的证明即可。长远来看对于主链上资源的节约是可观的。

第四节 用户状态 MerkleTree 的路径证明

4.4.1 实现思路

根据 4.2.2 节提到的用户状态变化值，本文需要生成三个证明，分别需要证明接收方变动之后的哈希值在用户状态树中，发送方变动之后的哈希值在用户状态树中，以及二者金额变动相等。同时由于主链为公开视角，用户状态的哈希值无法从主链上进行直接获取，是隐私的一部分，同时金额以及金额的变动也属于隐私的范畴，无法只通过获取主链上的 MerkleRoot 获取。

4.4.2 实现代码解析

MerkleTree 路径验证生成的代码片段：

```
std::vector<std::vector<libff::bit_vector>> levels(tree_depth);
        //level 2 leaves left most --> right most
    int leaf_count = std::pow(2, tree_depth);
        for (int i = 0; i < leaf_count; i++) {
            libff::bit_vector tmp = hash256<HashT>(argv[i+2]);
            //std::cout << *binToHex<HashT>(tmp) << std::endl;
            levels[tree_depth - 1].push_back(tmp);
        }

    calcAllLevels<HashT>(levels, tree_depth-1);
        libff::bit_vector input = levels[0][0];
        input.insert(input.end(), levels[0][1].begin(), levels[0][1].end());
        root = HashT::get_hash(input);

        address = std::stoi(argv[10]);
        leaf = levels[tree_depth-1][address];
        std::cout << address << std::endl;
        int addr = address;
```

```
for (int i = 0; i < tree_depth; i++) {
    int tmp = (addr & 0x01);
    address_bits[i] = tmp;
    addr = addr / 2;
    std::cout << address_bits[tree_depth-1-i] << std::endl;
}

//Fill in the path
size_t index = address;
for (int i = tree_depth - 1; i >= 0; i--) {
    path[i] = address_bits[tree_depth-1-i] == 0 ? levels[i][index+1] : levels[i][index-1];
    index = index / 2;
}
std::cout << "root is " << *binToHex<HashT>(root) << std::endl;
```

MerkleRoot 证明主要依赖 merkle_authentication_path_variable 和 merkle_tree_check_read_gadget这两个gadget依赖，其中第一个gadget为merkletree提供了一条路径，而第二个从名字来看便知道是检查给定的一个叶子结点看能否计算出正确的MerkleRoot。该电路有两个接口函数generate_r1cs_constraints和generate_r1cs_witness，前者用来生成R1CS依赖，后者用来给所有的电路赋值。该证明中公开信息为MerkleRoot的bit数，验证过程需要Root哈希值和生成的证明文件prove和验证文件verify_key。

第五节 金额变动数值证明

4.5.1 实现思路

根据 4.4.2 节提到的接收者和发送者金额变动的大小必须相等的约束条件，需要生成一个能验证两个数相等的证明，同样采取 R1CS 的约束方式。同 4.4 节中使用 libsnark 进行电路验证的方式，本节也采用多项式验证的方式生成电路、证明文件进行证明、验证。

4.5.2 实现代码解析

实现金额变动相等证明的代码片段：

```
// Create protoboard
protoboard<FieldT> pb;

pb_variable<FieldT> x, y, is_equal;

x.allocate(pb, "x");
```

```
    y.allocate(pb, "y");
    is_equal.allocate(pb, "is_equal");

    // Add R1CS constraints
    pb.add_r1cs_constraint(r1cs_constraint<FieldT>(1, x-y, 0), "x = y");

    const r1cs_constraint_system<FieldT> constraint_system = pb.get_constraint_system();

    // Generate keypair
    const r1cs_gg_ppzksnark_keypair<libff::default_ec_pp> keypair =
r1cs_gg_ppzksnark_generator<libff::default_ec_pp>(constraint_system)
```

输入的变量赋值在pb上的x和y形成多项式x-y，与1相乘。给变量is_equal赋值0，计算为R1CS的多项式约束条件1 × (x-y)==0。通过R1CS_gg_ppzksnark_keypair生成证明的prove_key和验证的verify_key，添加witness即需要证明的x和y值（需要提前写死在电路生成代码中）。再利用R1CS_gg_ppzksnark_prover直接生成证明，将两个key和prove进行保存，需要验证时可以使用r1cs_gg_ppzksnark_verifier_strong_IC函数引入验证钥verify_key，证明prove和公开输入即可对该证明进行验证。

第六节 实验综述

4.6.1 实验工具

本文主要采用 python 脚本和 libsnark 进行压缩模型的实现。其中 python 脚本用于处理同步到的以太坊区块信息，提取需要构建用户状态 MerkleTree 的信息进行整合，并且由于无法从头完整的构建用户状态因此只能选取中间的区块进行赋初值操作。之后采用 libsnark 库文件生成对于 merkletree 维护的 zk-SNARK 证明，该工具主要用于非交互式零知识证明，通过简短的电路证明高效地验证信息正确性。

4.6.2 实验结果及分析

实验结果采用前后文件占用空间大小的对比进行说明，由于数据结构发生了变化，并且随着交易数量的增加，只有证明数会与之匹配增加，因此本文仅对前后需要在主链上新存储的文件大小和原始数据进行比例计算得出压缩比例，体现压缩的效果。

$$\varepsilon_{ratio} = \frac{\text{新结构存储大小}}{\text{原区块数据大小}} \tag{3}$$

表 2　以太坊区块数据大小、证明文件大小

Block_all_information.txt	包含所有信息的一个区块文件	185Kb
Users_deposit.txt	整合出的记录用户地址和余额的列表文件	17Kb
Merkle_prove.raw	MerkleTree 节点证明	137Bytes
Merkle_vk.raw	MerkleTree 证明 verify_key	11Kb
Balance_prove.raw	验证两数相等证明	137Bytes
Balance_vk.raw	两数相等证明的 verify_key	1.1Kb
Merkle_Root	用户状态 MerkleRoot	32Bytes

可以看出上述文件的内存占用差距是非常大的。而本文的实验对象仅仅为一个 Block 中的交易数（大约为 230 笔），每个 Block 中的交易数据有所不同。而主链上便从此不需要再将所有的交易信息存储下来形成共识链，而可以以维护一个 32Bytes 的 MerkleRoot 的方式并不断地附加证明的方式，通过 zk-SNARK 证明区块上节点的有效性和交易的有效性，即某个叶子结点发生变化之后处于 MerkleTree 中，并且某笔交易造成的金额变动是相等的。但是由于现阶段 libsnark 库文件研究时间和整个模型的不够熟悉，只能做到一次对一笔交易进行证明生成和验证。由于证明数会随着交易数增加而增加，统计该区块新结构所需结构为 104Kb，和原始的一个区块交易记录的大小 185Kb 内存形成对比，由（3）式计算压缩率 $\zeta \approx 56.3\%$。

并且长期来看，相较于冗余的区块数据和交易细节记录，基于账户模型的以太坊更倾向于直接对于账户余额的操作，而区块链仅仅是对交易有效性的一种辅佐证明作用。在主链上，所有用户都需要进行访问的空间内，无需所有的信息，因为对于交易细节的深挖和探索可以转接到 Layer2 架构中，并且绝大多数的使

用者并不需要知道内部的细节，而是关注和自己相关的账号信息、交易内容以及能亲自验证此时区块链的有效性和合法性即可。除此之外基于以太坊的 Rollup 技术也会带来新的影响：

1. 更高的交易处理能力：以太坊 Rollup 将多个交易打包在一个区块中，并通过主链上的证明来验证交易的正确性，从而可以实现更高的交易处理能力。这可以使以太坊支持更多的交易，从而提高了其应用场景和用户数量。

2. 更低的交易成本：以太坊 Rollup 将多个交易打包在一个区块中，可以减少区块链上的交易数量，从而降低了交易成本。此外，Rollup 还可以实现更高的交易速度和更短的交易确认时间，进一步降低了交易成本。

3. 更高的安全性：以太坊 Rollup 通过将验证过程转移到以太坊主链之外，可以减少对主链的负担，从而提高了以太坊的安全性。此外，Rollup 还可以采用零知识证明等技术来保护用户隐私和安全。

第七节 本章小结

本章具体描述了 zk-SNARK 实现零知识证明的研究工作。根据新交易到来会对原状态树改变的内容提供了证明的方案和实验实现。由于技术和时间的限制，本文仅仅实现了对一笔交易的到来进行处理生成证明的过程，而以太坊中每天需要处理上千上万笔交易，因此短期内来看该方法还不足以改变以太坊存储的规则和实现扩容的目的。但是这种想法对于最初按照区块的生成、交易的验证等方案的提出是具有跨越式的提升的。

第五章 总结及展望

第一节 总结

当今时代背景下，加密货币逐渐兴起并引发了广泛的关注，其一是这种技术自身具有的稀有性保证了其内在价值，其二是背靠互联网技术的发展，其安全性的约束也需要被人们广泛研究和重视。同样的，会有越来越多的使用者和研究者参与到这一项技术的研究中，因此其产生的数据量只会与日剧增。距第一块比特币区块被挖出已经过去了25年，比特币所有区块的数据量已经达到了447GB（截止2023年5月），如果一个新节点建立需要同步接近一周的时间；而以太坊因为智能合约的部署，只算线下全区块存储的数据量已经需要658GB（截止2023年1月）的内存，线上同步时间无法预估，同时由于各种交易需要计算汽油费、打包形成区块，这就导致主链上带宽每时每刻都处于拥堵的状态。而根据本文的压缩效率计算，如下表所示，会对存储空间产生一定的优化。

表3　压缩前后内存对比

	压缩前	压缩后
比特币	447GB	138GB
以太坊	658GB	12Kb+n×0.4KB

因此数据压缩、降低能耗的工作必不可少。

第二节 展望

正如前文中所提到的，未来对于比特币数据处理的工作或许由有更多更精妙的数据结构来实现，能够在检具信息含量的情况下尽量降低空间消耗；而以太坊的 Rollup 技术正是为这种基于账户模型的交易策略提供了一种思考模式。通过构建用户状态 MerkleTree 的方式，维护 32bytes 的根节点即可正名所有数据的有效性。但同时，其交易的安全性就会收到损坏和质疑，zk-SNARK 的使用需要能够同步处理多笔交易对 MerkleTree 产生的影响，通过减少证明数来进一步减少带宽消耗。伴随未来叶子结点会越来越多，对于 MerkleTree 证明的升级问题急需解决。

参考文献

[1] 袁勇，王飞跃. 区块链技术发展现状与展望，《自动化学报》，2016 年 4 月，第 42 卷第 4 期

[2] 孙知信，张鑫，相峰，陈露. 区块链存储可扩展性研究进展，《软件学报》，2020-7-27

[3] 喻辉，张宗洋，刘建伟. 比特币区块链扩容技术研究，《计算机研究与发展》，2017 年, 54(10)：2390-2403

[4] Chang Honglin, Yuan Xie, W. Wolf. LZW-based code compression for VIE W embedded systems, i-eee，2005, (ASAP'05)

[5] Thomas Lavaur, Jerome Lacan, Caroline P. C. Chanel. Enabling Blockchai n Services for IoE with Zk-Rollups. Sensors 2022, 22(17), 6493. 2022.8.29

[6] Hartwig Mayer. Zk-SNARK explained: Basic Principles. CoinFabrik. 2016.1 2.13

[7] Shuhao Jiang, Jiajun Li, Shijun Gong, Junchao Yan, Guihai Yan, Yi Sun, Xiaowei Li. BZIP: A compact data memory system for UTXO-based blockchians. Journal of System Architecture. 2020.10, Volume 109, 101809.

[8] Alessandro Sforzin, Matteo Maso, Claudio Soriente, Ghassan Karame. On the Storage Overhead of Proof-of-Work Blockchain. 2020 IEEE International Conference on Blockchain.

[9] Vitalik Buterin. An incomplete guide to Rollups. 个人博客主页。2021.1.5. https://vitalik.ca/general/2021/01/05/rollup.html

[10] StarLI-Trapdoor. Example to create merkle path proof using libsnark. 2020. 7.27. https://github.com/StarLI-Trapdoor/libsnark_sample

致　谢

在本次的论文撰写过程中，我得到了苏明老师的细致帮助与指导。从论文的选题开始直到论文的定稿与答辩的过程中，老师都耐心地解决我提出来的疑问，给予我修改与完善论文的方向与意见，使我对于学术论文写作的过程与规范有了清晰的认识。在此，我想对苏明老师的耐心帮助与指导表达由衷地感谢。

大学的生活即将过去，在这几年的专业学习中，我从一堂堂悉心准备的精致课程和一次次思想碰撞的小组协作中受益匪浅，网安学院知识渊博、兢兢业业的老师们与积极进取、热情开朗的同学们为我的大学生涯画卷添上了最浓墨重彩的一笔。在此，我想对南开大学网安学院的老师和同学们表达由衷地感谢。

回首开心惬意的课下时光，那一个个记忆里欢乐美好的场景总少不了志同道合的朋友们和包容友好的 407 室友们的身影，他们在我最需要帮助时伸出自己的援手，他们在我最迷茫时为我提供前进的动力。在次，我想对我的朋友们和室友们表达由衷地感谢。

最后，这些年的大学生活离不开父母的物质支持与亲人们的精神鼓励，父母亲朋长辈们在以无微不至的温暖关怀为我塑造了温馨的童年生活后，又以丰富的人生经验与阅历为我提供未来道路的参考。在此，我想对我的亲人和父母们表达由衷地感谢。