学校代码： 10246

复旦大学

硕 士 学 位 论 文

（专业学位）

基于安全多方计算的区块链密钥管理研究

Research on Blockchain Key Management Based on Secure Multi-party Computation

编　　号： 17210240258

专业学位类别（领域）： 计算机技术

院　　系： 计算机科学技术学院

完 成 日 期： 2019 年 10 月 8 日

目录

摘要

自2008年比特币问世以来，比特币的底层技术-区块链技术不断的发展壮大。区块链不再是仅仅只像比特币一样作为一种价值承载的加密货币。在区块链基础架构上引入图灵完备的虚拟机后，区块链成为了一个理论上可以承载所有事务的状态机。不管是公有链还是许可链，至今已有众多行业的业务信息在区块链上记录着，包括公益、防伪溯源、医疗信息、供应链金融、司法存证、版权、数字身份等等领域和应用。

区块链上众多的业务承载着几千亿的价值，而区块链的安全性取决于公钥密码体系的安全性。从 2009 年比特币上线至今，发生了众多区块链资产被盗或者密钥丢失无法找回的事件。黑客窃取用户的区块链账户私钥，将用户的上万个比特币全部转走；交易所热密钥泄漏，导致交易所损失千万级的资产；用户因丢失了写有私钥的纸钱包失去了账户的控制权等等因密钥丢失或者泄漏的事件层出不穷。为了防止密钥泄漏和丢失发生，工业界和学术界探讨了多种密钥管理的方式，使用助记词或关键词替代密钥；开发热钱包和冷钱包；使用硬件设施存储密钥；使用硬件设施进行签名；使用第三方托管服务存储密钥等方式。但是现有的密钥管理方式要么硬件设施成本太高不易携带，冷热钱包开发商良莠不齐可能存在漏洞和后门，第三方托管机构的可依赖性也需要打个问号。所以本文提出一种基于安全多方计算的安全易用的密钥管理方法，以期促进区块链技术的进一步发展。

现存的密钥管理方法存在以下三个问题：密钥的权力不够分散，单点泄漏就导致整个账户失控；密钥不能刷新，不能提供动态安全性；密钥太长且无规律，不易记忆。针对上述这三个问题，本文基于安全多方计算的 ECDSA 签名算法，将区块链账号的权力分散在各个密钥份额上，签名时需要多方共同参与；并且提供密钥刷新的能力，在部分密钥份额泄漏时，更新密钥份额达到密钥管理的动态安全性；同时针对难记忆的问题，设计了一种基于集成加密技术的方法使得用户可以使用更为熟悉的密码口令管理密钥，且提供可配置的安全性选择。基于上述的算法，设计实现了一个基于安全多方计算的密钥管理系统，以提供安全易用的区块链密钥管理方式。

关键词：密钥管理，区块链，安全多方计算

中图分类号：TP391

Abstract

Since the advent of Bitcoin in 2008, Bitcoin's underlying technology blockchain technology has continued to grow, the blockchain is no longer a cryptocurrency that is only a value bearer like Bitcoin. After the introduction of Turing's complete virtual machine on the blockchain infrastructure, blockchain becomes a state machine that can theoretically carry all kinds of transactions. Whether it is the public chain or the Permissionless chain, business information of many industries including public welfare, anti-counterfeiting, medical information, supply chain finance, judicial deposit, copyright, digital identity and other fields and applications has been recorded in the blockchain.

The numerous services in the blockchain carry hundreds of billions of dollars, and the security of the blockchain depends on the public key cryptosystem. Since the launch of Bitcoin in 2009, there have been many incidents in which blockchain assets have been stolen or lost. Hackers steal the user's blockchain account private key and transfer all of the user's thousands of bitcoins; Exchange's hot key leaks, causing the exchange to lose tens of millions of assets; users lose the paper with the private key written, account can no longer open on the chain and user lose control of the account due to the loss of keys. In order to prevent key leakage and loss, the industry and academia have explored a variety of key management methods, using mnemonics or keywords instead of keys; developing hot wallets and cold wallets; using hardware facilities to store keys; using hardware facilities Signature; using a third-party hosted service to store keys. However, the existing key management method is that the hardware facilities are too expensive to carry, and there may be backdoors in the hot and cold wallet. The dependency of the third-party hosting organization also needs to be questioned. Therefore, this paper proposes a secure and easy-to-use key management method based on secure multi-party computation to promote the further development of blockchain technology.

The existing key management methods have the following three problems: the power of the key is not scattered enough, and the leakage can causes the entire account to be out of control; they cannot provide dynamic security because keys cannot be refreshed; the key is too long and irregular to remember. In view of the above three problems, this paper is based on the secure multi-party computation of ECDSA signature algorithm, which distributes the power of the blockchain account to many key

shares. Multiple parties need to participate together When signing; and the ability to provide key refresh is partially dense. When the key shares leak, the updated key shares reach the dynamic security of key management. At the same time, for the problem of hard to remember the key, an integrated encryption-based method is designed to enable users to manage keys with more familiar passwords which provide configurable security options. Based on the above algorithm, a key management system is designed to provide a secure and easy-to-use blockchain key management.

keywords: key management, blockchain, multi-party computation
CLCN: TP391

第一章 绪论

1.1 研究背景与研究意义

随着区块链技术的飞速发展，生产生活中使用该技术的新场景和新业务越来越多。2008 年中本聪发布了比特币的白皮书 -- 一种基于分布式网络基础和一致性共识算法的加密货币实现[1]，2009 年比特币主网正式发布上线，比特币的去中心化和分布式的特性得益于其底层的区块链技术。但是比特币的设计机制决定其支持的功能很少，采用交易脚本将其自身的能力限制在只能承载加密货币的价值。2014 年 Vitalik Buterin 在比特币会议上宣布了以太坊项目并详细介绍了以太坊的底层原理和架构设计[2]，大大扩展了区块链技术的能力。以太坊上设计实现了支持图灵完备的智能合约语言的虚拟机 EVM（Ethereum Virtual Machine），可以作为一个“世界状态机”的状态转移部件执行运行在以太坊上面的智能合约(Smart Contract)，并通过共识算法确保区块链上世界状态账户数据的一致性。根据《全球公链项目技术评估与分析蓝皮书》数据显示，2017 年到 2018 年底至少有 2 万条公链出现，还有众多更适用于企业项目场景的联盟链，直到现在，全球众多区块链上运行着数十万数量级的去中心化应用 DAPP（Decentralized Application），涉及金融、物流、供应链、存证、医疗等等众多领域的应用[3][4][5]，承载着上万亿的价值。

从比特币系统开始，到目前为止几乎所有的区块链系统都是基于公钥密码体系中的椭圆曲线密码学（Elliptic Curve Cryptography，ECC）来保证安全性的。1976 年，美国斯坦福大学的 Diffie 和 Hellman 共同发表了论文：密码学的新方向[6]，提出公钥密码学体制（Public Key Infrastructure， PKI），不仅加密算法本身可以公开，甚至加密用的密钥也可以公开。区块链账户地址与一个或者多个密码学公钥绑定，对应着一个或者多个私钥，拥有公钥对应的私钥也就掌握了区块链账户的所有权。区块链上的所有交易都需要通过私钥进行签名，共识节点验证签名通过才会将交易打包到区块中再经过共识过程上链更新世界状态的数据。每个账户对应着私钥，私钥就是身份的证明。用户必须保管好私钥不泄漏不被窃取且不能忘记，使用最广泛的椭圆曲线密码学的私钥一般是长度为 256 比特的随机字符串，相对应的是 32 位的十六进制字符串，而用户直接记忆 32 位长度的随机字符串是很困难的。丢失私钥或者私钥被盗都会造成很严重的后果。2015 年，Bitstamp 多个转账钱包遭到攻击，导致 19,000 比特币丢失，由于该公司一名员工无意中下载了一个恶意文件，使得攻击者可以直接访问包含 wallet.dat 文件以及公司热钱包的密码；2018 年近千万 EOS 被盗事件，黑客通过盗取受害人的私钥

而盗走了其价值近千万的 EOS；2019 年 QuadrigaCX 创始人意外死亡使得私钥丢失，从而导致 1.9 亿美元的资产无法找回等等。有统计数据表明，因丢失私钥导致区块链账户资产无法被使用的价值达到了数百亿美元。因私钥泄漏导致区块链账户的资产被窃取盗用、转移的价值也达到了数百亿美元的规模。

本文认为，出现私钥丢失和私钥泄漏的主要原因如下：

1、区块链账户私钥难以记忆：区块链用户为了方便日常生活生产使用，要么将私钥明文直接记录备份在连接了网络的移动设备上（如手机，个人电脑），要么使用密码通过第三方托管机构将私钥直接托管存储在外部服务器上。在以上两种情况下因为存储设备的安全问题很容易被攻击者窃取，甚至被不可信的第三方托管机构窃取盗用，攻击者将可以轻松访问由该私钥控制的区块链账户资产并窃取转移。

2、密钥备份机制不完善：为了防止出现密钥在网络上被窃取或者需要用户完全信任第三方服务商等问题。用户使用纸钱包的方式记录密钥信息，将密钥原文记录在个人的纸张（纸钱包）或者硬件设备（UKey 等）上，当需要密钥时再手动输入或者将硬件设备插入到签名设备中对交易进行签名。线下备份的问题是纸张和硬件设备容易污损、破坏和丢失，备份的介质一旦遇到物理破坏就无法使用，导致密钥丢失区块链资产无法找回，因为重新生成相同的私钥在计算上是不可行的。

区块链技术有信任机器、可追溯、防篡改和去中心化等等特点，已经在众多领域展示了它的使用价值，但是让区块链技术进一步赋能实体应用到更多的业务场景中还有许多问题，比如交易处理性能 TPS、可扩展性、可维护性、落地成本等。除此之外，还有一个限制着将区块链技术使用到更多领域的问题：密钥管理方法不够友好，普通用户接入成本和门槛较高。因为区块链是去中心化的组织，在密钥丢失时没有中心机构可以像传统的互联网应用一样通过手机或者邮箱认证找回或者修改密码。本文的主要目的是提供一种安全方便的区块链密钥管理方式，其中的关键问题是解决区块链密钥的安全性和易用性，建立一个完整可靠的密钥管理系统，包括密钥的备份和密钥的使用方式。从而降低用户使用区块链时的成本，对区块链技术的落地和发展起到推动作用。

1.2 国内外研究现状

现代密码学主要分为对称密码学和公钥密码学（也称非对称密码）两个体系，如果要保证被加密信息或者数字签名的安全性，两个密码体系都要求保管好加解密的私钥，一旦私钥丢失，信息和资产就会有泄漏和被盗用的风险。所以区块链系统需要一个完善的密钥管理方法来保证密钥的安全，也即保证了区块链账号的

安全。密钥的管理包含以下两个方面：

1、密钥存储：私钥只有在需要解密和签名的时候使用到，所以要将私钥存储在安全的环境中。用来存储密钥的环境要有足够的安全措施保证存储的信息不丢失且不泄漏[7]，使用时需通过特定的方法获取。

2、密钥使用：密钥的使用与密码体制有关，比如多重签名方案[8]需要多个私钥一起使用对消息进行签名才能生效、门限签名[9]要求有足够多的密钥份额才能构造出完整有效的签名。密码体系不同，使用密钥时的计算方式和交互也不一样。

目前存在的对区块链密钥的使用和管理，主要有如下 6 种方式：

（1）本地存储直接使用：用户直接将密钥存放在本地文件或者本地能访问的云服务中，在使用时直接获取密钥并输入到密码算法中；或者用户先加密密钥信息再存储下来，在需要时先解密获取得到密钥原文再进一步使用。该方法在使用时需要有完整的密钥才能够进行签名的过程，但是用户的个人设备的环境安全是无法保证，个人设备很可能因为个人的使用原因存在木马或者病毒，或者在使用过程中被攻击者通过边信道攻击监听获取到密钥的信息。该方法一旦密钥泄漏，整个区块链账户的所有权就会完全暴露给攻击者。

（2）助记词和分层确定性钱包：用户在生成区块链账户时，借助于一定数量的相对好记忆的单词或者诗句来计算出私钥和公钥，比如利用 11 个助记词通过密钥派生方法（Key Derivation Function，KDF）来生成私钥，进而用该私钥对交易哈希签名。或者是使用分层确定性钱包的方法，通过私钥种子来衍生出多个子私钥[10]，在使用时根据衍生规则生成私钥，再利用该私钥对交易签名。上述两种方法在一定程度上解决了密钥难记忆的问题，但是助记词的数量不能太少，太少容易被暴力破解而数量太多则又带来难以全部记住的问题；同样的在使用分层确定性的方法中，最关键的是密钥种子，而如何安全方便的管理好该密钥种子仍然是一个亟需解决的问题。

（3）多重签名:多重数字签名的概念首先被 Okamoto 和 Itakura 等人发表提出并同时设计出了一个具体的算法方案[8]，多重签名可以将一个账户的权力分散在多个密钥上，防止一个密钥丢失导致整个账户失控的情况。一个账户的资产操作交易需要多个相对独立的私钥对一个消息明文进行签名，当签名数量足够多时交易才生效。因为在区块链系统的账户体系中直接支持多重签名算法需要更改整个区块链的架构设计，所以该方法的具体实现多用于智能合约中，事先规定好签名的账号和需要的签名数量阈值。那么多重签名方法的安全性就取决于智能合约的设计和实现，而且灵活性不高，因为智能合约一旦部署之后就很难更改。多重签名方法在一定的场景下能发挥其安全性，但是较低的灵活性很难得到广泛的

使用。

（4）中心化托管服务：将密钥存储在相对可信的中心化机构。用户需要对消息进行解密或者签名的时候，将消息原文发送给托管服务方，服务方根据用户的需求处理结束后将结果返回给用户。这种方案用户不用自己保存密钥，即使忘记密码也可以通过托管的中心化服务找回，没有丢失密钥的风险。但是托管机构是一个中心化的服务，有可能受到传统的拒绝服务攻击而无法为用户提供服务，还可能因为服务安全措施不够完善导致服务系统被入侵，甚至可能因为社会工程的问题托管服务方内部监守自盗或者被攻击者获取到用户的密钥。完全的将密钥托付给中心化机构保管会有很大的风险。

（5）基于安全多方计算的签名：姚期智院士在 80 年代提出了 GC(Garbled circuit)与 OT(Oblivious transfer) 相结合的安全多方计算(Multi-Party Computation)的理论框架，描述为“一组互不信任的参与方之间在保护隐私信息以及没有可信第三方的前提下的协同计算问题”[11]。在无可信第三方的情况下，各方不需透露己方信息利用密码学方法得到多方数据整合在一起共同计算的结果。密钥管理中，多方分别持有私钥的份额，私钥份额只有自己持有，在计算时利用 GC + OT 或者同态加密的方法将各自的签名份额进行整合得到最终的完整签名。2017 年，以色列巴伊兰大学的 Lindell 用两方计算的方法实现 ECDSA 签名[12]，还有科学界和业界的科学家们在进一步实现安全多方计算的门限签名[13][14]。

（6）基于安全多方计算的门限签名：90 年代，门限密码学不断被提出，门限签名方案被认为是密钥管理的一种重要途径[9]。门限密码学存在着很多成熟的方案，包括 RSA[15]，ElGamal[16]和 Schnorr[17]签名等。虽然 DSA 和 ECDSA 是一种广泛使用的标准，但是 DSA 和 ECDSA 都很难以构建有效的阈值签名协议。如[12]中所讨论的这是由于需要在不知道 k 的情况下计算 k 和 k^{-1} ，这是很难构造出来的。2018 年 Lindell 和 Nof 提出了第一个真正实用的全门限 ECDSA 签名协议，它具有相对快速的签名和密钥分发的特点。解决了上述长达数年的开放性问题，并为现在急需的门限 ECDSA 签名的实际应用打开了大门[13]。但是门限 DSA 签名的效率低下，在密钥分发和签名阶段都需要进行多轮的信息交互和零知识证明，有待进一步的研究和发展。

综合以上的对密钥管理的进展研究，总结出目前区块链密钥管理存在的三个问题：

1、密钥权力不够分散：当将一个完整的密钥直接存储时，就潜在着整个完整密钥直接泄漏的风险，完整密钥直接泄漏将导致丢失账户的所有权。本地存储直接使用和中心化托管服务的方式都会遇到这个问题。针对该问题，可以利用秘密共享[18]或者多方计算的思想将密钥的权力分散，在生成密钥时就将密钥份额

分散在多个参与方手中，当少数参与方的密钥份额泄漏时也不会直接影响账户的安全，保证不会出现单点泄漏的问题。

2、不能刷新密钥：多重签名和目前发表的基于安全多方计算的签名方法中的密钥在生成后都无法进行便捷的刷新，这种情况下攻击者可以一个一个的攻击破解获取每一个密钥份额，最终获取到所有的份额就可以完全掌控该区块链账户的资产。如果存在安全便捷的密钥刷新方法，在部分密钥泄漏时可以将密钥份额进行刷新，已泄漏的旧的密钥份额则不再有实际作用。

3、易用性差：用户还是需要记忆很复杂的助记词或者密钥字符串，对于终端用户而言体验很差，这是阻碍区块链大规模发展的一个重要原因。为了让用户更容易的使用区块链系统和应用，使用方便记忆的密码口令来保护区块链密钥是目前用户最容易接受的方式[19]。

1.3 研究内容和工作

区块链技术是当今科技发展最热门的众多技术之一，与 AI 技术、IOT 技术一并成为众多公司和学府重点快速发展的技术。工业界已经使用区块链为金融服务、存证服务和供应链等领域带来了新的应用场景和技术支持，希望能够在区块链技术之上构建信任的基础，推动互联网行业的进一步发展。本文在对相关区块链技术与密码学原理做了充分的研究的基础上，结合安全多方计算 ECDSA 标准签名算法，开发了集安全性与易用性的密钥管理系统。

本文的研究内容包括以下几点:

1、研究 Lindell 的两方计算的 ECDSA 签名算法[12]，该方案只支持两方计算所以在密钥管理系统应用中能力受限。本文将其改进扩展为多方计算，支持多方参与 ECDSA 的密钥生成和签名过程，增强实用性。

2、在安全多方计算的基础上，设计实现刷新（Refresh）各参与方密钥份额的算法，同时保证主公钥不变。该算法在一部分密钥份额被攻击者窃取的情况下可以对全部或者部分密钥份额进行刷新，保证密钥管理系统的强鲁棒性和密钥的动态安全性。

3、使用集成加密技术对密钥份额进行加密存储备份，集成加密技术采用了用户习惯的密码口令保护密钥份额的方法，可以根据需要的安全保护程度选择安全强度不同的 KDF 算法和加密算法。让用户使用最熟悉的密码口令保护密钥，同时提供灵活自定义的安全控制。

4、将上述的密码学原理和技术方案整合设计成为一个密钥管理系统（Key Management System，KMS），为用户提供便捷的密钥生成、数字签名、密钥刷新和密钥备份的接口，该系统可以整合到任意的区块链应用中，解决前文探讨的区

块链密钥使用过程中遇到的安全性和易用性的问题。

使用安全多方计算计算实现密钥管理方案，可以防范密钥单点泄漏导致整个账户资产信息被盗，攻击者需要同时通过攻击获得多个私钥份额才能成功控制区块链账户，而当某一参与方被攻击时可以及时刷新私钥份额使得之前的密钥份额失效。密钥管理方案的整个生命周期中不会出现主私钥，签名时各方只需使用自己的私钥份额进行计算，计算完成将各方结果进行整合，没有私钥恢复的过程可以提供强有力的密钥保护。利用集成加密技术让用户只需使用较简单的密码口令即可使用，提高区块链的易用性，为区块链密钥的提供一种安全易用的使用方案。

1.4 本文结构

本论文共分 5 个章节，每个章节的内容如下：

第一章，绪论。第一章阐述了本论文的研究背景和业界、学界在该方向上的国内外研究现状，分析了已有方法存在的问题并阐述本文的主要工作内容和创新点。

第二章，技术概述。介绍了与本文主要工作相关的底层技术和原理，包括区块链技术、密码学等原理。

第三章，算法设计与分析。介绍了基于安全多方计算的密钥生成、数字签名、密钥份额刷新的具体算法以及使用集成加密技术备份密钥的详细方案。

第四章，系统设计和实现。根据前文提出的密码学方案和密钥备份方案，描述了具体的密钥管理系统架构设计，包括密钥管理系统的分层设计和每个模块的详细设计以及相互依赖关系。根据系统设计，介绍了密钥管理系统的工作流程，并进行了仿真实验。

第五章，总结与展望。对全文的方案和设计实现做出总结，展望未来的工作。

第二章 技术概述

2.1 区块链

2.1.1 区块链结构

区块链技术起源于比特币系统，2008 年中本聪发布了比特币的白皮书《比特币:一种点对点的数字货币系统》，一种基于分布式网络基础和一致性共识算法的加密货币实现[1]。比特币的目的是实现一种没有可信第三方机构的去中心化的，任何人可以参与的，可以直接相互交易的网络支付系统。比特币的本质是一个去中心化的分布式存储系统，存储系统的数据状态通过公认的共识算法更新转移，这个存储系统就被称为区块链。

在区块链系统中，数据以交易（Transaction）的形式被打包成一个区块（Block），每个区块有一个指向前一个区块的 hash 指针、本区块的 hash 值以及全部交易列表和其他相关的必要信息，因此区块与区块之间通过哈希指针相连组成了区块链的链式结构，如图 2.1:

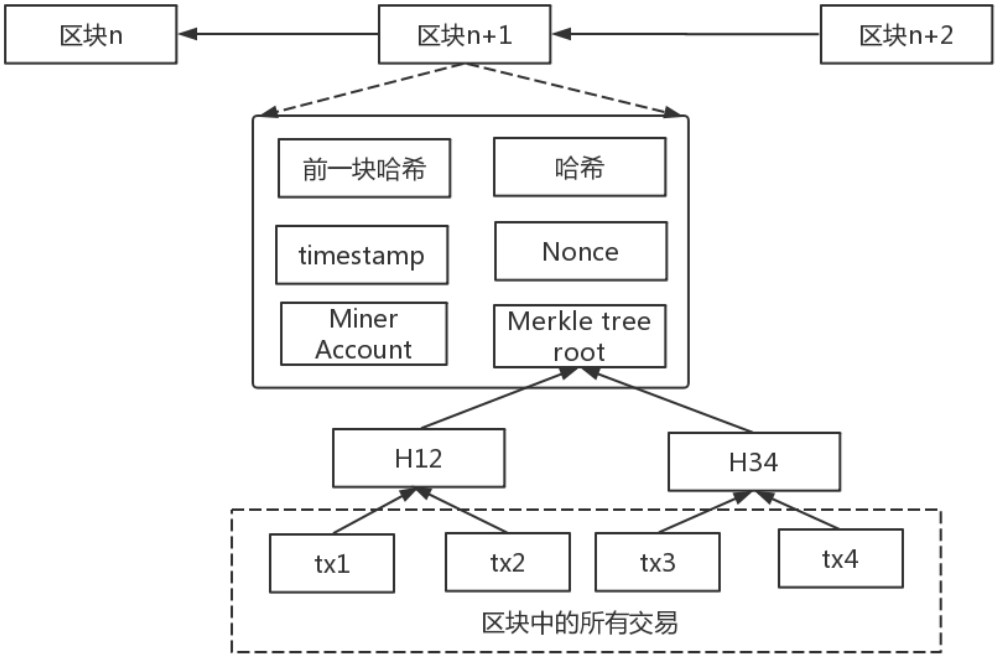


图 2.1 区块链结构示意图

2.1.2 区块链交易验证

区块链系统作为一个分布式存储系统，也是状态系统，全节点会存储所有的从创世区块（也称为第 0 块）以来的所有历史数据且不断追加新的共识数据，数据状态就在这个“世界状态机”里面通过交易的执行不断转移。大多数的区块链系统以账户为主体，每个账户地址在链上是唯一的，账户的信息是以 key-value 的形式存储在区块链账本中的。用户在链下对交易进行签名，将签名过的交易发送到到区块链网络节点上，共识节点先验证交易的有效性，接着将有效的交易打包成区块经过其他的共识节点确认后存储下来，根据每笔交易的内容更新存储系统中的账户数据。交易在区块链网络上的处理过程如图 2.2 所示：

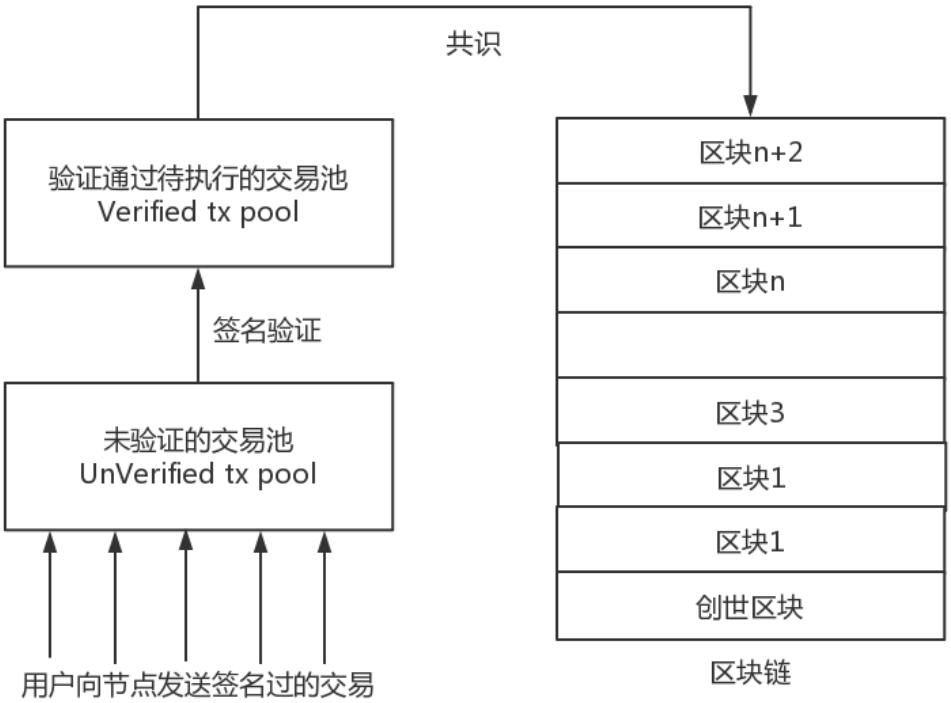


图 2.2 区块链验证签名的流程

账户交易的签名是对交易 hash 值的签名，当今主流的区块链系统（如比特币、以太坊、EOS 等等）使用的区块链签名算法都是 ECDSA，有少部分为了兼容传统行业（如银行、证券等）新增支持 RSA 的签名，或者兼容多种账户公钥类型的签名算法（比如同时支持 RSA、ECDSA、DSA 等）的混合平台。因为 RSA 的门限签名比较成熟且传统行业使用的比较多，本文不做详细描述，主要针对安全多方计算的 ECDSA 算法进行分析。

2.1.3 区块链架构

区块链技术发展至今，针对不同行业场景的功能和权限等需求衍生出了公有链和许可链（也称联盟链）。公有链对全世界开放，人人都可以参与到公有链的建设中来，通过共识机制和激励机制建设全世界范围的去中心化的组织，用户参

与到公有链的途径是创建合法账户，通过对交易进行签名在链上交易资产或者部署业务智能合约，也可以贡献本地的计算能力成为公有链的出块节点和区块验证节点。许可链针对企业不想将业务数据开放给全世界用户查看的问题，在小范围内建立区块链系统。企业通过与合作伙伴共同建立区块链的方式，各方维护不同的节点，所有的业务往来数据都要通过多方共识才能够写入到联盟区块链账本中。许可链更适用于传统企业保护商业数据隐私的需求。表 2-1 对公有链和许可链的重要特征进行对比。公有链去中心化程度最高，但是因为参与的节点分布在全世界各处，所以共识过程普遍效率较低，但是人人都可以参与进来。许可链是小范围的去中心化程度较低的区块链系统，要接入到许可链中需要有其他成员的授权，许可链更适合于交易速度较快且业务信息不可公开的场景。

	公有链	**许可链**
参与用户	任何用户	准入和授权的成员
中心化程度	完全去中心化	去中心化程度根据业务场景而定
项目特点	数据公开，交易执行速度慢，交易确认时间长	共识速度快，可以做保护数据的方案，更适用于商业场景
著名项目	BTC，ETH…	Fabric,，Quorum,，Fisco-Bcos…

表 2.1 公有链、许可链对比

一个完整的区块链系统由数据存储、网络 P2P、共识算法、智能合约虚拟机和账户协议等部分组成，并利用密码学的安全性来保证数据的一致性和不可篡改。

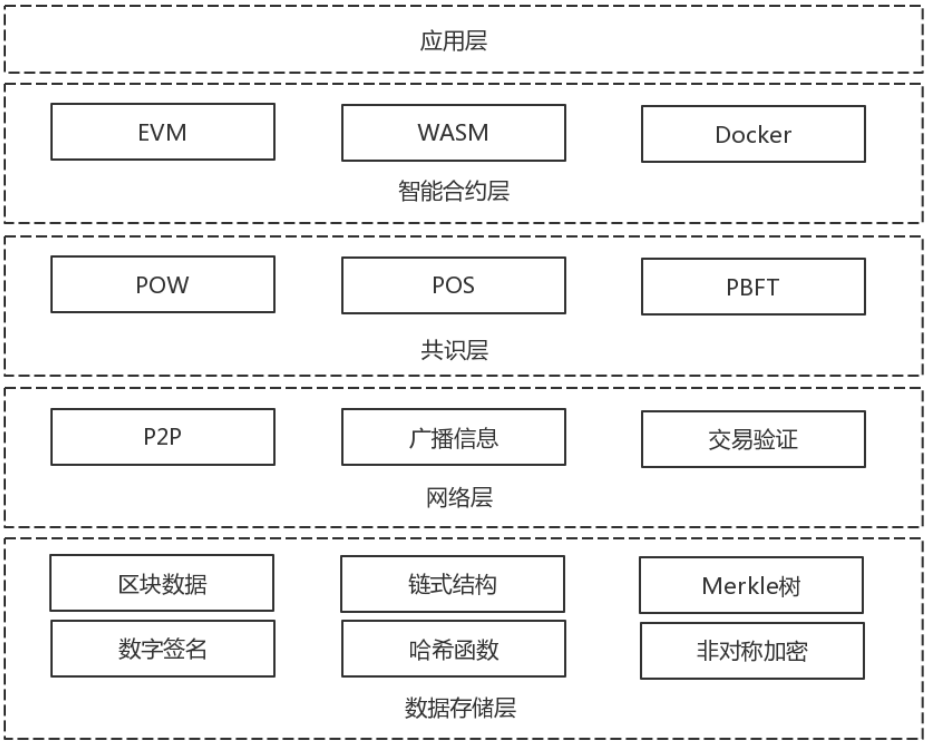


图 2.3 区块链架构图

图 2.3 是区块链系统的架构图。比特币系统的链上数据使用 Merkel 树结构进行存储，Merkel 树是一个二叉树，每个父节点由两个子节点的 hash 值计算得到，不断向上的计算最终会得到一个 Merkel Root 作为数据的标识存放在新区块中，如果区块的 Merkel root 不相等则说明区块数据不一致。Merkel 树可以用来很方便的做 SPV 验证，减轻区块链轻节点的存储压力，在共识过程中也利用 Merkel 树来检验区块中的交易是否完整。

共识算法的发展来源于分布式系统的一致性算法，共识算法的目的是在区块链系统中大多数节点是诚实节点的情况下保证链上合法数据的一致性。PoW 算法最先在比特币系统中使用，核心思想是通过节点的计算能力来计算区块的 nonce 值使得区块的 hash 值小于特定数值（hash 值的验证很简单只需验证数值的大小，但是需要穷举尝试计算 nonce 的值来计算 hash）来选取出块节点，计算速度越快拥有区块记账权的概率就越大，那么谁就能得到区块链的奖励（简称挖矿奖励）[20]。POW 共识算法会有大量没有实际效用的计算消耗大量的资源，但是该共识算法能防范不多余 50%的恶意攻击。BFT 类的共识算法多用于许可联盟链，BFT 最早在 1982 看由 Lamport 提出[21]，在恶意节点数量少于总节点数量 1/3 的情况下，区块链系统都可以正常且稳定的运行。PBFT 是在许可链中使用频率最高的 BFT 类共识算法[22]，经过三阶段的通信确定新区块的内容。还有为了减少 POW 能量消耗的基于权益证明的共识算法 POS 和委托股权证明 DPOS[23][24]。

Szabo 在 1994 年提出了智能合约[25]的概念，智能合约旨在解放人力审核和人力校验的成本。将智能合约的内容在部署执行之前先定义好，后续根据交易条件运行指定的合约业务逻辑。但是智能合约需要有一个各方都信任的执行环境且还要保证智能合约不会被随意篡改。所以区块链去中心化、不可篡改的特性让智能合约有了施展的空间。以太坊在 2014 年首次提出并实现了用于执行智能合约的以太坊虚拟机 EVM，EVM 支持图灵完备的智能合约语言 Solidity，理论上可以执行任意的合约逻辑代码。近几年智能合约不断发展，Hyperledger Fabric 支持 Go 和 JAVA 语言的虚拟机，EOS 支持 WASM 虚拟机可以执行 C++智能合约。智能合约虚拟机的发展，区块链的 TPS 提升了几百倍的量级，为区块链支持更多的业务场景提供了更多的可能。

2.2 密码学

2.2.1 哈希函数

密码学上的哈希算法用于检验数据的完整性：对任意长度的明文 m，经由哈

希函数 H 可产生固定长度的哈希值 H(m) 。这个值可以看作是明文 m 的指纹（fingerprint）或者摘要（message digest），一旦数据改变，哈希值就随之改变。所以可以通过数据的哈希值验证数据是否完整或者一致。哈希函数要求具有单向性、抗弱碰撞性和抗强碰撞性，并且不容易被暴力破解。

最早被提出的哈希算法是 MD4，紧接着 MD5、SHA1、SHA2 等不断成为标准算法，在 SHA3 之前的所有哈希函数都是基于 Merkle-Damgård 架构来构造的[26]，也是 Rivest 首次在 MD4 中使用的计算架构，MD4 为后续哈希算法的发展打下了基础的设计思路。但是基于 Merkle-Damgård 架构的 MD4、MD5、SHA1 都已经被证明不够安全[27]，学术界也总结出了针对该架构的哈希函数的通用攻击方法。从 2008 年开始历时 7 年时间，SHA3 的标准竞赛结束，2015 年 SHA3 成为 NIST 的标准，为了防范未来可能出现的针对 Merkle-Damgård 架构的有效普遍攻击，SHA3 算法明确说明不能使用 Merkle-Damgård 架构，而是使用了 Sponge construction 新架构。

下面针对主流的哈希算法进行性能测试。在 Ubuntu 17.04 4.10.0-19-genericntel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz，内存 32GB 的机器上，运行多次哈希算法的总时间对比。如图 2.4 所示，每个 hash 算法的在不同原文信息大小下计算一万次的平均时间对比。最早提出的的 MD4、MD5 和 SHA1 的计算速度很快，但是不具有实用性；SHA2 家族是目前使用最多的哈希算法，性能相对较低，但是 SHA2 家族是目前较为安全的哈希算法选择。SHA3 家族作为最新的哈希算法，性能跟 SHA2 家族的相差不大，未来工业界可能会慢慢向 SAH3 转变。

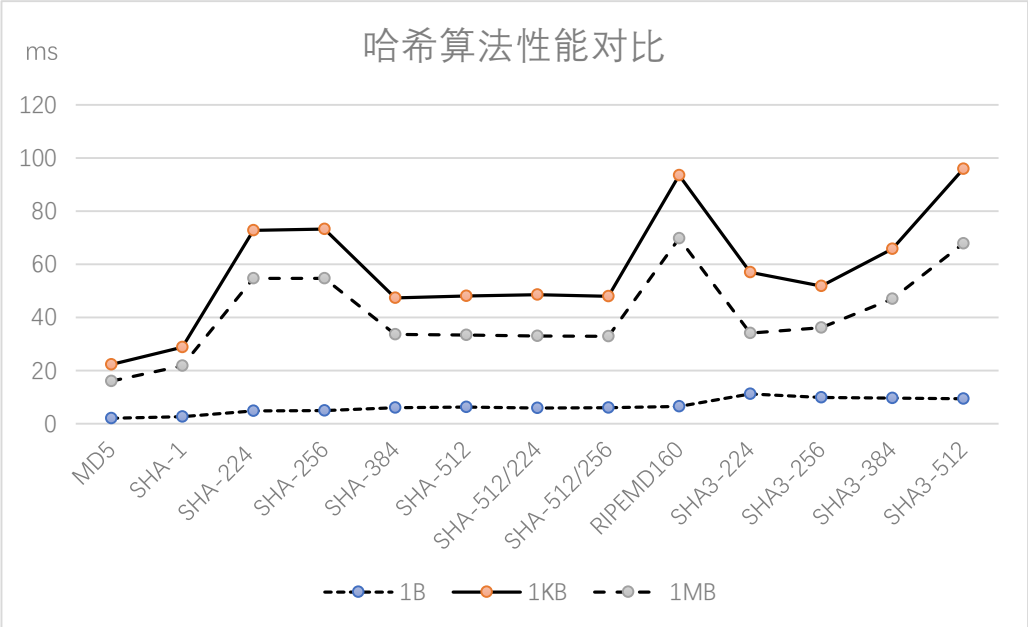


图 2.4 哈希算法性能对比

2.2.2 Schnorr 签名

Schnorr 签名是一种数字签名方案，方案的安全性是基于离散对数难题[17]。

初始条件有群 G，生成元 g，生成元的阶 q，以及哈希函数 $H:\{0,1\}^* \rightarrow \mathbb{Z}_q$ 。签名者拥有私钥 x 和公钥 $y = g^x$。

Schnorr 的签名过程如下：

1. 随机选取 $k \in \mathbb{Z}_q^*$；
2. 计算 $r = g^k$；
3. 计算$e = H(r \ || \ M)$;
4. $s = k - xe$;
5. 输出签名(s, e)。

Schnorr 的验签过程如下：

1. 计算 $r_v = g^s y^e$；
2. 计算 $e_v = H(r_v \ || \ M)$；
3. 判断 $e_v = e$。若相等则签名正确。

Schnorr 签名具有签名短且速度快的特性，特别适用于验证离散对数问题，证明者根据私钥计算 Schnorr 签名，验证者验证签名的正确性即可判断证明者是否拥有私钥。

2.2.3 椭圆曲线数字签名算法

Miller 和 Koblitz 等人在 1985 年提出了基于椭圆曲线的公钥密码体制[28][29]，基于椭圆曲线的离散对数问题(Elliptic Curve Discrete Logarithm Problem)被证明是难解的。基于椭圆曲线密码学的签名算法与 RSA、ElGamal 密码体系相比，需要更少的密钥长度就可以保证同等的安全强度，使得椭圆曲线密码学越来越受欢迎。ECDSA 于 2013 年成为 NIST 标准[30]。

ECDSA 的签名过程如下：

1. 使用约定好的哈希函数对消息原文求 hash 值 $z = HASH(m)$，
2. 随机选取一个整数值$k \in [1, n-1]$，n是生成员的阶，
3. 计算点$(x_1, \ y_1) = k \ \times \ G$，
4. 令 $r = x_1$，若$r = 0$，重新执行第二步，
5. 计算$s = k^{-1}(z \ + \ rd_A) \ mod \ n$，若$s = 0$，重新执行第二步，
6. 输出签名$(r, \ s)$。

ECDSA 的验证过程如下：

1. 检验 r 和 s 是否都属于$[1, n-1]$，若不属于则签名无效，
2. 使用约定好的 hash 函数对消息原文求 hash 值 $z = HASH(m)$ ，
3. 计算 $u_1 = zs^{-1} \ mod \ n$ 和 $u_2 = rs^{-1} \ mod \ n$,
4. 计算点 ，如果 ，则签名无效,
5. 如果，则签名验证通过。

椭圆曲线数字签名算法被普遍使用在身份认证、密钥交换、信息通信等领域，区块链系统就利用了其良好的身份认证特性。

2.2.4 Paillier 加密算法

Paillier 算法是在 1999 年由法国密码学家 Pascal Paillier 设计的公钥密码学方案[31]。加解密过程类似于 RSA 加密算法，该加密算法支持同态加密，与 RSA 不同的是 Paillier 满足同态加法特性。加法同态是指在不需要知道 x 和 y 值的条件下，从 $Enc(x)$ 和 $Enc(y)$ 通过同态加法运算计算出 $Enc(x+y)$ ：

同态加密：

$$
\begin{aligned}
&Enc_{pk}(m_1, r_1) \cdot Enc_{pk}(m_2, r_2) \\
&= g^{m_1} r_1^n \cdot g^{m_2} r_2^n \\
&= g^{m_1+m_2} r_3^n \\
&= Enc_{pk}(m_1 + m_2, r_3)
\end{aligned}
$$

解密：

$$
\begin{aligned}
&\mathrm{Dec}_{sk}\left(Enc_{pk}(m_1, r_1) \cdot Enc_{pk}(m_2, r_2) \ \ mod \ \ n^2\right) \\
&= m_1 + m_2 \ \ mod \ \ n
\end{aligned}
$$

支持加法同态特性的加密算法通常被使用在数值型数据的场景中，区块链隐私保护中可以使用 Paillier 算法对账户资产信息进行加密，链上只存储加密后的数据，同时支持在加密状态下通过同态加法更新账户的资产信息，只有拥有私钥的用户才能解密密文查看账户的真实数据，从而达到隐私保护的目的。Paillier 也常被用到零知识证明方案中，利用同态的特性隐藏证明者的知识，并维持数据可验证的状态。

2.2.5 零知识证明

零知识证明（Zero-Knowledge Proof）的概念是由 Goldwasser 和 Micali 等人在文献[32]中提出来的。在交互系统中，交互双方互相证明自己拥有的信息，当证明过程中双方都只是证明自己拥有信息但是信息的具体内容并没有泄漏出去，对方也不能从交互的信息中推断出信息的具体内容，那么这样一个过程称为零知识证明过程。

零知识证明被用到了很多密码算法中。区块链系统让零知识证明算法有了更多的应用场景。Zcoin 是一个可以隐藏交易 origin 的区块链系统，他使用了 Zerocoin 零知识证明协议[33]。Zcash 是一个可以隐藏交易的 origin、value 和 destination 的区块链支付系统，该系统使用的 Zerocash 协议借鉴了 zkNSNARKs 零知识证明算法[34]，zkNSNARKs 使用了 Pinocchio 协议基于椭圆曲线 pairing 的原理使得区块链的交易的具体数据不被挖掘出来的同时保证交易的有效性[35]。零知识证明也被广泛使用在云计算上，为了验证云服务计算的结果的正确性，零知识证明被用在了可验证计算上，只要用户在验证计算结果的时间消耗比自己进行计算的消耗小，那么可验证计算就是有实际应用价值的，至今能够在工业上实用的可验证计算协议是 Pinocchio 协议[36]。

2.2.6 Range Proof

Range proof 首次在 Brickell 等人的文献[37]中提出。Range proof 是用来证明在给定的承诺中，承诺的值属于一定的范围。最简单的 Range proof 是先给出数值 x 的承诺 commitment，在证明的时候直接将数值 x 发送给验证者，即可直接进行验证。零知识 Range proof 在广义范围证明的基础上加上了零知识的特性，在证明承诺的数值是属于一定范围的时候，不泄漏具体的数值，验证者只能够知道该数值确实是在验证范围内的。在实际使用时，零知识 Range proof 才有更多的应用场景，比如电子支付系统[38]。Boudot 在论文[39]中提出了一种可以证明一个数值在任意范围内的方法，该方法可以用在保险行业对体检信息的保护中，也可以用于做贷款行业的资产证明。BulletProof [40]则使用电路证明的方法，被广泛应用于区块链支付系统的金额证明和保护。

2.2.7 分布式密钥生成

分布式密钥生成是分布式签名的基础，在分布式密钥生成阶段各个参与方会分别生成一个密钥份额，通过可验证秘密共享 (verifiable secret sharing，VSS)和同态承诺的算法进行交互，最终计算出主公钥。VSS 是分布式密钥生成

（Distributed Key Generation，DKG）的基本 ,VSS 的概念在[41]中被 Chor 和 SGoldwasser 首次提出。VSS 用来验证每个参与者生成且承诺的密钥份额是正确的，才能保证最终生成的主公钥也是正确的。

2.4 本章小结

本章主要介绍了本文方案需要使用到的区块链背景知识和相关密码学算法。首先介绍了区块链的起源与发展现状，区块链已经被大量的企业使用并落地到实际项目中，总结了区块链的架构和区块链的工作原理。区块链的安全性依赖于密码学的安全，包含数字签名、哈希算法、零知识证明、Range Proof 和分布式密钥生成等等密码学的基础算法和高级扩展算法。分别介绍了每个密码算法的基本原理及其应用。

第三章 算法设计与分析

本文根据第一章对区块链密钥管理国内外现状的深入研究和分析，总结了1.2 节中的三个问题，针对这三个问题分别给出以下三点解决方案：

1、密钥权力不够分散：当将一个完整的密钥直接存储时，就潜在着整个完整密钥直接泄漏的风险，完整密钥直接泄漏将导致丢失账户的所有权。本文使用安全多方计算的思想，将主密钥“分成”多份分别存放在各个参与方手中。采用分布式密钥生成的方式，由多个参与方参与密钥的生成，每个人持有密钥的一个份额；在签名阶段采用安全多方计算的方式，各方分别利用手中的密钥份额进行计算，最终将计算结果整合到一起生成完整的 ECDSA 签名。

2、不能刷新密钥：如果密钥份额不能刷新，攻击者只需将密钥份额一个一个的攻击窃取到手，最终就可以掌控整个区块链账户。所以需要提供能够刷新密钥份额的能力。在本文的方案中，基于分布式密钥生成的协议，新增支持分布式密钥刷新的算法和接口，用户在部分密钥泄漏的情况下可以对全部或者部分密钥份额进行刷新，保证密钥的动态安全性。

3、易用性差：众多普通用户还是更适应于使用密码口令来保护信息，记忆很复杂的密钥字符串对于终端用户而言体验很差。本文设计了使用集成加密模式来存储备份密钥的方法，用户只需使用密码口令即可进行分布式密钥生成、分布式签名和分布式密钥刷新的功能。

3.1 密钥生成与签名

2017 年 Lindell 在文献[12]中提出了支持两方计算的 ECDSA 签名算法，两方计算可以在一定程度上分散密钥控制区块链账户的权力，当两个密钥份额中的一份泄漏时也不至于导致账户的资产受到直接威胁。Lindell 的方案支持两方密钥生成和两方分布式签名，但是两方计算的局限性和弱适用性使得该方案不能很好的使用在各种密钥管理系统中，2018 年 Lindell 在[13]中提出了多方门限 ECDSA 算法，该算法因为引入了门限机制所以需要进行多轮的交互和分别使用三种零知识证明方案才能保证算法的正常运行和安全性要求，而且该方法不支持密钥刷新的功能，在效率上和实用性上有待进一步的改进和优化。所以本章在[12]的基础上对算法进行改进，将原本的两方计算改进为多方计算，使得能够支持三方及以上数量的参与者参与到密钥生成和签名的过程中。使用两方计算的原理实现多方计算的功能，这样就没有传统的多方计算里面需要解决诚信的大多数（honest majority）的问题，除了上述将两方扩展为多方的改进部分，本文的方案还支持密钥刷新的功能，支持密钥全局刷新和局部刷新，提供密钥管理的动态安

全性，并对分布式密钥生成过程中的 range proof 进行性能优化。

3.1.1 分布式密钥生成

分布式密钥生成是在生成新的账户或者新公私钥对时必须进行的过程，过程中需要多个参与方共同参与，每个参与方根据协议随即生成密钥份额，并将密钥份额对应的公钥信息和证明信息发送给其他参与方，必要时需要进行零知识证明的过程，最终结果为每个参与方个字持有密钥份额且都计算得到一个相同的主公钥。整个计算和通信的过程使用到 Schnorr 签名、Paillier 同态加密算法、Range proof 算法和零知识证明，下面是分布式密钥生成过程的具体 6 个步骤，使用共有三个参与方的情况进行说明，后面很容易从三方计算过程扩展至更多参与方的过程。如图 3.1：

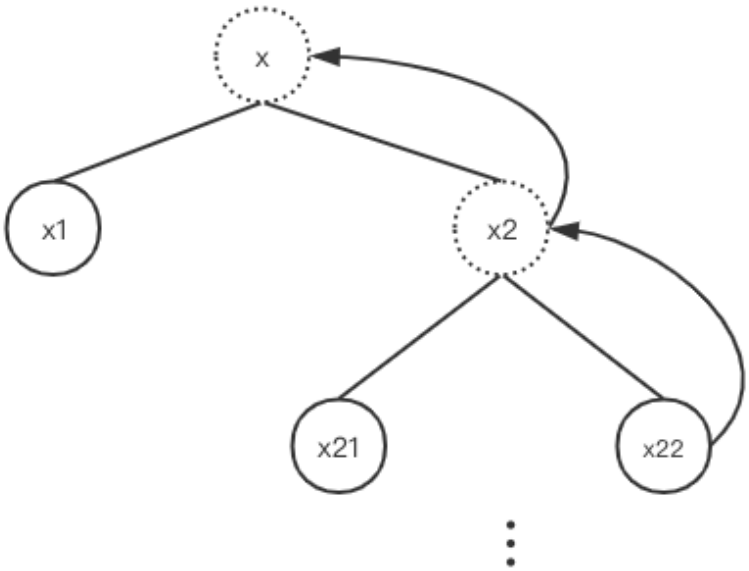


图 3.1 分布式密钥生成

$\mathrm{KeyGen}(\mathbb{G},\mathrm{G},\mathrm{q})$：系统初始化时生成了循环群 $\mathbb{G}$，生成元G，生成元的阶q。参与方为$\mathrm{P_i}$，以上参数是每个参与方都已知且认同的参数。图 3.1 中 x 表示主私钥，主私钥由两部分 $\mathrm{x_1}$ 和 $\mathrm{x_2}$ 组成，但是 $\mathrm{x_2}$ 不真实存在，$\mathrm{x_2}$ 由 $\mathrm{x_{21}}$ 和 $\mathrm{x_{22}}$ 计算得到。虚线框中的信息不真实存在，每个参与方真正随机生成的是实线框中的数据。

1. $\mathbf{P_1}$

（a）P_1 随机选取 $x_1 \in \mathbb{Z}_{q/3}$，并计算$\mathrm{Q_1} = x_1 \cdot \mathrm{G}$；$x_1 \in \mathbb{Z}_{q/3}$ 的范围不是 $\mathbb{Z}_q$ 的原因在 3.1.3 节进行详细阐述。

（b）输入 $(com-proof, 1, Q_1, x_1)$ 给 $\mathcal{F}_{com-zk}^{R_{DL}}$，发送$\mathrm{Q_1}$的承诺和$\mathrm{x_1}$的 Schnorr 签名。

2. P_{21}, P_{22}

（a）P_{21}, P_{22} 从 $\mathcal{F}_{zk}^{R_{DL}}$ 收到 $(proof-receipt, 1)$；

（b）P_{21} 随机生成密钥份额 $\mathrm{x_{21}}$，P_{22}随机生成密钥份额 $\mathrm{x_{22}}$，并分别计算密钥份额对应的公钥 $\mathrm{Q_{21}} = x_{21} \cdot G$， $Q_{22} = x_{22} \cdot G$；

（c）P_{21} 计算 x_{21} 的 Schnorr 签名，P_{22}计算 $\mathrm{x_{22}}$ 的 Schnorr 签名；

（d）P_{21}将 $\mathrm{Q_{21}}$ 和签名发送给，P_{22}将 Q_{22} 和签名发送给；

（e）P_{21}计算 $\mathrm{Q_2} = x_{21} \cdot Q_{22}$，$P_{22}$计算$Q_2 = x_{22} \cdot Q_{21}$；

（f）将各自计算得到的 $\mathrm{Q_2}$ 和签名信息 $(prove, 2, Q_2, Q_{2i}, x_{2i}), i \in [1,2]$ 输入给 $\mathcal{F}_{zk}^{\mathrm{RDL}}$，并发送给$\mathrm{P_1}$。

3. $\mathbf{P_1}$

（a）首先 $\mathrm{P_1}$ 从 $\mathcal{F}_{zk}^{\mathrm{RDL}}$ 收到 $(proof, 2, Q_2, Q_{2i}), i \in [1,2]$，验证其他参与方传输过来的 Schnorr 签名以及检查 $\mathrm{Q_2}$ 是否相等。若没有收到，过程错误退出。若有一处验证不通过，过程错误退出；

（b）$\mathrm{P_1}$ 将步骤一中承诺的签名值发送给其他参与者，输入$(decom-proof, 1)$到 $\mathcal{F}_{com-zk}^{\mathrm{RDL}}$；

（c）$\mathrm{P_1}$ 随机生成规定长度的 Paillier 密钥对(pk, sk)，并使用 Paillier 加密算法计算$c_{key} = Enc_{pk}(x_1)$；

（d）$\mathrm{P_1}$ 将 $(prove, 1, N, (p_1, p_2))$ 输入到 $\mathcal{F}_{zk}^{\mathrm{RP}}$ 证明生成的 Paillier 密钥 (pk, sk) 是符合配置要求的，其中$\mathrm{pk} = \mathrm{N} = \mathrm{p_1} \cdot p_2$[42]。并将 c_{key} 发送给其他参与方。

4. $\mathbf{P_1}$

$\mathrm{P_1}$ 向其他全部参与者证明 $(c_{key}, pk, Q_1) \in L_{PDL}$。该步骤的作用是向其他参与者以零知识证明的方式证明 $\mathrm{c_{key}}$ 是 pk 对 $\mathrm{Q_1}$ 的离散对数 $\mathrm{x_1}$ 加密的密文，即 $\mathrm{c_{key}} = \mathrm{Enc_{Paillier}}(x_1, pk)$，并且证明 $\mathrm{P_1}$ 知道加密 $\mathrm{x_1}$ 的 pk 对应的 Paillier 私钥 sk 的值。该零知识证明算法使用到了 Paillier 算法的同态加法特性和加解密方法，并且在证明过程中需要与验证者进行两轮通信，是一个交互式的零知识证明过程，详细的证明过程见[12]中第 6 节“Zero-Knowledge Proof for the Language $\mathrm{L_{PDL}}$”，。

5. $\mathbf{P_{21}, P_{22}}$

除 p1 外的所有参与方要进行下面的三项验证，并且保证每项验证都验证通过。
（a）从 $\mathcal{F}_{com-zk}^{\mathrm{R_{DL}}}$ 收到 $(decom-proof, 1, Q_1)$ 和从 $\mathcal{F}_{zk}^{\mathrm{R_P}}$ 收到 $(proof, 1, N)$ 并都验证通过；
（b）$\mathrm{P_1}$ 向每个参与者证明 $(c_{key}, pk, Q_1) \in L_{PDL}$ 通过；
（c）$\mathrm{pk = N}$ 的大小在设置范围内。

6. Output

（a）$\mathrm{P_1}$ 计算 $\mathrm{Q = x_1 \cdot Q_2}$，并将$(x_1, Q)$存储下来；
（b）$\mathrm{P_{2i}}$ 分别存储(x_{2i}, c_{key})。

上述 KeyGen 过程结束后，获得的主公私钥对是$(x = x_1 \cdot x_{21} \cdot x_{22},\ X = x \cdot G)$，x 不真实存在。

分布式密钥生成 KeyGen 的过程中，每个参与方都需要用自身随机生成的密钥份额 $\mathrm{x_I}$ 对相关信息（如“分布式密钥生成”字符串）进行 Schnorr 签名，向其他全部参与方证明其拥有该密钥份额，Schnorr 签名的数量与参与方数量成正比，需要 $O(n)$ 次 Schnorr 签名和验证计算。$\mathrm{P_1}$ 对 $\mathrm{x_1}$ 进行一次 Paillier 加密，并分别进行了两次不同的证明，一次证明 Paillier 的公钥在合理范围内；一次以零知识的方式进行了 L_{PDL} 的证明，所以的证明过程中都需要与每个参与方进行交互，所以参与方越多计算和交互的消耗越大，在 Paillier 的公钥合法证明和 L_{PDL} 的证明过程中也需要 $O(n)$ 的计算。在进行 L_{PDL} 证明的同时对 $\mathrm{x_1}$ 进行了一次 Range Proof 证明 $\mathrm{x_1}$ 在合理范围内，证明者之进行一次计算，每个验证着进行一次验证，计算复杂性为 $O(n)$。

3.1.2 分布式签名

分布式签名过程需要分布式密钥生成过程中的所有参与者共同参与，每个参与方利用自身的密钥份额 $\mathrm{x_i}$ 以及随机生成的 $\mathrm{k_i}$ 计算签名中间值。利用 Paillier 加密算法的同态特性，对密钥份额和随机数进行加密。通过参与方之间通信交换信息，分布式签名过程的结果是一个完整的 ECDSA 签名信息。整个过程需要使用到 Schnorr 签名、Paillier 同态加密算法，下面是分布式签名过程的具体步骤，如图 3.2 所示：

签名接口 $\mathrm{Sign(sid, m)}$，输入本次签名 id 以及要签名原文信息。系统约定的哈希函数 $HASH(m)$。所有参与方都可以计算 $\mathrm{m' = }HASH(m)$，以上参数是

每个参与方都已知且认同的参数。图 3.2 中 k 表示 ECDSA 签名过程中的随机数（ECDSA 签名算法是非确定性的，计算过程中有随机数参与计算），随机数由两部分 k_1 和 k_2 组成，k_2 由 k_{21} 和 k_{22} 计算得到。虚线框中的信息不真实存在，每个参与方真正随机生成的是每个叶子节点中的数据，实线框中的非叶子节点的数据是通信过程中根据其他参与方的信息计算得到的，实线框中的信息都存储在相应参与方的环境中。

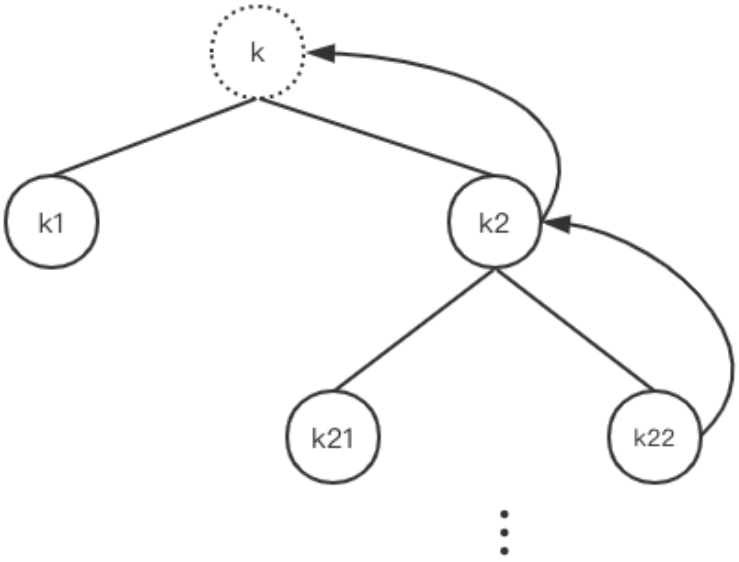


图 3.2 分布式签名示意图

1. $\mathbf{P_1}$

（a）P_1 随机选取 $k_1 \in \mathbb{Z}_q$，并计算$\mathrm{R}_1 = k_1 \cdot \mathrm{G}$；

（b）输入 $(com-proof, \mathrm{sid}||1, R_1, \mathrm{k}_1)$ 给 $\mathcal{F}_{com-zk}^{R_{DL}}$，发送$\mathrm{R}_1$的承诺和$\mathrm{k}_1$的 Schnorr 签名。

2. $\mathbf{P_{21}, P_{22}}$

P_{21}, P_{22} 运行一次 coin tossing，生成 k_2 和 R_2：

（a）P_{21}, P_{22} 从 $\mathcal{F}_{zk}^{R_{DL}}$ 收到 $(proof-receipt, \mathrm{sid}||1)$；

（b）P_{21} 随机生成密钥份额 k_{21}，P_{22}随机生成密钥份额 k_{22}，并分别计算$\mathrm{R}_{21} = k_{21} \cdot G$，$R_{22} = k_{22} \cdot G$；

（c）P_{21} 计算 k_{21} 的 Schnorr 签名，P_{22}计算 k_{22} 的 Schnorr 签名；

（d）P_{21}将 R_{21} 和签名发送给 P_{22}，P_{22}将 R_{22} 和签名发送给 P_{21}；

（e）P_{21} 计算 $\mathrm{R}_2 = k_2 \cdot G = HASH(k_{21} \cdot R_{22}) \cdot G$，$P_{22}$计算 $R_2 = k_2 \cdot G = HASH(k_{22} \cdot R_{21}) \cdot G$；

（f）将各自计算得到的 R_2 和签名信息 $(prove, sid||2, R_2, R_{2i}, k_{2i}), i \in [1,2]$ 输入给 $\mathcal{F}_{zk}^{R_{DL}}$，并发送给$P_1$。

3. P_1

（a）P_1 从 $\mathcal{F}_{zk}^{R_{DL}}$ 收到 $(proof, sid||\ 2, Q_2, Q_{2i}), i \in [1,2]$，验证其他参与方传输过来的 Schnorr 签名以及检查 R_2 是否相等。若没有收到，过程错误退出。若有一处验证不通过，过程错误退出；

（b）P_1 将步骤一中承诺的签名值发送给其他参与者，输入 $(decom - proof, sid||1)$ 到 $\mathcal{F}_{com-zk}^{R_{DL}}$；

4. P_{21}, P_{22}

（a）从 $\mathcal{F}_{com-zk}^{R_{DL}}$ 收到 $(decom - proof, sid||1, R_1)$ 并都验证通过；

（b）P_{21}，P_{22} 分别计算 $R = k_2 \cdot R_1$，令$R = (r_x, r_y)$，则$r = r_x \ mod \ q$；

（c）P_{21}, P_{22} 再运行一次 coin tossing，生成 $\rho \in \mathbb{Z}_{q^2}$；

（d）P_{21} 计算 $c_{21} = Enc_{pk}(x_{21})$ 并发送给P_{22}，P_{22}计算 $c_{22} = Enc_{pk}(x_{22})$ 并发送给 P_{21}；

（e）$c_1 = Enc_{pk}(\rho \cdot q + [k_2^{-1} \cdot m' \ mod \ q])$；

（f）P_{21} 计算 $c_2 = k_2^{-1} \cdot r \cdot x_{21} \odot c_{22} \odot c_{key}$，然后计算 $c_3 = c_1 \oplus c_2$。将 c_3 发送给 P_1。

（g）P_{22} 计算 $c_2 = k_2^{-1} \cdot r \cdot x_{22} \odot c_{21} \odot c_{key}$，然后计算 $c_3 = c_1 \oplus c_2$。将 c_3 发送给 P_1。

5. P_1

（a）P_1 分别计算 $R = k_1 \cdot R_2$，令$R = (r_x, r_y)$，则$r = r_x \ mod \ q$；

（b）P_1 收到其他参与者的 c_3 信息，检查全部 c_3 信息相等，否则退出；

（c）P_1 解密计算 $s' = Dec_{sk}(c_3)$，$s'' = k_1^{-1} \cdot s' \ mod \ q$，

（d）P_1 令 $s = \min\{s'', q - s''\}$

（e）输出签名(r, s)。使用外部的 ECDSA 验证函数验证签名是否正确。否则输出错误。

上述 Sign 过程结束后，签名者将获得签名信息(r, s)。下面是签名信息的正确性证明：

P_{21} 和 P_{22} 在步骤 4 中同时计算出来的 c_2 和 c_3：

$$c_2 = k_2^{-1} \cdot r \cdot x_{21} \odot c_{22} \odot c_{key} = Enc_{pk}(k_2^{-1} \cdot r \cdot x_{21} \cdot \ x_{21} \cdot x_1)$$

$$c_3 = c_1 \oplus c_2$$

$$= Enc_{pk}(\rho \cdot q + [k_2^{-1} \cdot m' \ mod \ q]) \oplus Enc_{pk}(k_2^{-1} \cdot r \cdot x_{21} \cdot x_{21} \cdot x_1)$$

$$= Enc_{pk}(\rho \cdot q + [k_2^{-1} \cdot m' \ mod \ q] + k_2^{-1} \cdot r \cdot x_{21} \cdot x_{21} \cdot x_1)$$

P_1 收到 P_{21} 和 P_{22} 发送的 c_3 后解密得到 s'：

$$s' = Dec_{sk}(c_3) = \rho \cdot q + [k_2^{-1} \cdot m' \ mod \ q] + k_2^{-1} \cdot r \cdot x_{21} \cdot x_{21} \cdot x_1$$

$$s'' = k_1^{-1} \cdot s' \ mod \ q$$

$$= k_1^{-1}(\rho \cdot q + [k_2^{-1} \cdot m' \ mod \ q] + k_2^{-1} \cdot r \cdot x_{21} \cdot x_{21} \cdot x_1) \ mod \ q$$

$$= k_1^{-1}k_2^{-1} \ (m' \ mod \ q + r \cdot x_{21} \cdot x_{21} \cdot x_1) \ mod \ q$$

$$= k^{-1}(m' + r \cdot x) \ mod \ q$$

在计算$r = r_x \ mod \ q$时：

$$R = k_1 \cdot R_2 = (k_1 k_2) \cdot G$$

P_1 最终计算得到的签名信息是 (r, s)，满足 ECDSA 签名标准的要求并可验证通过。

分布式签名的过程中，每个参与方都需要用自身随机生成的随机数 k_i 对相关信息（如“分布式签名”字符串）进行 Schnorr 签名，向其他全部参与方证明其拥有该密钥份额，在生成随机数 $\rho \in \mathbb{Z}_{q^2}$ 时也需要对随机数进行证明，Schnorr 签名的数量与参与方数量成正比，需要 $O(2n)$ 次 Schnorr 签名和验证计算。除 P_1 之外的其他参与方都需要计算 $c_i = Enc_{pk}(x_i)$ 和 $c_1 = Enc_{pk}(\rho \cdot q + [k_2^{-1} \cdot m' \ mod \ q])$ 并计算c_2 和 c_3 的值，即 $O(2n)$ 次 Paillier 加密运算和 $O(2n)$ 次 Paillier 同态计算，整个过程中没有零知识证明的过程。

3.1.3 Range proof 算法优化

文献[12]中借鉴了[37]的方法对随机数进行 Range proof 的证明，以零知识证明的方式证明生成 P_1 生成的私钥 x_1 是属于 $\mathbb{Z}_q$ 的。使用到的算法是一个扩展率为 3 的算法，即一个数 $x \in [b_1, b_2]$，证明其属于范围 $[B_1, B_2]$，其中 $B_2 - B_1 / b_2 - b_1 = 3$。所以在分布式生成阶段随机生成的 $x_1 \in \mathbb{Z}_{q/3}$，才能证明 $x_1 \in \mathbb{Z}_q$。

在零知识证明系统中，证明者和验证者不可避免的要进行信息交互，而当交互的次数越多时因为网络传输的成本比计算成本高，这个零知识证明过程的时间消耗就会很大。所以根据 Fiat–Shamir heuristic 原理[43]将使用的 Range Proof 改为非交互模式，减少整个过程的交互次数和信息量。证明者只需进行一次的证明

计算，将证明信息 π 传输出去，所有的参与者即可同时验证，大大提升零知识证明 $\mathrm{L_{PDL}}$的性能。

下面介绍优化后的 Range Proof 详细过程：

初始条件：

证明者（Prover，P）拥有信息：(c, x, r) 和 Paillier 的密钥信息$(N, \emptyset(N))$，其中$\mathrm{c} = \mathrm{Enc_{pk}}(x; r)$。

验证者（Verifier，V）拥有信息：c 和 Paillier 公钥 N。

约定参数值：$\mathrm{q}, l = q \,/\, 3, t$，其中 t 是安全参数。

1. Prover

（a）验证者随机选取 t 个数：$\mathrm{w}_1^1, \ldots, w_1^t \leftarrow \{l, \ldots, 2l\}$，接着计算 $w_2^i = w_1^i - l$，其中 $i = 1, \ldots t$；

（b）证明者以独立 $1/2$ 的概率交换每一个 w_1^i 和 w_2^i 的值；

（c）计算 $\mathrm{c}_1^i = Enc_{pk}(w_1^i, r_1^i)$ 和 $\mathrm{c}_2^i = Enc_{pk}(w_2^i, r_2^i)$，其中$\mathrm{i} = 1, \ldots, t$，且 $\mathrm{r}_1^i, \mathrm{r}_2^i \in \mathbb{Z}_N$ 随机选取。

（d）计算 $hash = HASH(c_1^1, \ldots, c_1^t, c_2^1, \ldots, c_2^t)$，如果 hash 的比特长度小于 t，$\mathrm{hash} = hash \mid HASH(hash) | HASH(HASH(hash)) | \ldots$，直到 hash 的长度大于等于 t。e 是 hash 的前 t 比特；

遍历 e 的每一个比特$\mathrm{e_i}, \mathrm{i} = 1, \ldots \mathrm{t}$

（e）如果 $\mathrm{e_i} = 0$，令 $\mathrm{z_i} = (w_1^i, r_1^i, w_2^i, r_2^i)$；

（f）如果 $\mathrm{e_i} = 1$，令 $\mathrm{z_i} = (j, x + w_j^i, r \cdot r_j^i \ mod \ N)$，其中 $\mathrm{j} \in \{1,2\}$，使得 $x + w_j^i \in \{l, \ldots 2l\}$；

（g）发送 $c_1^1, \ldots, c_1^t, c_2^1, \ldots, c_2^t$ 和 $\mathrm{z}_1, \ldots z_t$ 给验证者。

2. Verifier

（a）验证者收到证明着发送的$c_1^1, \ldots, c_1^t, c_2^1, \ldots, c_2^t$ 和 $\mathrm{z}_1, \ldots z_t$；

（b）计算 $hash = HASH(c_1^1, \ldots, c_1^t, c_2^1, \ldots, c_2^t)$，如果 hash 的比特长度小于 t，$\mathrm{hash} = hash \mid HASH(hash) | HASH(HASH(hash)) | \ldots$，直到 hash 的长度大于等于 t。e 是 hash 的前 t 比特；

遍历 e 的每一个比特$\mathrm{e_i}, \mathrm{i} = 1, \ldots \mathrm{t}$

（c）如果 $\mathrm{e_i} = 0$，首先验证 $\mathrm{z_i} = (w_1^i, r_1^i, w_2^i, r_2^i)$ 中的 $\mathrm{w_1^i}, \mathrm{w_2^i}$ 一个属于 $\{0, \ldots l\}$，另一个属于$\{l, \ldots 2l\}$。而且满足 $\mathrm{c}_1^i = Enc_{pk}(w_1^i, r_1^i)$ 和 $\mathrm{c}_2^i = Enc_{pk}(w_2^i, r_2^i)$。

（d）如果 $\mathrm{e_i} = 1$，验证 $\mathrm{z_i} = (j, w_i, r_i)$ 中的 $w_i \in \{l, \ldots 2l\}$，而且 $c \ \oplus c_j^i = Enc_{pk}(w_i, r_i)$；

（c）若全部验证通过，则可以认为证明者的 $x \in q$。

该算法的结果是概率型的，当验证者验证通过时说明可以有 $1-2^{-t}$ 的概率相信证明者的数值在指定范围内的。

表 3.1 是优化前与改为非交互模式后 range proof 的在 t 值为 40、80、128、256 下的性能对比。从表中可以看出在 t 增大的情况，时间消耗都会不断增长，因为 t 越大需要进行解密和传输的信息量就变多。在 t 值不同时，优化后的版本性能都比初始版本的好，而且随着 t 的增加，优化效果更加明显。

在参与方数量增多的情况下，非交互模式的优势将更加明显，因为证明者只需进行一次证明计算，将证明发送出去即可。而初始版本中证明者将需要和每个验证者进行交互，交互次数与参与者数量成正比。

时间(ms)	40	80	128	256
初始版本	16.40	33.34	46.23	100.31
优化版本	15.88	30.60	42.14	86.32
时间对比(百分比)	96.8%	91.8%	91.2%	86.1%

表 3.1 Range Proof 优化前后性能对比

3.2 密钥刷新算法

基于 3.1 节的密钥生成与签名算法，密钥份额分散在各个参与方手中。在本节的方案中，基于分布式密钥生成的协议，设计支持分布式密钥刷新的算法，用户在部分密钥泄漏的情况下可以对全部或者部分密钥份额进行刷新，保证密钥的动态安全性。

密钥刷新的基本原理如下：

密钥$\mathrm{x} = x_1 \cdot x_2 = x_1 \cdot (x_{21} \cdot x_{22}) = x_1 \cdot [(x_{211} \cdot x_{212}) \cdot (x_{221} \cdot x_{222})] = \cdots$。在进行密钥刷新时要保证主公钥和主私钥不变，即 (x, Q) 不变。根据有限域上的乘法原理，只需乘法双方中一方乘以一个数，另一方乘以这个数的逆元，即可保证乘积不变，即保证了原私钥不变。

下面分别给出密钥局部刷新和密钥全局刷新的具体算法。

3.2.1 密钥局部刷新

密钥刷新的功能可以支持密钥管理系统具有动态安全性。但是当参与方数量很多时，密钥份额全部刷新的时间和计算消耗成本会很高。本方案设计实现了密钥局部刷新的算法，当局部信息泄漏时，可以只对该局部的参与方的密钥份额进行刷新即可而不影响其他参与方的密钥信息。如图 3.3 中，圆形虚线框中的数据不真实存在但是要保证其数值不变，方形虚线框中的数值更新，而保持 x_2 不变

从而保证主私钥 x 不变。

算法接口 $Refresh(P_{21},P_{22})$ 输入要刷新密钥的参与方。参与局部密钥刷新的所有参与方要先进行 coin tossing 的过程，各自随机生成一个随机数 f_i，首先进行 Schnorr 签名发送给其他参与方，验证通过后，每个参与方计算得到自身密钥份额的一个变化量 f ，并相互验证变化量正确，即可进行密钥的更新计算过程，更新完成后将旧的密钥份额丢弃即可。

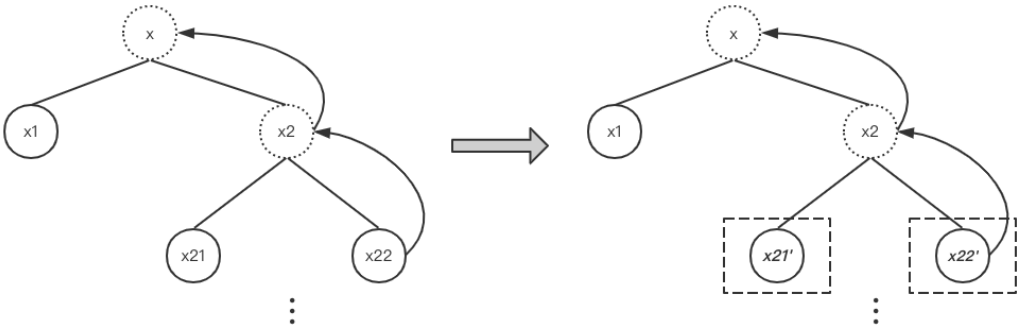


图 3.3 密钥局部刷新

1. P_{21}, P_{22}

P_{21}, P_{22} 运行一次 coin tossing，生成 F_2：

（a）P_{21} 随机生成刷新系数 f_{21}，计算$F_{21} = f_{21} \cdot G$；

（b）P_{22} 随机生成刷新系数 f_{22}，并计算 $F_{22} = f_{22} \cdot G$；

（c）P_{21} 计算 f_{21} 的 Schnorr 签名 ，P_{22}计算 f_{22} 的 Schnorr 签名；

（d）P_{21} 将 F_{21} 和签名发送给 P_{22}，P_{22}将 F_{22} 和签名发送给 P_{21}；

（e）P_{21} 计算 $F_2 = f_{21} \cdot F_{22}$，$P_{22}$计算$F_2 = f_{22} \cdot F_{21}$；

2. P_{21}

（a）P_{21} 计算 $f_2 = HASH(F_2)$ ；

（b）发送 $number = \mathrm{HASH}(f_2)$ 给 P_{22}。

3. P_{22}

（a）P_{22} 计算 $f_2 = \mathrm{HASH}(F_2)$ ；

（b）发送 number = $\mathrm{HASH}(f_2^{-1})$ 给 P_{21} 。

4. P_{21}

（a）P_{21} 验证 P_{22} 发送过来的 number = $HASH(f_2^{-1})$，若不相等则退出并通知 P_{22}；

（b）P_{21} 计算 $x'_{21} = x_{21} \cdot f_2 \mod q$；

（c）完成退出。

5. P_{22}

（a）P_{22} 验证 P_{21} 发送过来的 number = $HASH(f_2)$，若不相等则退出并通知 P_{21}；

（b）P_{22} 计算 $x'_{22} = x_{22} \cdot f_2^{-1} \mod q$；

（c）完成退出。

未刷新前：

$$x_2 = x_{21} \cdot x_{22} \mod q$$

刷新后：

$$x_2 = x'_{21} \cdot x'_{22} = (x_{21} \cdot f_2 \mod q) \cdot (x_{22} \cdot f_2^{-1} \mod q) = x_{21} \cdot x_{22} \mod q$$

所以在上述的密钥局部刷新过程完成后，P_{21} 和 P_{22} 的密钥份额发生了变化，而 P_1 的密钥份额保持原来的数值。而且在密钥刷新后，x_2 保持不变，所以依然可以使用 3.1.2 节的分布式签名算法对消息进行签名，且主公钥和主私钥保持不变。

局部密钥刷新的过程不涉及到 P_1 的密钥份额刷新，所以不需要对 x_1 进行重新加密和零知识证明。每个参与方只需要对生成的随机数 f_i 对相关信息（如“密钥局部刷新”字符串）进行 Schnorr 签名并发送，接着验证相关参与方计算出来的变化量是否正确即可更新密钥份额。所以局部刷新算法的计算复杂度是 $O(n)$ 次 Schnorr 签名与验证。

3.2.2 密钥全局刷新

极端情况下，攻击者已经攻击获取得到了大部分的密钥份额且还在进行攻击，为了在这种情况下保护区块链账户的密钥安全，就有了刷新全部密钥份额的必要性。密钥全局刷新是指将所有参与方的密钥份额都进行更新，每个密钥份额在刷新完成后都更新为另一个不同的数值。密钥全局刷新的时间和计算消耗会很高，但是带来最高的密钥动态安全性。因为密钥生成时是以二叉树的形式展开的，在密钥更新时就需要从二叉树的叶子结点出发，每两个参与者（一个节点的两个子节点）生成随机值计算得到父节点的随机值，像 Merkle 树一样不断往上计算直

到得到 x_1 的兄弟节点 x_2 的随机值 f_2，与 x1 的随机值 f_1 结合计算得到整个密钥更新过程 x_1 与 x_2 的偏移量 f，得到偏移量 f 之后，从 x_2 节点开始遍历其所有子节点，直到所有子节点密钥更新完毕。全局密钥刷新的过程才可以完成退出。

密钥刷新的根本要求是主私钥 x 和主公钥 Q 不变，也即图 3.4 中显示的树根中的虚拟值 x 不变，二组成 x 的所有子节点的值（方形虚线框中）都将发生变化。下面给出密钥全局刷新的具体算法步骤，如图 3.4:

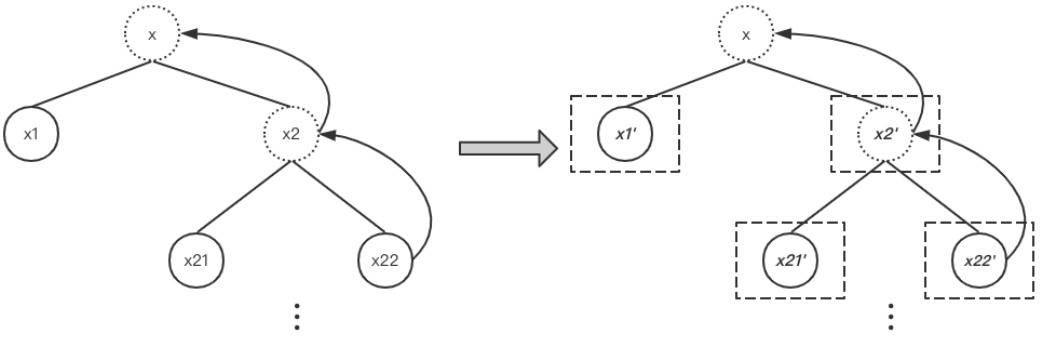


图 3.4 密钥全局刷新

算法接口 $RefreshAll()$ 输入要刷新密钥的参与方。

1. $\mathbf{P_1}$

（a）P_1 使用 x_1 对消息签名，发送签名给其他参与方，表示要进行密钥全刷刷新。

2. $\mathbf{P_{21}, P_{22}}$

P_{21}, P_{22} 运行一次 coin tossing，生成 F_2：

（a）P_{21} 随机生成刷新系数 f_{21}，计算$F_{21} = f_{21} \cdot G$；

（b）P_{22} 随机生成刷新系数 f_{22}，并计算 $F_{22} = f_{22} \cdot G$；

（c）P_{21} 计算 f_{21} 的 Schnorr 签名 ，P_{22}计算 f_{22} 的 Schnorr 签名；

（d）P_{21} 将 F_{21} 和签名发送给 P_{22}，P_{22}将 F_{22} 和签名发送给 P_{21}；

（e）P_{21} 计算 $F_2 = f_{21} \cdot F_{22}$，$P_{22}$计算$F_2 = f_{22} \cdot F_{21}$；

（f）将各自计算得到的 F_2 和签名信息 $(prove, 2, F_2, F_{2i}, f_{2i}), i \in [1,2]$ 输入给 $\mathcal{F}_{zk}^{\mathrm{RDL}}$，并发送给$P_1$。

3. $\mathbf{P_1}$

（a）P_1 从 $\mathcal{F}_{zk}^{\mathrm{RDL}}$ 收到 $(proof, 2, F_2, F_{2i}), i \in [1,2]$，验证其他参与方传输过来的

Schnorr 签名以及检查 F_2 是否相等。若没有收到，过程错误退出。若有一处验证不通过，过程错误退出；

（b）P_1 随机选取 $f_1 \in \mathbb{Z}_q$，并计算$\mathrm{F}_1 = f_1 \cdot \mathrm{G}$；

（c）计算 $F = f_1 \cdot F_2$；

（d）计算 $f = HASH(F)$；

（e）计算 $\mathrm{x}_1' = x_1 \cdot f \ mod \ q$。如果 x_1' 不小于 $\mathrm{q}/3$，返回步骤（b）重新选择 f_1；

（f）使用 x_1 对 HASH(F) 做 Schnorr 签名。将 F 和签名信息发送给其他参与方。等待其他参与方的确认回复，若全部接受确认进行下一步，否则提示错误退出；

（g）P_1 随机生成规定长度的 Paillier 密钥对(pk, sk)，并使用 Paillier 加密算法计算$c_{key} = Enc_{pk}(x_1')$；

（h）P_1 将 $(prove, 1, N, (p_1, p_2))$ 输入到 $\mathcal{F}_{zk}^{\mathrm{RP}}$ 证明生成的 Paillier 密钥 (pk, sk) 是符合配置设置的，其中$\mathrm{pk} = \mathrm{N} = \mathrm{p}_1 \cdot p_2$。并将 c_{key} 发送给其他参与方。

4. $\mathbf{P_1}$

P_1 向其他全部参与者证明 $(c_{key}, pk, Q_1) \in L_{PDL}$。该步骤的意义是向其他参与者以零知识证明的方式证明 $\mathrm{c_{key}}$ 是 pk 对 Q_1 的离散对数 x_1 加密的密文，而且 P_1 知道 x_1 和 pk 对应的 Paillier 私钥 sk。该步骤与分布式密钥生成阶段一致，因为 x_1 更新所以需要重新证明密钥与密文的关系。

5. $\mathbf{P_{21}}, \mathbf{P_{22}}$

除 p1 外的所有参与方要进行下面的三项验证，并且保证每项验证都验证通过。

（a）从 $\mathcal{F}_{zk}^{\mathrm{RDL}}$ 收到 $(proof, 1, F_1)$ 和从 $\mathcal{F}_{zk}^{\mathrm{RP}}$ 收到 $(proof, 1, N)$ 并都验证通过；

（b）P_1 向每个参与者证明 $(c_{key}, pk, Q_1) \in L_{PDL}$ 通过；

（c）$\mathrm{pk} = \mathrm{N}$ 的大小在设置范围内。

接下来各个参与方更新自己的密钥份额。

6. $\mathbf{P_{21}}$

（a）P_{21} 计算 $\mathrm{f}_2 = HASH(F_2)$；

（b）发送 $number = \mathrm{HASH}(f_2)$ 给 P_{22}。

7. P_{22}

（a）P_{22} 计算 $f_2 = HASH(F_2)$；
（b）发送 number = $HASH(f_2^{-1})$ 给 P_{21}。

8. P_{21}

（a）P_{21} 验证 P_{22} 发送过来的 number = $HASH(f_2^{-1})$，若不相等则退出并通知 P_{22}；
（b）P_{21} 计算 $x'_{21} = x_{21} \cdot f_2 \cdot f^{-1} \ mod \ q$；
（c）向 P_1 发送数据正确确认信息；
（d）完成退出。

9. P_{22}

（a）P_{22} 验证 P_{21} 发送过来的 number = $HASH(f_2)$，若不相等则退出并通知 P_{21}；
（b）P_{22} 计算 $x'_{22} = x_{22} \cdot f_2^{-1} \ mod \ q$；
（c）向 P_1 发送数据正确确认信息；
（d）完成退出。

未刷新前：

$$x = x_1 \cdot x_2 = x_1 \cdot (x_{21} \cdot x_{22} \ mod \ q)$$

刷新后：

$$\begin{aligned} x = x'_1 \cdot x'_2 &= x'_1 \cdot (x'_{21} \cdot x'_{22}) \\ &= (x_1 \cdot f) \cdot (x_{21} \cdot f_2 \cdot f^{-1}) \cdot (x_{22} \cdot f_2^{-1}) \ mod \ q \\ &= x_1 \cdot x_{21} \cdot x_{22} \ mod \ q \end{aligned}$$

所以在上述的密钥全局刷新过程完成后，P_1, P_{21} 和 P_{22} 的密钥份额发生了变化。而且在密钥刷新后，x 保持不变，所以依然可以使用 3.1.2 节的分布式签名算法对消息进行签名，且主公钥和主私钥保持不变。

全局密钥刷新的过程中每个参与方都需要用自身随机生成的密钥份额 f_i 对相关信息（如“密钥全局刷新”字符串）进行 Schnorr 签名，向其他全部参与方证明其拥有该随机数，接着验证相关参与方计算出来的变化量是否正确，若全部验证通过即可更新密钥份额。所以全局刷新算法的需要进行 $O(n)$ 次 Schnorr 签名与验证。整个过程涉及到 P_1 的密钥份额 x_1 的变化，所以需要对新的密钥份额 x'_1 重新加密，P_1 重新生成 Paillier 公私钥对并对 x'_1 进行一次 Paillier 加密，接着跟分布式密钥生成阶段一致分别进行了两次不同的证明，一次证明重新生成

的 Paillier 的公钥在合理范围内（为了更高的安全性，Paillier 的公私钥对也重新生成，如果为了节省计算成本，也可以直接使用先前步骤生成的密钥对进行计算），一次以零知识的方式进行了 L_{PDL} 的证明，在进行 L_{PDL} 证明的同时对 x_1' 进行了一次 Range Proof 证明 x_1 在合理范围内。

3.3 算法性能分析与对比

在 3.1 和 3.2 小节中分别详细说明了分布式密钥生成、分布式签名以及局部和全局的密钥刷新算法，因为是采用安全多方计算的思路，每个参与方在算法中都需要各自生成密钥份额或者随机数，进而要与其他参与方进行多轮问交互和计算。为了保证算法的安全性和完整性，算法流程中需要使用到随机数生成、Schnorr 签名、密码学承诺、Paillier 加解密和零知识证明等等基础密钥学算法.

表 3.2 中将各个算法中使用到基础密码学过程统计出来，以便对算法进行复杂度和性能上的对比分析：

	密钥生成	签名	密钥全局刷新	密钥局部刷新
Schnorr 签名	n	2n	n	n
Paillier 加解密	1	2n	1	-
Paillier 同态计算	-	2n	-	-
零知识证明	**2n**	**-**	**2n**	**-**
Range Proof	**n**	**-**	**n**	**-**

表 3.2 算法复杂度对比表

如表 3.2 统计的信息，每个算法流程中都需要经过与参与方数量 n 成正比的计算和通信过程。在零知识证明过程中，验证者与证明者进行两轮交互，交互过程中需要使用 Paillier 解密算法和同态计算并结合 challenge 和密码学承诺进行证明。因为该零知识证明过程是交互式的，所以信息的通信量随着参与者数量 n 增加而线性增长。密钥生成阶段需要进行 n 次 Schnorr 签名和验证，一次 Paillier 加密计算，一次 Range Proof 证明计算和 n 次 Range Proof 验证计算，最消耗时间的是零知识证明的过程，零知识证明的计算和时间消耗占据了将近分布式密钥生成阶段总时间的 98%。同样的，在密钥全剧刷新的过程中，首先先进性密钥份额变化量的计算过程，每个参与方生成随机数，并共享给其他参与方，在计算得到新的密钥份额 x_1 后 P_1 重新开始对密钥份额的密文 c_{key} 进行证明与分布式密钥生成阶段的步骤一致，所以密钥全剧刷新的时间消耗会密钥生成阶段的多出密钥份额变化量的步骤。密钥局部刷新过程只需生成随机数并通信共享给其他参与方即可，时间消耗最少。分布式签名过程中，每个参与方需要生成随机数 k_i 和 ρ_i，

并通过 Paillier 的加密算法和同态计算将随机数信息加密发送出去，P_1 获取到随机数信息后即可计算出签名信息。所以签名过程不需要零知识证明的过程，使用 Schnorr 签名和 Paillier 计算这些计算复杂度和时间消耗很小的步骤即可完成。

在三个参与方的场景下分布式密钥生成的时间消耗是分布式签名的 133 倍，签名所需平均时间 20ms，密钥生成需要平均时间 2.67s。当参与方的数量增加时，所有算法的时间消耗都随之增加，安全性也随之加强。所以在性能与安全性上的选择是一个权衡，需要有更高的安全性那么计算时时间消耗就会增加，想要有更高的计算效率那么就必须适当降低安全性的要求。

3.4 密钥存储与恢复

现有的密钥备份与恢复方法主要有：密钥热存储、离线冷存储、平台托管、硬件 Ukey 等。密钥热存储方法是将密钥直接存储在应用中，应用需要签名或者解密数据时直接从内存中获取密钥进行操作，一般应用是连接网络的，如果应用本身没有做好防范攻击的措施，密钥就很容易被窃取或者误操作将密钥泄漏出去；离线冷存储方法是密钥不存储在有任何网络连接的地方，比如将密钥抄写在纸上放进保险箱中，或者将密钥的其他形式记录下来比如助记词，离线冷存储的密钥在需要使用时再手动输入密钥可以保证密钥不会泄漏，但是一旦记录密钥的介质丢失损毁，将不可能再找回，且每次输入多位数密钥很不方便；平台托管方法是将密钥直接托管给中心机构保管，当需要使用时从托管平台获取，托管方式最大的问题是密钥直接暴露给托管机构，存在很大的安全隐患；硬件 Ukey 等硬件方法是将密钥硬编码到硬件中，应用需要有读取硬件密钥的接口，使用时必须要链接硬件设备，该方法有较高的安全性，但是实际使用很不方便且有易丢失的风险。

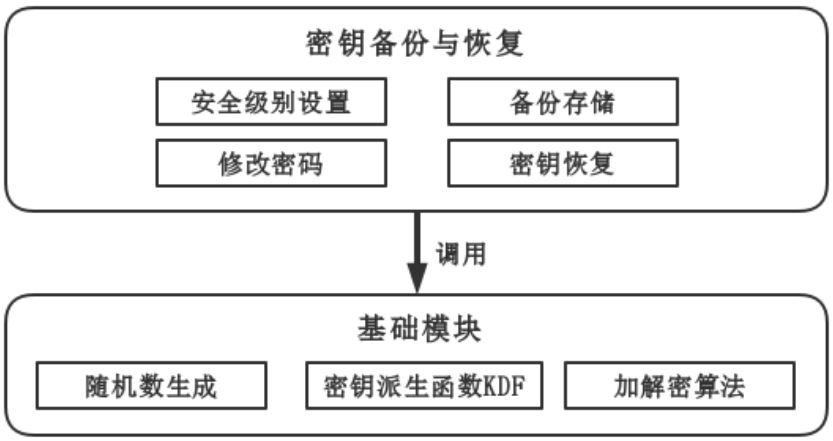


图 3.5 密钥存储模块图

综合以上问题本节设计了一种基于集成加密模式的密钥存储与恢复方法，用来存储备份分布式密钥生成阶段生成密钥份额。本方法是一种基于集成加密技术的密钥备份与恢复方法，为解决密钥容易丢失和泄漏等问题，提供了不同的安全等级以实现安全易用的密钥备份和密钥恢复的方案。如图 3.5 所示具体技术方案分为以下四个部分：

1. 可配置安全级别模块：

对密钥 privateKey 进行加密需要有加密函数 Enc() 和加密密钥 encKey，密码学中不同加密算法的安全级别不同，安全级别也与加密密钥 encKey 有关。所以在存储密钥时进入集成加密模块之前，设置了一个可配置的安全级别选择模块，用户在加密之前选择需要的安全级别 level，相应的安全级别配置了不同的加密算法和用户需要输入的信息数量和密码口令的复杂度要求，每个级别对应一个算法四元组：

（level、Enc| Mode、Complex、Number）

Level 表示安全级别，级别越高安全强度越大；

Enc | mode 表示加密算法和加密时使用的模式；

Complex 表示密码复杂度，当安全强度越高时要求用户在使用时输入相应强度的密码口令；

Number 表示所需个人信息数量，安全强度与个人信息有关，安全强度越高需要的信息数量越多。表 3.3 是安全级别选择表的示例，总共有个 n 个安全级别，每个级别中详细说明了要使用的加密算法和加密模式，定义了用户需要输入的密码口令强度和需要的用户身份信息数量。

安全级别	加密算法|加密模式	密码复杂度	个人信息数量
Level1	AES|CBC	低	2
Level2	AES|GCM	中	3
…	…	…	…
Leveln	ECIES	高	4

表 3.3 安全级别选择表

选择安全级别后，集成加密模块中的所有计算都要满足安全配置要求。

2. 集成加密模块：

每个用户可以备份任意多个密钥 privateKey，每次备份存储一个密钥时需要根据选择的安全等级输入满足安全等级要求的信息，根据用户输入的生物特征信息 informations 和隐私信息 secrets，比如生日、邮箱、密码等信息，将上述信息

作为密钥派生函数 KDF 的输入，输出一个用于集成加密的密钥 encKey 和用于消息认证的密钥 macKey（生成消息认证码的过程不是必须的流程，如果不做消息认证过程可以不生成 macKey）。加密过程如图 3.6 的时序图，图中的托管机构是存储已加密密钥的位置，见下一节存储模块描述，计算公式如下：

$$encKey，macKey = KDF（informations，secrets）$$

根据用户的生物特征信息和私密信息得到加密密钥 encKey 后，将用户要备份存储的密钥 privateKey 通过加密算法进行加密，加密算法根据用户需要可以任意选择，包括对称加密和非对称加密。

选择对称加密时直接使用 encKey 加密：

$$encPrivateKey = Enc（privateKey，encKey）$$

选择公钥加密算法时，通过 encKey 获得 encPublicKey：

$$encPublicKey = getPublic（encKey）$$

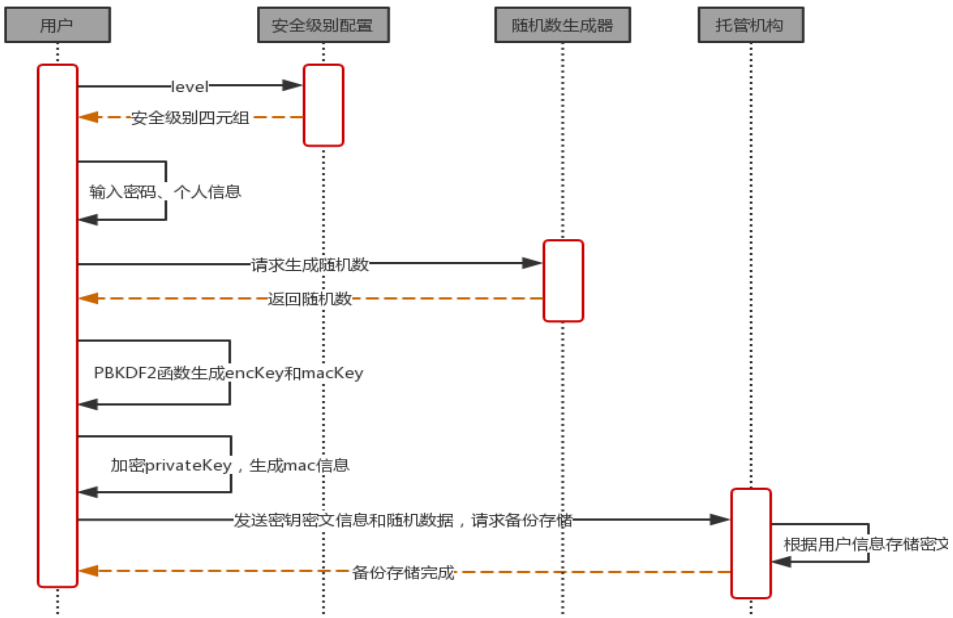
再利用 encPublicKey 对 privateKey 进行加密：

$$encPrivateKey = Enc（privateKey，encPublicKey）$$

加密完成后，如果要进一步对 encPrivateKey 进行消息认证，则需要下一步的 MAC 过程，生成消息认证信息 tag：

$$tag = MAC（encPrivateKey，otherinfo，macKey）$$

其中，otherinfo 是可以根据实际需要指定的其他信息。到此，集成加密模块图的过程结束，得到的 encPrivateKey 和 tag 组合在一起以及一些解密时需要的随机信息传输给存储模块进行备份存储。



3.6 加密存储流程时序图

3. 存储模块：

集成加密模块生成的 encPrivateKey、KDF 的临时参数和安全级别信息等都需要进行存储。因为这些信息中没有足够的能够解密得到 privateKey 的参数，所以可以将这些信息委托给第三方存储，比如密钥托管机构、分布式存储系统等等，也可以用户自行进行冷、热存储，只要保证上述信息不会丢失即可方式不限。存储模块可以在用户自定义的模块中定义，为用户提供一个接口，由用户根据实际情况具体实现。

4. 密钥恢复模块：

当需要使用密钥进行解密或者签名时，需要将加密存储过的密钥解密恢复出来。首先从存储中获取 encPrivateKey、KDF 的临时参数和安全级别等信息，根据安全等级输入满足安全等级要求的信息，根据用户输入的生物特征信息 informations 和隐私信息 secrets 通过密钥派生函数 KDF 得到集成加密的密钥 encKey 和用于消息认证的密钥 macKey，并通过安全要求检验。接着验证 MAC 信息的合法性，检验通过则说明 encPrivateKey 正确，再使用 encKey 解密：

$$privateKey = Dec(encPrivateKey, encKey)。$$

得到 privateKey 后直接传输到密码模块中即可。

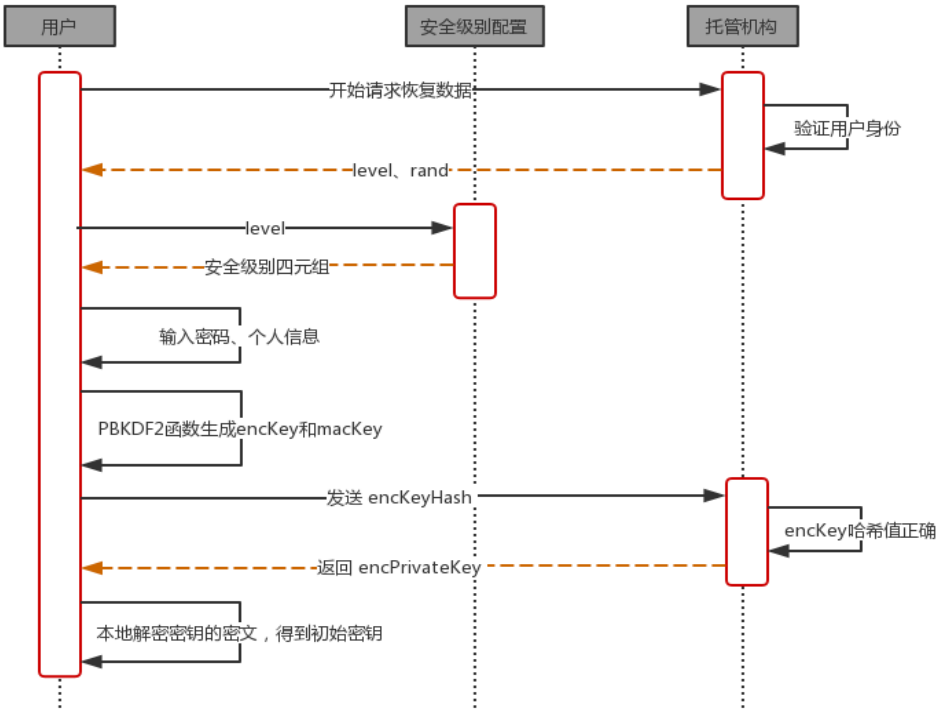


图 3.7 解密流程时序图

5. 修改密码模块：

当需要修改密码时，需要先获取得到所有的初始密钥明文，过程如图 3.7 所示。获得到密钥明文后要重新进行加密，用户重新选择相应的安全级别，输入必要的信息，经过图 3.6 的过程得到新的的密码加密的密钥。在存储模块中找到旧密钥的存储位置，更新覆盖旧的密钥密文数据，修改密码的操作过程完成。为了防止新密码忘记，可以将旧数据存储存储下来，一段时间后再将就数据删除，兼容更多的突发异常情况。

3.5 本章小结

本章主要介绍了安全多方计算 ECDSA 的分布式密钥生成、分布式签名和分布式密钥刷新三个改进的密码学算法，这些算法是密钥管理系统的核心算法。除了以上三个算法，还对分布式密钥生成阶段的 Range Proof 进行优化，将交互式的零知识证明过程改进为非交互式，加快分布式密钥生成的计算性能，并给出了改进前后的性能对比。本章最后设计实现了基于集成加密模块的密钥存储与恢复方法，可以根据用户选择的加密安全强度适配相应的加密算法和密码口令强度，用户无需接触到密钥份额本身，只需使用密码口令即可使用。以上为密钥管理的安全性和易用性给出了相关的算法设计。下面一章介绍如何使用以上算法，设计成为一个高可用的密钥管理系统。

第四章 系统设计和实现

区块链系统的安全性是基于公钥密码体系的椭圆曲线密码学安全来保证的，椭圆曲线密码学与 RSA 相比有安全性高和密钥长度短的优点。现在最被接受并且广泛使用的是 256 比特长度的密钥长度，比如 Prime256V1 和 Secp256k1。

基于上述章节的需求分析与算法设计，本章采用 Golang 语言设计实现了一个基于安全多方计算的密钥管理系统（Blockchain-KMS），本系统采用传统的四层结构，用解耦合、插件化的架构设计以及使用多种通信协议进行参与方之间的信息传输来为用户提供密钥管理的服务，为区块链账户用户和区块链应用提供一个安全易用的密钥管理工具。该密钥管理系统可以以 SDK 或者 Library 的方式嵌入到应用代码中，是一个基础通用的密钥管理库，也可以作为一个独立的应用为用户提供密钥使用和存储的功能，比如建立托管服务并为用户提供移动端的 APP 使用。

4.1 系统技术架构

密钥管理系统采用传统的四层结构：数据存储层，网络通信层，密码服务层，API 服务层。如图 4-1 所示：

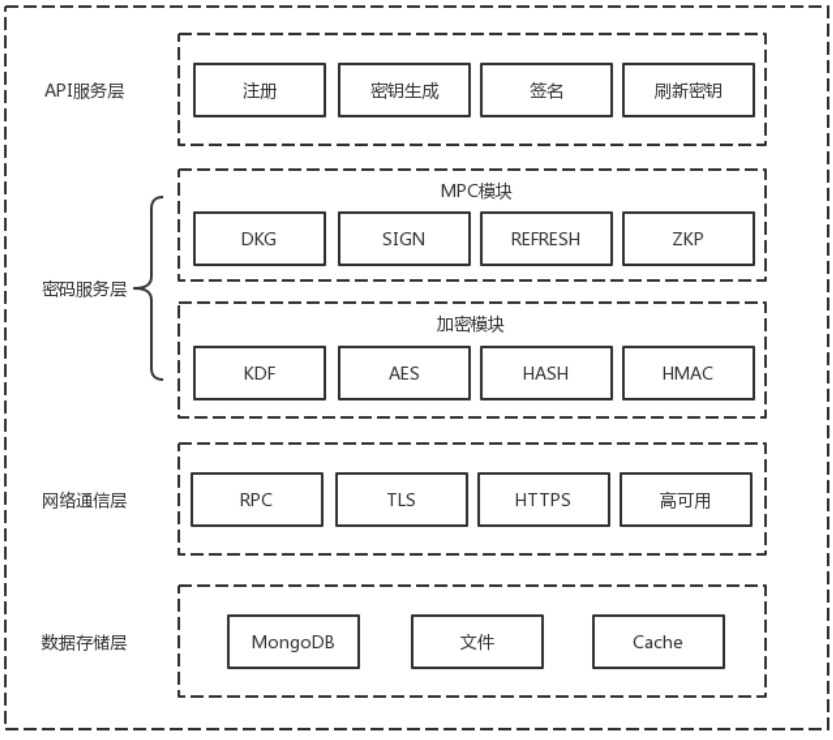


图 4.1 密钥管理系统架构图

4.1.1 数据存储层

数据存储层负责备份存储密钥信息。用户在密钥生成阶段和密钥刷新阶段获得的密钥份额经过集成加密模块加密后，相应的密文信息会根据账户 ID 和密钥 kid 存储在数据模块中，本文使用的数据存储数据库是关系型数据库 MySQL，数据库中存放了两类不同的数据：不同安全等级的加密算法信息、加密密钥的加密算法和加密后的密文信息。当该系统被创建为一个密钥托管服务时，会有很大的用户数据量和账户、密钥信息，在 MySQL 中创建索引，在数据量较大的情况下加快数据的查询速度。表 4.1 描述了数据库中存放的数据详细信息。

表 ID	说明	备注信息
CryptoInfo	加密算法信息表	存储相应安全等级的加密算法信息
KeyInfo	密钥备份表	存放所有加密后的密钥信息

表 4.1 数据库表

CryptoInfo 数据表负责存储相应安全等级的加密算法信息。表 4.2 详细说明了数据表中的每个字段。

元素名称	字段名称	类型及长度	必填	主键	相关说明
安全强度	Level	INT	是	是	安全强度等级
加密算法	Enc	CHAR(64)	是	否	要使用的加密算法
加密模式	Mode	CHAR(32)	否	否	加密时使用的加密模式
口令复杂度	Complex	INT	是	否	密码口令的复杂度要求
身份信息数量	Number	INT	是	否	加解密时所需的用户身份信息数量

表 4.2 CryptoInfo 表设计

KeyInfo 负责存放所有加密后的密钥信息。表 4.3 详细说明了数据表中的每个字段。

元素名称	字段名称	类型及长度	必填	主键	相关说明
ID	ID	CHAR(64)	是	是	加密信息的 ID，由账号和相应 kid 组成
安全强度	Level	INT	是	否	信息加密时使用的安全强度

数据	Data	CHAR(512)	是	否	密文数据

表 4.3 KeyInfo 表设计

数据库的具体配置和使用方式取决于用户设计，可以在这个设计的基础上加强数据库的安全性和容灾性。

数据存储层中适配了缓存机制，使用最近最少使用算法（Least Recently Used，LRU），将使用频率较高的数据缓存在内存中，加快签名和密钥刷新的性能。

如果该系统是在轻客户端上面使用的，比如在移动设备手机上，传感器设施上等不方便建设数据库的地方，系统提供了文件存储的功能，因为这些设备需要存储的信息量不会太多，可以将每个加密的信息存在文件中，需要时直接从文件上读取即可。

4.1.2 网络通信层

网络通信层负责在安全多方计算阶段与其他参与方进行数据通信。为了适配不同业务场景使用到的通信协议，系统在 TCP 协议上统一封装了 RPC、TLS 和 HTTPS/HTTP 等网络协议。用户在初始化密钥管理服务时通过配置文件设置所需的协议即可。同时为了适配不同系统平台的编码格式，统一使用 Google 高效的压缩编码格式 Protocol Buffers[44]进行数据的传输。每种协议都封装如图 4.2 的统一接口：

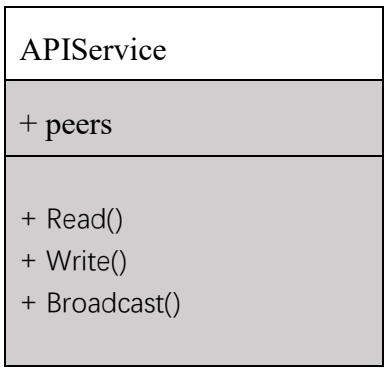


图 4.2 网络通信接口说明

因为网络传输原因，可能丢包或者网络连接断开。为了提高安全多方计算的成功率，在网络通信层添加了高可用的措施，当网络连接断开时自动重连，当长时间网络连接上没有收到消息时重新发送请求，在重新请求三次后仍然失败则通知上层用户，需要由用户排查其他参与方的服务状态。

4.1.3 密码服务层

密码服务层采用插件化的形式为 API 服务层提供加密服务。密码服务层分为两个模块：集成加密模块和 MPC 模块。集成加密模块中包含了所有会使用到

的 KDF、哈希算法、对称加密算法等等为备份密钥份额提供保障。加解密时根据用户选择的安全强度配置判断用户密码的安全强度并接受足够数量的用户身份信息，安全强度不匹配时会返回提示信息，接着选取相应的 KDF 算法和加密算法加密保存密钥份额信息；MPC 模块是支持多方计算 ECDSA 的核心模块，包含密钥生成、签名和密钥刷新的算法，该模块是密钥管理系统的核心算法部分。用户创建区块链账户需要使用分布式密钥生成算法，已有区块链账户发起交易需要使用分布式签名算法，在紧急情况下需要调用分布式密钥刷新接口对密钥份额进行更新，保护区块链账户。MPC 模块的运作依赖于网络通信模块，MPC 参与方之间需要保证网络通信正常，才能在规定时间内收到相关的证明信息和交互信息，从而保证 MPC 算法的正确执行。

插件化的架构方便用户添加自主化的密码模块，用户只要实现了定义好的接口就能在加密密钥时使用自主实现的算法，比如用户可以使用 TEE 环境存储密钥信息，让用户有更灵活的使用方式。设计成一个开放的框架有利于适配现有的传统行业中的密钥保护模块，比如 HSM（Hardware Security Modules）和各种硬件设施。

4.1.4 API 服务层

API 服务层是用户直接使用到的，为用户提供的上层接口。包含初始化配置、密钥生成、签名、密钥刷新和修改密码等接口。用户在实例化密钥管理系统后，可以直接在业务的流程中调用上述接口。类图如图 4.3 所示，在 4.2.2 节会详细说明这些 API 接口的具体定义。

APIService
无
+ Init(config Config)error + KeyGen(number int, keyID string, params interface{})（string，error） + Sign(message string, keyID string, params interface{}) (string, error) + Refresh()(keyID string, params interface{}) error + RefreshAll()(keyID string, params interface{}) error + UpdatePassword(keyID string, oldParams interface{}, newParams interface{}) error

图 4.3 API 接口说明

4.2 系统接口说明

本小节详细说明密码服务模块和 API 服务层中的接口定义。

4.2.1 密码服务模块接口

为了让用户可以在密码服务模块中添加自主实现的密钥管理功能，在密码服务模块中定义了统一的接口，密码服务模块是对密钥份额进行加解密的父类，并不具体实现接口中的实例函数。系统中会内置实现默认的加解密模块函数，用户只要实现了该接口设计，就可以接入用户自定义的管理功能，若用户不自定义则直接使用默认实现的密码模块。密码服务模块接口类图如图 4.3 所示：

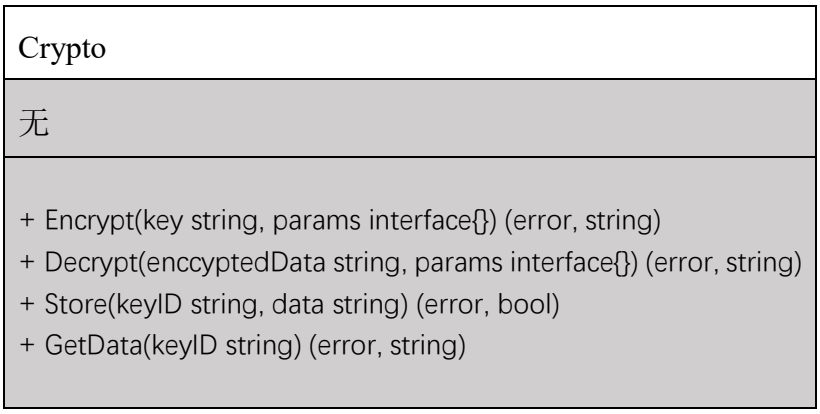


图 4.3 密码服务模块类图

密码服务模块插件化的接口详细说明：

Encrypt(key string, params interface{}) (error, string) 的详细设计如表 4.4 和表 4.5 所示。输入要加密的密钥份额原文的十六进制字符串和加密时需要使用到的参数，如安全强度、加密算法和加密模式、密码口令、身份信息等等。加密完成后输出两个值，正常情况下输出密钥的密文，error 为空，如果有发生加密错误则密文为空，error 为错误信息。

输入

参数	类型	是否必须	举例
Key	密钥十六进制字符串	是	“1a47f7f962b13e60c4741bcb83e710ca4f328c8c5d4847a2b2189ee51ae0119e”
Params	加密时的必要参数	根据实现情况而定	1，“AES”，“GCM”，“123456”，“邮箱”

表 4.4 Encrypt 接口输入信息

输出

类型	举例
String	“2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824”

Error	Null | “加密失败”

表 4.5 Encrypt 接口输出信息

Decrypt(enccyptedData string, params interface{}) (error, string) 的详细设计如表 4.6 和表 4.7 所示。输入要解密的密文信息和解密时需要使用到的参数，如安全强度、加密算法和加密模式、密码口令、身份信息等等。加密完成后输出两个值，正常情况下输出密钥的原文，error 为空，如果有发生解密错误则密文为空，error 为错误信息，如“解密口令强度不匹配”。

输入

参数	类型	是否必须	举例
enccyptedData	密文十六进制字符串	是	“2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824”
Params	解密时的必要参数	根据实现情况而定	1，“AES”，“GCM”，“123456”，“邮箱”

表 4.6 Decrypt 接口输入信息

输出

类型	举例
String	“1a47f7f962b13e60c4741bcb83e710ca4f328c8c5d4847a2b2189ee51ae0119e”
Error	Null | “解密失败”

表 4.7 Encrypt 接口输出信息

Store(keyID string, data string) (error, bool) 的详细设计如表 4.8 和表 4.9 所示。输入要存储的密钥密文信息和该信息的 keyID。存储完成后输出两个值，正常情况下为 true，error 为空，如果有发生存储错误则输出 false，error 为错误信息，如“存储出错”。

输入

参数	类型	是否必须	举例
keyID	ID 字符串	是	“Alice_1”
data	要存储的密文信息	根据实现情况而定	“2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824”

表 4.8 Store 接口输入信息

输出

类型	举例
bool	true
Error	Null | “存储出错”

表 4.9 Store 接口输出信息

GetData(keyID string) (error, string) 的详细设计如表 4.10 和表 4.11 所示。输入要获取的信息的 keyID。查询获取完成后输出两个值，正常情况下输出密钥的密文字符串，error 为空，如果有发生查询错误则密文为空，error 为错误信息，如“keyID 未找到”，

输入

参数	类型	是否必须	举例
keyID	ID 字符串	是	“Alice_1”

表 4.10 Store 接口输入信息

输出

类型	举例
bool	true
Error	Null | “未找到次 ID”

表 4.11 Store 接口输出信息

用户在使用时只需实现上述 4 个接口，就能实例化 Crypto 类。并且在运行时会使用用户自定义的接口函数来加解密和存储获取密钥信息。

在设计系统逻辑时，可以将恢复过程当作是签名算法或者解密的算法的一部分，每次恢复出来处密钥只做临时存储，每次签名或解密结束后释放临时存储，避免长期存储带来的安全隐患。

4.2.2 API 服务接口

为了让用户在使用密钥管理系统时不需要关注底层算法和架构设计的具体实现。更方便后续对密钥管理系统做升级优化，当用户已经将该管理系统应用到业务上面时，如果对接口进行任意的变更，都可能引起业务系统的崩溃和资产损失。所以对用户统一开放了 6 个接口，每个接口详细定义了输入输出的类型和含义及相应的样例以及使用的场景。

下面根据图 4.2 说明 API 副层接口的详细设计：

Init(config Config)error 的详细设计如表 4.12 和表 4.13 所示。输入初始化时密钥管理系统的配置参数。初始化完成后正常情况下 error 为空，如果有发生加

密错误则输出 error 为错误信息，如“config invalid”。 Init 接口在调用密钥管理服务之前必须先调用，初始化配置信息。Config 的结构体定义：前半部分定义了需要使用的椭圆曲线，输入合法的椭圆曲线参数；后半部分说明在 KeyGen 阶段零知识证明算法的安全参数。

```
{
  "P":"",        // 有限域的阶
  "N":"",        // 生成元的阶
  "B":"",        // 椭圆曲线参数
  "Gx": "",
  "Gy":"",       // 生成元的 x 轴 y 轴
  "BitSize":256, // 有限域的大小

  "NPaillierBits": 2048, //Paillier 密钥长度
  "NthRootSecBits":128,  //证明 Paillier 密钥的安全参数数
  "RangeSecBits": 40     //Range proof 的安全参数
}
```

输入

参数	类型	是否必须	举例
config	Config	是	配置信息

表 4.12 Init 接口输入信息

输出

类型	举例
Error	Null	“config invalid”

表 4.13 Init 接口输出信息

KeyGen(number int, keyID string, params interface{})（string，error）的详细设计如表 4.14 和表 4.15 所示。输入要生成的密钥 ID 和参与方个数，以及要加密存储密钥的参数。KeyGen 完成后正常情况下 error 为空，并且返回主公钥字符串，如果有发生错误则输出 error 为错误信息，如“network error”。当要在区块链上创建新账户时，就需要有一个或多个公钥信息，KeyGen 是创建新的公私钥信息的接口，获取到主公钥信息后，即可到区块链上进行账号注册或直接发送交易。

输入

参数	类型	是否必须	举例
number	Int	否	默认为 1
keyID	ID 字符串	是	“Alice_1”
Params	加密时的必要参数	根据实现情况而定	1，“AES”，“GCM”，“123456”，“邮箱”

表 4.14 KeyGen 接口输入信息

输出

类型	举例
String	“b2225c7b859e0b3f14535dc7e5df9f08d73428b622c27 0e21ad9b1eff07e174169dbdde92478bfd34081e7425639 1558090e47ca1aa37bb4f4f56c020b00a127”
Error	Null | “network error”

表 4.15 KeyGen 接口输入信息

Sign(message string, keyID string, params interface{})(string, error) 的详细设计如表 4.16 和表 4.17 所示。输入要签名的信息 message，使用的密钥 ID，以及解密存储密钥的参数。签名完成后正常情况下返回签名值（R||S）和空 error，如果有发生错误则输出 error 为错误信息，如“invalid key”。在生成公私钥信息后，就可以与其他参与者使用密钥进行交互对交易进行签名，得到签名值再将交易发送到区块链系统上。

输入

参数	类型	是否必须	举例
message	String	是	“hello”
keyID	ID 字符串	是	“Alice_1”
Params	加密时的必要参数	根据实现情况而定	1，“AES”，“GCM”，“123456”，“邮箱”

表 4.16 **Sign** 接口输入信息

输出

类型	举例
String	“909dac8768a859ea17d9f89a36a5aa3f61a6dde4d8e19894f80 05b37a7c0b6820192e39d1c436e589b184afd7aeb607f3f57216f 703b3b83d574620b61489627”
Error	Null | “invalid key”

表 4.17 **Sign** 接口输入信息

Refresh()(keyID string, params interface{}) error 该接口只进行局部的密钥刷新，调用该接口的参与者通过网络层通知其密钥生成二叉树上的兄弟节点进行密钥刷新的过程。详细设计如表 4.18 和表 4.19 所示。输入要刷新的密钥 ID，以及解密存储密钥的参数。刷新密钥完成后正常情况下返回 error 为空，如果有发生错误则输出 error 为错误信息，如“mpc error”。当用户察觉到自身的密钥份额

可能泄漏，则第一时间内发起 Refresh 的调用，与在密钥生成二叉树上互为兄弟节点的参与者进行局部密钥刷新。

输入

参数	类型	是否必须	举例
keyID	ID 字符串	是	“Alice_1”
Params	加密时的必要参数	根据实现情况而定	1，“AES”，“GCM”，“123456”，“邮箱”

表 4.18 **Refresh** 接口输入信息

输出

类型	举例
Error	Null | “mpc error”

表 4.19 **Refresh** 接口输入信息

RefreshAll()(keyID string, params interface{}) error 该接口进行全局的密钥刷新，调用该接口的参与者通过网络层通知其他的全部参与者进行密钥刷新的操作。输入输出与 Refresh 接口的表 4.16 和表 4.17 一致。密钥全局刷新在特殊情况下才会发起，比如当大部分密钥已经泄漏，那么发起全局密钥替换并加强每个密钥份额的安全措施至关重要。

UpdatePassword(keyID string, oldParams interface{}, newParams interface{}) error 的详细设计如表 4.14 和表 4.15 所示。输入要更换密码的密钥 ID 以及解密存储密钥的参数和新的加密密钥的参数。UpdatePassword 完成后正常情况下 error 为空，如果有发生错误则输出 error 为错误信息，如“password invalid”。更新密码口令是日常安全防护的一个手段，定期更换密码，有利于保护加密存储的密钥信息。

输入

参数	类型	是否必须	举例
keyID	ID 字符串	是	“Alice_1”
oldParams	解密时的必要参数	根据实现情况而定	1，“AES”，“GCM”，“123456”，“邮箱”
newParams	加密时的必要参数	根据实现情况而定	1，“AES”，“GCM”，“abcdef”，“邮箱”

表 4.20 UpdatePassword 接口输入信息

输出

类型	举例
Error	Null | “password invalid”

表 4.21 UpdatePassword 接口输入信息

4.3 系统工作流程

本文基于安全多方计算的区块链密钥管理系统的设计目的是作为一个密钥管理工具，使用于区块链应用系统中，为密钥使用提供一个安全便捷的方式。用户在开发区块链应用时，对交易的签名步骤必不可少。相对于传统的直接在业务系统中内嵌密码学数字签名模块，密钥管理系统的使用与传统方法没有太大的区别，业务流程中直接传入交易 hash 值，调用签名 Sign 接口即可。

如图 4.3 所示，应用系统中直接与密钥管理模块开放的接口进行交互。密钥管理系统中的所有参与方组成一个完整的密钥管理模块，在模块内部保证密钥的安全性，对外只提供 4.2.2 小节中介绍的六个接口。用户也可以将密钥管理模块部署成微服务，供更多的用户使用。

下面介绍如何使用密钥管理模块的具体流程。

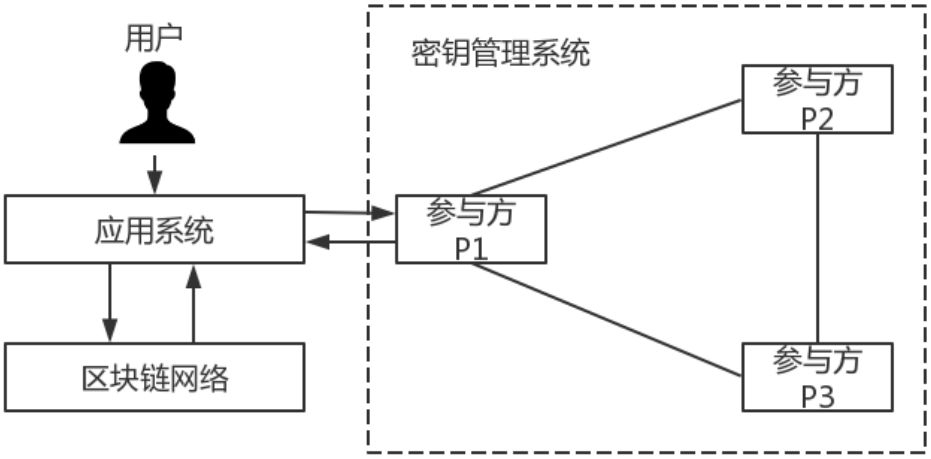


图 4.3 区块链应用架构图

4.3.1 环境介绍

本系统全部使用 Golang 语言编写，支持 Golang 1.11 及以上版本，系统中安装 Golang 环境即可使用。本系统是一个密钥管理工具，需要与具体业务结合才能展示业务的数据场景，所以接下来的使用说明中用命令行模式展示系统的工作流程。

4.3.2 初始化

首先配置初始化参数 Config：使用 Secp256k1 曲线，Paillier 密钥长度为 2048 比特，进行 Range proof 证明时安全参数为 40。

```
{
  "P":"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F",
  "N":"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141",
  "B":"0000000000000000000000000000000000000000000000000000000000000007",
  "Gx":"79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798",
  "Gy":"483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8",
  "BitSize":256,

  "NPaillierBits": 2048,
  "NthRootSecBits":128,
  "RangeSecBits": 40
}
```

初始化完成后可以查看命令行的接口信息：

```
./mpcec -h

-init
      输入配置信息
-keygen
      分布式密钥生成
-refresh
      密钥局部刷新
-refreshAll
      密钥全局刷新
-sign
      输入交易 hash 签名
-updatePassword
      更改密码
```

4.3.3 密钥管理操作

KeyGen

调用分布式密钥生成接口，参与方数量 3 人，使用 AES|CBC 模式加密，密码为“123456”。实验环境中将三个参与方各自生成的密钥份额显示出来，并输出主公钥信息。

```
./mpcec keygen 3 “account1” “AES” “CBC” “123456”

x1:  09faefc14c3fde37966c3795860e7c2d18cd981105a7f8922eb4a4c96432babc
x21:  8698e3e45e4881bd426edfdec8f5bf6a4d781214ade4b3dd5402b497aed67cce
x22:  c450357e7b046b69659aea1ca57f3a0e9b251f312c17a978d03dc07fc82fc428

Q:
46a026f1cf7f99d0176d4b9deabb4883808ab9c252252b4ebca6acc9baa148b3133d9f380222ea6
ae9537e195f7edcbe44b771815912eb56ae5178cc81402166
```

Sign

使用已经生成的密钥对信息进行签名，输入要签名的消息字符串，以及从存储模块中获取密钥的加密信息和密码。签名接口输出签名的结果（R||S）以及相应的主公钥方便验证签名的有效性。

```
./mpcec sign “hello” “account1” “AES” “CBC” “123456”

R:  7c2d1510e0fe886f3a092870680fbc5812786051a8fffd06b7b8142aa23d8373
S:  a69c716e66a0e8c4eb640761552715db638ca6d55943e2db08a8261d3b0366be

Q:
46a026f1cf7f99d0176d4b9deabb4883808ab9c252252b4ebca6acc9baa148b3133d9f380222ea6
ae9537e195f7edcbe44b771815912eb56ae5178cc81402166
```

Refresh

调用刷新局部密钥接口，刷新 P_{21} 和 P_{22} 的密钥份额。输出刷新后的密钥份额信息，与 KeyGen 阶段生成的密钥进行对比，其中 x_{21} 和 x_{22} 发生了变化而 x_1 不变，而且刷新之后主公钥 Q 不变。

```
./mpcec refresh “account1” “AES” “CBC” “123456”

x1:  09faefc14c3fde37966c3795860e7c2d18cd981105a7f8922eb4a4c96432babc
x21:  0d31c7c8bc91037a84ddbfbd91eb7ed5e0414742ac80c77ee8330aa28d76b85b
x22:  62281abf3d8235b4b2cd750e52bf9d074d928f98960bd4bc681ee03fe417e214

Q:
46a026f1cf7f99d0176d4b9deabb4883808ab9c252252b4ebca6acc9baa148b3133d9f380222ea6
ae9537e195f7edcbe44b771815912eb56ae5178cc81402166
```

紧接着使用刷新后的密钥对消息进行签名，仍然可以通过主公钥的验证。

```
./mpcec sign “hello” “account1” “AES” “CBC” “123456”

R:  3379624def8919a554900f6ddb7b816b06e528aaa229b6b55acb2679d47bbfa3
S:  04db7977070a8bc9b395f739963559526fca813fe6bfa787ae209c8d26f55cbe

Q:
46a026f1cf7f99d0176d4b9deabb4883808ab9c252252b4ebca6acc9baa148b3133d9f380222ea6
ae9537e195f7edcbe44b771815912eb56ae5178cc81402166
```

RefreshAll

调用刷新全局密钥接口，刷新 P_1，P_{21} 和 P_{22} 的密钥份额。输出刷新后的密钥份额信息，与 KeyGen 阶段生成的密钥进行对比，其中 x_1，x_{21} 和 x_{22} 都发生了改变，且刷新之后主公钥 Q 不变。

```
./mpcec refreshAll “account1” “AES” “CBC” “123456”

x1:  13f5df82987fbc6f2cd86f2b0c1cf85a319b30220b4ff1245d694992c8657578
```

```
x21:  1a638f91792206f509bb7f7b23d6fdabc0828e8559018efdd06615451aed70b6
x22:  188a06afcf608d6d2cb35d4394afe741d364a3e62582f52f1a07b80ff905f885

Q:
46a026f1cf7f99d0176d4b9deabb4883808ab9c252252b4ebca6acc9baa148b3133d9f380222ea6
ae9537e195f7edcbe44b771815912eb56ae5178cc81402166
```

紧接着使用刷新后的密钥对消息进行签名，也仍然可以通过主公钥的验证。

```
./mpcec sign “hello” “account1” “AES” “CBC” “123456”

R:  f9675653aaf9501a3220d8d56281770c09670dd67bfaca691c855e8cc3783b41
S:  0141fda5727e164a0d706ce82bd88b782ce2bfb392407509d2c3f4b41c2d3e9b

Q:
46a026f1cf7f99d0176d4b9deabb4883808ab9c252252b4ebca6acc9baa148b3133d9f380222ea6
ae9537e195f7edcbe44b771815912eb56ae5178cc81402166
```

本节展示了密钥管理系统从环境配置要求到初始化，进而调用每个接口进行实验，展示了每个步骤的输入输出信息，从实验测试的角度将本文的密钥管理系统的设计和实现展现出来。

4.4 本章小结

本章针对上一章对区块链密钥管理系统的需求分析和算法的设计，将算法模块化设计出密钥管理系统的四层架构，以解耦合和插件化的形式详细说明了架构之间每个服务模块的关系和工作的流程，详细介绍了对用户开放的所有密码模块接口和 API 服务接口，并对系统工作流程进行详细阐述，从用户的角度说明密钥管理系统的环境配置、初始化以及自主接口实现和接口调用的流程。

第五章 总结与展望

5.1 总结

本文通过对区块链技术原理与发展和区块链相关的密码学原理的研究，分析了区块链密钥管理在现阶段的主要问题。针对现存密钥管理方案存在的三个问题：1、密钥权力不够分散：当将一个完整的密钥存储下来时，就有整个密钥一起泄漏的风险。2、不能刷新密钥：密钥在生成后无法进行便捷的刷新，攻击者可以一个一个的破解获取密钥份额，最终可以完全把控区块链账户的资产。3、易用性差：用户需要记忆很复杂的密钥字符串，是阻碍区块链大规模发展的重要原因。给出对应的算法设计：1、使用安全多方计算的思想，将密钥分成多份分别存放在各个参与方手中。采用分布式密钥生成的方式，由多个参与方参与密钥的生成，每个人持有密钥的一个份额；在签名阶段，各方分别利用手中的密钥份额进行计算，最终将计算结果整合到一起生成完整的 ECDSA 签名。2、基于分布式密钥生成的协议，新增支持分布式密钥刷新的算法和接口，用户在部分密钥泄漏的情况下可以对全部或者部分密钥份额进行刷新，保证密钥的动态安全性。3、使用集成加密模式来存储和备份密钥的方法，用户只需使用密码口令即可进行分布式密钥生成、分布式签名和分布式密钥刷新的功能。

基于上述算法设计实现了一个四层架构的区块链密钥管理系统，并在文中详细地介绍了系统的架构说明和接口描述，并从用户的角度对密钥管理系统进行测试，对测试结果进行了分析。本文为区块链用户提供了一种区块链密钥安全易用的使用方式。

5.2 研究展望

现阶段学术界和工业界都在不断的探索区块链技术的可扩展性、高性能、安全性等问题的解决方法。区块链技术也将不断的完善并被使用到更多的业务场景中。在区块链技术大势所趋的环境之下，后续将对以下 2 个方面做进一步的研究：

移动互联网时代下，想要让更多的用户参与到区块链业务中来，就需要设计开发出用户使用更加方便的工具，比如将密钥管理系统集成到 Android 和 IOS 客户端中，让用户随时随地可以参与到区块链系统中。而移动终端甚至是 IOT 设备中，其计算能力有限，所以需要进一步对算法进行优化，使其对算力较低的设备更加友好。

本方案的密码学算法还不支持安全多方计算的门限功能，如果有门限秘密共

享的能力，可以提供在密钥丢失后从其他参与者手中足够的密钥份额中进行恢复，以提供更强大的密钥恢复能力。因此在后续的工作中可以继续研究安全多方计算下的门限签名以及在业务系统中的使用场景。

参考文献

[1] Nakamoto S. Bitcoin: A peer-to-peer electronic cash system[J]. 2008.
[2] Wood E. A Secure Decentralised Generalised Transaction Ledger[J]. 2018.
[3] Aras S T, Kulkarni V. Blockchain and Its Applications–A Detailed Survey[J]. International Journal of Computer Applications, 2017, 180(3): 29-35.
[4] Huckle S, Bhattacharya R, White M, et al. Internet of things, blockchain and shared economy applications[J]. Procedia Computer Science, 2016, 98: 461-466.
[5] 苏汉.工信部发布《2018 年中国区块链产业发展白皮书》[J].中国汽配市场,2018(02):15.
[6] Diffie W, Hellman M. New directions in cryptography[J]. IEEE Transactions on Information Theory, 1976, 22(6): 644-654.
[7] Eskandari S, Clark J, Barrera D, et al. A first look at the usability of bitcoin key management[J]. arXiv preprint arXiv:1802.04351, 2018.
[8] Okamoto T. A digital multisignature scheme using bijective public-key cryptosystems[J]. ACM Transactions on Computer Systems (TOCS), 1988, 6(4): 432-441.
[9] Desmedt Y, Frankel Y. Shared generation of authenticators and signatures[C]//Annual International Cryptology Conference. Springer, Berlin, Heidelberg, 1991: 457-469.
[10] Wuille P. BIP32: Hierarchical Deterministic Wallets[J]. https://github.com/genjix/bips/blob/master/bip-0032. md, 2012.
[11] Yao A C. Protocols for secure computations[C]//23rd Annual Symposium on Foundations of Computer Science (sfcs 1982). IEEE, 1982: 160-164.
[12] Lindell Y. Fast secure two-party ECDSA signing[C]//Annual International Cryptology Conference. Springer, Cham, 2017: 613-644.
[13] Lindell Y, Nof A. Fast secure multiparty ecdsa with practical distributed key generation and applications to cryptocurrency custody[C]//Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2018: 1837-1854.
[14] Gennaro R, Goldfeder S. Fast multiparty threshold ecdsa with fast trustless setup[C]//Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2018: 1179-1194.
[15] Rivest R L, Shamir A, Adleman L. A method for obtaining digital signatures and public-key cryptosystems[J]. Communications of the ACM, 1978, 21(2): 120-126.
[16] ElGamal T. A public key cryptosystem and a signature scheme based on discrete logarithms[J]. IEEE Transactions on Information Theory, 1985, 31(4): 469-472.
[17] Schnorr C P. Efficient identification and signatures for smart cards[C]//Conference on the Theory and Application of Cryptology. Springer, New York, NY, 1989: 239-252.
[18] Shamir A. How to share a secret[J]. Communications of the ACM, 1979, 22(11): 612-613.

[19] Herley C, Van Oorschot P. A research agenda acknowledging the persistence of passwords[J]. IEEE Security & Privacy, 2011, 10(1): 28-36.
[20] 张亮, 刘百祥, 张如意, et al. 区块链技术综述[J]. 计算机工程, 2019, 45(05):7-18.
[21] Lamport L, Shostak R, Pease M. The Byzantine generals problem[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1982, 4(3): 382-401.
[22] Castro M, Liskov B. Practical Byzantine fault tolerance[C]//OSDI. 1999, 99(1999): 173-186.
[23] King S, Nadal S. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake[J]. self-published paper, August, 2012, 19.
[24] Larimer D. Delegated proof-of-stake (dpos)[J]. Bitshare whitepaper, 2014.
[25] Szabo N. The idea of smart contracts, 1997[J]. 1997.
[26] Merkle R C, June S. Authentication and Public Key Systems[D]. Ph. D. Dissertation, Stanford University, 1979.
[27] Wang X, Yin Y L, Yu H. Finding collisions in the full SHA-1[C]//Annual International Cryptology Conference. Springer, Berlin, Heidelberg, 2005: 17-36.
[28] Miller V S. Use of elliptic curves in cryptography[C]//Conference on the Theory and Application of Cryptographic Techniques. Springer, Berlin, Heidelberg, 1985: 417-426.
[29] Koblitz N. Elliptic curve cryptosystems[J]. Mathematics of Computation, 1987, 48(177): 203-209.
[30] Gallagher P. Digital signature standard (dss)[J]. Federal Information Processing Standards Publications, volume FIPS, 2013: 186-3.
[31] Paillier P. Public-key cryptosystems based on composite degree residuosity classes[C]//International Conference on the Theory and Applications of Cryptographic Techniques. Springer, Berlin, Heidelberg, 1999: 223-238.
[32] Goldwasser S, Micali S, Rackoff C. The knowledge complexity of interactive proof systems[J]. SIAM Journal on Computing, 1989, 18(1): 186-208.
[33] Miers I, Garman C, Green M, et al. Zerocoin: Anonymous distributed e-cash from bitcoin[C]//2013 IEEE Symposium on Security and Privacy. IEEE, 2013: 397-411.
[34] Sasson E B, Chiesa A, Garman C, et al. Zerocash: Decentralized anonymous payments from bitcoin[C]//2014 IEEE Symposium on Security and Privacy. IEEE, 2014: 459-474.
[35] Ben-Sasson E, Chiesa A, Tromer E, et al. Succinct non-interactive zero knowledge for a von Neumann architecture[C]//23rd {USENIX} Security Symposium ({USENIX} Security 14). 2014: 781-796.
[36] Parno B, Howell J, Gentry C, et al. Pinocchio: Nearly practical verifiable computation[C]//2013 IEEE Symposium on Security and Privacy. IEEE, 2013: 238-252.
[37] Brickell E F, Chaum D, Damgård I B, et al. Gradual and verifiable release of a secret[C]//Conference on the Theory and Application of Cryptographic Techniques. Springer, Berlin, Heidelberg, 1987: 156-166.

[38] Chan A, Frankel Y, Tsiounis Y. Easy come—easy go divisible cash[C]//International Conference on the Theory and Applications of Cryptographic Techniques. Springer, Berlin, Heidelberg, 1998: 561-575.
[39] Boudot F. Efficient proofs that a committed number lies in an interval[C]//International Conference on the Theory and Applications of Cryptographic Techniques. Springer, Berlin, Heidelberg, 2000: 431-444.
[40] Bünz B, Bootle J, Boneh D, et al. Bulletproofs: Short proofs for confidential transactions and more[C]//2018 IEEE Symposium on Security and Privacy (SP). IEEE, 2018: 315-334.
[41] Chor B, Goldwasser S, Micali S, et al. Verifiable secret sharing and achieving simultaneity in the presence of faults[C]//26th Annual Symposium on Foundations of Computer Science (sfcs 1985). IEEE, 1985: 383-395.
[42] Hazay C, Mikkelsen G L, Rabin T, et al. Efficient RSA key generation and threshold paillier in the two-party setting[C]//Cryptographers' Track at the RSA Conference. Springer, Berlin, Heidelberg, 2012: 313-331.
[43] Fiat A, Shamir A. How to prove yourself: Practical solutions to identification and signature problems[C]//Conference on the Theory and Application of Cryptographic Techniques. Springer, Berlin, Heidelberg, 1986: 186-194.
[44] Protocol Buffers https://developers.google.com/protocol-buffers

附录 1 硕士期间学术论文与科研成果

专利

1、本人第二发明人（导师为第一发明人），用于获取区块链中的数据的方法、装置及存储介质，公开号：CN109492419A

2、本人第二发明人（导师为第一发明人），一种基于集成加密技术的密钥备份与恢复方法 (申请中)

3、本人第二发明人（导师为第一发明人），一种支持密钥刷新的两方椭圆曲线数字签名算法 (申请中)

4、本人第二发明人（导师为第一发明人），一种基于安全多方计算的区块链密钥管理系统 (申请中)

软件著作权

1、本人第二发明人（导师为第一发明人），基于属性密码的去中心化身份管理平台软件（简称：ABE 身份管理平台），登记号：2019SR0293429.