A Blockchain-Based PGP Key Sharing Mechanism for Secure Email Communication

September, 2025

Md. Biplob Hossain

Graduate School of
Natural Science and Technology
(Doctor's Course)

OKAYAMA UNIVERSITY

DOCTORAL THESIS

A Blockchain-Based PGP Key Sharing Mechanism for Secure Email Communication

Author:	Md. Biplob Hossain
Supervisor:	Yasuyuki NOGAMI
Co-supervisors:	Kazuhiro UEHARA
	Yukinobu FUKUSHIMA

A dissertation submitted to
OKAYAMA UNIVERSITY
in fulfillment of the requirements for the degree of
Doctor of Philosophy in Engineering
in the
Graduate School of Natural Science and Technology

September, 2025

To Whom It May Concern

We hereby certify that this is a typical copy of the original doctor thesis of

Md. Biplob Hossain

Signature of
The Supervisor

Seal of

Prof. Yasuyuki Nogami

Graduate School of
Natural Science and Technology

Declaration Authorship

This dissertation and the work presented here for doctoral studies were conducted under the supervision of Professor Yasuyuki Nogami. I, Md. Biplob Hossain, declare that this thesis titled, "**A Blockchain-Based PGP Key Sharing Mechanism for Secure Email Communication**" and the work presented in it are my own. I confirm that:

- The work presented in this thesis is the result of original research carried out by myself, in collaboration with others, while enrolled in the Graduate School of Natural Science and Technology at Okayama University as a candidate for the degree of Doctor of Philosophy in Engineering.
- This work has not been submitted for a degree or any other qualification at this University or any other institution.
- The published work of others cited in this thesis is clearly attributed. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help to pursue this work.
- The experiments and results presented in this thesis and in the articles where I am the first author were conducted by myself.

Signed: _____

Date: September 17, 2025

Abstract

Email remains a fundamental mode of communication across personal, corporate, and governmental domains. However, the inherent insecurity of standard email protocols has led to widespread reliance on cryptographic schemes like Pretty Good Privacy (PGP). Despite its strengths, PGP suffers from several critical limitations—particularly in the key sharing process, which remains vulnerable to interception, manipulation, and centralized trust dependencies.

Traditional PGP systems transmit the session key alongside the encrypted email. If the recipient's private key is compromised, both the message and key are at risk. Moreover, the use of centralized key servers or manual exchange introduces further risks of man-in-the-middle attacks, poor scalability, and usability challenges in multi-user environments.

This dissertation presents a blockchain-based architecture to address the vulnerabilities in PGP key distribution. The research is developed over four studies: the first proposes a secure key sharing scheme using blockchain and ECC; the second introduces zk-SNARKs for privacy-preserving key access; the third evaluates the system using variable ECC key lengths under different case study scenarios; and the fourth enhances scalability by supporting multi-recipient key distribution through indexed encryption.

The first study designs a decentralized key sharing mechanism by integrating Ethereum blockchain smart contracts with ECC-based encryption. The encrypted session key is uploaded to the blockchain, and only the authorized recipient can retrieve it. This system improves the security, integrity, and availability of the session key, as confirmed by a detailed evaluation of gas cost, storage efficiency, and successful retrieval rate using Truffle and Ganache platforms.

The second study focuses on preserving recipient anonymity and transaction metadata privacy by incorporating zk-SNARKs into the key access protocol. This privacy-enhancing upgrade ensures that access verification can be performed without revealing the user's identity or intent. Experiments with ZoKrates demonstrate that privacy can be achieved with only moderate increases in gas usage, and zk-SNARKs based transactions maintain scalability and security.

In the third study, a case-driven simulations has been introduced that vary ECC key sizes (32 bytes, 48 bytes, and 64 bytes) to balance security with performance. Experimental results show that higher ECC sizes increase transaction gas cost and memory overhead, but significantly improve security and resistance to brute-force attacks. This evaluation provides guidance for tailoring key sizes to application-specific sensitivity levels.

Finally, the fourth study addresses the scalability bottleneck of single-recipient PGP systems by extending the smart contract design to support multiple recipients. By introducing a tuple of Key ID and encrypted key, the system reduces redundancy and optimizes storage. The correlation between number of recipients and gas cost is evaluated, demonstrating predictable linear scalability ($R^2 = 0.999$), making the system viable for mass communications in secure environments.

Overall, this research offers a comprehensive and modular framework for sharing PGP key securely in email communication using blockchain technologies. By progressively addressing vulnerabilities in key sharing, privacy, scalability, and performance optimization, the proposed architecture delivers both theoretical robustness and practical viability. The integration of smart contracts, zk-SNARKs, and variable ECC key lengths enables a flexible, secure, and scalable key distribution mechanism applicable across diverse communication scenarios. Future directions include refactoring smart-contract along with modularizing reusable logic to minimize code duplication and reduce execution overhead without altering the core design. These improvements can make the system cost-effective and enhancement in gas efficiency for practical deployment in organizational email environments where secure group communications are critical.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Professor Yasuyuki Nogami, for his exceptional support, guidance, and encouragement throughout my doctoral studies at Okayama University. His insightful advice and generous mentorship have been invaluable to my academic and research development.

I am deeply grateful to my co-supervisors, Professor Kazuhiro Uehara and Associate Professor Yukinobu Fukushima, for their continuous support, constructive feedback, and for imparting essential knowledge through their courses.

I would like to extend my special thanks to Associate Professor Yuta Kodera for his inspirational attitude towards research, which greatly influenced and motivated me throughout my PhD journey. I am also deeply thankful to all the faculty members who have shared their knowledge with me through the seminars conducted in the Information Security Engineering Laboratory.

I would like to express my heartfelt gratitude to Professor Md. Arshad Ali of Hajee Mohammad Danesh Science and Technology University, Bangladesh, for his unwavering support and insightful guidance throughout my PhD journey. I also extend my sincere thanks to Assistant Professor Samsul Huda at Okayama University for his valuable scientific discussions and contributions.

I would also like to acknowledge all the members of the Information Security Engineering Laboratory for creating a positive and stimulating research environment. Their camaraderie and support made my academic journey more enjoyable and productive.

My heartfelt appreciation goes to Ms. Maya Rahayu, with whom I shared many fruitful research discussions. Her collaboration and support in co-authoring several publications were instrumental in the success of my doctoral work.

Special thanks to Ms. Yuri Kunisada and Ms. Yumi Sato for their kind administrative support, which helped me manage various academic procedures smoothly during my stay at Okayama University.

Last but not least, I am deeply indebted to my mother for her unwavering dedication, encouragement, and determination in allowing me to pursue doctoral studies. Her constant support and sacrifice have been a source of strength throughout my academic journey.

Publications

Peer-Reviewed Journal Paper (First author):

1. **Md. Biplob Hossain**, Maya Rahayu, Md. Arshad Ali, Samsul Huda, Yuta Kodera, and Yasuyuki Nogami, "A Blockchain-based Approach with zk-SNARKs for Secure Email Applications," *International Journal of Networking and Computing*, vol. 14, no. 2, pp. 225–247, 2024.
(Acceptance rate 14.06% from CANDAR'23 selected papers).

Peer-Reviewed International Conference Papers (First author):

2. **Md. Biplob Hossain**, Maya Rahayu, Md. Arshad Ali, Samsul Huda, Yuta Kodera, and Yasuyuki Nogami, "A Smart Contract Based Blockchain Approach Integrated with Elliptic Curve Cryptography for Secure Email Application," *Eleventh International Symposium on Computing and Networking Workshops (CANDARW)*, Matsue, Japan, pp. 195–201, Nov. 2023.
(Acceptance rate 37.2%).

3. **Md. Biplob Hossain**, Maya Rahayu, Samsul Huda, Md. Arshad Ali, Yuta Kodera, and Yasuyuki Nogami, "A Blockchain-Based Approach for Secure Email Encryption with Variable ECC Key Lengths Selection," *The 8th International Conference on Mobile Internet Security (MobiSec)*, Sapporo, Japan, pp. 1–14, Dec. 2024.
(Acceptance rate 30.1%).

Peer-Reviewed Journal Papers (Co-author):

4. Maya Rahayu, **Md. Biplob Hossain**, Samsul Huda, Yuta Kodera, Md. Arshad Ali, and Yasuyuki Nogami, "The Design and Implementation of Kerberos-Blockchain Vehicular Ad-Hoc Networks Authentication Across

Diverse Network Scenarios," *sensors*, vol. 24, no. 23, Art. no. 7428, pp. 1–29, 2024.
(Acceptance rate 56%).

5. Samsul Huda, Yasuyuki Nogami, Maya Rahayu, Takuma Akada, **Md. Biplob Hossain**, Muhammad Bisri Musthafa, Yang Jie, and Le Hoang Anh, "IoT-Enabled Plant Monitoring System with Power Optimization and Secure Authentication," *Computers, Materials & Continua*, vol. 81, no. 2, pp. 3165–3187, 2024.

Peer-Reviewed International Conference Papers (Co-author):

6. Samsul Huda, Yasuyuki Nogami, Takuma Akada, Maya Rahayu, **Md. Biplob Hossain**, Muhammad Bisri Musthafa, Le Hoang Anh, and Yang Jie, "A Proposal of IoT Application for Plant Monitoring System with AWS Cloud Service," *2023 International Conference on Smart Applications, Communications and Networking (SmartNets),* Istanbul, Turkiye, pp. 1–5, Jul. 2023.

7. Maya Rahayu, **Md. Biplob Hossain**, Md. Arshad Ali, Samsul Huda, Yuta Kodera, and Yasuyuki Nogami, "An Integrated Secured Vehicular Ad-Hoc Network Leveraging Kerberos Authentication and Blockchain Technology," *Eleventh International Symposium on Computing and Networking Workshops (CANDARW)*, Matsue, Japan, pp. 260–266, Nov. 2023.
(Acceptance rate 37.2%).

8. Samsul Huda, Yasuyuki Nogami, **Md. Biplob Hossain**, Yang Jie, Le Hoang Anh, Muhammad Bisri Musthafa, Maya Rahayu, and Takuma Akada, "A Secure Authentication for Plant Monitoring System Sensor Data Access," *2024 IEEE International Conference on Consumer Electronics (ICCE),* Las Vegas, NV, USA, pp. 1–2. Jan. 2024.

9. Maya Rahayu, **Md. Biplob Hossain**, Samsul Huda, Md. Arshad Ali, Yuta Kodera, and Yasuyuki Nogami, "An In-depth Analysis of Kerberos

and Blockchain Integration on VANETs' Security and Performance," *2024 International Conference on Consumer Electronics - Taiwan (ICCE-Taiwan)* Taichung, Taiwan, pp. 391–392, Jul. 2024.
(Acceptance rate 69.4%).

10. Samsul Huda, **Md. Biplob Hossain**, Maya Rahayu, Andri Santoso, and Yasuyuki Nogami, "Design of Blockchain-Based Secure Device Authentication for IoT Plant Monitoring Systems," *2025 International Conference on Consumer Electronics - Taiwan (ICCE-Taiwan)* Kaohsiung, Taiwan, Jul. 2025. [Accepted for publication]

Peer-Reviewed Book Chapter:

11. Springer Communications in Computer and Information Science (CCIS) book series. [Accepted for inclusion]

    The paper will be published as part of the following volume:

    - Series Title: Communications in Computer and Information Science
    - Book Title: Mobile Internet Security
    - Book Subtitle: 8th International Conference, MobiSec 2024, Sapporo, Japan, December 17–19, 2024, Revised Selected Papers
    - Paper Title: A Blockchain-Based Approach for Secure Email Encryption with Variable ECC Key Lengths Selection

Table of Contents

List of Figures

List of Tables

Notations and Abbreviations

Entities

PGP	Pretty Good Privacy
ECC	Elliptic Curve Cryptography
zk-SNARKs	zero-knowledge Succinct Non-interactive ARguments of Knowledge
RSA	Rivest-Shamir-Adleman
S/MIME	Secure/Multipurpose Internet Mail Extensions
PKI	Public Key Infrastructure
CA	Certification Authority
SBTM	Secure Blockchain Trust Management
ZKP	Zero-knowledge Proof
DAP	Decentralized Anonymous Payment
CSP	Cloud Service Provider
ABE	Attribute-Based Encryption
DAP	Decentralized Anonymous Payment
ZKRP	Non-interactive Zero-knowledge Range Proof
MITM	Man-In-The-Middle
DoS	Denial of Service
Pk	public key
Sk	Secret Key (private key)
MREKS	Multi-recipient Public Key Encryption Scheme with Keyword Search
KGC	Key Generation Center
AES	Advanced Encryption Standard

IDEA	International Data Encryption Algorithm
EVM	Ethereum Virtual Machine
ABI	Application Binary Interface
CRS	Common Reference String
E_p	Elliptic Curve
$\mathbb{F}_p$	Finite Field
p	Prime Number
P_r	Private Key
G	Base Point
n	Order of Base Point
K	Random Key
r	Random Number
K_e	Encrypted Key
dApps	Decentralized Applications
IDE	Integrated Development Environment
GUI	Graphical User Interface
RPC	Remote Procedure Call
ETH	Ether
GWEI	Giga Wei Dai
(vk)	Verification Key
π	Non-interactive Proof

λ	Security Parameter
(K_{id})	Key ID
IBE	Identity-Based Encryption
FP	Public-Key Fingerprint
$H(\cdot)$	Cryptographic Hash
SHA	Secure Hash Algorithm
LSB	Least Significant Bit

Chapter 1

Introduction

1.1 Introduction

In the current era of digital transformation, email continues to serve as a primary channel for personal, organizational, and institutional communication. Despite the widespread adoption of email as a convenient and efficient tool for message exchange, conventional email systems are frequently vulnerable to various security threats, including unauthorized access, spoofing, tampering, and privacy breaches [1]. To mitigate these risks, cryptographic solutions such as Pretty Good Privacy (PGP) have been employed for decades, offering encryption and digital signature functionalities [2]. However, PGP's effectiveness is often hindered by its reliance on centralized or semi-centralized key distribution models, which present scalability limitations and privacy vulnerabilities—especially in dynamic environments involving multiple recipients. This thesis addresses the limitations of traditional PGP key sharing by introducing a blockchain-integrated framework that evolves through four sequential research phases.

The first study involves the adoption of blockchain technology and Elliptic Curve Cryptography (ECC) to facilitate a decentralized, immutable, and auditable key-sharing process. This foundational study demonstrates how blockchain can be effectively used to store encrypted session keys (K_e) securely, where each key is intended for a single recipient. It ensures that once the key is recorded on the chain, it cannot be tampered with or deleted, thereby establishing the trust and transparency necessary for secure email communication.

The second study introduces privacy-preserving enhancements to the key-sharing framework through the integration of zero-knowledge Succinct Non-interactive ARguments of Knowledge (zk-SNARKs). This advancement ensures that while encrypted keys are stored and accessed via blockchain, the underlying identity of users and the content of interactions remain confidential. zk-SNARKs allow one party to prove knowledge of a valid decryption key or the authenticity of an encryption transaction without revealing the actual key or message, thereby elevating the privacy guarantees of the system.

In the third study, the system's cryptographic agility is analyzed through a case-driven evaluation of ECC key length variations. Different ECC key sizes—namely 256, 384, and 521 bits—are implemented to understand their trade-offs in terms of security strength, gas consumption, memory usage, and transaction cost when stored and accessed via Ethereum smart contracts. This investigation enables flexible adoption of encryption standards based on application-specific security requirements and resource constraints, paving the way for practical deployment in real-world scenarios.

Finally, the fourth and most advanced phase of this research focuses on enhancing the system's scalability by transitioning from a single-recipient to a multi-recipient key distribution model. A novel Key ID mechanism is introduced, derived from the least significant 64 bits of the recipient's public key fingerprint, enabling precise and compact recipient identification. This innovation allows a sender to upload a single transaction containing multiple (K_{id}, K_e) tuples, each corresponding to a designated recipient, thereby significantly reducing computation, transaction cost, and storage overhead compared to performing separate operations for each recipient. The performance of the proposed system is rigorously evaluated across 1 to 25 recipients, with corre-

lation analysis used to validate its linear scalability and efficiency in terms of gas usage and on-chain memory requirements.

Collectively, these four contributions provide a holistic and progressive framework for secure, scalable, and privacy-aware email communication. This dissertation consolidates the theoretical models, architectural designs, and empirical findings from each phase to offer a unified perspective on next-generation secure email systems that leverage the decentralized and immutable nature of blockchain while integrating advanced cryptographic primitives for practical and privacy-preserving usage.

1.2 Problem Statement and Motivation

Email encryption relies on exchanging cryptographic keys securely between the sender and recipient. In typical schemes such as PGP or S/MIME, the sender encrypts the email body with a random symmetric session key and then encrypts that key under the recipient's public key [3]. The recipient uses their private key to recover the session key and then decrypt the message. This hybrid approach combines symmetric and asymmetric encryption: it leverages the speed of symmetric ciphers for large message payloads while using public-key cryptography for secure key distribution. In fact, by using the recipient's public key to wrap the session key, these schemes enhance security and facilitate the efficient management and distribution of cryptographic keys [4].

This key sharing mechanism has clear advantages. By encrypting the session key with a recipient's public key, the sender can safely transmit the key over an open channel: only the holder of the corresponding private key can

decrypt it. As per the previous studies, this hybrid encryption framework not only enhances security but also facilitates the efficient management and distribution of cryptographic keys. Thus, public-key based key sharing ensures end-to-end confidentiality and enables scalable key management in email systems, removing the need for a trusted courier or secure channel to share keys in advance.

Despite these advantages, the key sharing steps itself can be vulnerable. In traditional PGP email, the encrypted message and the encrypted session key are typically sent together. An adversary who intercepts the message could potentially capture the ciphertext of the session key and later try to compromise its confidentiality. As noted in our earlier work, if an attacker obtains the encrypted session key during transit, the confidentiality of the PGP key is compromised. Ensuring the secure distribution of the encryption key remains a critical challenge in PGP deployment [2]. This challenge—securing the PGP key transfer—is precisely the problem our first study was designed to solve.

In my first study [5], a novel blockchain-based solution has been proposed to secure the PGP key sharing mechanism. Specifically, we integrated blockchain technology and Elliptic Curve Cryptography (ECC) into the PGP workflow to eliminate traditional intermediaries in key sharing. By design, blockchain immutability serves as a safeguard against the vulnerabilities in PGP's key sharing mechanism, and smart contracts remove the requirement of a Man-in-the-Middle in key delivery. The encrypted PGP key is stored on-chain, ensuring its confidentiality and integrity during transfer.

I implemented and evaluated this blockchain-based key sharing mechanism to verify its feasibility. The experiments showed that the encrypted PGP

key could be successfully transmitted via the smart contract. However, a limitation remained: the blockchain transactions themselves are public. Although the key is encrypted, metadata about the transaction is visible to all participants. This residual privacy issue motivated my second study [6].

In my second study [6], I enhanced the blockchain key-exchange scheme by integrating zero-knowledge proofs to address the privacy shortfall. zk-SNARKs was incorporated into the protocol to enable verification of encrypted key transfers without revealing sensitive data. This integration preserved the confidentiality of the key while also hiding metadata from the blockchain. The results confirmed that privacy is preserved and that transaction costs were minimized. However, the system still used a fixed ECC key length. This limitation—lack of flexibility to adapt encryption strength to the context—was addressed in our third study [7].

In my third study [7], variable ECC key length was introduced to allow users to dynamically adjust encryption strength. The sender could now choose between ECC-256, ECC-384, or ECC-521 depending on the sensitivity of the communication and available computational resources. Experimental evaluation confirmed the feasibility and efficiency of this feature. However, like its predecessors, this design still assumed a single recipient per transaction, making it inefficient for multi-user email. This limitation motivated our fourth study.

In my fourth study, I addressed the multi-recipient scenario by proposing a scalable group key distribution framework. A multi-recipient extension mechanism was designed to the blockchain-based PGP system using a single smart contract transaction to distribute encrypted keys to multiple recipients using unique Key IDs. I implemented and evaluated the proposed system with vary-

ing numbers of recipients. Results confirmed strong scalability and significant reductions in gas costs compared to repeated single-recipient transactions. In summary, these studies addressed the sequential challenges of secure PGP key sharing, privacy, encryption strength flexibility, and scalability in PGP email encryption systems.

1.3 Major contributions

This section outlines the major contributions of the thesis, which are presented across four successive studies. Each contribution builds upon the previous, addressing key limitations and progressively enhancing the security, scalability, and efficiency of blockchain-based secure email communication systems. The following are the main contributions of this dissertation:

- **Implementation of a Secure PGP Key Sharing Scheme with Blockchain Technology**

  This study was published in the *Proceedings of the Eleventh International Symposium on Computing and Networking Workshops (CANDARW)*, 2023 [5], and its implementation details are comprehensively discussed in Chapter 4. The key contributions of this work are as follows:

  - Developed a blockchain-based key sharing mechanism that securely distributes PGP session keys using Ethereum smart contracts, ensuring immutability, transparency, and auditability.
  - Implemented ECC-based key pair generation and encrypted key sharing, offering lightweight cryptographic operations suitable for

resource-constrained environments compared to traditional RSA-based PGP.

  - Designed and deployed Solidity smart contracts to register users and securely store encrypted keys on the Ethereum blockchain, enabling decentralized and tamper-resistant key access.
  - Evaluated performance metrics, including gas consumption, transaction cost, and storage usage, demonstrating the feasibility of the approach for secure and cost-efficient email key sharing.

- **Integration of zk-SNARKs with Blockchain for Privacy Preservation**

  This study was published in the *International Journal of Networking and Computing (IJNC)*, vol. 14, no. 2, 2024 [6], and is elaborated in Chapter 5. The primary contributions of this work are as follows:

  - Integrated zk-SNARKs into the blockchain-based key sharing mechanism to conceal sender–recipient mappings and preserve transaction-level privacy without revealing sensitive metadata.
  - Employed ZoKrates for generating and verifying zero-knowledge proofs, ensuring compatibility with Ethereum smart contracts and enabling privacy-preserving verification of encrypted key validity.
  - Designed a modified smart contract framework that verifies the correctness of encrypted keys via zk-SNARK proof submission, avoiding direct exposure of key-related data on-chain.
  - Performed a comparative analysis of gas cost and computational overhead with and without zk-SNARKs, confirming that privacy

enhancements incur a reasonable performance trade-off within acceptable bounds.

- **Case-Driven Analysis of ECC Key Length Variations in PGP Key Sharing**

  This study was presented at the *8th International Conference on Mobile Internet Security (MobiSec)* [7] and is detailed in Chapter 6. The key contributions of this work are summarized as follows:

  - Investigated the impact of varying ECC key sizes (32-byte, 48-byte, and 64-byte) on a blockchain-based key sharing mechanism to assess performance implications.
  - Proposed three distinct use-case scenarios—personal, professional, and governmental—to guide ECC key length selection based on the required security-performance balance.
  - Implemented the ECC key sharinfg approach using Solidity smart contracts and deployed on the Ethereum test network, analyzing the gas cost, transaction delay, and on-chain memory usage for each key length.
  - Demonstrated how longer key sizes provide enhanced security but introduce higher computation and gas overheads, thus informing users on appropriate configurations under different application contexts.

- **Enhancing Blockchain Scalability via Multi-Recipient in PGP Key Sharing**

This study is comprehensively discussed in Chapter 7. The key contributions are as follows:

- A blockchain-based system that efficiently handles encryption for multiple recipients within a single transaction, eliminating redundant encryptions, reducing computational overhead and storage requirements.
- Adopt Key ID Mechanism using the least significant 64 bits of the public key's fingerprint to create compact identifiers, enabling efficient key sharing without compromising security.
- Maintained support for different ECC key lengths (256, 384, and 521 bits) while extending functionality to multi-recipient scenarios, reducing computational overhead, storage requirements, and transaction costs.
- Comprehensive evaluation of the system under varying numbers of recipients (1–25), measuring key performance metrics including gas consumption, transaction cost, and memory requirements. Results demonstrate strong linear correlation between resource usage and recipient count, confirming the system's scalability and cost-efficiency.

1.4 Outlines of This Dissertation

This section provides an overview of the organization of this dissertation. This dissertation is structured into eight chapters, each focusing on different aspects of blockchain-based secure key sharing and email encryption systems.

It highlights the publication venues where each core contribution was published and outlines the structure of the dissertation chapter-wise.

Several components of this dissertation have been formally disseminated through peer-reviewed international journals and conferences. The core developments described in Chapter 4 were initially presented at the *2023 Eleventh International Symposium on Computing and Networking Workshops (CANDARW)* [5]. Chapter 5 builds upon work published in the *International Journal of Networking and Computing (IJNC)* [6], focusing on privacy-preserving blockchain integration. The comprehensive performance assessment detailed in Chapter 6 is based on findings shared at the *8th International Conference on Mobile Internet Security (MobiSec 2024)* [7]. Finally, Chapter 7 presents an enhanced multi-recipient encryption framework.

The structure of the dissertation is organized as follows:

- **Chapter 1:** Establishes the context and motivation of the study related to blockchain-based secure email applications. It highlights the research contributions made through four major studies, and summarizes the overall structure of the dissertation.

- **Chapter 2:** Presents a comprehensive analysis of existing studies related to PGP key sharing scheme. It synthesizes the findings from peer-reviewed papers, categorizing thematically, highlighting limitations in previous works and motivating the proposed solutions.

- **Chapter 3:** Presents the foundational concepts and technologies relevant to this research. It includes an overview of the PGP system, blockchain characteristics pertinent to secure key sharing, consensus mech-

anisms, zk-SNARKs integration, and the open-source tools adopted throughout the implementation.

- **Chapter 4:** Details the design and implementation of a blockchain-based secure key sharing system using elliptic curve cryptography (ECC). It outlines the system architecture, implementation process, and performance evaluation, followed by a security analysis and discussion of results.

- **Chapter 5:** Introduces a privacy-preserving enhancement to the PGP key sharing by integrating zk-SNARKs with blockchain. It discusses the underlying principles, presents the proposed framework, implementation details, and evaluates the performance and privacy impact through zk-SNARK-based security analysis.

- **Chapter 6:** Conducts a case-driven analysis of PGP key sharing by varying ECC key lengths. It outlines the proposed system, details implementation and performance evaluation across different scenarios, and investigates smart contract security and potential vulnerabilities arising from varying cryptographic configurations.

- **Chapter 7:** Presents the implementation of a multi-recipient PGP key sharing mechanism using variable ECC key lengths. It revisits the limitations of earlier approaches, introduces an enhanced system architecture to support scalable recipient management, and evaluates performance in terms of gas cost, memory usage, and smart contract vulnerabilities.

- **Chapter 8:** Summarizes the overall findings of the research presented throughout the dissertation. It reflects on the key outcomes, discusses the

limitations encountered, and outlines future directions to further enhance the scalability and efficiency of blockchain-based secure key exchange systems.

Chapter 2

Literature Review

2.1 Introduction

Email remains one of the most widely used communication methods globally, but it is inherently vulnerable to a wide range of security threats. Traditional PGP (Pretty Good Privacy) systems provide a foundational layer of confidentiality by encrypting both the message and the symmetric session key [8], yet they suffer from key distribution limitations, centralized trust models, and susceptibility to man-in-the-middle attacks. These limitations are especially pronounced when extended to modern, scalable communication models involving multiple recipients or decentralized access patterns.

Recent studies have aimed to address these concerns by leveraging blockchain technology for secure and transparent key management. In my first study [5], I proposed a blockchain-based key sharing mechanism using Elliptic Curve Cryptography (ECC) to improve secure key distribution for PGP email systems. Building on this, my second study [6] introduced zk-SNARKs (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge) to enhance privacy during key retrieval processes, mitigating metadata leakage. The third paper [7] conducted a case-driven analysis on the impact of varying ECC key lengths, showing trade-offs in gas usage and memory overhead depending on the security demands. The recent work, addressed the scalability limitations by proposing a multi-recipient key distribution framework using blockchain, which incorporated key indexing and reduced redundancy to handle large-scale secure communication.

This chapter synthesizes the contributions of these works alongside other contemporary literature to identify the gaps, challenges, and opportunities in secure key sharing for PGP systems. It categorizes prior efforts thematically—focusing on cryptographic security, decentralization, privacy preservation, and scalability—to establish the motivation for our integrated blockchain-based solutions.

2.2 Related Works in Literature

This section provides a comprehensive overview of the existing research and developments related to secure email communication, blockchain-based key management, elliptic curve cryptography (ECC), multi-recipient encryption, and privacy-preserving technologies such as zk-SNARKs.

2.2.1 PGP in Email Communication

Email is one of the most important communication services used daily by people to share data across the world [1]. The number of email circulating in the world per day is growing due to its convenience features. Although traditional email system authenticates only on the email server according to the user name and password, while the information itself is stored in plain text on the server [3]. For this reason, it becomes necessary to encrypt and sign the email before sending it from the sender to the recipient. There are many secure email encryption techniques available, and the most popular of them are Secure/Multipurpose Internet Mail Extensions (S/MIME) and PGP [2]. Encryption techniques secure the email contents, though a major challenge in this case is the secure exchange of the key that is used to encrypt the email.

In public key authentication system to PGP, at first sender generate a private key, which is known as random key. After that, sender encrypt the message by this random key. Then the random key is also encrypted using the receiver's public key to protect it during transmission. Both the message and the encrypted random key are sent to the receiver then. At the receiving end, the receiver first decrypt the random key using his/her private key. Then, using this random key, receiver can decrypt the original message and read it. By encrypting and signing the messages in the network, PGP can give us confidentiality and integrity [9]. However, as PGP sends the encrypted message and encrypted key together, we may lose both if the attackers are able to get hold of them. So, in PGP, it still remains a challenge to share the encrypted key from the sender to the recipient.

2.2.2 Blockchain Technology in PGP System

Blockchain system can be used in this case to reduce the vulnerabilities for PGP's key sharing mechanism. Due to its distributed and immutable characteristics, blockchain can share the encrypted key separately from the sender to recipient. As there is no central authority governing the system, it reduces the risk of a single point of failure during the transmission of encrypted key. Whenever the sender stores the encrypted key in the blockchain, it requires transaction verification from all the users. It needs the same verification when the desired user tries to access the encrypted key. For this, attackers cannot directly access the encrypted key as it requires many steps of verification to store and access it from the blockchain due to the written conditions of smart contracts. It also removes the encrypted key deletion or modification during the transactions. On the other hand, transactions recorded on a blockchain

are visible to all users in the network, which increases trust among the users and makes the users equally responsible for transaction execution. Blockchain also removes the Man-in-the-Middle risk during encrypted key transmission.

Blockchain is a distributed ledger where all nodes are independent to execute their tasks, though verify all the transactions by cooperation to maintain an identical ledger [10]. A Smart contract makes blockchain more flexible to create computer code to define the steps and the management of the system [11]. It automates the transactions and ensures that all nodes follow the same rules. The cryptographic hashing algorithm makes the blocks unbreakable and tamper-proof. For these cryptographic characteristics, blockchain consider in secure email application [12]. In [13], a secure email solutions based on blockchain was proposed, where the authors identified the security problems of email communication and proposed possible solutions based on a flexible and immutable blockchain architecture. However, the authors did not implement any blockchain architecture to solve the problems, they review several research works to identify the problems and make a direction to solve them. The security and privacy of email communication using blockchain with blowfish algorithm was proposed in [14]. In this case, the node data are converted into hashing text format and then stored it in individual blocks using the blowfish algorithm.

Blockchain technology has the potential to provide security and privacy in various applications including email transmission, data sharing, file sharing and so on. Blockchain employs a hybrid encryption scheme along with a consensus algorithm [15], and it integrates a distributed file system as well as smart contracts [16]. A solution for certified electronic mail using blockchain technology has been proposed in [17]. The solution preserves the confiden-

tiality of the mail exchange between the sender and receiver, although the mail contents in the plain-text format. In [18], authors proposed a blockchain based framework that marks down an outgoing mail to provide an decentralized and secure solution for mail verification. Although, the authors explained the technical details about the working procedure of the proposed system, did not provide any specific solutions.

One of the analyzed work was proposed in [19], to replace email protocols with smart contracts. Developing a decentralized system, the proposed framework can minimizes the common problems in email services such as spam, phishing, and spoofing. Although the solution may face challenges in terms of adoption and implementation.

Recently, blockchain technology has been introduced in the PGP system to reduce challenges such as Man-in-the-Middle risks and inefficient certificate revocation due to synchronization delays. A new PGP management framework using blockchain technology was proposed in [2] to solve the security challenges of PGP key servers. The framework offers fast propagation of certificate revocation and grants users access control to update their own PGP certificates. However, sending together the original message and secret key is still like conventional PGP. So, there is a risk of attacks from the attackers, and we may lose both the message and the key, as it sent together. In our proposed scheme, the original message is sent like PGP. Although we consider the secure transfer of secret key from the sender to the receiver using blockchain technology separately. In [20], the authors proposed a Bitcoin-based PGP certificate that leverages aspects to address the shortcomings of the existing PGP certificate model. The proposed framework can perform identity verification transactions using the Bitcoin address for each generated PGP certificate. However, the

proposed certificate implementation and adoption may face challenges due to the need for widespread adoption of Bitcoin infrastructure. Moreover, it can increase the complexity of the certificate issuance and verification process for identity verification [21, 22].

Based on this literature, a blockchain-based framework is proposed to exchange the encrypted key of the PGP system securely from the sender to the receiver while the encrypted message is transmitted similar to PGP [5].

2.2.3 Zero Knowledge Proof Techniques in Blockchain

However, nodes in a blockchain network must come to an agreement and for this the user's need to disclose their personal information to the network for executing the transaction verification. This poses significant privacy issues for the users. That is, the privacy of user data cannot be guaranteed by the blockchain alone. Some other techniques can be added to increase the privacy of the transaction. Several privacy-preserving methods, like ring signature, homomorphic encryption, and zero-knowledge proof (ZKP), have been used to blockchain applications to address the privacy issues in the blockchain [23].

ZKP is a protocol designed to provide interactive verification between two or more participants. Without giving any valuable details, this technique allows verifiers to be certain that a prover holds certain confidential data. This characteristic makes it possible for ZKP technology to ensure data validity without disclosing any confidential details about the data. The proving key and verification key must be created for the zero-knowledge proof protocol to be implemented, and the proof must enable multiple verifications using the verification key without disclosing any further pertinent information to the opposition throughout the entire verification procedure. ZKP is often used to

implement blockchain technology, anonymous credibility, remote verification, and asset settlement [24].

ZK-SNARKs (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge) are a kind of zero-knowledge proof (ZKP) cryptography technique that is essential to protecting the confidentiality of data. Without sharing the real data, the two individuals may use zk-SNARKs to confirm their mutual accuracy in an assertion or piece of information. Because of this, zk-SNARKs are a very effective technique for protecting privacy of information [25]. Without disclosing the specifics of the transaction, a user may demonstrate that they have sufficient balance to complete the transaction. This facilitates the accomplishment of blockchain's primary objective, which is safeguarding and disclosing information while preserving user privacy [26].

In blockchain and cryptocurrency applications, zk-SNARKs enable users to prove the validity of transactions while maintaining privacy. In transaction privacy preservation, the primary objective is to enhance the confidentiality of various transaction attributes, including sender and receiver addresses, transaction amounts, and the linkage between the transaction parties. By ensuring the anonymity of both sender and receiver addresses, the identities of the parties involved can be effectively concealed. Several innovative approaches have been proposed to achieve transaction privacy preservation through zk-SNARKs.

Miers et al. [25] introduce Zerocoin, a cryptographic extension to Bitcoin, that allows for fully anonymous currency transactions by augmenting the protocol without introducing new trusted parties or changing the security model of Bitcoin. It eliminates the need for coin issuers by allowing individual Bitcoin clients to generate their own coins. However, sender and receiver addresses as

well as transaction amounts were still revealed by this method. To address this issue, Ben-Sasson et al. [26] proposed Zerocash , a digital currency that provides strong privacy guarantees by hiding the payment's origin, destination, and amount, leveraging zero-knowledge Succinct Non-interactive ARguments of Knowledge (zk-SNARKs). Zerocash is a practical implementation of a decentralized anonymous payment (DAP) scheme that is more efficient and anonymous than Zerocoin and competitive with plain Bitcoin.

To make even stronger transaction privacy preservation, Guan et al. [27] introduces BlockMaze , a privacy-preserving account-model blockchain based on zk-SNARKs, to address the privacy issues in blockchain technology. BlockMaze achieves strong privacy guarantees by hiding account balances, transaction amounts, and the linkage between senders and receivers. It introduces a dual-balance model for account-model blockchains, consisting of a plaintext balance and a zero-knowledge balance for each account. This was accomplished by using zero-knowledge transactions between senders and receivers to hide transaction links in conjunction with a secure commitment system for transaction amounts and account balances. Moreover, transaction privacy was preserved by the idea of a decentralised mixing system, which was introduced by Song et al. [28]. This method concentrated on hiding the sender and recipient addresses in addition to the connection between transactions.

Xu et al. [29] propose a privacy-respecting approach utilizing zk-SNARKs to mitigate privacy concerns. It analyzes the security features and performance of the proposed approach to demonstrate its effectiveness in these applications. The proposed approach does not require any special party in the blockchain system and does not change the overall design of existing applications. Privacy preservation within energy trading systems was addressed by Hou et al. [30].

In order to ensure anonymity during energy trading, this research combines blockchain technology with a double auction technique, using zk-SNARKs to check bid accuracy without disclosing personal information.

Blockchain was used by the authors to improve the confidentiality and integrity of data in real estate contract systems introduced Jeong et al. [31]. It enables online contract management and discrimination of contract forgery through blockchain technology. The zero-knowledge proof algorithm ensures confidentiality and prevents fraud until the contract is concluded and terminated. Gai et al. [32] propose a blockchain-based identity verification system that protects user identities and characteristics by using zk-SNARKs. It discusses the implementation of the adopted architecture of Zero-Knowledge Proofs (ZKP) using the development toolkit Zokrates, which allows for off-chain generation of ZKP and on-chain verification based on Ethereum smart contracts. The authors made sure that identity indications and identity traits were protected by using zk-SNARKs, allowing for secure authentication without compromising privacy.

Based on this literature, a blockchain-based framework is proposed to exchange the encrypted key of the PGP system securely from the sender to receiver using blockchain technology without disclosing the details of the transactions [6].

2.2.4 ECC Key Lengths in PGP

However, due to a predetermined key length, users may lose the opportunity to customize the encryption strength according to the specific nature of the data [33]. For instance, highly sensitive data may necessitate more robust encryption through the utilization of longer keys, which would consequently

hinder processing speed. In contrast, data of lesser significance might only require a shorter key length to facilitate swifter encryption. A fixed key length forces all data to use the same level of security and performance, regardless of its sensitivity [34]. In normal cases, PGP employs a variable-length key pair to encrypt random keys, thereby catering to diverse security and performance requirements.

In PGP, individuals have the option to select extended key lengths to enhance security when the situation demands it or to opt for shorter keys to facilitate expedited processing and reduced computational cost when security imperatives are less stringent. For example, considering government entities such as intelligence agencies, defense departments, or foreign affairs ministries handling national security information. Government agencies frequently handle classified or sensitive information that must be transmitted securely among officials and departments. This information is particularly valuable for inter-agency communications, such as sending classified data on national security, citizen data, or diplomatic information. To safeguard this information, the organization may select a longer key length for the purpose of encrypting the random key, as it provides an exceptionally elevated degree of security. While encrypting with a longer key requires a powerful device capable of handling the increased computational load, the agency acknowledges this compromise as the security of the information is far more critical.

Based on this literature, I varied the length of ECC key pairs such as ECC-256, ECC-384, and ECC-521, which produce various lengths of encrypted keys and share them using our blockchain-based system. Because of this, three different scenarios for case studies have been considered. In case study I, the system is considered for casual personal email communication, where the con-

tents of the email are less sensitive, but the information can be shared without noticeable delays. Case study II is implemented for professional email communication, where the email contents are moderately sensitive business information. It also emphasizes the necessity of striking a balance between security and performance. In the case of study III, the system is acknowledged for highly confidential government communication, where the email contents contain extremely sensitive information that requires the highest level of security but demands high-performance, secure workstations to implement the system. Based on the studies, we also evaluate the blockchain's performance and the feasibility of the proposed system in terms of gas consumption, transaction costs, and memory requirements [7].

2.2.5 Multi-recipient Email Communication

However, the previous approaches focus on single-recipient communication, thereby neglecting the imperative of disseminating same email content to numerous recipients within collaborative environments. In many real-world scenarios, email communication correspondence involves multiple recipients, rendering it crucial to concurrently disseminate same content to all recipients to enhance operational efficiency [35]. Moreover, encrypting a single message for multiple recipients helps eliminate the need to send multiple copies of the same content, thereby conserving system resources. In conventional or single-recipient email, each email is encrypted and transmitted individually to every recipient. This approach is characterized by inefficiency and a significant consumption of resources, particularly in collaborative or enterprise-level communications, wherein the same content must be securely delivered to multiple recipients [36].

In a single-recipient email architecture, each recipient necessitates a separate email encryption and transactional process [37]. Emails are subjected to encryption utilizing a symmetric key. Subsequently, this symmetric key is encrypted independently for each recipient through the utilization of their respective PGP public keys to generate the encrypted keys. The encrypted content of the email is transmitted from the sender to the recipient employing the PGP framework. Following this, each encrypted key is uploaded independently to the blockchain via a blockchain-based mechanism. Consequently, each transaction incurs individual gas costs for the upload of each encrypted key, which ultimately exacerbates computational overhead, network utilization, and storage demands.

In contrast to a multi-recipient system, email content is encrypted once with a symmetric key and the encrypted email contents are transmitted to recipients via PGP. The symmetric key is then encrypted individually for each recipient using their respective PGP public keys, producing a set of encrypted keys [38]. Unlike the single-recipient approach, these encrypted keys are uploaded to the blockchain in a single transaction. This significantly reduces the number of blockchain operations, as only one transaction is needed to serve multiple recipients. As a result, this approach minimizes gas consumption, lowers computational overhead, and reduces network and storage usage, making the system more efficient and scalable for group communication.

In a single-recipient email system, each email is encrypted and sent individually to separate recipients, resulting in distinct communication instances for each user. Because every transmission is independently recorded on the blockchain, there is no centralized or collective reference that links all recipients to the same original message. This lack of linkage limits transparency

regarding who else received the information, making it difficult to ensure uniformity of communication across recipients [39]. Consequently, collaborative efforts may suffer, as team members cannot easily confirm whether everyone involved has received identical content or updates. This disconnect can negatively impact coordinated decision-making, particularly in distributed or remote work environments where real-time synchronization is crucial.

In contrast, a multi-recipient system encrypts the email content once and stores the associated set of encrypted symmetric keys on the blockchain in a single, consolidated transaction. This approach links all recipients through a common blockchain entry, guaranteeing uniform message access for every participant. By maintaining a shared record, the system inherently supports transparency, where each recipient can verify both the message content and the involvement of other recipients. This collective visibility improves collaboration by ensuring all users reference the same version of the communication, encouraging mutual accountability and alignment. Such a design is particularly advantageous in distributed or interdepartmental teams, where synchronized, trackable exchanges are essential for efficient and reliable coordination [40].

To consider these advantages of a multi-recipient email system, this research advances our blockchain-based PGP key distribution framework to facilitate scalable multi-user communication. We propose a key ID mechanism, which is derived from the least significant 64 bits of the fingerprint of the public key, to formulate a compact identifier for each user's public key. This innovation enables efficient key dissemination, thereby eleminating the necessity to encrypt the identical message for every recipient. The system retains the variable ECC key length support established in our prior investigation while enhancing the architecture to accommodate multiple recipients within a single

blockchain transaction. In the act of registering, the participant forms a duo of public and private keys (Pk, Sk) employing ECC. The key ring securely holds the public key (Pk), while the private key (Sk) is sent back to the individual. In the context of PGP key distribution, after generating a key ID, it is integrated with the encrypted PGP key and thereafter documented on the blockchain. Recipients are enabled to access the relevant encrypted key by utilizing the key identifier, thus facilitating secure and efficient communication.

2.2.6 Comparison of Proposed Literature with some Relevant Existing Literature

Table 2.1 summarizes the comparison among proposed literature with some existing literature. Authors in [41], presented a multi-party certified mail protocol on Ethereum that executes smart contracts under conflict, maintaining end-to-end confidentiality while minimizing on-chain operations. In [42], authors implemented a decentralized email system using Ethereum smart contracts and off-chain storage to address spam, phishing, and authentication. Authors in [43] designed a hybrid public/private blockchain email framework emphasizing decentralization and authentication, yet lack a detailed multi-recipient encryption scheme. An Ethereum-based framework "BlockPGP," was developed by the authors in [44], for storing PGP certificates on-chain to enhance public-key management for encrypted email.

Authors in [45] merged dynamic AES file encryption with blockchain key management, focusing on data confidentiality and integrity primarily for single-recipient scenarios. In [6], authors improved blockchain-based PGP key sharing by integrating zk-SNARKs, encrypting PGP keys with ECC and employing zk-

SNARK proofs for ciphertext verification, but still considering single recipient model. A decentralized mail system (INCET 2023) was proposed in [46], utilizing smart contracts and ECC encryption to eliminate central servers and combat spam, although their work serves as a proof-of-concept with limited performance metrics. In [47], authors introduced an identity-based one-way group key agreement for email, while in [48], authors presented a blockchain-based certificate-less bidirectional searchable encryption for cloud email, enabling keyword search with dual authentication.

In contrast, the proposed multi-recipient system work advances these schemes by enabling efficient multi-recipient encryption and flexible key management. In particular, it natively supports one-to-many communication (up to 25 recipients) with nearly linear gas-cost scaling, rather than requiring N repetitions of a single-recipient protocol [6, 45, 46]. We furthermore allow flexible choice of ECC key strength (256, 384 or 521 bits), whereas prior works generally fixed a single key size [6, 44, 46, 48]. Moreover, our smart contracts are optimized for gas and storage efficiency: heavy cryptographic operations are performed off-chain, and on-chain data structures are minimized. This design yields significantly lower gas costs and memory usage than prior on-chain encryption schemes, making our approach more scalable and cost-effective for multi-recipient secure email on blockchain.

2.3 Summary

This chapter provided a comprehensive review of relevant research addressing secure key shari8ng mechanisms, particularly within the context of PGP-enhanced email systems. While traditional email encryption models face

limitations in trust, scalability, and privacy, blockchain-based approaches show significant promise in overcoming these challenges. Our reviewed literature including both foundational studies and our own research, highlights the advantages of using smart contracts, ECC, zk-SNARKs, and key indexing strategies. These approaches collectively offer improved security, transparency, and system scalability. The insights gained from this literature review form the conceptual backbone for the proposed blockchain-based solutions discussed in subsequent chapters.

Table 2.1: Comparative study of proposed literature with existing secure email and key distribution schemes.

Reference	**Blockchain platform**	**Key Length /Algo**	**Simulation Tool**	**Gas /Cost Analysis**	**Perf. Metrics Evaluated**	**Recipient Model**
[41]	Ethereum	Standard	Truffle /Ganache	Yes	Gas cost	Multi-recipient
[42]	Ethereum + IPFS	PGP /PKI	IPFS +Solidity	Yes	Security features	Single /Group
[43]	Conceptual Blockchain	Not specified	Design model	No	Security features	Single /Group
[44]	Ethereum	RSA /ECC	Web3.js	No	Latency	PGP Server
[45]	Permissioned Ledger	Standard Email Keys	Prototype	Yes	Latency, Scalability	Single
[6]	Ethereum	ECC	Truffle /Ganache, ZoKrates	Yes	Gas, cost and memory	Single
[46]	Ethereum + IPFS	Standard	Prototype	No	Functions	Single /Group
[47]	Ethereum	Not specified	Solidity, Web3.js	No	Latency	Multi-recipient
[48]	Ethereum	ECC	Conceptual	No	Cost	Multi-recipient
Proposed Work (2025)	Ethereum	Variable ECC	Ganache, Truffle, ZoKrates	Yes	Gas, cost, memory, correlation	Single and Multi-recipient

Chapter 3

Background and Preliminaries

3.1 Introduction

Secure communication has become a pivotal requirement in the digital age, especially with the widespread use of email for personal, professional, and governmental communication. Ensuring confidentiality, integrity, and authenticity in email transmission necessitates robust cryptographic mechanisms. This chapter provides a foundational understanding of the key technologies and tools employed in this dissertation. It begins by discussing the principles and workflow of PGP systems, followed by an overview of blockchain technology, smart contracts, Solidity programming, and consensus mechanisms. The chapter also explores the integration of zk-SNARKs in blockchain environments and introduces the open-source tools used for implementation and testing, including Truffle, Ganache, Remix, and ZoKrates.

3.2 Working Procedure of PGP

Pretty Good Privacy (PGP) is a cryptographic protocol designed to provide secure email communication by utilizing a combination of symmetric and asymmetric encryption along with digital signatures. Originally developed by Phil Zimmermann in 1991 [49], PGP remains a widely adopted standard for personal and organizational data protection. Its primary functions include confidentiality, authentication, and data integrity.

PGP operates by using a hybrid encryption scheme. When a sender wants

to transmit a secure message, they first generate a random session key to encrypt the message using symmetric encryption (typically AES or IDEA). This session key is then encrypted using the recipient's public key through an asymmetric encryption algorithm such as RSA or ECC. The resulting encrypted session key, along with the encrypted message, is sent to the recipient. Upon receiving the data, the recipient decrypts the session key with their private key and uses it to decrypt the message [20].

Despite its strengths, PGP faces persistent challenges, particularly in key distribution and scalability. Traditional PGP requires a user to manually obtain and validate recipients' public keys through key servers or direct exchange, which is prone to spoofing and lacks automation. This is especially problematic in multi-recipient scenarios, where the message must be encrypted separately for each user [5]. Our work aims to overcome these issues by leveraging blockchain technology to decentralize and automate public key distribution while improving scalability and integrity through smart contracts and zk-SNARKs [6].

Recent studies have further explored the integration of blockchain with PGP to provide immutable, transparent, and verifiable key storage and access. Your earlier studies also adopt this approach by developing smart contract-based key registration mechanisms, enhancing the trustworthiness of PGP public key handling. Additionally, the proposed system eliminates redundant encryptions in multi-recipient environments by introducing Key ID mapping strategies [7], providing a scalable and privacy-preserving solution.

Thus, while PGP provides a solid foundation for secure communication, evolving communication patterns and scalability requirements demand architectural improvements—many of which are addressed through the blockchain-

integrated solutions developed in this dissertation.

3.3 Blockchain Technology and Characteristics

This section provide the architecture of blockchain technology and the characteristics of it particularly suitable for secure PGP key sharing and email encryption.

3.3.1 Blockchain Technology

Blockchain is a distributed ledger technology that allows data to be recorded across multiple nodes in a network in a secure, transparent, and immutable manner [50]. Originally developed for cryptocurrencies, blockchain has evolved into a general-purpose architecture applicable in domains such as finance, healthcare, supply chain, and secure communication systems, including email encryption.

Each block in a blockchain contains a set of transactions, a timestamp, and a cryptographic hash of the previous block, thereby forming an immutable chain [51]. This design ensures the integrity and chronological order of records. Smart contracts—self-executing scripts stored on the blockchain—enable programmable logic to facilitate automated operations, such as secure key sharing without the need for trusted intermediaries [52].

Figure 3.1 illustrates the basic structure of blockchain technology. Blockchain's application in secure key sharing has gained traction due to its ability to record cryptographic keys in a tamper-proof and publicly verifiable way [53].

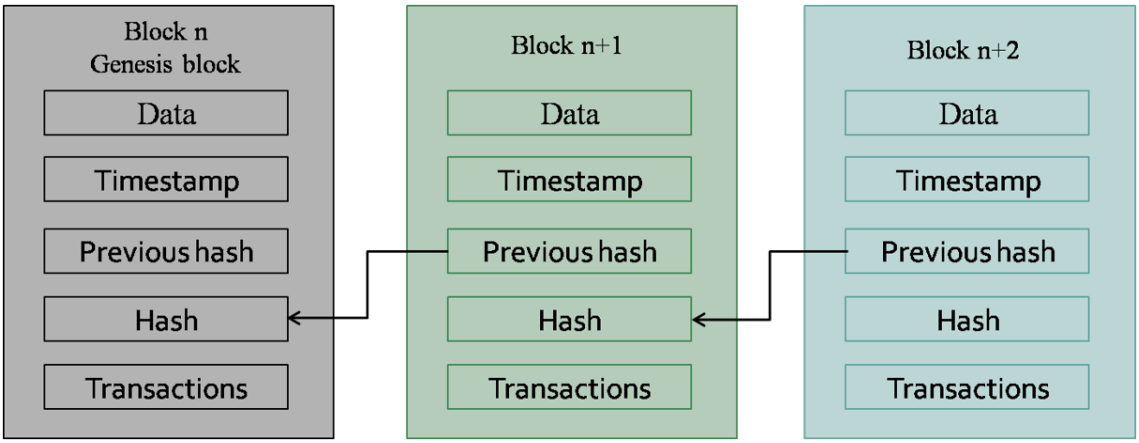

Figure 3.1: General architecture of blockchain technology.

3.3.2 Blockchain Characteristics

Several core characteristics of blockchain technology make it particularly suitable for applications involving secure key share and email encryption. These features are especially beneficial for systems where confidentiality, integrity, and traceability are essential:

- **Immutability:** Once data is stored on the blockchain, it cannot be altered or deleted, ensuring that recorded cryptographic keys remain tamper-proof.
- **Transparency:** All nodes have access to the same data, promoting verifiability and traceability of key transactions without compromising security.
- **Decentralization:** No single entity controls the blockchain, thereby removing single points of failure often associated with centralized key servers.
- **Auditability:** Historical records of key registration and access events can be traced and audited efficiently.

- **Security via Cryptography:** The blockchain employs cryptographic primitives such as digital signatures and hash functions to validate data integrity and user authenticity.
- **Programmability:** Smart contracts enable automated enforcement of security policies, such as controlling who can upload or retrieve public keys.

These characteristics collectively enhance the reliability of blockchain-based key management systems and justify their integration in secure email communication frameworks.

3.3.3 Smart Contracts

These are the written agreements among the parties and all the parties are bound by the contracts [54]. The three stages of smart contracts are as follows.

1) Contract Generation: It covers the parties needs, the contract's details, and clarifies between the contract's participants, among other things. A contract code will be created after the precise definition and fixing of the contract terms.

2) Contract Release: The signed contract is sent to the nodes in peer-to-peer modes after it has been formed, and it is kept there until a consensus is established.

3) Contract Execution: A contract cannot be altered after it is published and the nodes have come to an agreement. The contract automatically comes into action if the predetermined requirements are met [55].

3.3.4 Solidity

High-level programming languages like Solidity were developed especially for Ethereum blockchain smart contract development. Runs on the Ethereum Virtual Machine (EVM), it is an object-oriented language[56]. The solidity code is created and then compiled to make sure it is accurate, effective, and prepared for deployment. Solidity code is transformed into Application Binary Interface (ABI) and bytecode artefacts upon compilation. The names of the individual functions of the smart contract, as well as the kinds of data they receive as inputs and return as outputs, are included in the ABI. It serves as a manual for data delivered to and from contracts, including encoding and decoding. The real logic and functionality of the contract, presented in a way that the EVM can comprehend and carry out, are included in the machine-understandable instructions, or bytecode.

3.3.5 Consensus Mechanisms in Blockchain

Blockchain offers decentralized privacy and security characteristics via the use of consensus techniques in a peer-to-peer network. The security, scalability, and accessibility of the blockchain are all significantly impacted by the consensus method [57]. It is regarded as an essential component of a distributed ledger that is decentralised. It establishes how the blockchain network's blocks, transactions, and contract executions are validated. It is a procedure for managing abusive conduct and achieving a consensus among users [58]. In order to guarantee that only authorised transactions are recorded in the blockchain, it is helpful to verify and review the data before it is put to the block. It contributes to the stability and integrity of a blockchain network by using con-

sensus to make sure that all modifications are approved uniformly by all nodes [59].

3.4 zk-SNARKs on Blockchain Technology

zk-SNARKs are a class of cryptographic proof systems [21]. Without disclosing any details about the assertion itself, they allow one person, the prover, to persuade another party, the verifier, that a statement is true. The concept of “zero-knowledge" describes the situation when the evidence reveals nothing that might compromise the veracity of the claim. The constituents of zk-SNARKs are decomposed as follows:

- Zero-Knowledge: With the use of a proof, one may show that a statement is true without disclosing any information about the assertion itself.
- Succinct: The proof is brief and simple to check. Scalability depends on this, particularly for blockchain applications.
- Non-Interactive: There is no need for back-and-forth correspondence between the prover and verifier since the proof is produced in a single step.
- Arguments of Knowledge: Without disclosing the information itself, the proof shows that the prover is in possession of a certain piece of knowledge.

Zk-SNARKs are useful in many areas, but they are most prominently used in blockchain technology to facilitate private transactions. A user may demonstrate the legitimacy of a transaction using zk-SNARKs without disclosing the sender, recipient, or transaction amount.

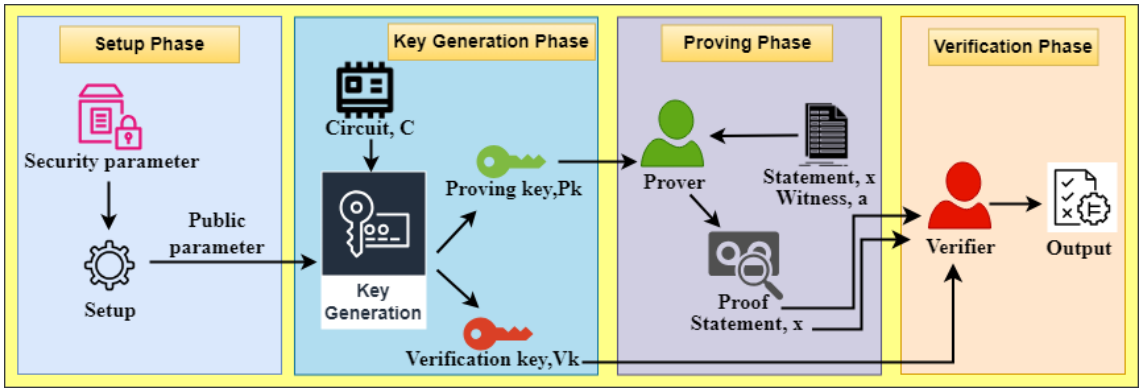

Figure 3.2: The framework of zk-SNARKs.

Steps to implement zk-SNARKs on blockchain technology is illustrated in Figure 3.2 and described as follows [29]:

1. Setup Phase: It needs a one-time trustworthy setup. In this stage, a security parameter is used to build a collection of public parameters. These parameters are obtained from secret parameters during a trusted setup procedure, and together they form the Common Reference String (CRS).

2. Key Generation Phase: This stage generates a pair of keys known as the proving key and the verification key.

3. Proving Phase: Without disclosing any particulars about the statement, the prover creates the proof of a statement using the proving key from the setup step, a public statement, and a private witness.

4. Verification Phase: Anyone may easily verify the correctness of the proof by using the verification key from the setup phase, a public statement, and the proof produced from the proving step. The transaction will be successful if the verification is successful; if not, the transaction request will be rejected.

3.5 ECC Encryption and Decryption

ECC is a public key cryptography. In blockchain, it can be used for encryption, decryption, key share, and key agreement schemes [60]. For the utilization of ECC encryption and decryption procedures, let us analyze an elliptic curve denoted as $E_p(a, b)$, wherein the parameters a and b govern the formulation of the curve's equation and have to fulfill the condition $4a^3 + 27b^2 \neq 0$. The elliptic curve is constructed over a finite field identified as $\mathbb{F}_p$, wherein p represents a prime number ($p > 3$) [61]. The initiator of the communication commences the procedure by formulating a private key, P_{rA} (with 0 to n), which is subsequently employed to derive the associated public key, $P_A = P_{rA} \cdot G$, where G is a base point and n denotes the base point's order. In a parallel manner, the recipient formulates a private key, denoted as P_{rB} (with 0 to n), which is utilized to derive the associated public key, expressed as $P_B = P_{rB} \cdot G$.

For the purpose of encryption, the random key k is encrypted by the sender. A random whole number r (within the limits of 0 to n) is picked to function as a fleeting private key, and the related public key, $R = r \cdot G$, is derived. The shared secret point, $S = R \cdot P_{rB}$, is established, and the x-coordinates of the shared secret S are utilized to derive a symmetric key, S_k. Subsequently, the sender encrypts the key k in order to generate the encrypted key, $k_e = \{S_k, k\}$.

Throughout the decryption phase on the receiver's end, the receiver first acquires k_e alongside the temporary public key, R, and subsequently computes the shared secret point, $S = R \cdot P_{rB}$. This is followed by the extraction of the shared secret S_x-coordinates to establish a symmetric

key, S_k. Ultimately, k_e is decrypted to obtain the original key, expressed as $k = \{S_k, k_e\}$.

3.6 Standard PGP Email Communication Workflow

Pretty Good Privacy (PGP) is a widely used system for end-to-end email encryption that combines symmetric and asymmetric cryptography in a hybrid approach. In a typical PGP scenario, each user has a public/private key pair. To send an encrypted email, the sender must have access to the recipient's public key (obtained beforehand through a key server, exchange of key files, or other means). The high-level process of standard PGP email encryption and decryption is shown on Figure 3.3. The overall process are summarize as follows:

(a) Session Key Generation: The sender generates a random symmetric session key (a one-time key, e.g. 128-bit AES key). This session key will be used to encrypt the email's content.

(b) Message Encryption: Using a fast symmetric cipher (e.g. AES), the plaintext email body is encrypted with this session key. This produces the ciphertext of the email content.

(c) Session Key Encryption: The session key itself is then encrypted using the recipient's public key (e.g. via RSA or ECC). This step "locks" the session key so that only the corresponding private key (held by the intended recipient) can unlock it.

(d) Email Composition: The sender creates the PGP email by attaching the encrypted session key (often in a PGP "Key Packet") to the encrypted message. (In practice, PGP email formats include a block that contains the session key encrypted for the recipient, followed by the block of ciphertext for the message.)

(e) Transmission: The encrypted email (containing both the encrypted message and the encrypted session key) is sent over the standard email network to the recipient.

(f) Session Key Decryption (by Receiver): Upon receiving the email, the recipient uses their private key to decrypt the encrypted session key. Because the session key was encrypted with the recipient's public key, only their matching private key can recover it.

(g) Message Decryption: Finally, the recipient uses the recovered session key to decrypt the email's ciphertext, obtaining the original plaintext message. At this point, the recipient can read the email in its original form.

This PGP workflow ensures confidentiality by using the efficiency of symmetric encryption for the bulk of the data and the security of asymmetric encryption for sharing the key. Notably, an attacker cannot decrypt the message without both the encrypted message and the correct private key to decrypt the session key.

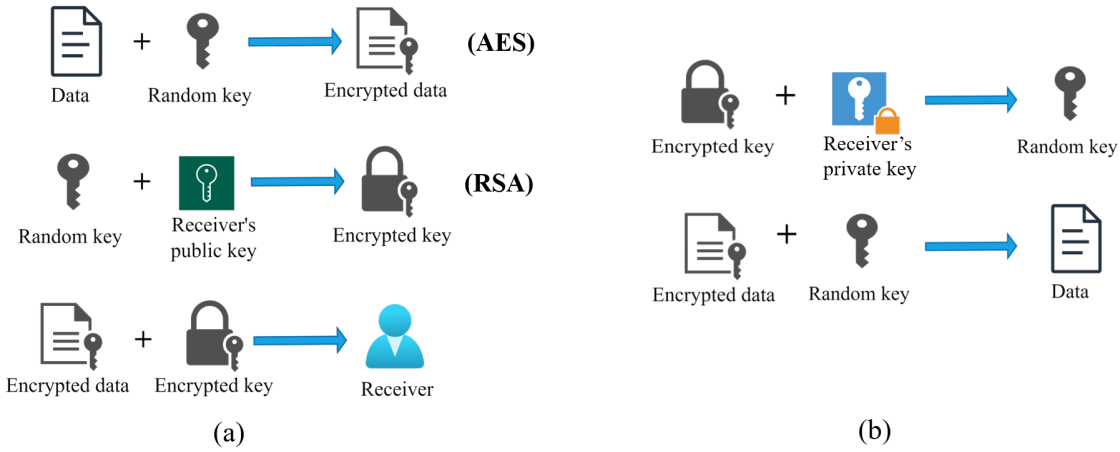

Figure 3.3: Standard PGP email (a) encryption and (b) decryption process.

3.6.1 Blockchain-Based Key Sharing vs. Traditional PGP

The proposed blockchain-based system preserves the above PGP process for encryption and decryption, but it modifies the key-sharing mechanism used to deliver the session key to the recipient. In traditional PGP, as described, the session key (encrypted with the recipient's public key) is sent alongside the email ciphertext to the receiver. This means an eavesdropper who intercepts the email in transit obtains a copy of the encrypted session key. Although the key is encrypted, its presence in transit could be exploited – for example, if the adversary later compromises the recipient's private key, they could decrypt the session key and read the message retroactively. Moreover, distributing public keys or certificates via central servers or direct exchange can introduce risks of manipulation or man-in-the-middle (MITM) attacks in a traditional PGP setup.

In the blockchain-based approach, only the key distribution step is changed,

the rest of the PGP workflow remains unchanged. The sender still generates a random symmetric session key and encrypts the email message with it as usual. However, instead of attaching the encrypted session key to the email, the sender uploads the encrypted session key to a blockchain (via a smart contract transaction) designated for the intended recipient. In other words, the session key (already encrypted with the recipient's public key) is stored on the blockchain network rather than being sent through the email channel. This blockchain based process is shown in Figure 3.4. In our proposed Blockchain based system, we share the key separately from sender to the receiver. We also introduced ECC as an asymmetric algorithm to replace RAS, considering the storage capability of Blockchain. For the same level of security, ECC require less sizes of key compared to RSA, that actually reduce the size of encrypted key to become more feasible to store it in Blockchain. After that, Blockchain technology will be used to share the encrypted key, the encrypted data (email contents) will be sent as usual using PGP. So there will be no risk to send the data and key together like normal PGP.

The receiver, upon getting the encrypted email content, will fetch or receive the encrypted session key from the blockchain. This may involve the receiver querying the smart contract (proving their identity or using a lookup key) to retrieve the portion of data that contains their encrypted session key. Only the authorized recipient is intended to access this encrypted key from the blockchain (e.g. the contract might be designed such that only the intended user's address can retrieve that data). Even if the blockchain is public, the session key data is safe because it remains encrypted with the recipient's public key, no one else can decrypt it

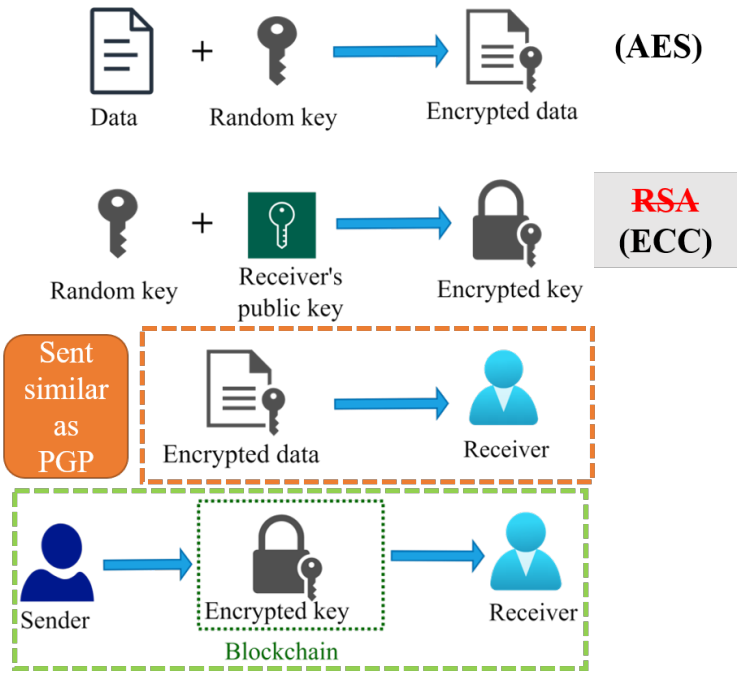

Figure 3.4: Proposed blockchain approach to share the encrypted key.

without the corresponding private key.

Once the receiver obtains the encrypted session key from the blockchain, they use their private key to decrypt it, recovering the session key just as in standard PGP. They then decrypt the email's ciphertext with this session key to read the message. Thus, aside from the delivery route of the session key, the core PGP encryption/decryption operations are the same in both systems. The blockchain serves only as a secure, decentralized ledger for transferring the session key, replacing the traditional direct email-based key sharing.

This modification of the key-sharing process confers several security and reliability benefits. By not transmitting the encrypted session key with the email, the system reduces the exposure of that key to potential interceptors. An adversary who taps the email traffic would not find the ses-

sion key in the message at all. Furthermore, using a blockchain to publish the encrypted key provides an immutable and tamper-evident record of the key exchange. The blockchain's consensus mechanism means that an attacker cannot alter or spoof the session key, unlike a rogue key server or intercepted email which could potentially be manipulated. It also removes the dependence on centralized PGP key servers for key distribution – trust is placed on the decentralized blockchain network instead of a single server or manual exchange, mitigating traditional vulnerabilities of the Web of Trust or PKI.

In summary, the blockchain-based PGP system retains the standard PGP hybrid encryption workflow (symmetric encryption for the message, public-key encryption for the session key) and only changes how the session key is shared with the recipient. The encrypted session key is delivered through a smart-contract on a blockchain rather than via email, providing enhanced security against interception and tampering while still allowing the intended recipient to obtain the key and decrypt the message as usual. All other aspects of PGP (key generation, message encryption, decryption, and optional signing) remain unchanged, ensuring that the integration of blockchain affects only the key exchange component of the communication.

3.7 Adopted Open Tools

The authors have adopted several open tools. A brief introduction about these open tools are as follows:

- Truffle: A world class development environment, testing framework and asset pipeline for blockchains using the Ethereum Virtual Machine (EVM), aiming to make the developer's life easier [27]. With Truffle, we get built-in smart contract compilation, linking, deployment and binary management. It also provides advanced debugging with breakpoints, variable analysis, and step functionality. Truffle can be used in automated contract testing for rapid development. In our proposed system, we used truffle as a development framework of Ethereum. The encrypted key that is generated in the off-chain environment is accessed first from truffle platform. We uploaded and accessed the encrypted key from truffle and connected in to the ganache for visualizing the transaction.

- Ganache: Ganache serves as a valuable Ethereum development tool, providing the capability to simulate a blockchain environment locally and conduct tests on deployed smart contracts. Throughout the entire development process, Ganache proves useful, allowing developers to create, deploy, and assess their decentralized applications (dApps) within a secure and predictable setting. Since smart contracts, once deployed on the blockchain, are immutable, thorough testing and debugging are essential before deployment [62]. Because of this, in this study, we choose a local blockchain environment, Ganache, as it alleviates developers from transaction costs and delays.

- Remix: Remix Ethereum IDE (Remix IDE) is the online development environment for Ethereum. Remix IDE is used for the entire

journey of smart contract development by users at every knowledge level [27]. It requires no setup, fosters a fast development cycle, and has a rich set of plugins with intuitive GUIs. Remix IDE is a powerful open-source tool that helps you write Solidity contracts straight from the browser. Remix IDE has modules for testing, debugging, and deploying of smart contracts and much more. Remix is a full stack web framework that let us to focus on the user interface and work back through web fundamentals to deliver a fast, slick, and resilient user experience that deploys to any Node.js server and even non-Node.js environments at the edge. We used remix for executing the solidity code of zk-SNARKs.

- ZoKrates: Zero-knowledge proofs (ZKPs) are a family of probabilistic protocols, first described by Goldwasser, Micali and Rackoff in 1985. One family of ZKPs is described as zero-knowledge Succinct Non-interactive ARguments of Knowledge (zkSNARKs). zk-SNARKs are the most widely used zero-knowledge protocols, with the anonymous cryptocurrency Zcash and the smart-contract platform Ethereum among the notable early adopters [29]. Ethereum runs computations on all nodes of the network, resulting in high costs, limits in complexity, and low privacy. zkSNARKs are difficult to understand and manipulate, and they can only be used to validate on-chain computations for a fraction to calculate the cost of running them. ZoKrates bridges this gap. It helps us to create off-chain programs and link them to the Ethereum blockchain, expanding the possibilities for your DApp. In our solidity code, we implemented the zk-SNARKs idea using the ZoKrates tool. It is

necessary to create a separate file first called ZoKrates (.zok) in order to integrate zk-SNARKs into the solidity code using ZoKrates. The zero-knowledge proof's reasoning is explained in this file. After that ZoKrates generates proving key to create the proof. Subsequently, the proof is confirmed using the verification key generated by the ZoKrates tool.

3.8 Summary

This chapter provided an overview of the fundamental technologies underpinning this research. It began by explaining the PGP system and its limitations in secure key distribution. The principles of blockchain technology were introduced, emphasizing decentralization, immutability, and suitability for secure key sharing. The role of smart contracts and Solidity in automating trustless operations was discussed, along with relevant blockchain consensus mechanisms. Additionally, zk-SNARKs were explored as a privacy-preserving enhancement to blockchain-based systems. Finally, essential development tools such as Truffle, Ganache, Remix, and ZoKrates were outlined. Together, these foundational elements establish the technical groundwork upon which the subsequent proposed systems, implementations, and evaluations are built in the following chapters.

Chapter 4

Implementation of a Secure PGP Key Sharing Scheme with Blockchain Technology

4.1 Introduction

Email is one of the most important communication services used daily by people to share data across the world [1]. The number of email circulating in the world per day is growing due to its convenience features. Although traditional email system authenticates only on the email server according to the user name and password, while the information itself is stored in plain text on the server [3]. For this reason, it becomes necessary to encrypt and sign the email before sending it from the sender to the recipient. There are many secure email encryption techniques available, and the most popular of them are Secure/Multipurpose Internet Mail Extensions (S/MIME) and PGP [2].

Secure key sharing is a fundamental challenge in cryptography and communications. In email encryption systems such as Pretty Good Privacy (PGP), a common approach is to use hybrid encryption. In PGP, each user has a public/private key pair; the public key is used by others to encrypt messages to the owner, and the private key is kept secret by the owner to decrypt received messages and to digitally sign outgoing messages. Because asymmetric encryption of large messages is computationally expensive, PGP uses a hybrid encryption approach: the actual email message is encrypted using a symmetric cipher with a one-time session key (often called a *message key*), and then this session key is encrypted with the recipient's public key (typically using RSA

or elliptic curve cryptography) for transmission. This ensures confidentiality of the message content and the integrity of the message through signing. However, a significant vulnerability remains in the step of sharing the session key with the recipient.

4.2 Background Problem and Motivation

In traditional PGP, the encrypted session key is sent along with the encrypted email message; if an attacker intercepts this package, both the encrypted message and the encrypted key can be captured at once. An attacker who manages to compromise the recipient's private key could potentially decrypt the session key and thus the message. Therefore, the secure distribution of the email's encryption key (the PGP session key) is critical, and a breach at this stage can undermine the entire email security scheme.

Blockchain technology offers a promising solution to strengthen the key sharing process. Blockchains provide an immutable, tamper-resistant ledger distributed across many nodes. Once data (such as a cryptographic key) is recorded on a blockchain and confirmed via consensus, it becomes extremely difficult for any adversary to alter or delete that data without detection. This property can be leveraged to store and transfer encryption keys in a way that is resilient to tampering and unauthorized access. Moreover, blockchain transactions are transparent and verifiable by all participants in the network, which can increase trust in the key sharing mechanism. Every transaction (such as a key upload or retrieval) must be validated by a majority of nodes, preventing a malicious actor from unilaterally inserting or altering a key. In addition, using a blockchain removes reliance on a centralized key server or certificate

authority; there is no single point of failure, and the risk of man-in-the-middle attacks during key sharing is significantly reduced, since the key retrieval follows pre-defined smart contract rules and consensus verification rather than an ad-hoc direct transfer.

In this chapter, a blockchain-based secure key sharing scheme is presented to address PGP's key sharing vulnerability. The scheme integrates Ethereum blockchain smart contracts with PGP's encryption workflow, using the blockchain as a distributed key distribution platform. The approach is as follows: first, a random symmetric key is generated by the sender to encrypt an email (just as in standard PGP). Next, instead of encrypting this session key with the recipient's public key and sending it directly to the recipient, the session key is encrypted using Elliptic Curve Cryptography (ECC) to produce an encrypted key payload. This encrypted key is then uploaded to the blockchain via a smart contract transaction. The intended recipient can later retrieve this encrypted key from the blockchain by invoking the corresponding smart contract function, and then decrypt it using their private key to recover the original session key.

By separating the transmission of the encrypted key from the email itself and placing it on a tamper-proof ledger, the system enhances security. An adversary would need to compromise the blockchain in addition to intercepting the email, making attacks far more difficult. The proposed solution not only preserves the confidentiality of the session key (through ECC encryption) but also ensures its integrity and availability via the blockchain. Smart contracts are employed to automate the key sharing transactions and enforce access control policies. For example, only the intended recipient is allowed to retrieve the key, and all actions are logged transparently on the ledger.

The contributions of this chapter are listed below:

- We adopt blockchain to share the encrypted key. Due to the distributed characteristics of blockchain, attackers can not easily change the blocks, and it also removes Man-in-the-Middle risk, which ensures the key security.
- All the transactions carried out between participants in the network are recorded on the blockchain. Before adding a transaction to the blockchain, it is validated by network nodes using a consensus method based on the majority. So, the framework increases trust among the users.
- The encrypted key is stored in blocks that are linked together in a chain. One can not delete or modify the chain without consensus from the network, which removes the possibility of encrypted key modifications.

4.3 Proposed System Architecture

Figure 4.1 provides an overview of the proposed key sharing architecture, depicting the interactions between the sender, the blockchain (smart contract), and the receiver [5]. The workflow consists of a sequence of steps that ensure a secure upload and retrieval of the encryption key. The main idea is that the sender will encrypt the session key using the receiver's public key (via ECC) off-chain, then utilize a smart contract on the blockchain to store this encrypted key and later allow the receiver to fetch it. All participants in the blockchain network collectively verify the transactions for storing and accessing the key, providing a decentralized security check.

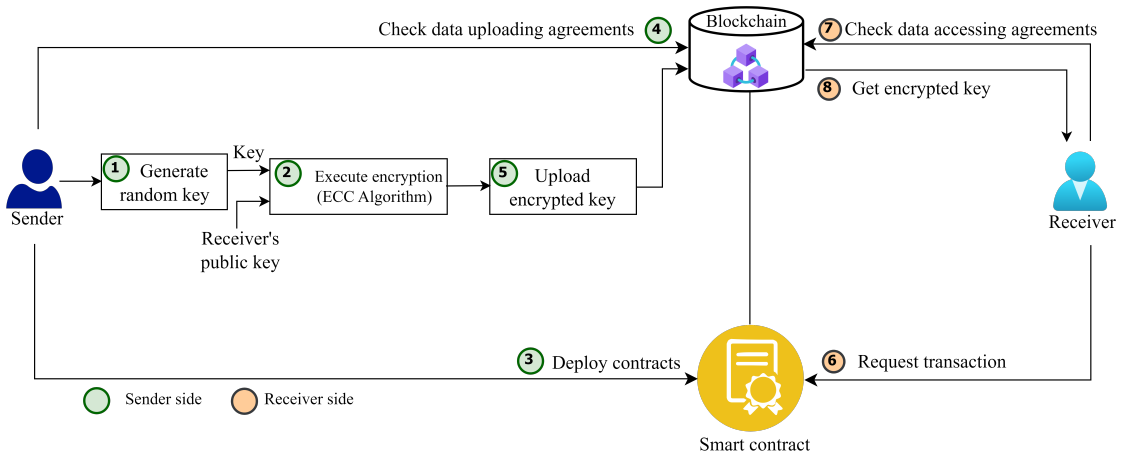

Figure 4.1: Proposed system to share the encrypted key.

The sender and receiver must have to register first before using the blockchain network. It contains a total of 8 steps which are described as follows:

Step 1. Generate random key

At first, the sender generates a random key and it is used to encrypt the email contents. In our proposed system, this random key is generated using a random key generator.

We used the **randomBytes()** function to generate random bytes equal to the desired length of the key. PGP uses a random key length of 128 to 256 bits. In our experiment, we used the random key length equal to 128-bits (16 bytes), 192-bits (24 bytes), and 256-bits (32 bytes). The next step is to convert the generated random bytes to a hexadecimal string. This conversion represents the key to a more readable format. Next, trim the hexadecimal string to ensure it matches the desired key length. The resulting trimmed hexadecimal string is returned as the random key, K.

Step 2. Execute encryption (ECC algorithm)

After the key is generated, we use the ECC algorithm to encrypt the key on the sender side and get the encrypted key, K_e. This K_e is transmitted from

the sender to receiver using our proposed blockchain framework. The receiver first receives this K_e. After that, uses the same ECC algorithm to decrypt it and get K.

Encryption and Decryption:

Consider an elliptic curve $E_p(a, b)$, where a and b define the curve equation. The elliptic curve is defined over a finite field $\mathbb{F}_p$ of prime order p. The base point G and n be the order of the base point. The sender first generates a private key P_{rA} $(0 < P_{rA} < n)$, to compute the corresponding public key $P_A = P_{rA} \cdot G$. The receiver also generates a private key P_{rB} $(0 < P_{rB} < n)$, to compute the corresponding public key $P_B = P_{rB} \cdot G$.

For encryption, the sender encrypts the randon key k. First, choose a random number r $(0 < r < n)$, as a temporary private key and compute the temporary public key $R = r \cdot G$. Next, calculate the shared secret point $S = r \cdot P_B$ and derive the shared secret S_x-coordinates as a symmetric key S_k. Then, encrypt the random key k to generate the encrypted key $K_e = \{S_k, K\}$.

For decryption, receiver side first receives the encrypted key k_e and the temporary public key R. Next, compute the shared secret point $S = R \cdot P_{rB}$ and derive the shared secret S_x-coordinates as a symmetric key S_k. Then, decrypt K_e to get back random key $K = \{S_k, K_e\}$.

Step 3. Deploy contracts

The agreements and the conditions are clearly defined by the sender and receiver before initializing the system. These agreements are written in Solidity programming language, known as smart contracts. After the contract is created, the sender deploys the contract at the beginning at once to initialize the system. After that, the other functions of the smart contracts are executed as per the requirements of the sender and recipient.

Step 4. Check data uploading agreements

When the sender sends a request for the transaction to upload the encrypted key K_e (generated in step 2) in the block of the blockchain, data uploading agreements are checked by every user. If all the data uploading agreements are satisfied by the transaction, it will be successful, otherwise the transaction request will be ignored.

Step 5. Upload encrypted key

After the successful transaction, the sender sends the encrypted key K_e to the blockchain to store it and it is stored as a transaction data.

Step 6. Request transaction

The receiver now wants to access the encrypted key K_e, as it is stored successfully. For this, the receiver makes a transaction request in the deployed smart contract to access K_e.

Step 7. Check data accessing agreements

After the transaction request from the receiver, the data accessing agreements are checked by each user of the blockchain network. If all the agreements of data access are satisfied by the transaction, it will be successful, otherwise the transaction request will be ignored.

Step 8. Get the encrypted key

After the successful transaction, the receiver can access the encrypted key K_e from the blockchain. The receiver now uses the ECC algorithm to decrypt K_e and to get K.

This architecture ensures that at no point is the raw session key K exposed in transit or stored in plaintext on a server. It is either protected by ECC encryption or handled within the secure execution of the smart contract. The use of blockchain means that any attempt to illegitimately obtain or modify

the key would require compromising a majority of the network or the contract's logic, which is vastly more difficult than attacking a single server or tricking a user into a false key sharing. Furthermore, because every access is mediated by the contract, the system can enforce fine-grained policies (for instance, keys can only be accessed by a specific recipient, or only within a certain time window, or only under multi-signature approval if desired).

4.4 Implementation Details

To demonstrate the practicality of the proposed scheme, a prototype implementation was developed using the Ethereum blockchain platform. The implementation involved writing smart contracts in Solidity, setting up a local Ethereum network for testing (using Ganache), and writing client-side scripts (with Truffle and Node.js) to simulate the sender and receiver operations. Key generation and ECC encryption were done off-chain in the client code, while the uploading and retrieval of keys were done via smart contract calls on-chain.

4.4.1 Implementation Environments

To analyze the feasibility of the proposed system, this section introduces the evaluation in details. The implementation environments, model, the required technical tools, and software are introduced in Table 4.1.

The implementation is divided into two parts: the **off-chain components** and the **on-chain smart contract**. The off-chain part of the system handles tasks that are computationally intensive or involve sensitive operations that need not (or should not) occur on the blockchain. In this scheme, the key generation, ECC encryption, and ECC decryption steps are all done off-chain

Table 4.1: Implementation environments.

Software	Configuration/Version
OS	Windows 11 pro
CPU	3.20 GHz Intel Core i7
RAM	16 GB memory
Truffle	5.11.0
Ganache	2.5.4.0
Node.js	16.16.0

by the sender's and receiver's software.

4.4.2 Smart Contract Execution

This section describes the proposed system including the necessary functions required. It also summarizes the smart contract creation, deployment, and execution.

1. Smart Contract Creation

Based on the proposed system, in smart contract we created three functions to store encrypted key K_e in the blockchain by the sender and to access it by receiver. These three functions are: **RegisterUser()**, **UploadData()**, and **AccessData()** functions. At the beginning of the process, all users need to register in the system with the **RegisterUser()** function. In our proposed system, we considered a case study scenario to create smart contracts with these functions. Suppose in a private organization, the "Chief Executive Officer (CEO)" wants to share the encrypted key K_e, with the "Manager". Based on this case, users need to provide input parameters like 'user name', 'user email id', 'organization name', and 'user position' for the registration. We made 'user email id' as the unique parameter and an email id only used one time to register a user. The **UploadData()** and **AccessData()** are used to

store K_e in the block and get it back from the block, respectively.

2. Smart Contract Deployment

Before deploying the smart contract, the contract needs to connect with the testing platform of Ethereum. In our proposed system, we tested the contract in the Ganache platform. Firstly, the contract is connected to the Remote Procedure Call (RPC) server to allows external applications to interact with the Ethereum blockchain. After that, for the contract to be usable on the Ethereum network, it needed to be deployed. After the deployment, the bytecode of the contract is stored in the blockchain as transaction data and cannot be changed or modified. This immutability ensures that the contract's logic and state remain consistent and secure.

4.4.3 Access Control Transactions

The access control transactions for the proposed system are implemented with consensus and validation policy using smart contracts. In our proposed system, access control indicates the users who can upload the encrypted key K_e and access it in the blockchain. The steps involved in the **UploadData()** transaction are shown in Figure 4.2. The explanation of the steps for **UploadData()** transaction by the sender are as follows:

I) The sender generates K_e and embeds the smart contract, and then deploys it in the blockchain.

II) The sender makes a **UploadData()** transaction request to store K_e in the block.

III) The smart contract broadcasts the transaction in the blockchain network.

IV) A new block will be produced for the validated transaction when the transaction has been verified by all of the users of the blockchain network. Finally, the K_e is stored in the newly created block.

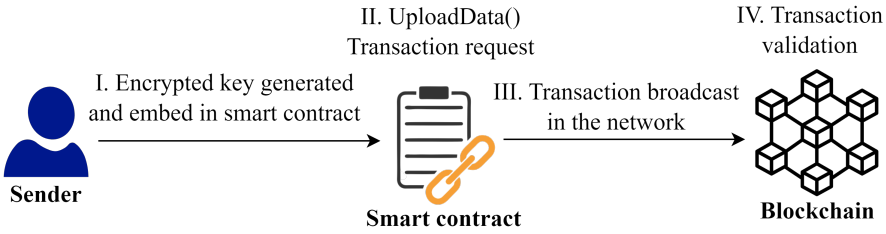

Figure 4.2: UploadData() transaction: Sender upload data.

The user asking to access the k_e using **AccessData()** transaction as shown in Figure 4.3. The detailed steps for a receiver asking to access k_e are as follows:

I) The receiver sends an **AccessData()** transaction request to access K_e from the block.

II) The smart contract broadcasts the transaction in the blockchain network.

III) All the users of the blockchain network validate the transaction first and then a new block will be added according to the verified transaction. The receiver can now access the K_e from the blockchain network.

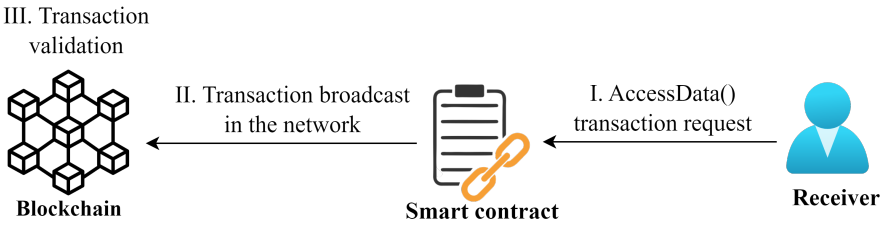

Figure 4.3: AccessData() transaction: Receiver asking to access data.

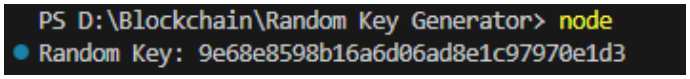

Figure 4.4: Generated random key.

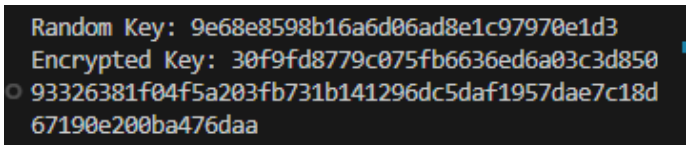

Figure 4.5: Encrypted key (using ECC encryption).

4.5 Evaluation and Result Analysis

The proposed system was evaluated in terms of its feasibility and performance on the Ethereum blockchain (using Ganache, a private Ethereum network, for controlled testing). The implementation is divided into two parts: the off-chain components and the on-chain smart contract. According to our proposed system, we consider the key generation, key encryption, and key decryption parts of their execution outside of the blockchain. In this case, the random key K, is generated first by the user. After that, the sender encrypts the random key using the receiver's public key to get the encrypted key K_e. The output of the generated random key and the encrypted key are shown in Figure 4.4 and Figure 4.5, respectively. In this case, we took a random key size of $K = 32$ bytes (256-bits). We used ECC-256-bit key pair encryption to encrypt K and got an encrypted key size of $K_e = 48$ bytes (384 bits).

For on-chain environment, we used Truffle as a development framework and ganache as the testing platform of Ethereum. The encrypted key K_e, generated from the off-chain environment is accessed first from the solidity code. Then it should be passed through the smart contract to visualize it in the ganache platform. In this case, **UploadData()** takes K_e string as an input and then it is passed as an argument of the function. After that,

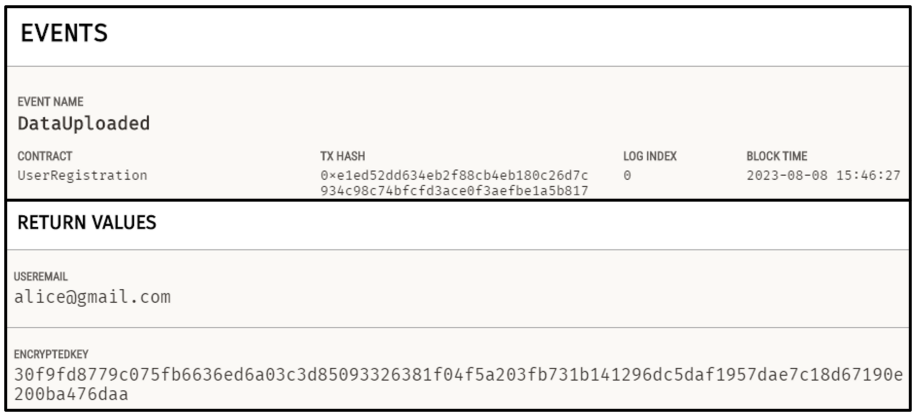

Figure 4.6: UploadData() transaction output.

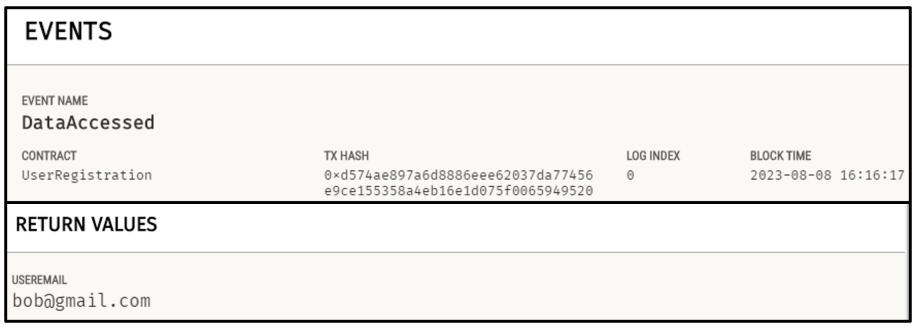

Figure 4.7: AccessData() transaction output.

UploadData() is executed as a transaction to store K_e in the blockchain. After the successful transaction, in ganache platform, we now see the bytecode of the transaction as transaction data. To see the original data from the blockchain, we created events within the function that emits the information from the smart contract to the user. Figure 4.6, illustrates the output of the **UploadData()** transaction. In this example, we consider "Alice" as a sender and "Bob" as a receiver. We also use the email addresses as "alice@gmail.com" and "bob@gmail.com" to register the sender and receiver, respectively in the system.

Now the receiver sends an **AccessData()** transaction request to access K_e. The output of this transaction is shown in Figure 4.7. After the successful transaction, the receiver now able to get K_e from the blockchain. After getting K_e, receiver use his/her private key to decrypt K_e and get the random key K, shown in Figure 4.8.

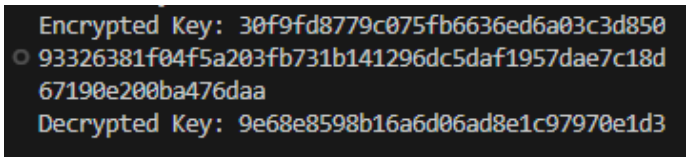

Figure 4.8: Decrypted key (using ECC decryption).

4.5.1 Gas and Transaction Cost Analysis

We have constructed a blockchain-based secure key-sharing system for email applications, and the smart contracts of this system have been compiled and deployed on the Ethereum blockchain platform. This platform can calculate the amount of work done in the form of a unit called gas. Table 4.2 shows the gas and computational costs of executing smart contract's several

operations. During the analysis period, the exchange rate of Ethereum and the US dollar in October 12, 2023 was 1Ether=1529.41USD. The minimum gas price for executing a trade is 1GWEI=10^{-9}Ether. Note that we have used the test Ethereum network and these gas values are just test values, not real cryptocurrency.

Table 4.2: Gas usage and transaction costs of smart contract operations.

Operation	**Gas (GWEI)**	**Cost (ETH)**	**Cost (USD)**
Deployment	1743924	0.001744	2.6672
RegisterUser()	134916	0.000135	0.2064
UploadData()	57144	0.000057	0.0872
AccessData()	36083	0.000036	0.0551

** 1 GWEI = 10^{-9} Ether, 1 Ether = 1529.41 USD on October 12, 2023.*

At first, the cost of the deployment of smart contacts is calculated. This deployment cost is calculated at the beginning at once to initialize the system. After that, the execution cost of the several operations of smart contracts is calculated. In our case study, the system requires a cost of $2.6672 to deploy the smart contracts and initialize the system. The cost required to execute RegisterUser(), UploadData(), and AccessData() functions are $0.2064, $0.0872, and $0.0551, respectively. This analysis demonstrates the feasibility of the proposed scheme in the real world concerning the required gas values and computational costs [63].

In the baseline scenario (one sender, one receiver, one encrypted key per transaction), we measured the gas consumption of each major operation in the protocol:

- **Contract Deployment:** Deploying the smart contract to the Ethereum network consumed roughly 1.74×10^6 gas. This was the highest single cost since it involves writing the contract's bytecode into the blockchain.

At a gas price of 1 GWEI, this corresponds to about 0.00174 ETH, which was approximately 2.67 at the time of testing. Deployment is a one-time cost; once the contract is deployed, it can handle an arbitrary number of key exchanges.

- **User Registration (`RegisterUser`):** Registering a user (which writes the user's info into the contract storage) used about 1.349×10^5 gas (0.000135 ETH, or around 0.20). This cost is incurred for each new user but is relatively low.

- **Key Upload (`UploadData` for one 32-byte key):** Uploading one encrypted key of 32 bytes used approximately 5.7×10^4 gas (0.000057 ETH, or roughly (0.087). This covers the cost of the function execution and storing 32 bytes of data on-chain. If the encrypted key size were larger, this cost would increase proportionally (storing more bytes costs more gas).

- **Key Access (`AccessData` for one key):** Retrieving the key (and emitting it in an event) was around 3.6×10^4 gas (0.000036 ETH, or about (0.055). This is mostly the cost of reading the stored data and generating an event.

These results, summarized in Table 4.2, demonstrate that the on-chain overhead of the scheme is quite manageable. The most expensive part, contract deployment, is a one-off setup cost. Each individual email key sharing (consisting of one upload and one access transaction) in proposed scheme costs on the order of 0.14 in transaction fees at 1 GWEI gas price, which is negligible for most use cases (and on a private network, the "cost" is not an actual concern, though gas usage still matters for performance).

Overall, the single-recipient case confirms that using the blockchain for key sharing is feasible and not prohibitively expensive in terms of gas. The cryptographic operations were off-loaded to off-chain software, so the on-chain work is limited to simple storage and verification tasks.

4.5.2 Security Analysis

The smart contract approach ensures that certain security policies are enforced automatically by code:

- **Access control:** Only authorized senders can upload keys, and only intended recipients can retrieve them. The contract checks the identity (address) and possibly role of the caller on each function call. For example, if an unauthorized address tries to call `UploadData`, the transaction will fail. Similarly, if someone other than the intended receiver tries to call `AccessData`, they will either get no data or a rejection.

- **Integrity and immutability:** Once a key is uploaded and the transaction is confirmed, that encrypted key cannot be altered or removed by a malicious party. The blockchain's immutability guarantees that K_e stays as originally recorded until the legitimate receiver accesses it. Even the sender cannot "take it back" or change it after the fact (without a new transaction), which prevents a malicious sender from swapping out the key after sending the email content.

- **Auditability:** Every upload and access leaves a trace on the blockchain. In an organizational context, this means there is a clear log of who shared what with whom and when. If there is ever a dispute or investigation, the blockchain records (and contract events) can provide evidence of key

distribution. This is an advantage over standard PGP email, where key sharing happens peer-to-peer and might not be recorded.

- **No Trusted Intermediary:** The smart contract itself acts as the intermediary for key sharing mechanism, removing the need to trust an email server or key server for key delivery. The rules of key sharing are hard-coded and transparent. This eliminates certain attacks, such as a compromised email server injecting its own key, because the contract would not authorize an unregistered key or user.

- **Mitigation of MITM:** Because the encrypted key is not sent directly from sender to receiver but retrieved through the blockchain after consensus verification, a traditional man-in-the-middle (who intercepts email traffic) cannot stop the receiver from obtaining the correct key, nor can they substitute a fake key unless they somehow gain majority control of the blockchain network or the smart contract (which is exceedingly difficult). In contrast, in a standard PGP exchange, a MITM could attempt to send the receiver a fake encrypted session key if they tricked the sender into using the wrong public key. Here, that scenario is mitigated by the blockchain verification steps.

It is also important to consider the confidentiality of metadata. Our scheme makes the encrypted key K_e visible on the blockchain (to all participants of the blockchain network). While K_e is safe by cryptographic means (assuming ECC and symmetric encryption are secure), the fact that a key was exchanged between two parties is public. In a public blockchain like Ethereum mainnet, this could raise privacy concerns; however, in a private or consortium blockchain (such as one within an organization), this might be acceptable or

even desired for oversight. Solutions like storing only a hash of K_e on-chain (with actual K_e in a secure off-chain storage accessible via the contract) or using zero-knowledge proofs could be explored to hide even the encrypted key content on a public ledger.

Another consideration is the long-term security of the keys. The blockchain will retain the encrypted key K_e indefinitely in its history (even if the contract "deletes" it, the transaction log remains). Thus, if the receiver's private key d_B were compromised at any point in the future, an adversary could retrieve K_e from the blockchain history and decrypt it. This is analogous to the usual PGP concern: if someone's private key is compromised, all past emails encrypted to that key become readable. Our scheme shares that limitation. Mitigations include using ephemeral recipient key pairs (so that even if one is compromised, it only affects messages in a certain time frame) or rotating keys frequently. In an enterprise scenario, if an employee leaves or a key is suspected to be compromised, the corresponding blockchain account and keys should be revoked and new ones used, and old keys should ideally be removed or made inaccessible in the contract (smart contracts can implement revocation lists or key versioning).

In summary, the implemented system uses blockchain and smart contracts to enforce that only legitimate users can share and access encryption keys, providing a robust defense against several attack vectors inherent in email key sharing. The cryptographic strength of ECC ensures confidentiality of the keys, while the blockchain ensures the integrity and controlled accessibility of those keys.

4.5.3 Discussion

The evaluation demonstrates that the integration of blockchain into the key sharing process is feasible with acceptable overhead. The absolute monetary cost in a public chain scenario is likely acceptable for secure communications in a business or high-security context. In a private blockchain deployment, the concept of cost is not directly relevant, but gas usage still correlates with performance and resource consumption on the nodes. Our gas usage figures indicate the approach is lightweight.

Latency is another factor: on a public Ethereum network, waiting for a block confirmation (perhaps 15 seconds or more) adds a delay to the key sharing mechanism. In many email scenarios this delay is not critical (emails are not always read instantaneously), but users might not want to wait too long for the key to become available. In a private network with faster block times or an optimized consensus, this could be in the order of a second or less. In any case, the security benefits have to be weighed against this delay.

From a security standpoint, the evaluation focused on the system working correctly under normal conditions. A full security analysis would also consider what happens under various threat scenarios (e.g., a malicious user trying to register multiple times, an attacker sniffing network traffic, etc.). Our implementation assumed honest behavior in registration (since verifying identity was out of scope), but in a real deployment, registration could be tied into an organization's identity management or require out-of-band approval to prevent fraudulent identities.

To strengthen the scheme further, one could incorporate cryptographic techniques like zero-knowledge proofs. For example, a zk-SNARK could allow

the contract to verify a user's right to access a key without revealing which key (adding privacy). Or attribute-based encryption (ABE) could allow one ciphertext to be decrypted by multiple recipients based on roles, potentially reducing on-chain storage by having one key for a group (though that reintroduces a single key shared by many, which has other drawbacks). These are possibilities for future enhancements.

In conclusion of the evaluation, the results show that the proposed blockchain-based key sharing scheme is not only functionally effective but also efficient enough to be practical. The additional overhead (in terms of gas and time) is small compared to the security gains of eliminating dedicated key sharing vulnerabilities. The scheme scales linearly and can handle both one-to-one and one-to-many key distribution scenarios within the capacity of typical blockchain parameters.

4.6 Summary

This chapter presented a blockchain-based scheme for secure key sharing mechanism in email systems, integrating it with the PGP encryption workflow. By leveraging blockchain and smart contracts, the perennial problem of safely sharing the encryption session key in PGP is addressed with a novel approach: rather than sending the key directly to the recipient along with the message, the key is encrypted with ECC and stored on an immutable ledger, and only the intended recipient can retrieve it after network-wide verification. This design effectively removes the need for a trusted central key server and mitigates the risk of man-in-the-middle attacks during key exchange.

The implemented Ethereum smart contract enforces rules that ensure only

authorized participants can upload or download keys, and it logs all such transactions for transparency and auditability. Experiments on a test network confirmed that the system is practical: the gas costs for using the contract are modest, and the overhead introduced is minimal (on the order of tens of thousands of gas for key operations, which translated to only cents in cost). The approach also scales linearly for multi-recipient distributions, meaning that it can be used for sending encrypted communications to groups without a prohibitive increase in cost or complexity.

In summary, the blockchain-based secure key exchange scheme demonstrates that distributed ledger technology can be effectively applied to enhance the security of email communication. By combining the immutability and decentralization of blockchain with robust encryption (ECC for content keys), the system ensures confidentiality, integrity, and availability of the keys that protect our digital communications. This contributes towards building a more secure and trust-minimized email infrastructure, and the methodology can be extended to other applications requiring secure key distribution.

Chapter 5

Integration of zk-SNARKs with Blockchain for Privacy Preservation

5.1 Introduction

Secure key sharing is critical for protecting confidential communication (e.g. email). Traditional email encryption methods like PGP require out-of-band sharing of private keys, which is vulnerable to interception or man-in-the-middle (MITM) attacks. Blockchain technology offers an immutable, decentralized ledger that can serve as a trustworthy key distribution channel. Once a key (or its ciphertext) is recorded on the blockchain, it cannot be altered without consensus, eliminating the single-point-of-failure in traditional key servers. In our scheme, we leverage this immutability: the sender stores the encrypted symmetric key (for example, a PGP session key) on-chain, ensuring it is tamper-resistant.

However, blockchains are by default public: any data put on-chain (like key identifiers or user IDs) can be seen by all. This transparency conflicts with privacy goals. To address this, I integrate zk-SNARKs (Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge) into the key sharing process. zk-SNARKs allow one party to prove knowledge of a secret (e.g. that they hold a valid role or key) without revealing the secret itself. In our system, a prover (the sender) generates a zk-SNARK proof attesting to the authorization conditions (such as "the uploader is the intended holder of the key") without revealing their identity or any private attribute. The smart contract on-chain

then verifies this proof; if valid, it releases the encrypted key to the recipient.

5.2 Background Problem and Motivation

While blockchain offers immutability, transparency, and decentralization, these very features can introduce privacy vulnerabilities in applications that handle sensitive information, such as secure email systems. Transactions recorded on public blockchains like Ethereum are visible to all participants, which can expose metadata including sender and receiver addresses, transaction content identifiers, and function calls [64]. In the context of blockchain-based key distribution for secure email, this transparency risks revealing not only the existence of a communication link but potentially inferring the key sharing pattern, compromising confidentiality [65].

To mitigate such privacy leakage, several privacy-preserving methods, like ring signature, homomorphic encryption, and zero-knowledge proof (ZKP), have been used to blockchain applications to address the privacy issues in the blockchain [23]. Zero-knowledge proofs (ZKPs) have emerged as a compelling cryptographic primitive in this case. Specifically, zk-SNARKs allow one party to prove possession of certain knowledge (e.g., a secret key) without revealing the knowledge itself and with minimal interaction. By integrating zk-SNARKs into blockchain smart contracts, we can ensure that a party can validate their right to access encrypted data or perform sensitive actions without exposing private inputs or operations to the public ledger.

This chapter presents a privacy-preserving key access mechanism by embedding zk-SNARKs within the Ethereum-based smart contract framework for secure email communication. In our design, the session key encrypted under

ECC is stored on the blockchain, similar to the previous chapter. However, instead of allowing the recipient to directly retrieve the encrypted key via a smart contract function (which would reveal access patterns and identifiers), the recipient must first generate a zk-SNARK proof off-chain using the ZoKrates toolkit [66]. When a user requests access to an encrypted session key stored on the blockchain, they submit a zero-knowledge proof instead of their actual identity or key. The smart contract verifies this proof to determine eligibility and grants access only if the proof is valid, without learning anything about the private input.

The proposed method ensures that the access control logic is verifiable without leaking any sensitive input, thus improving privacy. Unauthorized users cannot derive or infer key related operations even by monitoring the blockchain. Furthermore, since zk-SNARKs are succinct and efficient, the additional computational and gas overhead remains acceptable within Ethereum's constraints [67]. As a result, this approach balances the need for decentralization and transparency with the essential requirement for confidentiality in secure communications.

The contributions of this chapter are listed below:

- Integrated zk-SNARKs with blockchain to share the encrypted key of PGP system, which can ensure the anonymity of transaction participants and increases privacy of data.

- Kept the encrypted key in a chain of interconnected blocks. The option of changing an encrypted key is eliminated since it is difficult to remove or alter the chain without the entire network's consent.

- Created the necessary proofs of our system and stored them in the

blockchain framework, which can significantly reduce storage requirements.

5.3 Overview of The System

This section introduces proposed system of encrypted key sharing mechanism using blockchain with the integration of zk-SNARKs. In the previous paper as mentioned in [5], we only considered the blockchain technology for sharing the encrypted key. Now, we also added zk-SNARKs with the blockchain technology to share the encrypted key. For this, the proof of the zk-SNARKs have to be generated first as discussed in section 5.3.1. After that, the encrypted key have to be shared according to the system described in section 5.3.2.

5.3.1 zk-SNARKs integration for Transaction Verification

Security, efficiency, and privacy are the main goals of zK-SNARKs for blockchain-based identity sharing. Ensuring privacy is crucial since it allows users to authenticate themselves on the blockchain without disclosing private information. For real-time interactions, the system has to be computationally efficient to guarantee quick verification and proof creation. Moreover, conciseness, smaller proofs, and less storage to improve the system's scalability. Figure 5.1 shows a summary of how zk-SNARKs are integrated with blockchain for transaction verification. It contains the following steps which are described as follows:

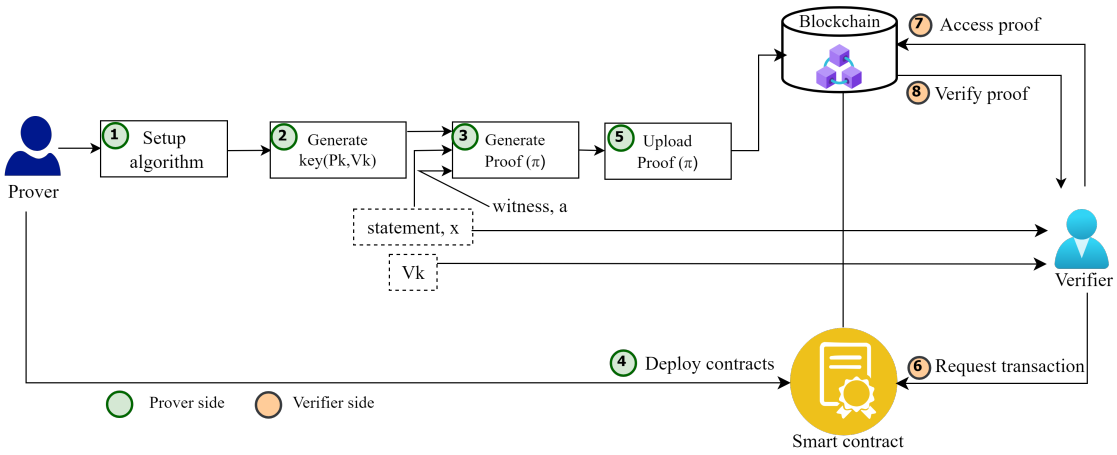

Figure 5.1: zk-SNARKs integration to the system for transaction verification.

Step 1. Setup algorithm

Initially, a security parameter λ is used to construct a set of public parameters pp. These values, which come from secret values during a trusted setup procedure, make up the common reference string (CRS). Information about the user's identification are indicated by the notation
$I = attribute1, attribute2...attributeN$.

Step 2. Generate Key (pk, vk)

Next, depending on the user's input data and blockchain-specific needs, a set of constraints representing

$Constraints = constraint1, constraint2, ..., constraintM$, the necessary characteristics must be created and designated. The key generator method then produces a verification key (vk) and a proving key (pk) given a circuit C. The prover generates proofs with the help of the proving key, and the verifier uses the verification key to confirm the proofs.

Step 3. Generate proof (π)

The prover uses the blockchain-adapted zk-SNARKs proof generating algorithm with a proving key (pk), a public statement (x), and a private witness

(a) as inputs to generate a non-interactive proof (π) for the statement. This proof demonstrates the relationship established by the circuit C between the variables x and a. This evidence verifies the accuracy of the identity-related data without disclosing the true values. To do this, one must solve the zk-SNARKs equation in the context of the blockchain: $blockchain - proof = prove(pk, x, a)$.

Step 4. Deploy contracts

Before the system is initiated, the sender and recipient explicitly declare the agreements and terms of the transactions. Smart contracts are used to write these agreements. Upon creation of the contract, the sender immediately deploys it to start the system from beginning. The remaining smart contract operations are then carried out in accordance with the sender's and receiver's specifications.

Step 5. Upload proof

Next stage to upload the generated blockchain specific proof on the blockchain, associating it with the user's blockchain address or identifier.

Step 6. Request transaction

The verifier now wants to verify the proof, as it is stored successfully. For this the verifier need to make a transaction request first to access the proof stored in blockchain.

Step 7. Access proof

After the successful transaction, the verifier now can access the blockchain proof to verifity it. The zk-SNARKs verification parameters like public statement (x), verification key (vk) also needed as a input to verify the proof.

Step 8. Verify proof

Now that the user's blockchain address or identification is linked to the blockchain, the verifier retrieves the identity evidence unique to the blockchain.

To ascertain if the proof is legitimate or not, the verifier runs the proof verification algorithm using a verification key (vk), a public statement (x), and a zk-SNARK proof (π) as values. The following equation is used for verification: $Verify(vk, x, \pi) = true\ or\ false$?

In the context of blockchain verification, this means assessing the blockchain-specific restrictions in proof and making sure they stay true while keeping the genuine identity traits hidden. If the verification is accurate, the blockchain-specific proof confirms successfully, and the verifier may rely on the identity characteristics' authenticity inside the blockchain context without having access to the user's private data.

We will get the solidity file *(verifier.sol)* based on the proof generated in this section. This proof will be used as an input parameter for the transaction verification in the next section (section 5.3.2). We also have to import this solidity file (*verifier.sol*) into the solidity file of key sharing scenario (section 5.3.2) to integrate the zk-SNARKs concept to the proposed system.

5.3.2 Encrypted Key Sharing using Blockchain

An overview of the key sharing scenario of proposed system is illustrated in Figure 5.2. The sender and receiver must have to register first before using the blockchain network. The solidity file (verifier.sol) that already generated (section 5.3.1) using ZoKrates tool have to import first to the proposed system's solidity file for transaction verification. It contains a total of 8 steps which are described as follows:

Step 1. Generate random key

At first, the sender generates a random key and it is used to encrypt the email contents. In our proposed system, this random key is generated using a

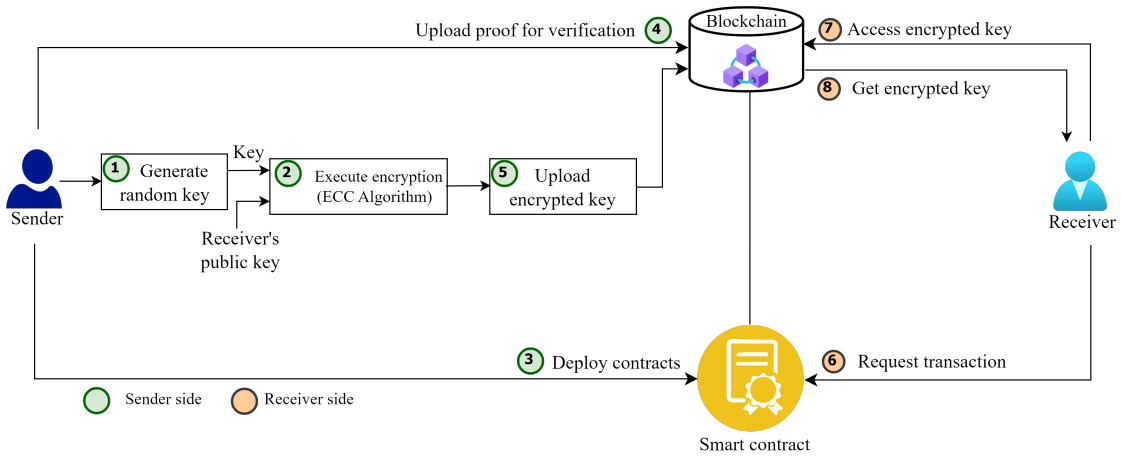

Figure 5.2: Proposed system to share the encrypted key.

random key generator.

We used the **randomBytes()** function to generate random bytes equal to the desired length of the key. PGP uses a random key length of 128 to 256 bits. In our experiment, we used the random key length equal to 128-bits (16 bytes), 192-bits (24 bytes), and 256-bits (32 bytes). The next step is to convert the generated random bytes to a hexadecimal string. This conversion represents the key to a more readable format. Next, trim the hexadecimal string to ensure it matches the desired key length. The resulting trimmed hexadecimal string is returned as the random key, K.

Step 2. Execute encryption (ECC algorithm)

After the key is generated, we use the ECC algorithm to encrypt the key on the sender side and get the encrypted key, K_e. This K_e is transmitted from the sender to receiver using our proposed blockchain framework. The receiver first receives this k_e. After that, uses the same ECC algorithm to decrypt it and get K.

Encryption and Decryption:

Consider an elliptic curve $E_p(a, b)$, where a and b define the curve equa-

tion. The elliptic curve is defined over a finite field $\mathbb{F}_p$ of prime order p. The base point G and n be the order of the base point. The sender first generates a private key P_{rA} ($0 < P_{rA} < n$), to compute the corresponding public key $P_A = P_{rA} \cdot G$. The receiver also generates a private key P_{rB} ($0 < P_{rB} < n$), to compute the corresponding public key $P_B = P_{rB} \cdot G$.

For encryption, the sender encrypts the random key k. First, choose a random number r ($0 < r < n$), as a temporary private key and compute the temporary public key $R = r \cdot G$. Next, calculate the shared secret point $S = r \cdot P_B$ and derive the shared secret S_x-coordinates as a symmetric key S_k. Then, encrypt the random key K to generate the encrypted key $K_e = \{S_k, K\}$.

For decryption, receiver side first receives the encrypted key K_e from the blockchain and the temporary public key R from the key server used by normal PGP. Next, compute the shared secret point $S = R \cdot P_{rB}$ and derive the shared secret S_x-coordinates as a symmetric key S_k. Then, decrypt K_e to get back random key $K = \{S_k, K_e\}$.

Step 3. Deploy contracts

The agreements and the conditions are clearly defined by the sender and receiver before initializing the system. These agreements are written in Solidity programming language, known as smart contracts. After the contract is created, the sender deploys the contract at the beginning at once to initialize the system. After that, the other functions of the smart contracts are executed as per the requirements of the sender and receiver.

Step 4. Upload proof for verification

When the sender sends a request for the transaction to upload the encrypted key K_e (generated in step 2) in the block of the blockchain, the transaction will be verified first. For this, the sender needs to upload the proof and

the receiver will verify it along with the statement and the verification key. If the proof verification becomes true for the transaction, it will be successful, otherwise the transaction request will be ignored.

Step 5. Upload encrypted key

After the successful transaction, the sender sends the encrypted key K_e to the blockchain to store it and it is stored as a transaction data.

Step 6. Request transaction

The receiver now wants to access the encrypted key K_e, as it is stored successfully. For this, the receiver makes a transaction request in the deployed smart contract to access K_e. If the proof of the transaction is valid, it will be successful, otherwise the transaction request will be ignored.

Step 7. Access encrypted key

After the successful transaction, the receiver now can access the encrypted key stored in the block of the blockchain.

Step 8. Get encrypted key

After accessing the block, the receiver now can take the encrypted key K_e from the blockchain. The receiver now uses the ECC algorithm to decrypt K_e and to get K.

5.4 Implementation Details

This section highlights the simulation environments, result analysis of the system and the comparison of proposed work with previous work in details. The necessary tools for the off-chain and on-chain environment are summarized in section 5.4.1. After creating the proof, and transmiting the encrypted key through blockchain, the necessary results are highlighted in section 5.5.

5.4.1 Implementation Environments

To analyze the feasibility of the proposed system, the details of the evaluation are described here. The implementation environments, model, required technical tools, and software are introduced in Table 5.1.

Table 5.1: Implementation environments.

Software	**Configuration/Version**
OS	Windows 11 pro
CPU	3.20 GHz Intel Core i7
RAM	16 GB memory
Truffle	5.11.0
Ganache	2.7.1
Node.js	16.16.0
ZoKrates	0.8.4
Remix	0.8.22

To execute and run our proposed system, we consider two environments. One is the off-chain environment that will work outside of the blockchain and another is the on-chain environment that works inside of the blockchain. According to our proposed system, we consider the proof generation, key generation, key encryption, and key decryption parts of their execution outside of the blockchain.

5.4.2 Implementation Components

We implemented the above scheme using standard Ethereum toolchains and the ZoKrates zk-SNARK framework. The main components are:

- **Cryptographic setup:** We used ZoKrates to write a .zok program specifying the authorization circuit. In our prototype, the circuit checks three boolean witnesses (a, b, c) indicating (1) the user is registered,

(2) belongs to company ABC, and (3) has position CEO (value 1 for all three). ZoKrates compiled this into a verification circuit and generated a proving key and a Solidity verifier contract.

- **Key generation and encryption:** Off-chain (in our test environment), the sender software generates a 32-byte random key k and encrypts it using ECC-256 (e.g. using a standard ECC library). The resulting ciphertext ke is 48 bytes (384 bits). This ciphertext is supplied as an argument to the smart contract call.

- **Smart contracts:** We developed a Solidity contract (named UserRegistration.sol) with two primary functions: UploadData(bytes ke, Proof pi) and AccessData(). The contract stores a mapping of transaction IDs to ciphertexts. In the non-ZK baseline, UploadData would require passing user fields (username, email, position) to record who uploaded. In our ZK-enabled version, UploadData instead accepts the zk-SNARK proof pi (the struct generated by Verifier.sol) and calls verifyTx(pi) from the imported Verifier contract. We modified the contract to unpack the proof components: constructing Pairing.G1Point and G2Point from the proof inputs as required by the verifier. If verifyTx returns true, the contract records the encrypted key and emits an event with the stored ciphertext. AccessData() is then a simple function the recipient calls to retrieve the ciphertext (also via event logs).

- **Development environment:** We used Truffle as the development framework and Ganache for a local Ethereum blockchain. Truffle migrations deployed the contracts onto Ganache. We also used the Remix IDE to interactively compile and debug the Solidity code. The UserRegis-

tration.sol contract was tested by simulating users “Alice” and “Bob” with corresponding roles (Alice as “CEO”, Bob as “Manager”). Each upload/access operation was a transaction on Ganache.

- **Integration of ZoKrates:** The ZoKrates toolkit was used to perform the trusted setup (setup) and compute proofs (compute-witness and generate-proof). For example, to generate a proof of CEO authority, we ran ZoKrates to produce the proving key and proof π given the private inputs (a=1, b=1, c=1). We then took the proof JSON output and fed its components into our Solidity UploadData call. The main contract includes the Verifier logic, so verification is done entirely on-chain.

- **Event logging:** To convey results to the users, our contract emits events. In UploadData, after successful verification, an event includes the transaction hash or stored key identifier. The recipient watches for this event and then calls AccessData() to get the ciphertext.

- **Scenario details:** For a concrete example, we implemented a scenario where Alice (CEO) uploads a key for Bob (Manager). Without zk-SNARKs, Alice would call UploadData with her identity fields. With zk-SNARKs, she instead submits the proof. In both cases, AccessData by Bob returns the encrypted key. In the ZK case, the verifier contract checks the proof and automatically authorizes the key retrieval without ever seeing Alice’s username or email.

We used ZoKrates tool to execute the zk-SNARKs concept in our solidity code. To integrate zk-SNARKs into the solidity code using ZoKrates involves creating a seperate file **ZoKrates (.zok)** first. This file describes the logic of

the zero-knowledge proof. The proof is generated based on the statement of the transaction, appropriate witness and the corresponding proving key. According to our study, for example, in a private organization, suppose the “Chief Executive Officer (CEO)”, wants to share the encrypted key with the “Manager". For doing this, we have a function in the solidity file called **UploadData()**, using this the “CEO" can upload the encrypted key into the blockchain. The input paramaters of the **UploadData()** are “USERNAME", “USEREMAIL" and “POSITION". So we want to create a zk-SNARKs proof that indicating the transaction comes from “CEO" of the organization without revealing any input information about the user uploading the key. To generate the proof, we have to follow several steps. Firstly, have to create a file **ZoKrates (.zok)** for the **UploadData()** to demonstrate that the user has the authority to upload data. The statement we want to prove is that the user uploading the key is the “CEO" of the organization. Secondly, we have to specify the witness of the proof. The witness is a set of private inputs that satisfy the constraints of ZoKrates file. In this case, the witness includes information that proves the “CEO" status without revealing the actual CEO’s identity. The witnesses for this case are as follows:

- a (User is registered): Witness for $a : 1$ (indicating the user is registered).
- b (User’s company is ABC): Witness for $b : 1$ (indicating the user is associated with the company ABC).
- c (User’s position is CEO): Witness for $c : 1$ (indicating the user holds the position of CEO).

Thirdly, based on the statements and the witnesses, ZoKrates generate a set of keys called proving key and verification key to generate and verify the proof,

respectively. Finally, the proof is generated based on the above statement, witnesses and corresponding proving key. After that, the proof is verified based on the verification key produced by ZoKrates tool. Based on this proof, it will be decided whether the transaction will be valid or not. The proof generated based on our proposed system in shown in Figure 5.3.

After generating the proof, the verifier key is used by the aggregator associated smart contracts. This file **(Verifier.sol)** contains the Solidity code for the ZoKrates verifier contract. Now, we have to import this verifier contract **(Verifier.sol)** to the main contract **(UserRegistration.sol)** of our proposed system. To do so, we need to include the content of the **"Verifier.sol"** file directly into the **"UserRegistration.sol"** file. According to the the Proof struct in the **"Verifier.sol"** file, we need to modify the main contract **(UserRegistration.sol)** file to replace Pairing.G1Point and Pairing.G2Point with the actual values from the proof generated by the ZoKrates. We need to extract the individual components of proof and construct the Pairing.G1Point and Pairing.G2Point objects accordingly. Based on this proof, now we can modify the **UploadData()** of the main contract.

In the key sharing scenario, the random key K, is generated first by the user. After that, the sender encrypts the random key using the receiver's public key to get the encrypted key K_e. In this case, we took a random key size of $K = 32$ bytes (256-bits). We used ECC-256-bit key pair encryption to encrypt K and got an encrypted key size of $K_e = 48$ bytes (384 bits).

For on-chain environment, we used Truffle as a development framework and ganache as the testing platform of Ethereum. The encrypted key K_e, generated from the off-chain environment is accessed first from the solidity code. Then it should be passed through the smart contract to visualize it in the

Figure 5.3: The generated proof using zk-SNARKs.

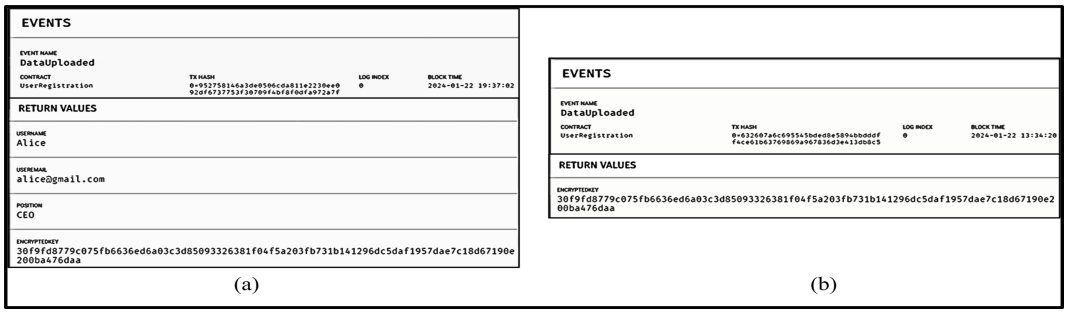

Figure 5.4: Transaction output for UploadData() (a) without zk-SNARKs and (b) with zk-SNARKs.

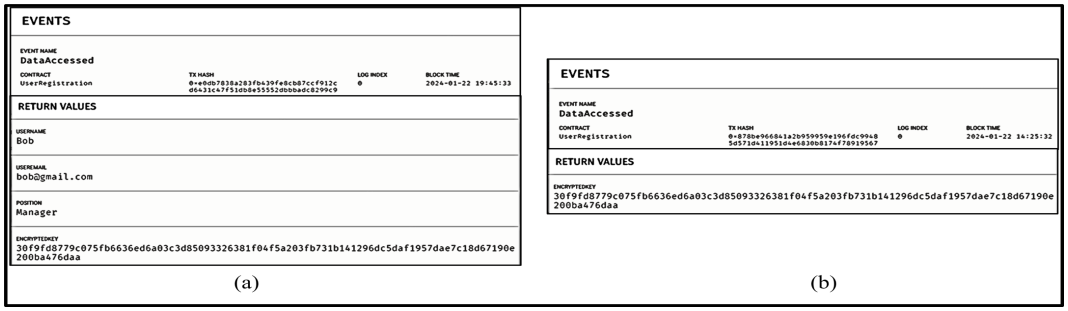

Figure 5.5: Transaction output for AccessData() (a) without zk-SNARKs and (b) with zk-SNARKs.

ganache platform. In this case, **UploadData()** takes K_e string as an input and then it is passed as an argument of the function. After that, **UploadData()** is executed as a transaction to store K_e in the blockchain. After the successful transaction, in ganache platform, we now see the bytecode of the transaction as transaction data. To see the original data from the blockchain, we created events within the function that emits the information from the smart contract to the user. After that receiver sends an **AccessData()** transaction request to access K_e from the blockchain. The output of the **UploadData()** transaction and **AccessData()** transaction without and with zk-SNARKs concepts are illustrated in Figure 5.4 and Figure 5.5, respectively.

In this example, we consider "Alice" as a sender and "Bob" as a receiver. We also use the email addresses as "alice@gmail.com" and "bob@gmail.com" to register the sender and receiver, respectively in the system. After that, we consider two environments, one is without zk-SNARKs and another is with zk-SNARKs to upload and access the encrypted key in blockchain and to observe the transaction output. In case of without zk-SNARKs, we need to put "USERNAME", "EMAIL" and "POSITION" as an input parameter to execute the transaction verification. We also make the "POSITION" of the sender as an unique parameter to valid the transaction.

On the other hand, when we integrated zk-SNARKs, at that case, we do not need to put any input parameter to validate the transaction. Only proof is required to verify the transaction. As the proof is generated based on the input statements and appropriate witness. As a result, the transaction verification process becomes more simpler. Moreover, it does not need to show any identity of the prover to verify the transaction that increases the privacy of the transaction.

After the successful transaction, the sender store K_e in the block and receiver now able to get K_e from the block. After getting K_e, receiver use his/her private key to decrypt K_e and get the random key K.

5.5 Evaluation and Result Analysis

We have constructed a blockchain-based secure key-sharing system for email applications, and later we also integrated zk-SNARKs to the system for the privacy preservation of the transaction. The smart contracts of this system have been compiled and deployed on the Ethereum blockchain platform. This platform can calculate the amount of work done in the form of a unit called gas.

5.5.1 Effects of zk-SNARKs on Gas Consumption and Transaction Cost Analysis

Table 5.2 shows the gas and computational costs of executing smart contract's several operations with and without zk-SNARKs. During the analysis period, the exchange rate of Ethereum and the US dollar in October 12, 2023 was 1Ether=1529.41USD. The minimum gas price for executing a trade is

1GWEI=10^{-9}Ether. Note that we have used the test Ethereum network and these gas values are just test values, not real cryptocurrency.

Table 5.2: The gas and cost of performing several operations with and without zk-SNARKs.

Operation	**Gas used (GWEI)**		**Tx cost (Ether)**		**Tx cost (USD)**	
	Without zk-SNARKs	With zk-SNARKs	Without zk-SNARKs	With zk-SNARKs	Without zk-SNARKs	With zk-SNARKs
Deployment	1743924	1089729	0.001744	0.001090	2.6672	1.6671
RegisterUser()	134916	134916	0.000135	0.000135	0.2064	0.2064
UploadData()	57144	26221	0.000057	0.000026	0.0872	0.0398
AccessData()	36083	24866	0.000036	0.000025	0.0551	0.0382

** 1 GWEI = 10^{-9} Ether, 1 Ether = 1529.41 USD on October 12, 2023.*

Table 5.3: The percentage of gas reduction due to the integration of zk-SNARKs.

Operation	**Gas (GWEI) reduction**
Deployment	37.5%
RegisterUser()	-
UploadData()	54.1%
AccessData()	31.1%

In our proposed system, there were four main operations of smart contracts. Every time, we calculated the used gas and the corresponding execution cost of operations considering with and without zk-SNARKs concept. At first, the cost of the deployment of smart contacts is calculated. This deployment cost is calculated at the beginning at once to initialize the system. After that, the execution cost of the several operations of smart contracts is calculated. From this table, we can observe that for the operation of **Deployment**, **UploadData()**, and **AccessData()** the required gas is lower while integrating

zk-SNARKs concept. As a result, the corresponding transaction cost also becomes lower as shown in Figure 5.6, considering the transaction with zk-SNARKs than the transaction without zk-SNARKs. This figure illustrates that the cost for the **Deployment**, **UploadData()**, and **AccessData()** becomes lower due to the integration of zk-SNARKs. While the cost for the **RegisterUser()** function is still same for both with and without zk-SNARKs cases. This is due to register a user we considered the same input parameters for with and without zk-SNARKs environments. The total cost required for the system without zk-SNARKs and with zk-SNARKs are $3.0159 and $1.9515, respectively. So integrating zk-SNARKs, it reduces the system total cost approximately 35.3%. This analysis demonstrates the feasibility of the proposed scheme in the real world concerning the required gas values and computational costs [22].

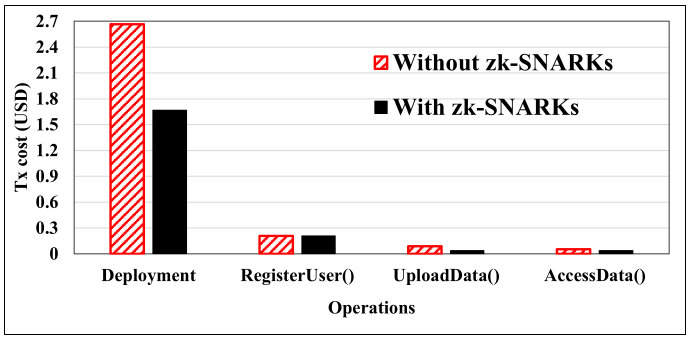

Figure 5.6: Transaction cost of the system with and without zk-SNARKs.

Due to the integration of zk-SNARKs, the amount of gas reduction is investigated in Table 5.3. From this table, we can observe that integrating zk-SNARKs, the required gas reduce for deployment, **UploadData()** and **AccessData()** while for **RegisterUser()** the gas was same as previous. It shows 37.5%, 54.1% and 31.1% gas reduction for the deployment, **UploadData()** and **AccessData()** functions respectively due to the integration of

zk-SNARKs. It makes the system more feasible.

5.5.2 Security Analysis Based on zk-SNARKs integration

The integration of zk-SNARKs in our framework ensure the following issues:

- **Data Minimization and Privacy Preservation:** By enabling users to provide just the information that is required without disclosing extra details, zk-SNARKs promote data reduction and privacy protection. zk-SNARKs, which are part of our proposed system, allow the verification of a certain characteristic (like the user's position) without revealing additional personal information (such the user's name or email). By doing this, the likelihood of data breaches is decreased and sensitive data exposure is avoided. Moreover, it also improves the data privacy.

- **Non-Interactivity:** Due to the non-interactive nature of zk-SNARKs, there is no need for back-and-forth communication between the prover and the verifier throughout the proof creation and verification procedure. This makes the verification procedure easier and is especially beneficial for blockchain transactions.

- **Trustless Verification:** By enabling trustless verification, zk-SNARKs do away with the need that parties have faith in one another. Without depending on the prover's integrity, the verifier may independently confirm the reliability of the evidence.

Besides, smart contract faces some weakness described by Chen et al. [**?**].

We used Ethereum platform and solidity code for smart contracts that may faces the following challenges:

- **Access Control Policy:** We used access control policy which is one of the key parameters in smart contract security. Some sensitive operations are only restricted to specific users due to this. In our scenario, we also used the check statements/conditions for the authorization of sender and receiver to exchange encrypted key securely. If the check is missing or invalid, attackers may be able to access the system and carry out risky actions which could ultimately result in vulnerabilities. To mitigate this issue, careful consideration is needed while writing the conditions.

- **Use of** *require*() **function:** In our proposed system, we used *require*() function for input validation. We created some specific conditions such as "USEREMAIL" should be unique and "POSITION" of the user should be fixed to execute the transaction. These conditions must met before executing the rest of the contract code. The transaction is reverted, and an exception is raised if the condition is not met.

5.5.3 Discussion

Smart contracts can be leveraged in an email system to securely transfer encrypted key by enforcing specific rules for the transaction. Smart contracts implemented in our framework ensure the following issues:

Firstly, to store the encrypted key on the blockchain, the sender must issue a transaction request under our suggested approach. However, in order for the recipient to access the encrypted key on the blockchain, a request for a transaction has to be sent. Based on the terms and conditions outlined in the

smart contracts, every network user confirms the transaction. A transaction is only regarded as legitimate if it satisfies the agreements. For instance, let's say that in a private company, the "Manager" and the "Chief Executive Officer (CEO)" would want to share the encryption key. According to this scenario, the encrypted key will be kept stored if the transaction request is sent by the "Chief Executive Officer (CEO)" alone; if not, it will not. Likewise, access to the encrypted key will only be permitted if the transaction request originates from the "Manager"; otherwise, it will not be accepted. Therefore, the blockchain network may be used to specify the terms for the exchange of encrypted keys thanks to smart contracts. In PGP, however, the encrypted key is sent straight to the recipient by the sender, with no further requirements. The amount of unauthorised access to the encryption key may be decreased by these extra requirements.

Secondly, the details of the contract are made available to all blockchain network users. Before introducing a transaction to the blockchain network, all users check over and confirm each one. Transparency is therefore encouraged. Furthermore, a smart contract cannot be modified after it has been implemented. It eliminates the potential for code modifications to the contract.

Thirdly, in order to complete the transaction successfully within the suggested framework, both the sender and the recipient must follow a number of procedures. It does not need the assistance of Man-in-the-Middle to carry out the steps, despite the fact that passing them is necessary. A smart contract contains the predetermined terms of the transaction. These terms are automatically evaluated and validated throughout the transaction. The system becomes automated by smart contracts.

In addition, there is a study that closely aligns with our proposal, pro-

viding relevant benchmark. At that study, Yakubov et al. [68] proposed a PGP management framework by utilizing blockchain technology for fast propagation of certificate revocation among PGP key servers. Differently in this work, we would like to exchange the encrypted key of the PGP system securely from the sender to receiver using blockchain technology without disclosing the details of the transactions. The detailed comparison between Yakubov et al. [68] and the proposed work are summarized in Table 5.4.

Table 5.4: Comparison between Yakubov et al. [68] and proposed work.

Characteristics	**Yakubov et al. [68]**	**Proposed**
Blockchain development platform used	Ethereum	Ethereum
PGP key generation	Conventional (RSA-3072)	ECC-256
Block size to store the encrypted key	Large memory required (encrypted key size=384 bytes)	Less memory required (encrypted key size=48 bytes)
New block should be approved by more than 50% of nodes for authorization	Yes	Yes
Gas and cost analysis of smart contracts (both ether and USD)	No	Yes
MITM risks solve by the default characteristics of blockchain	Yes	Yes
Zero Knowledge proof concept for privacy protection	Not considered	Considered

Both the work consider Ethereum as the development platform of blockchain. However, we used ECC algorithm to generate the PGP key rather than the conventional RSA algorithm. Using ECC, we produced smaller sizes encrypted key which are feasible to store in the block considering the storage capability of blockchain. We calculated the gas and cost required for every functions of smart contract to show the practicality of our proposed system. Moreover, we consider the user privacy protection by using zk-SNARKs.

5.6 Summary

In this paper, we proposed the integration of Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zk-SNARKs) and blockchain into PGP's key sharing mechanism to further bolster privacy protection. This ensures data verification without revealing sensitive details, strengthening privacy protection. We also employed Elliptic Curve Cryptography (ECC) to keep the PGP key confidential. This combined approach enhances the security of the PGP key, ensuring confidentiality, integrity, and user privacy. Additionally, we assessed gas consumption and transaction costs with and without zk-SNARKs. The results showed that our proposal consume less gas consumption and transaction costs compared to scenarios without zk-SNARKs. In future work, we will measure the time taken for generating and verifying proofs in the zk-SNARKs scheme, considering varying lengths of the encrypted key.

Chapter 6

Case-Driven Analysis of ECC Key Length Variations in PGP Key Sharing

6.1 Introduction

In the digital era, secure communication is paramount, especially with the increasing reliance on email for personal, professional, and governmental correspondence. Pretty Good Privacy (PGP) has been a cornerstone in email encryption, offering end-to-end security. However, traditional PGP systems face challenges in key distribution and management, often relying on centralized servers that can be points of failure or targets for attacks.

Blockchain technology, with its decentralized and immutable nature, presents an opportunity to enhance PGP systems by providing a distributed ledger for key management. Integrating Elliptic Curve Cryptography (ECC) into this framework allows for robust security with smaller key sizes compared to traditional algorithms like RSA, making it suitable for devices with limited computational resources.

6.2 Background Problem and Motivation

Pretty Good Privacy (PGP) traditionally uses fixed-length cryptographic keys for encrypting session keys, typically based on RSA or static ECC configurations. However, this static approach may not always be optimal across diverse security and performance scenarios. A shorter key may compromise security in high-risk environments, while a longer key may incur unnecessary

computational and financial costs in low-sensitivity use cases [69, 70]. This one-size-fits-all paradigm restricts the adaptability of secure email systems and increases the likelihood of either under-securing or over-engineering the encryption process.

To address this limitation, a dynamic encryption scheme that allows variable key lengths tailored to user context is necessary. Elliptic Curve Cryptography (ECC) provides such flexibility through multiple key length options like 256-bit, 384-bit, and 521-bit curves, which offer different trade-offs between security strength and processing cost [71]. Integrating this adaptability into a blockchain-based PGP key distribution framework enables users to select key sizes that suit specific application demands—ranging from casual personal use to critical government communications.

This chapter proposes a blockchain-integrated PGP system where senders can choose from different ECC key lengths when encrypting a session key, which is subsequently stored on the Ethereum blockchain. The implementation utilizes smart contracts that handle the secure upload and retrieval of the ECC-encrypted session key, as well as index it using a unique identifier for each transaction. We conducted three case studies: (i) personal-level encryption using a 32-byte key, (ii) business-level encryption with a 48-byte key, and (iii) government-grade encryption using a 64-byte key. Each case examines the performance trade-offs in terms of gas consumption, transaction costs, and memory overhead.

The experimental evaluation demonstrates that as key length increases, gas and memory usage also increase—but in a predictable, linear manner, making resource planning feasible. Moreover, the ability to choose key lengths enhanced user flexibility and optimized system performance across varying use

cases. The findings support the conclusion that variable ECC key lengths can significantly improve the practical utility and efficiency of secure email systems without compromising security.

The contributions of this chapter are enumerated as follows:

- Varied the length of ECC key pairs that produce different lengths of encrypted key (k_e). User can choose the length of the ECC key according to their security and cost requirements.
- Evaluated the performance of blockchain and the feasibility of the system, considering different lengths of ECC key.
- Evaluated the proposed method for three different scenarios based on low, moderate, and extreme sensitivity of information.

6.3 Overview of The System

An illustration of the key-sharing mechanism integrated into our proposed system is presented in Figure 6.1 [7]. In our prior research [5], we successfully facilitated the transmission of PGP's encrypted key through the utilization of blockchain technology. In the current key sharing scenario, we focus on generating various lengths of K_e and sharing them from sender to receiver. Based on this, we created three smart contract functions such as **RegisterUser()**, $\boldsymbol{K_e}$**Uploading()**, and $\boldsymbol{K_e}$**Accessing()** to register the user, to store K_e, and to access it in blockchain, respectively.

During the registration phase, users need to input parameters including 'userName', 'userEmail', 'organization', and 'position', with 'userEmail' being the unique identifier for registration. After registration, a user can be treated

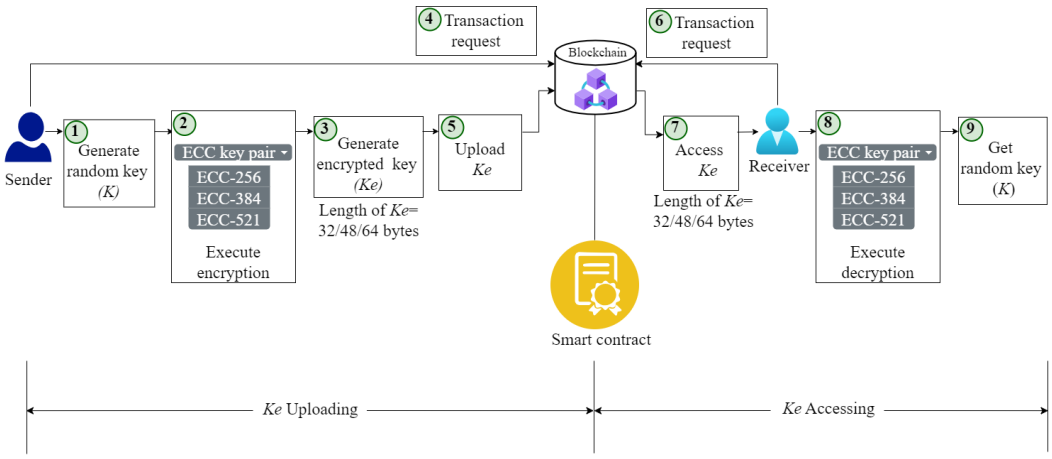

Figure 6.1: Blockchain system to share various length of encrypted key (K_e).

as a sender or receiver based on their 'position'. This means only the predefined user from the registered user can be either sender or receiver of the system. For example, if the 'position' of the user becomes "X," that user will be treated as the sender. On the other hand, if the 'position' of the user becomes "Y," that user will be treated as a receiver. Following this, the function $\boldsymbol{K_e}$**Uploading()** is utilized to store K_e within the block by the sender, while $\boldsymbol{K_e}$**Accessing()** is employed to retrieve it from the block by the receiver. The method for creating and exchanging K_e is delineated below:

Step 1. Generate random key ($\boldsymbol{K}$)

Initially, the sender generates a random key, denoted as K, to encrypt the email content, which is then transmitted using the PGP system. Our primary objective is to facilitate the exchange of this key. Utilizing a random key generator, we create K and employ **randomBytes()** function to ensure the desired length of K.

Step 2. Execute encryption

We employed the ECC algorithm to encrypt K and to produce an encrypted key, K_e. Because of this, the users need to follow several stages, such

as initiating the key generation process, selecting the key length, generating the key pair, storing the key pair, and finally using the key pair in encryption/decryption. Each of the stages is described as follows:

- **Key generation process:** User initiates this process by using the tool. In this case, we used 'eccrypto' library for key generation, key encryption, and key decryption purposes. It is a JavaScript library that provides implementations of ECC algorithms for Node.js. The library supports various elliptic curves.

- **Select key length:** The tool asks the user to choose the specific key length that the user wants to use for this encryption. User can adjust the encryption strength by choosing the appropriate key length based on the security of their data or the performance of their devices. Users can select the key length option from the drop-down menu of the system. In our experiment, we vary multiple ECC key lengths such as 256 bits, 384 bits, and 521 bits.

- **Generate key pair:** Based on the selected key length (256/384/521 bits), the tool generates two large primes and multiplies the prime numbers to create a modulus. The corresponding ECC public and private key pairs are derived from the modulus. Longer key lengths entail increased computational demands and time owing to the complexity of secure prime generation and larger bit calculations. In this experiment, adopting the 'eccrypto' library, the elliptic curves 'secp256r1', 'secp384r1', and 'secp521r1' is used to execute ECC-256, ECC-384, and ECC-521 key pair, respectively. After that, the corresponding public and private keys are generated.

- **Store key pair:** Upon the generation of the key pair, which comprises public and private keys, they are preserved within the user's keyring, which is a secure location designed to manage keys. In conjunction with the keys, pertinent metadata such as the key length, the date of creation, and the expiration date are also systematically recorded.

- **Use key pair in encryption:** Once a key pair, public and private keys are generated and stored in the keyring, the chosen key length is used during encryption/decryption operations. The system automatically detects the key length for encryption/decryption from the keyring based on the key length previously selected by the user while generating the key pairs. The public key belonging to the receiver is utilized in the process of encrypting random key, K (generated in step.1) to produce encrypted key, K_e in the sender side.

Step 3. Generate encrypted key (K_e)

The encryption process involves generating a shared secret between the sender and the receiver. Different curves will produce different shared secrets. So, even though the same K, the choice of elliptic curve affects how the shared secret is derived, leading to different lengths of K_e. In our proposed system, we used ECC-256, ECC-384, and ECC-521 encryption to encrypt K and got the length of K_e as 32 bytes, 48 bytes, and 64 bytes, respectively.

Step 4. Transaction request (from sender)

The sender generates K_e of various lengths and then sends a transaction request to upload it to the blockchain. Network users validate the smart contract transaction agreements, with the sender's 'position' designated as "X" for validation purposes. If the transaction adheres to the agreements, it proceeds

successfully; otherwise, it is ignored.

Step 5. Upload K_e

Following a successful transaction, the sender uploads K_e of designated length into the blockchain ledger, wherein it is stored as transaction records. In our proposed system, the length of K_e can be either 32 bytes, 48 bytes, or 64 bytes. Based on this variation of K_e, the required gas consumption of the several functions of the solidity code will also be varied. The procedure for uploading K_e is detailed in **Algorithm 1**.

Step 6. Transaction request (from receiver)

Once K_e is uploaded to the blockchain, the receiver commences a request for transaction initiation in order to access it. The transaction agreements are verified by network users, creating a unique field 'position' for the receiver as "Y" to legitimate the transaction. If the agreement is met, the transaction is successful; otherwise, it is ignored.

Step 7. Access K_e

Upon the completion of a successful transaction, the receivers retrieve K_e from the blockchain. The length of K_e can vary as 32/48/64 bytes. Based on this variation, the required gas consumption of the several functions of solidity code, will also be varied. After that, the receiver can access the same length of K_e sent by the sender. The procedure for accessing K_e is outlined in **Algorithm 2**.

Step 8. Execute decryption

Subsequently, the receiver employed the ECC algorithm for the decryption of K_e in order to retrieve the original key K. Because of this, the following stages are executed:

- **Select key pair length:** At first, the system selects the desired ECC key

pair for decryption based on the length of K_e accessed by the receiver. As the length of K_e are 32 bytes, 48 bytes, and 64 bytes, the system automatically detects the corresponding decryption algorithm of ECC-256, ECC-384, and ECC-521, respectively.

- **Use key pair in decryption:** After selecting the decryption algorithm, the corresponding private key is used to decrypt K_e on the receiver side.

Step 9. Get random key (K)

After the successful decryption of K_e, the receiver can obtain the random key, K. Subsequently, using this K, the receiver is now able to decrypt the original email content and read it.

Algorithm 1 Encrypted key (K_e) Uploading

```
if userIsRegistered(userEmail) then
    The user registers successfully in the system and obtains its identity;
    User user = getUserInfo(userEmail);
    if user.position == "X" then
        The user's position matches the stored position and can upload the encrypted key (K_e);
        string encryptedDataHash = getEncryptedDataHash(userName, userEmail, organization, position, encryptedKey); //retrieve hash of encrypted data
        string hashCiphertext = getHashCiphertext(encryptedDataHash); //get hash ciphertext
        storeHashCiphertext(userEmail, hashCiphertext); //store hash ciphertext in blockchain
    end if
end if
```

6.4 Implementation Details

This part outlines the assessment of the intended system, as depicted in Table 6.1, encompassing its implementation settings, framework, requisite

Algorithm 2 Encrypted key (K_e) Accessing

```
1: if userIsRegistered(userEmail) then
2:    The user registers successfully in the system and obtains its identity;
3:    User user = getUserInfo(userEmail);
4:    if user.position == "Y" then
5:       The user's position matches the stored position and can access the
   encrypted key (K_e);
6:       string encryptedDataHash = getEncryptedDataHash(userName,
   userEmail, organization, position, encryptedKey); //retrieve hash of en-
   crypted data
7:       string hashCiphertext = getHashCiphertext(encryptedDataHash);
   //get hash ciphertext
8:       sendHashToNetwork(userEmail, hashCiphertext); //send hash to net-
   work
9:       returnTransactionInfo(transactionID, blockNum); //get access of
   block
10:   end if
11: end if
```

technical tools, and software. We considered three case study scenarios, Case Studies I, II, and III, to evaluate the performance of the system. Case study I is examined for casual personal email, case study II for professional communication and case study III for highly confidential business communication.

6.4.1 Implementation Environments

To analyze the feasibility of the proposed system, the details of the evaluation are described here. The implementation environments, model, required technical tools, and software are introduced in Table 6.1.

6.4.2 Smart Contract Functions

The smart contracts are written in Solidity and include the following functions:

Table 6.1: Implementation environments.

Software/Component	Configuration/Version
Operating System (OS)	Windows 11 Pro
CPU	3.20 GHz Intel Core i7
RAM	16 GB Memory
Truffle	5.11.5
Ganache	2.7.1
Node.js	20.0.0

- `RegisterUser(string name, string email, string organization, string role)`: Registers a new user with their details.
- `KeUploading(bytes encryptedKey)`: Allows a user to upload their encrypted symmetric key to the blockchain.
- `KeAccessing(address userAddress)`: Enables a user to retrieve an encrypted key associated with a specific address.

The encryption and decryption processes are handled off-chain using Node.js scripts that utilize ECC libraries to perform the necessary cryptographic operations. This approach ensures that private keys remain secure and are not exposed on the blockchain.

6.4.3 Case Study Scenarios

Case study I (Casual Personal Email)

- User: Alice, a college student
- Device: Mid-Range smartphone
- Context: Sending a brief email to a friend about weekend plans
- Key Length: ECC-256
- K_e: 32 Bytes

- Principle: Low sensitivity of information, limited device processing power, and need for quick message composition and sending.

In this scenario, Alice uses ECC-256, which provides adequate security for personal communications while ensuring minimal computational overhead, making it suitable for mobile devices.

Case study II (Professional Communication)

- User: Bob, a marketing manager
- Device: Company-issued laptop
- Context: Sending a quarterly report to the management team
- Key Length: ECC-384
- K_e: 48 Bytes
- Principle: Moderate sensitivity of business information, standard corporate device with decent processing power, and need for a balance between security and performance.

Bob opts for ECC-384 to secure professional communications that require a higher level of confidentiality, balancing security and performance.

Case study III (Highly Confidential Government Communication)

- User: David, Head of the National Defense Agency
- Device: High-performance secure workstation with enhanced cryptographic capabilities
- Context: Discussing classified defense strategies with the heads of intelligence and foreign affairs ministries via encrypted emails. This discussion

involves sharing sensitive information about coordinated international defense protocols

- Key Length: ECC-521
- K_e: 64 Bytes
- Principle: Extremely sensitive nature of national security data and high stakes requiring the highest level of security, a powerful device capable of handling the increased computational load, and recipient expectations for top-tier encryption.

Carol selects ECC-521 to ensure maximum security for highly sensitive government communications, accepting the trade-off of increased computational requirements.

6.5 Evaluation and Result Analysis

The evaluation of the proposed system and it's result are described here in datails.

6.5.1 Gas and Transaction Cost Analysis

The proposed system employs dual environments: an external off-chain setting that operates outside the blockchain infrastructure and an internal on-chain setting that functions within it. The Ethereum blockchain platform compiles and deploys smart contracts, which calculate the required gas for different case studies and various lengths of K_e. In Ethereum, it is called computation cost, which refers to the gas required to execute a transaction or perform computations within a blockchain. This cost is represented in gas

units, where each computational operation (e.g., addition, storage, or function calls) has a specific gas cost.

Table 6.2 displays the gas needed to carry out every function of a smart contract, corresponding to different lengths of K_e specifically 32 bytes, 48 bytes, and 64 bytes for case study I, II and III, respectively. To calculate the required gas consumption for each case study, the user must initially enroll in the system by utilizing the smart contract function of **RegisterUser()**. Subsequently, the user (sender) uploads K_e to the block of blockchain using the function of $\boldsymbol{K_e}$**Uploading()**. After that, accessed this K_e from the block by another user (receiver) using the function of $\boldsymbol{K_e}$**Accessing()**. To do so, the gas values for every operation were calculated. The empirical findings indicate that within the entirety of the case studies examined, the maximum gas consumption recorded in case study III for the operation of deployment with $K_e = 64$ bytes amounted to 1747140 GWEI. This gas value is significantly lower than the maximal gas threshold of block 6721975 GWEI, thereby affirming the system's practicality.

Table 6.2: Gas usage for smart contract operations under different ECC key lengths.

Operation	**Case Study I** ($K_e = 32$ Bytes)	**Case Study II** ($K_e = 48$ Bytes)	**Case Study III** ($K_e = 64$ Bytes)
Deployment	1,733,302	1,743,924	1,747,140
RegisterUser()	134,916	134,916	134,916
K_eUploading()	96,321	116,755	137,190
K_eAccessing()	34,391	34,731	35,072

*

1 GWEI $= 10^{-9}$ *Ether,* *1 Ether* $=$ 2962.73 USD on May 3, 2024.

The cost of the transactions is likewise connected to the value of the gas. The minimal gas required to complete a trade in Ethereum is 1GWEI=10^{-9}Ether. On May 3, 2024, during the entire duration of the analysis phase, the prevail-

ing exchange rate that characterized the relationship between Ethereum and the United States dollar was recorded at a value of 1 Ether, being equivalent to 2962.73 USD. Firstly, a cost estimate is provided for the smart contract deployment. To initialize the system, this deployment cost is computed immediately at the beginning. Subsequently, the cost of the execution is computed, which is associated with each of the various operations pertaining to smart contracts. Figure 6.2 illustrates the cost associated with transactions within the system for various operations pertaining to smart contracts with three different case study scenarios and their corresponding K_e length. In our experiment, we used a shorter length of K_e as 32 bytes for case study I, a moderate length of K_e as 48 bytes for case study II, and a longer length of K_e as 64 bytes for case study III. From the analysis of this figure, it becomes evident that an enhancement in the length of K_e, transitioning from case study I to case study II, and subsequently from case study II to case study III, leads to a concomitant increase in the associated transaction costs for the operations of **Deployment**, $\boldsymbol{K_e}$**Uploading()**, and $\boldsymbol{K_e}$**Accessing()**. As these three operations are related to the length of K_e. For the operation of **RegisterUser()**, the transaction cost remains the same irrespective of the length of K_e as the parameters to register a user are the same in all cases. This analysis shows that taking into account the system's transaction cost for all case studies, the suggested strategy is feasible in the real world [63].

6.5.2 Analysis of Memory Requirements

The size of a block is dynamic on the Ethereum blockchain. Each Ethereum transaction involves a certain amount of computing work, denoted as gas. A block's gas limit dictates the maximum amount of computing work that can

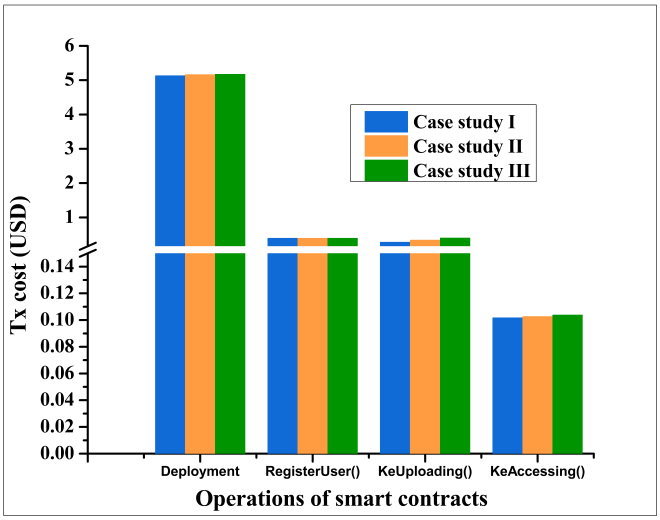

Figure 6.2: Transaction cost of the system for different case study scenarios.

be done there. Ethereum regulates the size of a block using gas. Gas calculates how much memory and processing time a transaction uses. The block gas limit sets a cap on the total gas within a block, which indirectly determines data storage capacity. The higher the gas limit, the more data can be put into a block [73]. Our investigation focuses on the storage of K_e in the block up to the length of 64 bytes. Figure 6.3 shows the memory required to store various lengths of K_e associated with three different case study scenarios. This illustration demonstrates the relationship between the length of K_e and memory storage in a block. From this illustration, we found that as the length of K_e escalates from case study I to case study II, and consequently progressing from case study II to case study III, there is a heightened requirement for memory allocation to accommodate its storage within the block. The present study employed the typical Ethereum block size ranging from 1-2 MB. The experimental results suggest that among the detailed assortment of case studies analyzed, the maximum memory storage necessitated in case study III

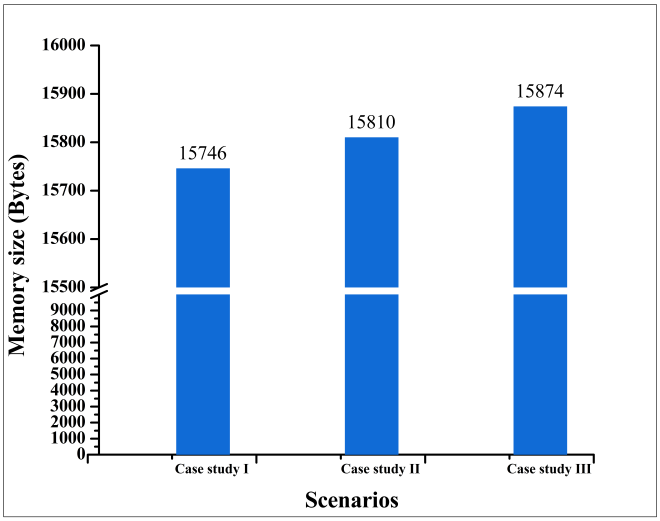

Figure 6.3: Memory size required to store K_e for different case study scenarios.

to store K_e for the length of 64 bytes was found to be 15874 bytes, which is considerably lower than the maximum block size (1-2 MB). This assessment highlights the feasibility of the system concerning its memory requirements.

6.5.3 Security and Bugs of Smart Contract Analysis

The gas cost of smart contracts can be optimized by considering a hybrid off-chain/on-chain storage environment. Large data can be offloaded to off-chain storage while keeping only essential or verification-related data on-chain. Moreover, cryptographic algorithms like asymmetric encryption (e.g., RSA and Elliptic Curve Cryptography) consume considerable computational resources; for this reason. it is better to execute them off-chain. The proposed system utilizes two environments: an off-chain and an on-chain environment. According to our system, we executed the **randomBytes()** function to generate a random key that is used to encrypt the email contents in an off-chain environment. Moreover, we also performed the cryptographic algorithm ECC

to encrypt and decrypt this random key in an off-chain environment. After encryption, the encrypted key K_e only interacts with the blockchain system to store it in the block by the sender and access it from the block by the receiver. These off-chain operations optimize gas values and computational resources for our system.

A bug in a smart contract is a flaw or error in the contract's code, often resulting from coding mistakes or logical errors. In a system using the Ethereum platform and Solidity code for smart contracts, vulnerabilities may arise due to the use of access control policy and **require()** function. Access control policy restricts sensitive operations to specific users. Our scenario involved using check statements to secure key encryption, highlighting the potential for vulnerabilities if the check is missing or invalid. on the other hand, **require()** function is used for input validation, requiring specific conditions like unique 'userEmail' and fixed user 'position' for our system. These conditions must be met before executing the contract code, and if not, transactions are reverted and exceptions raised. Careful consideration is needed to avoid using **require()** frequently or inappropriately within loops.

The proposed key-sharing mechanism combined with blockchain technology performed well in terms of security. Blockchain can help mitigate a range of attacks such as MITM (Man-in-the-Middle), DoS (Denial-of-Service), and others by providing a distributed, tamper-resistant ledger. In our smart contracts, we set several predetermined conditions for **RegisterUser()**, $\boldsymbol{K_e}$**Uploading()** and $\boldsymbol{K_e}$**Accessing()** to register the user, store the encrypted key and access it, respectively. These conditions are automatically evaluated and validated throughout the transaction. There is no need for the involvement of other nodes to execute the transactions, which mitigates the man-in-the-middle

(MITM) attack. Blockchain networks, being decentralized, have no central server to overwhelm. Moreover, our proposed scenario involves creating a field called 'organization' to restrict registration to specific organization users. This discourages attackers from spamming the network with malicious requests, as these consensus mechanisms consume considerable resources, making denial of service (DoS) attacks costly and impractical. we stored the encrypted key in the block to eliminate its modification. Additionally, any of the contents of the blockchain included by the nodes in the system were distributed among all nodes in the network to synchronize and validate. This makes it highly challenging for attackers to tamper with the encrypted key without triggering alerts or invalidating blocks that mitigate data modification attacks.

In the development phase, we utilized Truffle and Ganache to simulate the blockchain environment. This setup allowed us to test smart contract functionality, PGP key management, and transaction workflows in a controlled and cost-effective way. In the transition from a development environment to a live deployment scenario (Ethereum mainnet), we need to consider several things. Before deployment, we need to optimize the smart contracts further to minimize gas costs on the Ethereum mainnet. After optimization, we have to deploy the system to an Ethereum testnet (such as Ropsten or Goerli). Before mainnet deployment, a comprehensive security audit must be performed. After that, we need to set up real-time monitoring and logging to track performance and potential issues post-development.

6.6 Summary

In this paper, we manipulate the length of the ECC key to generate varying lengths of encrypted keys, which are subsequently disseminated through our blockchain-based PGP key distribution system. This research has facilitated the integration of functionality for multiple ECC key lengths (ECC-256, ECC-384, and ECC-521), thereby enabling users to adaptively adjust encryption strength in accordance with information security requirements, device specifications, and computational costs. We conducted a series of case studies encompassing three distinct scenarios to analyze the performance of blockchain technology and its potential application in real-world contexts. Case Study I examines less sensitive information utilizing shorter key lengths, Case Study II addresses moderately sensitive information with intermediate key lengths, while Case Study III investigates highly sensitive information employing extended key lengths. The findings show that the proposal is applicable to implement through the assessment of blockchain performance measurement criteria, encompassing gas consumption, transaction costs, and memory specifications. In future work, we will add multi-user scenarios with variable ECC key lengths and compute the performance of the system for secure email applications.

Chapter 7

Enhancing Blockchain Scalability via Multi-Recipient in PGP Key Sharing

7.1 Introduction

As digital communication becomes increasingly central to organizational operations, ensuring the security and privacy of email messages has never been more critical. While pretty good privacy (PGP) has been a widely adopted standard for email encryption, key distribution vulnerabilities remain a challenge, especially in multi-recipient scenarios. In existing work, we integrated blockchain technology with elliptic curve cryptography (ECC) and zero-knowledge proofs (zk-SNARKs) to enhance both the security and privacy of PGP key distribution. This system supported variable ECC key lengths (256, 384, and 521 bits), allowing users to adjust encryption strength based on their needs. Nevertheless, this implementation was constrained to single-recipient use cases.

In modern organizational communication, it is common to distribute the same message to multiple recipients, such as in corporate announcements or team collaborations. For example, in a single-recipient email architecture, each recipient necessitates a separate email encryption and transactional process [38]. Emails are subjected to encryption utilizing a symmetric key. Subsequently, this symmetric key is encrypted independently for each recipient through the utilization of their respective PGP public keys to generate the encrypted keys. The encrypted content of the email is transmitted from the

sender to the recipient employing the PGP framework. Following this, each encrypted key is uploaded independently to the blockchain via a blockchain-based mechanism. Consequently, each transaction incurs individual gas costs for the upload of each encrypted key, which ultimately exacerbates computational overhead, network utilization, and storage demands.

In contrast to a multi-recipient system, email content is encrypted once with a symmetric key and the encrypted email contents are transmitted to recipients via PGP. The symmetric key is then encrypted individually for each recipient using their respective PGP public keys, producing a set of encrypted keys [74]. Unlike the single-recipient approach, these encrypted keys are uploaded to the blockchain in a single transaction. This significantly reduces the number of blockchain operations, as only one transaction is needed to serve multiple recipients. As a result, this approach minimizes gas consumption, lowers computational overhead, and reduces network and storage usage, making the system more efficient and scalable for group communication.

7.2 Background Problem and Motivation

In traditional PGP systems and earlier blockchain-based implementations, the secure key sharing process is optimized primarily for one-to-one communication. Each sender encrypts the session key specifically for a single recipient, meaning that in scenarios where a message must be shared with multiple recipients, the sender must redundantly perform the key encryption and uploading process for every recipient individually. This design leads to repetitive transactions on the blockchain, increasing computational overhead, transaction costs, and on-chain storage usage. As the number of recipients increases, the sys-

tem's scalability deteriorates rapidly, rendering such frameworks inefficient for real-world applications that demand secure group or broadcast communication (e.g., organizational announcements or team coordination). Prior works such as [75, 76] have noted this bottleneck in traditional cryptographic and blockchain-assisted email systems.

To address this limitation, a scalable and efficient key exchange mechanism is required that supports secure distribution of session keys to multiple recipients in a single, unified process. Blockchain's inherent immutability and auditability can still be utilized, but the smart contract architecture and key distribution model must be adapted to accommodate multiple users without compromising security or increasing transaction complexity disproportionately.

In this chapter, we propose a multi-recipient secure key sharing system that builds on our earlier blockchain-integrated PGP framework. The core idea is to associate each encrypted session key with a compact Key ID (K_{id}), which uniquely identifies a recipient in the blockchain ledger. Rather than uploading separate keys for each recipient, the system uploads tuples of (K_{id}, K_e) using a looped or batched smart contract function. The recipient-specific Key ID helps efficiently map each user to their corresponding encrypted key during retrieval, minimizing blockchain writes and ensuring deterministic key access. This design not only reduces redundant transactions but also simplifies the logic required to manage multiple encrypted keys on-chain.

The proposed enhancement achieves significant gains in blockchain scalability while maintaining the cryptographic integrity of the key sharing process. Our experiments show that gas costs increase linearly (rather than exponentially) with the number of recipients, and the smart contract memory model effectively supports multi-user access. Moreover, the use of indexed Key IDs

introduces an additional access control layer, ensuring that only intended recipients can retrieve their session key. These improvements make the system highly suitable for enterprise and institutional applications requiring secure and scalable group communication. By combining compact ECC encryption, efficient smart contract logic, and structured recipient indexing, the multi-recipient model provides a robust solution to the limitations of earlier one-to-one PGP key sharing systems.

The key contributions of this chapter are as follows:

- Presents a scalable blockchain-based system for multi-recipient encryption that minimizes redundancy and overhead through a compact Key ID mechanism.
- Supports variable ECC key lengths and demonstrates cost-efficiency and linear scalability across 1–25 recipients.
- Measures key performance metrics including gas consumption, transaction cost, and memory requirements to confirm the the system's scalability and practicality.

7.3 Overview of the System

This section introduces a proposed system of encrypted key sharing mechanisms using blockchain technology. In the previous paper as mentioned in [5–7], we only considered the blockchain technology for sharing the encrypted key considering a single recipient. Now, we utilize the system for sharing the encrypted key for multi-recipient.

To overcome the limitations of single-recipient, we propose a multi-recipient extension to the blockchain-based PGP key distribution. The goal is to allow a sender to share a single PGP session key with any number of recipients in one protocol, without repeating redundant steps. This improves scalability by batching the key distribution on-chain. The sender generates a single random symmetric key (K) for the email and then prepares an encrypted copy for each intended recipient. For each recipient i, the sender uses that recipient's public ECC key to compute $K_{e,i} = \text{Enc}_{\text{ECC}}(K)$, just as in the single-recipient case. After that, a key ingredient Key ID (K_{id}) is created to uniquely identify each recipient and combine it with K_e to produce (K_{id}, K_e) tuples. However, instead of performing separate transactions for each key (in single-recipient), our system associates each (K_{id}, K_e) tuple and uploads all of them in a coordinated manner. At a high level, the architecture remains a blockchain smart contract that mediates key storage and retrieval. However, the contract now accepts multiple (K_{id}, K_e) tuples. Each recipient later queries the contract for their own K_e. The blockchain still provides immutability and collective verification: once the keys are stored, they cannot be altered or deleted, and any access request is checked against the contract logic. This decentralizes trust and leverages the same confidentiality guarantees as before.

Compared to the single-recipient scheme, the multi-recipient system requires only a single contract deployment and a combined transaction per batch of recipients, greatly reducing overhead for group emails. By indexing each K_e with a recipient-specific identifier K_{id}, we ensure that only the intended party can efficiently locate and decrypt their K_e. In this way, the proposed system retains authentication (through user registration and blockchain accounts), confidentiality (through ECC encryption), and integrity (through chain im-

mutability) while enabling one-to-many secure email distribution.

7.3.1 Key ID Mechanism

To securely index multiple recipients, we derive a unique Key ID (K_{id}) for each recipient from their public key fingerprint. Traditionally, this public key is a cryptographic key (e.g., RSA or ECC). However, to enhance usability and align with modern practices, the system can also employ the recipient's email address as their public key. This approach is supported by identity-based encryption (IBE) schemes, where a user's unique identifier, such as an email address, serves as the public key [77]. For instance, in Pretty Good Privacy (PGP) implementations, public keys are often associated with email addresses, facilitating straightforward key distribution and verification.

The procedure for generating Key ID (K_{id}) is as follows :

- Compute the recipient's public-key fingerprint: $FP_i = H(Pk_i)$, where Pk_i is the recipient's ECC public key and $H(\cdot)$ is a cryptographic hash (SHA-256).
- Derive a short Key ID: $K_{id_i} = \text{Truncate}(FP_i)$, for example we take the least 64 bits of FP_i (analogous to standard PGP key IDs).

Each K_{id_i} thus uniquely corresponds to Pk_i and importantly, K_{id_i} is just a pseudonymous index, it does not reveal Pk_i itself since it is a one-way hash/truncation. The sender uses K_{id_i} as a handle for recipient i when storing keys on-chain. In practice, each recipient computes their own K_{id} offline. When the sender posts (K_{id_i}, K_{e_i}) to the blockchain, only the holder of the corresponding private key Sk_i will recognize K_{id_i} as their entry and be

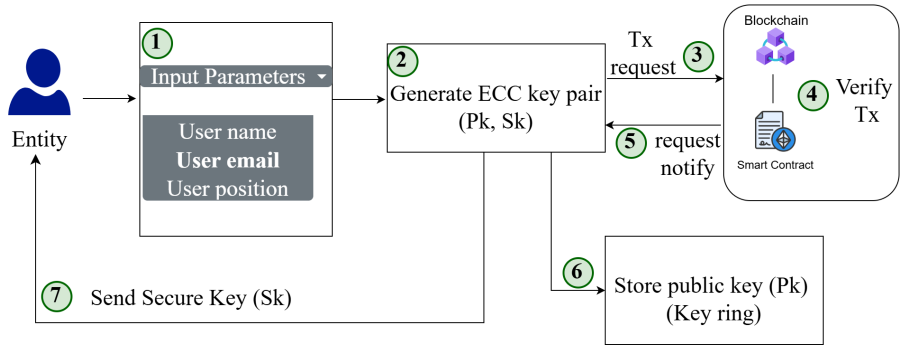

Figure 7.1: Registration phase.

able to decrypt Ke_i. This mechanism allows the contract to securely identify recipients by a compact identifier while preserving key privacy.

The advantages are threefold: (1) Compactness: a 64-bit Key ID saves space and gas compared to storing full hashes or public keys on-chain. (2) Indexability: the smart contract can use the Key ID as a mapping key, so a recipient can directly query with their Key ID to retrieve the associated K_e. (3) Privacy: the Key ID reveals no sensitive information by itself. Only someone who knows a recipient's public key can compute the same Key ID; outsiders cannot easily invert it to learn the user's identity or key. In this way, each uploaded entry (K_{id}, K_e) is effectively "tagged" for a specific recipient without exposing which user it is on the public ledger.

7.3.2 Multi-Recipient Key Distribution

The multi-recipient key distribution system starts with the registration phase of each entity. After that followed by the key distribution mechanism.

7.3.2.1 Registration Phase

Figure 7.1 elucidates the registration phase of an entity within a blockchain-oriented framework that employs ECC. This procedure guarantees secure identity registration of every user before utilizing the blockchain network.

In the preliminary phase **(1)**, the entity is mandated to provide specific identity-related parameters, which encompass the user's name, email address, and position in the organization. These characteristics are indispensable for the creation of a distinctive identity and for ensuring traceability within the network.

Subsequent to the input, in phase **(2)**, the system implements the ECC algorithm to produce a cryptographic key pair: a public key P_k a private key S_k. ECC is selected for its operational efficiency and robust security, which is achieved through shorter key lengths in comparison to conventional cryptographic techniques.

Thereafter, in phase **(3)**, a transaction request is initiated and forwarded to the blockchain network. This request functions to document the new entity registration on the distributed ledger and necessitates the invocation of a smart contract.

In phase **(4)**, the blockchain network, utilizing its smart contract framework, ascertains whether the entity is already registered. This procedure is critical for preventing duplicate or unauthorized registrations, thereby preserving the integrity and distinctiveness of each entity within the system.

Upon verification, as depicted in phase **(5)**, the blockchain transmits a notification response back to the key generation module, thereby confirming the successful registration of the entity.

In phase **(6)**, the generated public key P_k is archived in a publicly accessible key ring. This key ring acts as a decentralized repository of all registered public keys in the system and is available for all participants in the network. This facilitates the retrieval of the public key by other participants in the network when required for secure communications.

Lastly, in phase **(7)**, the private key S_k is securely conveyed back to the entity. It is imperative that this key remains confidential and is stored securely by the entity, as it will be utilized for the signing of transactions and the decryption of information.

7.3.2.2 Key Distribution Mechanism

In the multi-recipient email encryption system, numerous fundamental procedures remain similiar with the single-recipient system. In particular, the processes pertaining to the generation of a random key for the encryption of the email content, alongside the application of ECC algorithms to secure this random key, are indistinguishable across both communication contexts. Nevertheless, a significant obstacle emerges within the multirecipient setting: following the generation of multiple encrypted keys, the system is required to accurately and securely link each encrypted key to its designated recipient.

In order to tackle this challenge, we introduce for the implementation of a Key ID mechanism. This mechanism offers a streamlined and distinctive method to identify each designated recipient, while not imposing considerable overhead. The procedure commences during the user registration phase, wherein the public key of each user is securely stored within a key ring. When the necessity to generate the Key ID arises, the system retrieves the registered public key from the key ring and computes its fingerprint. From this

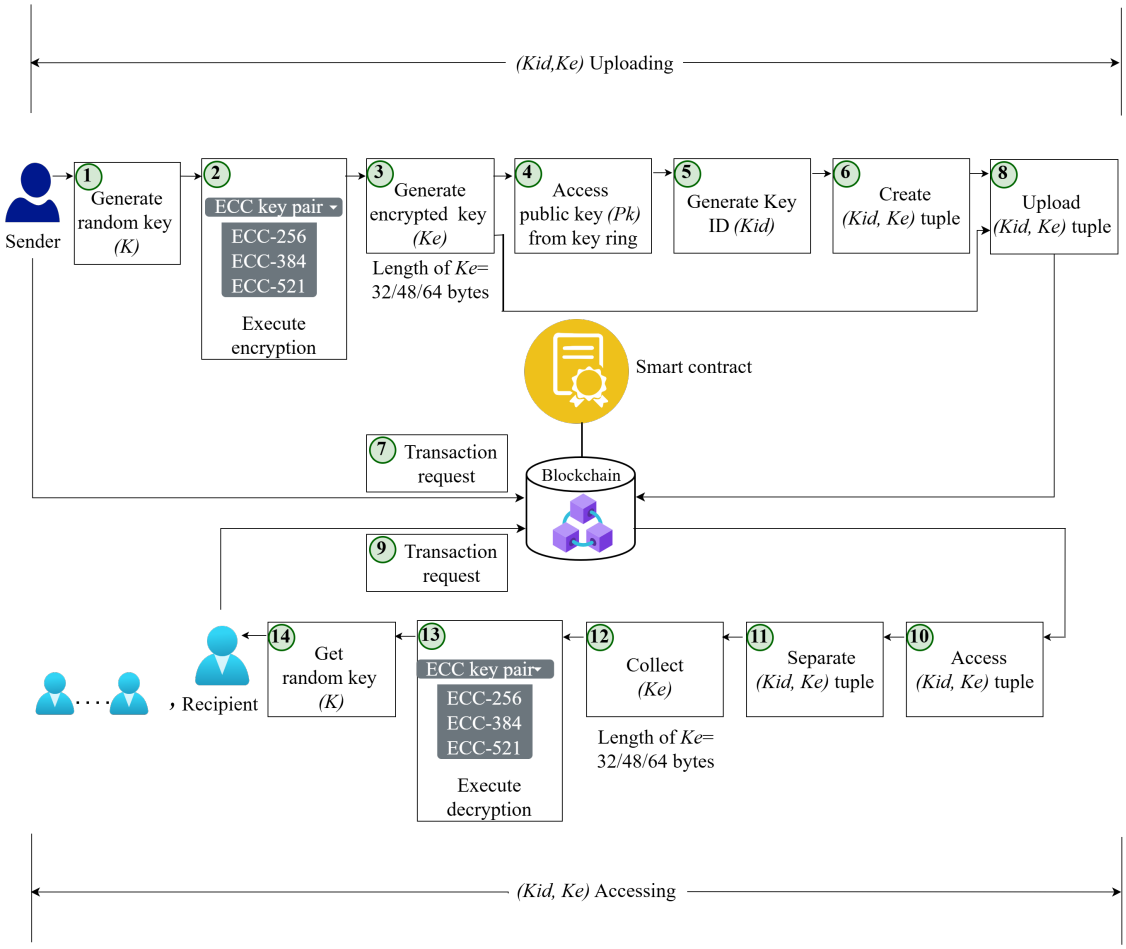

Figure 7.2: Proposed system to share various length of encrypted key (K_e) for multi-recipient.

fingerprint, the system extracts the least significant 64 bits to formulate a compact and distinctive identifier Key ID that corresponds to each user. The process flow, illustrated in Figure 7.2, consists of 14 key steps, each of which is explained in detail below:

Step 1. Generate random key (K)

To initiate the procedure, the sender generates a stochastic key, denoted as K, which is specifically intended for the encryption of the email's contents, and this key is subsequently transmitted through the PGP framework. The objective of this process is to facilitate the secure conveyance of this key. By employing a random key generation algorithm, we derive K and apply the **randomBytes()** function to ensure that K attains the necessary length.

Step 2. Execute encryption

In this step, the process of encryption is undertaken through the application of ECC algorithm to secure the symmetric key K and generate the encrypted key K_e. Initially, the key generation procedure is executed by the user utilizing a specific tool, wherein the 'eccrypto' library in Node.js is employed for the purposes of key generation, encryption, and decryption, facilitating the application of diverse elliptic curves. Subsequently, the selection of key length permits users to opt for varying levels of encryption strength (256, 384, or 521 bits) via a dropdown menu, thereby modulating security and performance parameters accordingly. Thereafter, the system systematically generates a key pair predicated on the chosen key length by formulating large prime numbers and deriving ECC public and private keys utilizing curves such as 'secp256r1', 'secp384r1', and 'secp521r1'. Upon completion of key generation, the system ensures the secure storage of the key pair within a user's keyring. Ultimately, the system employs the key pair in the encryption pro-

cess, whereby the sender's tool autonomously identifies the stored public key to encrypt the random symmetric key K, resulting in the production of the encrypted key K_e, which is prepared for secure transmission to the receiver.

Step 3. Generate encrypted key (K_e)

The encryption procedure facilitates the establishment of a mutual secret between the transmitting and receiving parties. Various elliptic curves produce unique shared secrets. As a result, while the same K is employed, the choice of elliptic curve significantly affects the computation of the shared secret, leading to diverse lengths of K_e. In the framework presented, we implemented ECC-256, ECC-384, and ECC-521 encryption methodologies, which correspond to K_e lengths of 32 bytes, 48 bytes, and 64 bytes, respectively.

Step 4. Access public key (P_k) from key ring

In the multi-recipient encrypted key distribution framework, the sender is required to acquire the requisite public keys of all designated recipients to advance with secure encryption. This procedure is of paramount importance since each encrypted key K_e must be distinctly associated with an individual recipient's public key to guarantee confidentiality and precise access. During the preliminary user registration process, the ECC public key of each recipient is securely archived within a centralized and tamper-proof storage system referred to as the key ring. The key ring serves as a cryptographically governed repository to retain each user's public key and the specifications related to the type and length of the cryptographic key. Upon the commencement of the encrypted key sharing procedure by the sender, the system autonomously queries the key ring to extract the ECC public keys $P_k$for all participants involved in that transaction. This extraction guarantees that the appropriate and most current key is employed for each recipient during the encryption process.

Step 5. Generate Key ID (K_{id})

The Key ID (K_{id}) functions as a succinct and distinct identifier that enables the system to precisely link each encrypted key K_e to its designated recipient, especially within a multi-recipient framework where effective mapping is of paramount importance. In order to guarantee both distinctiveness and confidentiality, the creation of K_{id} is executed through a bifurcated procedure: the generation of a fingerprint followed by the derivation of the Key ID.

- **Fingerprint Generation:** In the initial phase, a cryptographic fingerprint is derived through the application of a one-way hash function on the recipient's elliptic curve public key P_k. A widely utilized hashing algorithm for this specific objective is SHA-256, which is esteemed for its robust collision resistance and computational irreversibility. This procedure produces a 256-bit output that functions as a deterministic digital signature of the recipient's public key. The fingerprint offers a reliable and verifiable depiction of the public key without necessitating its direct disclosure. It also ensures that even minute alterations in the public key will lead to a markedly different fingerprint, thus enhancing recipient isolation and safeguarding data integrity.

- **Key ID Derivation:** Upon the completion of the fingerprint computation, the system generates the Key ID K_{id} by extracting the least significant 64 bits (i.e., the rightmost 64 bits) from the 256-bit fingerprint. This truncation process produces a compact yet adequately distinctive identifier that is appropriate for application in blockchain contexts, where efficiency in storage and transaction costs is of paramount

importance.By condensing the fingerprint to a 64-bit representation, the Key ID facilitates a degree of compactness conducive to efficient on-chain storage, allows for rapid deterministic indexing, and upholds privacy by obscuring the original public key along with the complete fingerprint.

Step 6: Create (K_{id}, K_e) tuple

Subsequent to the generation of the encrypted key K_e, it is securely paired with the corresponding Key ID (K_{id}) to establish the composite tuple (K_{id}, K_e). This integration guarantees a verifiable and tamper-resistant mapping between the encrypted key and its intended recipient. Because the Key ID is uniquely derived from the recipient's public key, any alteration in key identity or misdirection can be easily detected during key retrieval and decryption. Furthermore, this structure serves as an efficient routing reference, allowing for the management of multiple recipients' encrypted keys in a single transaction. It enhances access control while preserving the anonymity of identities and public keys on-chain, thus achieving a balance among security, scalability, and privacy.

Step 7. Transaction request (from sender)

The designated sender generates a set of tuples (K_{id_i}, K_{e_i}) for all intended recipients, where each K_{e_i} is the encrypted message key for recipient i and K_{id_i} is the corresponding Key ID. These tuples are bundled and submitted as a single transaction request to the blockchain. Network participants validate the sender's authorization marked by their assigned 'position' as "X", through the smart contract's access control logic. If the sender is authenticated and the transaction structure adheres to protocol requirements, the transaction is approved and stored on-chain; otherwise, it is rejected without execution.

Step 8: Upload (K_{id}, K_e) tuple

After the validation of transactions, the authorized sender uploads a collection of (K_{id}, K_e) tuples into the blockchain ledger, wherein each tuple is distinctly associated with a specific recipient. This integrated framework guarantees that encrypted keys are verifiably associated with their designated recipients through the use of lightweight identifiers. In the system we propose, the length of the encrypted key K_e can be configured to 32, 48, or 64 bytes. Furthermore, the quantity of designated recipients can vary from 1 to 25. Consequently, the gas consumption associated with the functions of the Solidity smart contract varies based on both the length of the keys and the number of recipients. The comprehensive methodology is delineated in **Algorithm 3**. It takes as input the 64-bit Key ID K_{id} which is computed from the recipient's public key, and the encrypted symmetric key K_e, which is generated using the recipient's ECC public key. The algorithm outputs a successful blockchain transaction that stores the K_{id}, K_e mapping in the smart contract.

Step 9. Transaction request (from recipient)

Each designated recipient initiates a transaction request aimed at retrieving the encrypted message key tuple (K_{idi}, K_{ei}) from the blockchain infrastructure. The request encompasses the recipient's distinct Key ID (K_{idi}), which was derived during the key distribution process. The smart contract enforces access control mechanisms by validating that the public key registered for the requesting entity aligns with the intended K_{idi}. Upon successful verification, the blockchain system authorizes the release of the corresponding encrypted key K_{e_i} to the authenticated recipient. Conversely, if the request fails to meet validation criteria—such as mismatched Key ID, unregistered identity, or unauthorized 'position', the transaction is rejected. It ensures that only

Algorithm 3 Multi-Recipient (K_{id}, K_e) tuple Uploading

Input: Recipient public key P_K, Encrypted session key K_e
Output: Transaction storing (K_{id}, K_e) on blockchain

if userIsRegistered(userEmail) **then**
  The user registers and obtains its identity
  user ← getUserInfouserEmail
  if user.position = "X" **then**
    Sender is authorized to upload multi-recipient data
    for all recipient ∈ recipientList **do**
      $P_k \leftarrow$ getPublicKeyrecipient
      $fp \leftarrow \text{SHA256}P_k$
      $K_{\text{id}} \leftarrow \text{LSB64}fp$
      $K_e \leftarrow \text{encrypt}K, P_k$
      tuple ← combineTupleK_{id}, K_e
      appendToPayloadtuple
    end for
    submitTransactionuserEmail, payload
    storeInBlockchainuserEmail, payload
  end if
end if

authorized recipients are granted access to their respective encrypted keys.

Step 10: Access (K_{id}, K_e) tuple

Once the combined (K_{id}, K_e) are uploaded to the blockchain, each assigned recipient is entitled to initiate a request for access to their respective encrypted key. Utilizing their ECC public key, the recipient systematically derives their Key ID (K_{id}) by executing the SHA-256 hashing function and extracting the 64 least significant bits. This Key ID is subsequently submitted in a formal request to the smart contract. The contract verifies whether the requester's public key is authorized and correctly mapped to the claimed (K_{id}). If verified, the contract retrieves and transmits the encrypted key (K_e) associated with that Key ID. This procedural framework guarantees that access to the encrypted key is exclusively available to the intended recipient, thereby preserving minimal on-chain overhead and ensuring cryptographic confiden-

tiality. The underlying logic is comprehensively outlined in **Algorithm 4**. It takes as input the same Key ID and the requester's public key. It verifies the validity of the requester by recomputing the Key ID from the submitted public key. If the verification succeeds, the corresponding encrypted key K_e is returned to the recipient. Otherwise, access is denied.

Algorithm 4 Accessing (K_{id}, K_e) by Designated Receiver

Input: Requestor's Public Key P'_k, Claimed Key ID (K_{id})
Output: Encrypted session key (K_e) if verification passes; else `null`

if userIsRegistered(userEmail) **then**
  The receiver registers and obtains identity;
  user ← getUserInfouserEmail
  P_k ← getPublicKeyuser
  fp ← SHA256P_k //Generate fingerprint
  K_{id} ← LSB64fp //Derive 64-bit Key ID
  payload ← requestPayloadK_{id}
  if verifyOwnership*user*, K_{id} **then**
    K_e ← retrieveEncryptedKey*payload*, K_{id}
    K_e
  else
    Unauthorized access attempt; reject
    Access Denied
  end if
end if

Step 11: Separate (K_{id}, K_e) tuple

Once the authorized recipient has successfully retrieved the composite tuple (K_{id}, K_e) from the blockchain, the next step is to isolate the encrypted key (K_e) from the tuple. The configuration (K_{id}, K_e) constitutes a composite data structure that associates a distinct 64-bit Key ID with the encrypted symmetric key, which serves the function of decrypting the email content. Upon extraction, the recipient systematically dissects the structure to isolate and eliminate the Key ID, as it has fulfilled its role in routing and identification. This step ensures that the receiver is left with only the cryptographic payload

(K_e), which can now be decrypted using their corresponding private ECC key.

Step 12. Collect (K_e)

After successfully isolating, each receiver collects their respective K_e. The length of K_e varies as 32 bytes, 48 bytes or 64 bytes depending on the encryption strength used previously. After that each receiver stores it temporarily in a secure memory buffer or storage container, preparing it for the next stage of decryption using the corresponding ECC private key.

Step 13. Execute decryption

After collecting their individually encrypted key (K_e), each designated recipient proceeds to decrypt it using ECC algorithm. The decryption process in the multi-recipient system follows a decentralized and recipient-specific path, where every user performs their own decryption locally, ensuring confidentiality and scalability. The following stages are executed for decryption:

- **Select key pair length:** Initially, the system identifies the requisite ECC key pair for decryption predicated on the length of K_e accessed by the recipient. Given that the lengths of K_e are 32 bytes, 48 bytes, and 64 bytes, the system autonomously ascertains the corresponding decryption algorithms of ECC-256, ECC-384, and ECC-521, respectively.

- **Employ key pair in decryption:** Following the selection of the decryption algorithm, the pertinent private key is utilized to decrypt K_e on the recipient's end.

Step 14. Get random key (K)

Upon the successful decryption of their respective encrypted key (K_e), each recipient independently acquires the original random key (K). This key functions as the symmetric decryption key that was initially employed by the

sender to encode the actual content of the email. Subsequent to the retrieval of K, the recipient applies it to decrypt the original email message content, thereby facilitating secure and private access to the communication. The confidentiality of the message is thus preserved end-to-end, from sender to each specific receiver, ensuring that each recipient reads only their intended message.

7.4 Implementation Details

The proposed system was implemented using the Ethereum blockchain environment, with smart contracts developed in Solidity and tested via the Truffle Suite on a local Ganache network. Our experimental setup supports variable ECC key lengths (256, 384, and 521 bits), and simulates multi-recipient scenarios ranging from 1 to 25 receivers. The performance was rigorously analyzed based on key metrics such as gas consumption, transaction cost, and memory usage, demonstrating linear scalability and system efficiency across increasing recipient numbers.

7.4.1 Implementation Environments

The proposed system is developed and tested in a dual-layer environment comprising an off-chain backend and an on-chain Ethereum layer. The development machine runs Windows 11 Pro (64-bit) on an Intel Core i7 CPU (3.2 GHz) with 16 GB RAM. This desktop-class hardware comfortably supports a local blockchain and cryptographic operations. The software stack includes Truffle v5.11.5 (Ethereum development framework) and Ganache as a local testnet, along with Node.js v20.0.0 for the off-chain logic.

7.4.2 Off-chain and On-chain Layer

On the off-chain layer, a Node.js (v20.0.0) backend implements key generation, encryption, and contract interaction. Node.js is well-suited for this because it provides the crypto library (wrapping OpenSSL) for secure random bytes and cryptographic primitives. When the sender needs to share a key, the Node app generates a fresh random symmetric key K (e.g. via crypto.randomBytes), and then encrypts K for each recipient. In practice we use an elliptic-curve hybrid scheme: we generate a random K, and then for each recipient retrieve their public key and use ECC to encrypt K under that key. After encrypting K for each user, the Node app bundles the ciphertexts along with a generated Key ID (K_{id}) that tags this key-sharing transaction. Finally, Node uses Web3.js (Ethereum JavaScript library) or Truffle scripts to send a transaction to the smart contract, passing (K_{id}, K_e) as arguments.

On the on-chain Ethereum layer, the smart contract (written in Solidity 0.8.x) receives and stores the encrypted keys. In the simplest design, each (K_{id}, K_e) pair is stored in the contract's state (for example, in a mapping or struct). Because each encrypted key is 32–64 bytes (depending on curve), storing it on Ethereum is expensive, about 20,000 gas per 32-byte word. Thus, supporting multiple recipients means storing multiple such entries. We use Ganache to deploy this contract locally and submit transactions to it. The transaction receipts (logged by Truffle/Ganache) show exactly how much gas each call consumes, so we can perform gas analysis. We use Truffle to compile and deploy our Solidity contract to the Ganache blockchain, to write migration and test scripts, and to obtain transaction receipts (including gas usage). Ganache is used to measure gas consumption as we scale the number of recip-

ients; it shows gas used per transaction as we can analyze cost growth.

7.5 Evaluation and Result Analysis

The result analysis focuses on four key aspects of the proposed system. First, we present the variation of ECC key lengths on gas and transaction cost. There are four basic operations of smart contracts in our proposed system. To deploy the contract, **Deployment** operation is used. The **RegisterUser()** is used to register each entity in the network. The remaining two functions k_e**Uploading()** and k_e**Accessing()** are used to upload and to access the (K_{id}, K_e) tuple, respectively. Second, we examine how increasing the number of recipients affects the gas usage and storage requirements of smart contract functions. Thirdly, correlation analysis is conducted to quantify the linear relationship between the number of recipients and key performance metrics, revealing strong positive correlations and confirming the scalability of the system. Lastly, a comparative analysis carried out based on proposed system with existing works.

7.5.1 Impact of Variable ECC Key Lengths on Gas Consumption and Transaction Cost

Analysis of gas consumption:

The proposed multirecipient system leverages a dual-environment design, comprising an off-chain application layer for key generation and encryption, and an on-chain Ethereum blockchain layer responsible for smart contract execution. Within the blockchain environment, the Solidity-based smart contracts

orchestrate the key registration, distribution, and retrieval processes for all recipients.

Table 7.1: Gas consumption for smart contract operations with varying K_e key sizes.

Operation	$K_e = 32$ **Bytes**	$K_e = 48$ **Bytes**	$K_e = 64$ **Bytes**
Deployment	1,736,765	1,743,684	1,750,603
RegisterUser()	134,916	134,916	134,916
K_eUploading()	116,814	137,249	157,683
K_eAccessing()	34,731	35,072	35,412

** 1 GWEI = 10^{-9} Ether, 1 Ether =* 2962.73 USD on May 3, 2024.

As shown in Table 7.1, we analyzed gas usage for the core smart contract functions under varying encrypted key lengths - 32 bytes, 48 bytes, and 64 bytes corresponding to ECC-256, ECC-384, and ECC-521 respectively. Each operation was evaluated by uploading a batch of recipient-specific key tuples (K_{id}, K_e), which increases the data complexity and impacts overall gas usage.

Compared to the single-recipient system [7], where only a standalone encrypted key K_e is stored, the multi-recipient system incurs a slight gas overhead due to the combined structure of Key ID and encrypted key being uploaded for each user. This design, while marginally more expensive in terms of gas, ensures accurate recipient mapping and enables auditable, scalable key management.

For example, the gas usage for the $\boldsymbol{K_e}$**Uploading()** function increases from 96,321 GWEI (for single recipient with 32-byte key) to 116,814 GWEI in the multi-recipient configuration, which represents an approximate **21.3%** increase. Similarly, for 64-byte keys, the uploading gas rises from 137,190 GWEI to 157,683 GWEI, marking a **14.9%** increase. This rise stems from the added data storage and encoding of the (K_{id}, K_e) tuple in the blockchain.

The **Deployment** operation exhibits a modest increase of roughly **0.2–0.3%** across all key sizes due to minor overhead from initializing metadata structures supporting multi-recipient indexing. The **RegisterUser()** and $\boldsymbol{K_e}$**Accessing()** functions experience minimal gas variation (typically within **0.1–0.6%**) due to their consistent procedural logic.

Despite these increases, the maximum recorded gas usage 1,750,603 GWEI for deployment with 64-byte K_e, which remains well below Ethereum's block gas limit (6,721,975 GWEI). Thus, the multi-recipient architecture is computationally feasible and deployable in practice, where minimal gas overhead yields significantly enhanced communication scalability.

Analysis of transaction cost:

The transaction cost analysis is conducted based on smart contract operations executed within the Ethereum environment. Similar to the single-recipient model, the system architecture includes both off-chain and on-chain environments. All smart contract functionalities are executed within the on-chain context, and each operation incurs a specific gas cost, subsequently converted to a monetary value in USD based on the Ether-to-USD exchange rate. The minimum amount of gas requisite for executing a transaction on the Ethereum blockchain is quantified as 1 GWEI, which is equivalent to 10^{-9} Ether. On the date of May 8, 2025, throughout the entirety of the analytical phase, the dominant exchange rate that delineated the correlation between Ethereum and the United States dollar was documented at a value of 1 Ether, corresponding to 1829.79 USD.

Figure 7.3 presents the transaction costs (in USD) for four core smart contract operations **Deployment**, **RegisterUser()**, $\boldsymbol{K_e}$**Uploading()**, and $\boldsymbol{K_e}$**Accessing()** across three key lengths: 32 bytes, 48 bytes, and 64 bytes.

These correspond to ECC-256, ECC-384, and ECC-521 encryption strengths, respectively. From the examination of this figure, it is apparent that an increase in the length of K_e, evolving from 32 bytes to 48 bytes, and subsequently from 48 bytes to 64 bytes, results in a corresponding rise in the associated transaction costs for the operations of **Deployment**,$\boldsymbol{K_e}$**Uploading()**, and $\boldsymbol{K_e}$**Accessing()**. These three operations are intrinsically linked to the length of K_e. In contrast, for the operation of **RegisterUser()**, the transaction cost remains constant regardless of the length of K_e, as the parameters required to register a user are uniform across all cases.

A notable distinction in our multi-recipient system is the inclusion of (K_{id}, K_e) tuples instead of merely transmitting K_e as in the single-recipient system. This structural change marginally increases the gas usage, as observed in all smart contract functions, particularly for $\boldsymbol{K_e}$**Uploading()**. However, despite this rise in gas units, the actual transaction cost in USD appears lower in the multi-recipient model. This is attributed to the change in the conversion rate between Ether and USD during the experiment. While the transaction cost for the single-recipient system [7] was calculated using the Ether price of **1 ETH = 2962.73 USD** (as per May 3, 2024), the multi-recipient system computations were performed using a lower Ether valuation of **1 ETH = 1829.79 USD** (as per May 8, 2025). Consequently, the overall transaction cost (USD) for each operation is reduced. This discrepancy underscores that while gas usage reflects the on-chain resource consumption, the actual monetary transaction cost is highly sensitive to the prevailing cryptocurrency exchange rates.

Despite of a marginal increase in gas consumption, the integration of Key IDs for the purposes of secure and traceable recipient identification does not

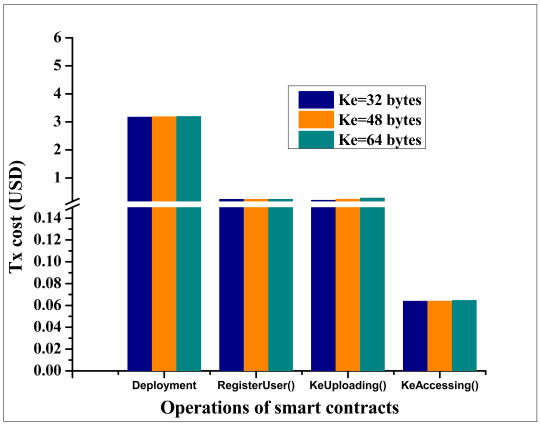

Figure 7.3: Transaction cost of the system for various length of K_e.

impose any significant financial encumbrance, thereby endorsing the viability of the proposed system within practical, multi-user secure communication contexts [63].

7.5.2 Effects of Multi-Recipient on Smart Contract Operations

In the Ethereum-based blockchain system, each computational operation consumes gas. So, the adding multi-recipient logic naturally increases gas usage. Among the contract's four core operations **Deployment**, **RegisterUser()**, $\boldsymbol{K_e}$**Uploading()**, and $\boldsymbol{K_e}$**Accessing()**, only **Deployment** and $\boldsymbol{k_e}$**Uploading()** exhibit higher gas consumption under multi-recipient system.

The **RegisterUser()** function takes same parameters such as 'userName', 'userEmail', 'organization', and 'position', similiar to single recipient system. Because of this, its logic does not depend on the number of recipients and its gas usage remains constant. Similarly, for $\boldsymbol{K_e}$**Accessing()** function, the system can autonomously identify the relevant (K_{id}, K_e) tuple for the specified

recipient in accordance with their Key ID. For this purpose, the recipients need only to retrieve their corresponding (K_{id}, K_e) tuple, which will remain unchanged despite an increase in the number of recipients, with no influence on gas values.

By contrast, **Deployment** now includes extra code and initialization to handle multiple recipients, increasing the contract's bytecode size and number of storage writes. As the Etherum gas consumption calculated on per byte of deployed code and per storage write, the more complex multi-recipient deployment incurs higher gas. Likewise,$\boldsymbol{K_e}$**Uploading()** has been extended to process up to 25 recipients. Typically by using a loop or group operation that stores each recipient's (K_{id}, K_e) tuple one by one; every extra (K_{id}, K_e) tuple added increases the gas values.

In Figure 7.4, which plots the gas used for contract **Deployment** versus the number of recipients. The gas consumption grows roughly linearly with the recipient count for all encrypted key K_e sizes. For example, for the smallest key length $K_e = 32$ bytes, the deployment gas rises from on the order of 2 million (at 1 recipient) to about 4–4.5 million at 25 recipients. For the length of key, $K_e = 64$ bytes, the corresponding rise is from roughly 2.4 million to about 5.6 million. Thus across the 1 to 25 range the gas increases by about 120–130% ($K_e = 32$ bytes) to 140% ($K_e = 64$ bytes). The differences between key lengths are modest but consistent. Larger key length consume more gas at every point, so doubling recipients roughly doubles storage gas. Moreover, the maximal gas threshold of Ethereum block is 6721975 GWEI. From the observation we can see that, the maximum gas used for the **Deployment** operation (when the number of recipient=25 and $K_e = 64$ bytes), is 2534871 GWEI, which is only 37.71% of the maximum limit of block. Consequently, this

serves to validate and substantiate the inherent practicality and operational effectiveness of the system, thereby reinforcing its applicability in real-world scenarios.

Figure 7.5 shows a similar trend for the $\boldsymbol{K_e}$**Uploading()** function, which uploads(K_{id}, K_e) tuple to the chain. Gas usage again increases almost linearly with recipients, and larger keys require more gas per recipient. Here the effect is even more pronounced, at 25 recipients, the 64-byte key case uses roughly 2.4–2.5 million gas versus about 1.4–1.5 million for the 32-byte key (an increase on the order of 70–80% over the smaller key). In percentage terms, going from 1 to 25 recipients increases gas by about 110% for $K_e = 32$ bytes, but by around 180–200% for $K_e = 64$ bytes. This indicates that the per-recipient gas increment is larger for larger key sizes. Notably, however, the overall gas scale for $\boldsymbol{K_e}$**Uploading()** is lower than for **Deployment** operation. It's roughly half as much which reflecting that the deployment likely includes additional fixed overhead (such as contract creation) beyond the per key storage. From the empirical analysis, it is evident that the highest volume of gas consumed during $\boldsymbol{K_e}$**Uploading()** operation, with a recipient count of 25 and a key size of $K_e = 64$ bytes, amounts to 2385112 GWEI, which constitutes merely 35.48% of the upper threshold of the blockchain. Consequently, this finding serves to validate and substantiate the inherent practicality and operational efficacy of the system, thereby enhancing its relevance within real-world applications.

In Figure 7.6, which plots the blockchain storage (memory) required to store the encrypted keys, K_e. The trends are again near-linear but with much smaller slopes. Each set of bars (black=32 Bytes, red=48 Bytes, blue=64 Bytes) grows roughly linearly with recipient count, and larger keys require proportionally more memory. For instance, at 25 recipients the stored key

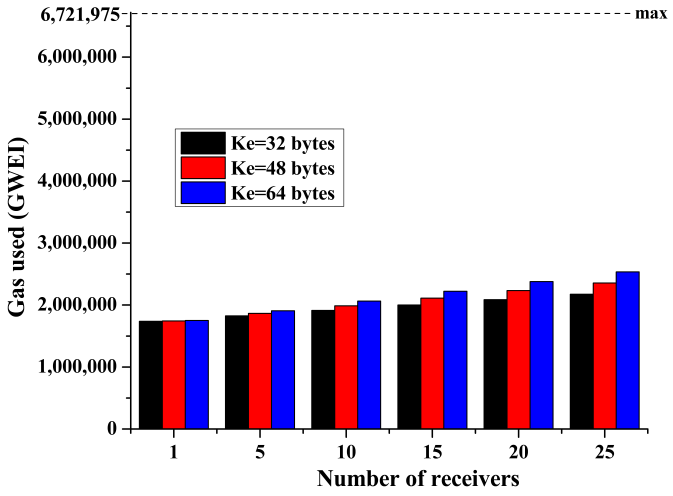

Figure 7.4: Gas used for multi-recipient system with various length of K_e for Deployment operation.

memory is on the order of 200,000–220,000 units (bytes) for $K_e = 64$ bytes, versus around 180,000–190,000 for $K_e = 32$ bytes. Over the range, memory grows by only 10–15% from the smallest to largest key for a fixed number of recipients, reflecting that key length has a direct but moderate effect on storage size. The overall increase in memory from 1 to 25 recipients is on the order of 30–40% for all key sizes, again indicating a roughly linear data scaling. The empirical findings indicate that within the comprehensive examined, the peak memory capacity required in case of the number of recipient,N =25 and $K_e = 64$ bytes was determined to be approximately 23 KB, which is significantly inferior to the maximum block size (1-2 MB), only (1.15%). This evaluation underscores the practicality of the system in relation to its memory specifications.

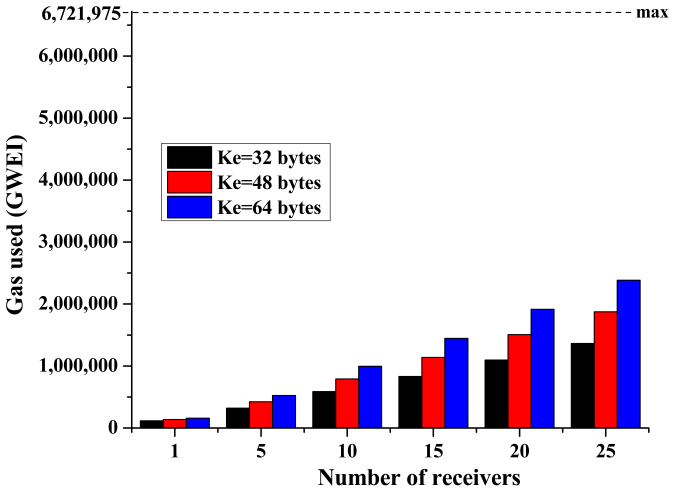

Figure 7.5: Gas used for multi-recipient system with various length of K_e for $K_eUploading()$ function.

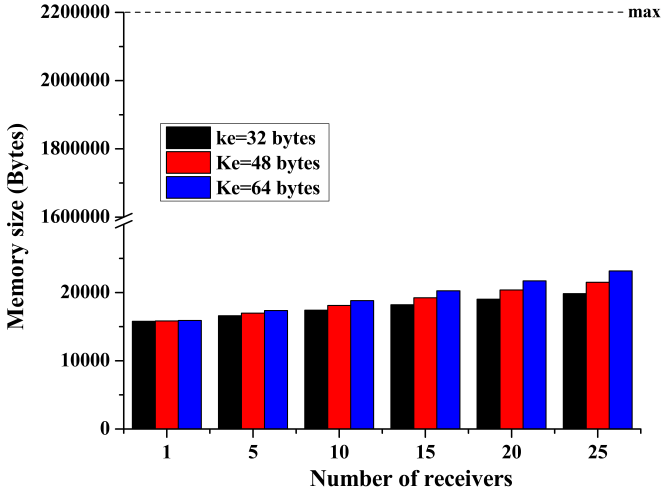

Figure 7.6: Memory required for multi-recipient system to store various length of K_e in the block.

7.5.3 Correlation Analysis

Correlation analysis plays a crucial role in evaluating the scalability and efficiency of the proposed multi-recipient encrypted key distribution system. By quantifying the linear relationships between the number of recipients and performance metrics such as gas usage and memory consumption, it enables data-driven validation of system behavior. This insight is essential for optimizing smart contract performance and forecasting system costs in real-world deployments.

Figures 7.7–7.9 illustrate the correlation analyses between the number of recipients and three critical metrics of the proposed multi-recipient blockchain-based email encryption system: (i) gas usage for **Deployment**, (ii) gas usage for $\boldsymbol{K_e}$**Uploading()**, and (iii) memory required to store the encrypted key (K_e). These analyses are specifically carried out for the case where (K_e = 32 bytes), as for (K_e = 48 bytes) and (K_e = 64 bytes) cases exhibit the same linear trend (with slightly larger slopes due to larger key sizes), but the scalability behavior is unchanged.

Figure 7.7, reveals a near-perfect linear relationship between the number of recipients and the gas consumed during the **Deployment** operation of the smart contract. The regression analysis yields a linear equation of the form:

$$y = 17843.68x + 1{,}731{,}649.56, \quad R^2 = 0.9989$$

where y denotes the gas used and x represents the number of recipients. The intercept (1,731,649.56 GWEI) signifies the baseline gas cost for initializing the contract with a single recipient, while the slope (17,843.68 GWEI) quantifies the incremental gas required for each additional recipient. The high

R^2 value (0.9989) confirms a very strong positive correlation, implying that the deployment cost increases consistently and predictably with recipient count. This trend supports our hypothesis that the underlying smart contract complexity, which includes state variables, mappings, and data structures for multi-recipient key sharing, leads to higher gas requirements as the recipient pool grows.

Figure 7.8 presents the correlation for the $\boldsymbol{K_e}$**Uploading()** function, which handles the upload of (K_{id}, K_e) tuples to the blockchain. The regression equation derived from this data is:

$$y = 51{,}715.55x + 66{,}540.32, \quad R^2 = 0.9998$$

This function exhibits an even stronger correlation with $R^2 = 0.9998$, highlighting near perfect linearity. The slope (51,715.55 GWEI) indicates the per recipient gas increase due to additional storage writes for each encrypted key. The relatively low intercept (66,540.32 GWEI) reflects the minimal overhead required for initial transaction preparation. These results validate the efficiency of the smart contract's batch key upload mechanism, which scales linearly and efficiently without introducing non-linear bottlenecks.

Figure 7.9 investigates the system's memory requirements, focusing on the bytes needed to store encrypted keys for multiple recipients. The linear regression model is:

$$y = 165.04x + 15{,}730.58, \quad R^2 = 0.9989$$

This again demonstrates a strong linear relationship between recipient

count and memory usage. The slope (165.04 bytes/recipient) closely aligns with the expected growth rate for storing 32 byte keys along with metadata (e.g., Key ID), while the intercept (15,730.58 bytes) likely captures the fixed memory required to deploy and maintain initial storage structures. The high R^2 score affirms that memory growth in the system is not only predictable but also tightly controlled, making it feasible for moderate to large recipient groups.

Overall, the correlation results underscore that both gas usage and memory requirement scale linearly with the number of recipients. The strong coefficients of determination ($R^2 > 0.998$ in all cases) confirm that the system's resource consumption is predictable and consistent, which is essential for evaluating cost effectiveness and blockchain feasibility. These findings highlight the practical scalability of the proposed multi-recipient encrypted key sharing system and reinforce its design viability for secure and efficient email communications in distributed environments.

7.5.4 Comparative Analysis among Proposed System and Existing Systems

To provide a comprehensive understanding of the functional and performance differences, here we discuss first a comparative analysis between the single recipient system [7] and our proposed multi-recipient encrypted key distribution framework. After that make a comparison of proposed work with existing real world literature.

Figure 7.10 indicates gas consumption for contract **Deployment** as a function of the number of recipients, shown on a logarithmic scale for clarity.

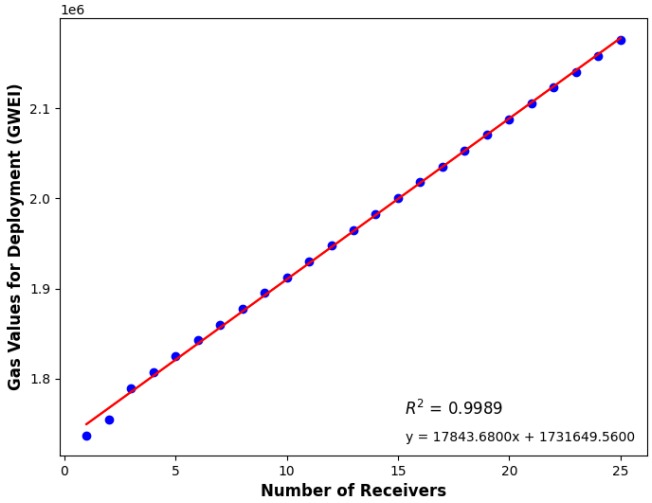

Figure 7.7: Correlation analysis of proposed system for Deployment operation ($K_e = 32$ bytes).

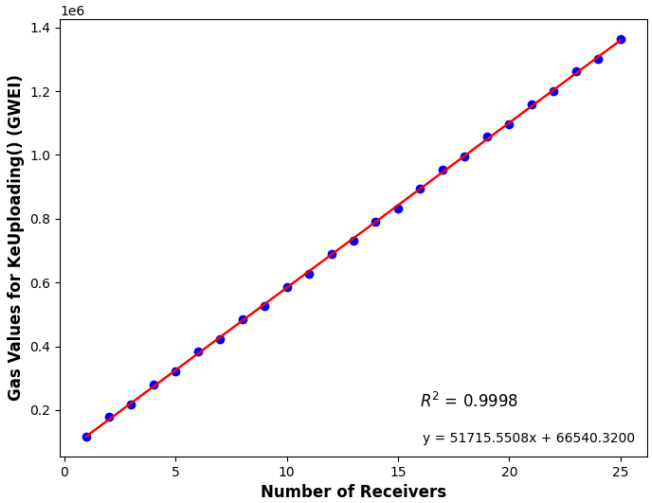

Figure 7.8: Correlation analysis of proposed system for $K_eUploading()$ function ($K_e = 32$ bytes).

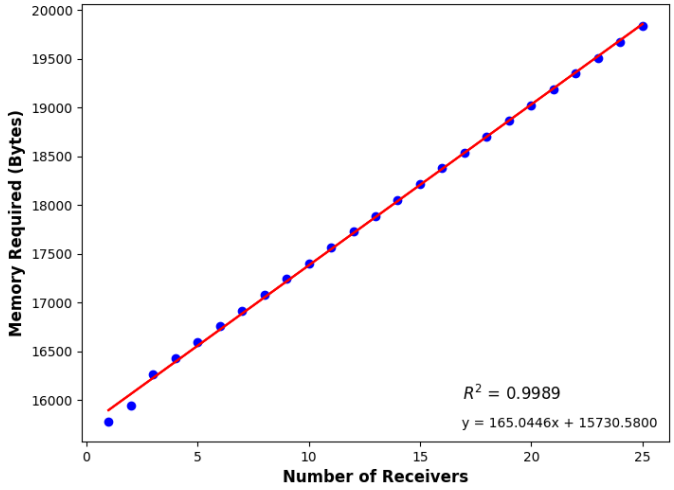

Figure 7.9: Correlation analysis of proposed system for memory requirement to store encrypted key ($K_e = 32$ bytes).

The single-recipient approach (dashed lines) requires deploying a separate contract for each recipient, causing the gas cost to grow linearly with the number of recipients. In contrast, the multi-recipient approach (solid lines with symbols) deploys one contract to serve all recipients, yielding a much flatter growth curve. For example, at 25 recipients, the multi-recipient deployment consumes on the order of 10^6 gas, whereas the single-recipient deployments cumulatively approach 4×10^7 gas. This corresponds to an approximate 95% reduction in gas for the multi-recipient system at the upper range of tested recipients. The log-scale plot highlights how the single-recipient curves rise steeply with the number of receivers, while the multi-recipient curves remain nearly horizontal. These results demonstrate excellent scalability of the multi-recipient architecture in the deployment phase by maintaining the one-time contract setup cost across many recipients, it avoids the repetition of expensive deployment operations [7].

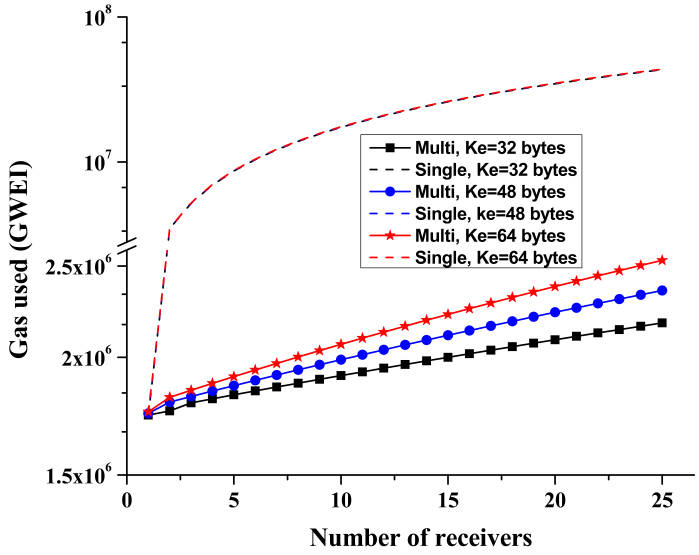

Figure 7.10: Gas usage comparison for Deployment operation in single vs. multi-recipient system.

Figure 7.11 presents gas consumption for the encrypted key-uploading operation $\boldsymbol{K_e}$**Uploading()** versus number of recipients (linear scale). Each single-recipient upload (dashed lines) is a separate transaction delivering an encrypted key K_e to one recipient, so serving multiple recipients requires repeated transactions. The total gas cost for N single uploads grows linearly as $N \times G_{\text{upload}}$, where G_{upload} is the gas for one upload. In the multi-recipient design (solid lines with symbols), a single $\boldsymbol{K_e}$**Uploading()** transaction can embed keys for all recipients at once. As a result, the multi-recipient gas curves grow much more slowly with N. We observe that for a small number of recipients ($N = 1$), the multi-recipient approach incurs a slight overhead (its transaction is more complex, so for example at $N = 1$ the multi upload uses ∼116k gas versus ∼96k for the single [7]. However, beyond one recipient, this overhead is quickly offset by savings: for $N \geq 2$, the multi-recipient

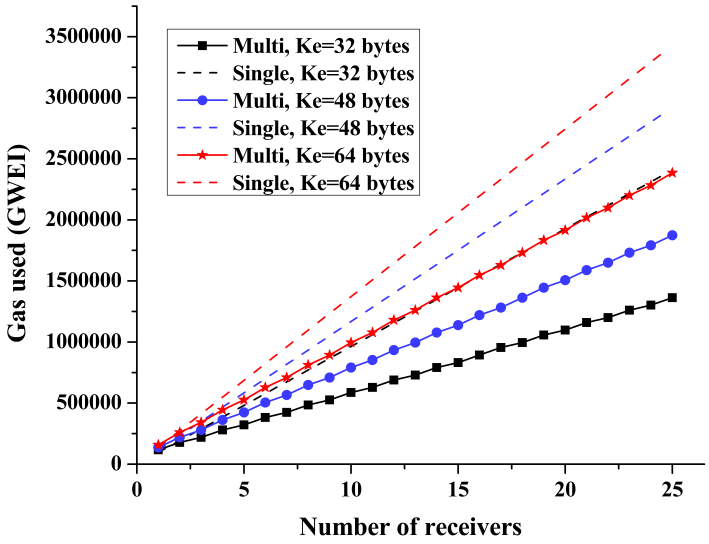

Figure 7.11: Gas usage comparison for $K_eUploading()$ function in single vs. multi-recipient system.

method consistently uses less total gas than the equivalent separate uploads. By $N = 25$, the multi-recipient upload consumes only about 1.36–2.38 million gas (for 32–64 byte keys, respectively) compared to 2.41–3.43 million if done via individual uploads. This represents a significant gas reduction (approximately 30–45% less gas at $N = 25$) achieved by batching the uploads into one transaction instead of many.

Table 7.2 presents a comparison spans across design architecture, smart contract behaviors, basic gas usage, memory implications, and application specific suitability. The single recipient system is designed for one-to-one secure email communication, where a random key K is encrypted using the receiver's public key, resulting in K_e which is then uploaded to the blockchain. The system is streamlined and lightweight but lacks efficiency and scalability in collaborative or group environments.

In contrast, the multi-recipient system enhances scalability by allowing a single sender to transmit different K_e values to multiple recipients simultaneously. This is achieved by associating each encrypted key with a unique Key ID (K_{id}, K_e) pair, thereby ensuring both traceability and access control. For the basic operation ($N = 1$), the multi-recipient scheme incurs slightly higher gas usage in the single-recipient case due to its added indexing and recipient-handling logic. For example, uploading a 32-byte encrypted key for one recipient costs about 116,814 GWEI in the proposed system versus 96,321 GWEI in the single-recipient system, roughly a 21% increase.

However, the multi-recipient approach becomes significantly more efficient as the number of recipients grows. Distributing the same key to 25 recipients with the single-recipient scheme would require 25 separate $\boldsymbol{K_e}$**Uploading()** transactions (one per recipient), consuming a total of approximately 2.4110^6 GWEI for a 32-byte key. In contrast, the proposed system can deliver the key to all 25 recipients in a single batched upload, with a total gas cost of only about 1.4610^6 GWEI. This batched operation reduces the number of transactions from 25 down to 1, amortizing the fixed transaction costs and yielding an overall gas saving of around 40% in this scenario.

Table 7.3 presents a comparison of the proposed work with several existing studies to demonstrate its applicability to real-world systems. The use of blockchain for secure key sharing has been explored in various domains. For instance, email-based systems are covered in [7, 44, 86]. Blockchain integration in cloud computing is discussed in [83, 84, 87], while web-based applications are presented in [82]. Furthermore, general PKI-based frameworks are analyzed in [79, 81]. Finally, IoT and vehicular applications are summarized in [78, 80, 85]. However, a deeper comparative analysis reveals that most of these studies

offer limited performance assessments, lack cryptographic flexibility, or do not incorporate privacy-preserving mechanisms.

Regarding **cryptographic flexibility**, we find that prior studies often use fixed encryption schemes or fail to mention cryptographic agility. Our system supports *variable ECC key lengths* (256-bit, 384-bit, and 521-bit), allowing users to tailor encryption strength according to device capability and security requirements. This flexibility is validated through case studies and performance evaluations previously discussed.

A major strength of the proposed method lies in its **gas and memory cost evaluation**. While[78, 82, 83] considered gas cost in blockchain operations, and [80] analyzed both gas and memory cost, most others omitted it. Our system not only evaluates gas usage but also analyzes memory footprint of smart contract storage, enabling developers and organizations to assess deployment feasibility and budgetary considerations.

Additionally, **correlation analysis** between variables such as recipient count, gas usage, and memory cost has not been studied in prior works. Our manuscript presents a statistical regression model with high R^2 values, confirming predictable scalability and validating the robustness of the proposed contract design.

Finally, **security attack analysis** was included in most of the existing works except [81], but not comprehensively. Our proposal includes mitigation of common blockchain threats such as man-in-the-middle (MITM) and unauthorized key access by design, with zk-SNARKs and Key ID mapping playing central roles.

Overall, the proposed system presents a holistic design that bridges the limitations of earlier schemes. It unifies multi-recipient scalability, crypto-

graphic adaptability, blockchain cost-memory analysis, and correlation analysis into a single framework tailored for secure email communication — an advancement not collectively addressed in any single prior work.

7.5.5 Security and Privacy Analysis

Our multi-recipient design maintains strong security and privacy properties while improving efficiency and scalability:

- **Privacy Preservation:** Only encrypted keys Ke_i and Key IDs ($\mathrm{K_{id_i}}$) are posted publicly. The actual PGP key K never appears on-chain. Since $\mathrm{K_{id_i}}$ is a one-way hash of the public key, observers cannot infer recipient identities or keys from the blockchain data. If desired, the scheme still supports zk-SNARKs: a sender could optionally attach a zero-knowledge proof (as in [6]), without revealing any secret information. Thus, the system can achieve the same level of anonymity and confidentiality as the single-recipient version.

- **Integrity, Authenticity, and Confidentiality:** Authentication is guaranteed by the blockchain: users must register and transact using their cryptographic wallets, and smart contracts enforce access controls. Confidentiality of the email is maintained because the actual message key K is never exposed; only its encrypted shares K_e are on-chain. Since K_e is encrypted with ECC, only the holder of the corresponding private key can decrypt it. We use strong curves (up to 521 bits) so that breaking the ECC encryption would be computationally infeasible.

- **Blockchain Immutability:** Blockchain immutability ensures that once a key tuple is recorded, it cannot be altered or removed. Our smart con-

tract enforces strict conditions on uploads and accesses (e.g., requiring correct registration and matching Key IDs). This prevents unauthorized modifications or forgeries. Because the Ethereum ledger is decentralized, there is no central point to compromise – man-in-the-middle or denial-of-service attacks at the application level are effectively mitigated by consensus. In summary, any $(\mathrm{K_{id}}, K_e$ entry is guaranteed to be untampered, and only the designated recipient can decrypt it.

- **Cost Efficiency:** By batching keys for multiple recipients, the scheme lowers per-recipient gas and storage costs. For example, sending to N users requires roughly one contract invocation (or a few) instead of N separate invocations. This amortizes the fixed transaction overhead across all recipients. Consequently, the total gas and transaction fees per user are significantly reduced compared to repeating the single-recipient protocol. Fewer on-chain writes also mean lower blockchain storage usage per email. These savings make the multi-recipient scheme substantially more scalable for group emails.

Overall, the proposed multi-recipient key distribution preserves the privacy and security guarantees of our previous single-recipient systems while addressing their scalability limitations. The use of Key IDs and bundled transactions ensures that multiple recipients can securely obtain the shared PGP key with minimal overhead, making the system practical for real-world multi-user email encryption.

7.6 Summary

This paper proposed a scalable blockchain-based PGP key distribution system to efficiently support multi-recipient email communication. The introduced Key ID mechanism, derived from the least significant 64 bits of the public key's fingerprint, creates a compact identifier for each user while maintaining security properties. By storing these Key IDs combined with encrypted PGP keys on the blockchain, our system enables recipients to securely access their encrypted keys by eliminating redundant message encryption.

Experimental evaluation with varying recipient counts (N =1-25) demonstrated that the multi-recipient architecture achieves approximately 40% gas savings compared to the single-recipient approach for group communications. This is despite incurring slightly higher initial costs for individual recipients due to added indexing logic. Statistical analysis confirmed strong linear scalability with high correlation coefficients ($R^2 > 0.99$) between recipient count and both gas consumption and memory usage. Thus, Performance metrics validate the system's viability for secure email communications in organizational settings where encrypted group messaging is frequently required.

Future work will focus on smart-contract refactoring along with modularizing reusable logic [88], to minimize code duplication and reduce execution overhead without altering the core design. These improvements can make the system cost-effective and enhancement in gas efficiency for practical deployment in organizational email environments where secure group communications are critical, such as financial institutions sharing encrypted reports, healthcare providers exchanging patient information, or government agencies distributing classified documents.

Table 7.2: Comparison of single [7] vs. multi-recipient system.

Feature	**Single Recipient System [7]**	**Proposed Multi-Recipient System**
Key Distribution Scope	One sender to one receiver	One sender to multiple receivers
Key Data Stored	K_e only	Tuple of (K_{id}, K_e) for each recipient
Smart Contract Complexity	Simple function calls	Enhanced functions with recipient handling and indexing
Deployment Gas (32B key) (N=1)	1,733,302 GWEI	1,736,765 GWEI (↑0.2%)
KeUploading() Gas (32B key) (N=1)	96,321 GWEI	116,814 GWEI (↑21.3%)
KeAccessing() Gas (32B key) (N=1)	34,391 GWEI	34,731 GWEI (↑ 1%)
RegisterUser() Gas (N=1)	134,916 GWEI	134,916 GWEI (same)
Transaction Cost Sensitivity	Depends on K_e length	Depends on both K_e length and number of recipients
Memory Usage Pattern	Stores one K_e per block	Stores multiple (K_{id}, K_e) tuples per block
Correlation (Gas vs Receivers)	Not applicable	$R^2 = 0.999$ (linear, predictable scaling)
Security Model	Targeted encryption with public key	Indexed encryption using Key ID for access control
Use Case Suitability	One-to-one secure email (e.g., manager to staff)	Collaborative/mass communication (e.g., manager to teams)

Table 7.3: Comparison of proposed system with existing blockchain-based key sharing mechanisms.

Reference / Year	**Encryption Technique**	**Gas / Cost Eval.**	**Memory Eval.**	**Correlation Analysis**	**Security Analysis**
[44] / 2020	Not defined	✗	✗	✗	✓
[78] / 2021	ECC	✓	✗	✗	✓
[79] / 2022	Not defined	✗	✗	✗	✓
[80] / 2022	ECC	✓	✓	✗	✓
[81] / 2024	CRT-based scheme	✗	✗	✗	✗
[82] / 2024	Not defined	✓	✗	✗	✓
[7] / 2024	ECC	✗	✗	✗	✓
[83] / 2023	AES + ECC	✓	✗	✗	✓
[84] / 2024	Not defined	✗	✗	✗	✓
[85] / 2022	ECC	✗	✗	✗	✓
[86] / 2023	ECC	✗	✗	✗	✓
[87] / 2024	AES + ECC	✗	✗	✗	✓
Proposed / 2025	**Variable ECC**	✓	✓	✓	✓

Chapter 8

Conclusion and Future Works

8.1 Conclusion

This dissertation presented a comprehensive exploration of secure key distribution mechanisms for email encryption using blockchain technology, focusing on scalability, performance optimization, and privacy preservation. The research unfolded through four progressive studies, each addressing critical challenges in email security while extending the capabilities of blockchain-integrated systems.

The first study introduced a secure email encryption system by integrating Elliptic Curve Cryptography (ECC) with Ethereum-based smart contracts. It demonstrated the feasibility of replacing conventional PGP key-sharing mechanisms with a blockchain-enabled framework, offering enhanced transparency, immutability, and decentralization. Performance metrics such as gas consumption and memory cost were analyzed, validating the practicality of the proposed approach.

Building upon the initial framework, the second study incorporated zero-knowledge proofs through zk-SNARKs to address the privacy vulnerabilities of exposing key access patterns on-chain. By integrating ZoKrates with Solidity-based contracts, this work established a privacy-preserving mechanism that enabled secure email decryption without revealing recipient identities. The study emphasized the importance of balancing verifiability with confidentiality in decentralized environments.

To optimize cryptographic efficiency, the third study investigated the ef-

fects of variable ECC key lengths on system performance. A detailed analysis across multiple scenarios showed how increasing key sizes impacted gas usage, memory, and transaction latency. This case-driven evaluation highlighted the trade-offs between cryptographic strength and computational cost, guiding suitable key selection for context-specific email communication needs.

The final study addressed scalability by designing a multi-recipient encryption framework capable of securely sharing a single encrypted email with multiple recipients in a single blockchain transaction. A compact Key ID mechanism was introduced to support efficient recipient identification, minimizing redundancy in message encryption and reducing on-chain storage demands. This extension not only improved throughput and reduced gas consumption but also paved the way for scalable group communications in secure email systems.

From the experimental analyses and implementation results discussed across all chapters of this dissertation, it is evident that the proposed blockchain-integrated secure email systems significantly enhance cryptographic performance, scalability, and privacy. Each progressive study has contributed to solving specific limitations of the previous one while improving the practicality of secure key sharing for real-world email communication. The key experimental insights and performance improvements observed throughout this research are as follows:

- Successfully integrated Elliptic Curve Cryptography (ECC) with Ethereum smart contracts to replace traditional PGP-based key sharing.This analysis demonstrates the feasibility of the proposed scheme in the real world concerning the required gas values and transaction cost.

- The zk-SNARKs enhanced system provided full privacy for recipient identities during email decryption. Experimental result shows that integrating zk-SNARKs into the system significantly reduces gas usage of 54.1% while reducing total transaction costs of 35.3%.

- Performance analysis of ECC key lengths revealed that 32-byte keys consumed 45.6% less gas than 64-byte keys. On-chain memory usage increased by 38.2% when scaling ECC key lengths from 32 to 64 bytes.

- The scalability of the system enhanced by introducing multi-recipient scenario based on Key ID indexing. Experimental evaluation demonstrated that the multi-recipient architecture achieves approximately 40% gas savings with high correlation coefficient($R^2 > 0.99$) between recipient count and both gas consumption and memory usage.

Overall, the dissertation demonstrated that blockchain-based solutions when combined with ECC, zk-SNARKs, and smart contract optimization can significantly advance the security, privacy, and scalability of encrypted email communication. The proposed methodologies and their systematic evaluations contribute valuable insights for future developments in decentralized identity management and secure data exchange.co

8.2 Future Works

Future work will focus on smart-contract refactoring along with modularizing reusable logic [88], to minimize code duplication and reduce execution overhead without altering the core design. These improvements can make the system cost-effective and enhancement in gas efficiency for practical deploy-

ment in organizational email environments where secure group communications are critical, such as financial institutions sharing encrypted reports, healthcare providers exchanging patient information, or government agencies distributing classified documents.

References

[1] E. Altulaihan, A. Alismail, M. M. H. Rahman, and A.A. Ibrahim, “Email Security Issues, Tools, and Techniques Used in Investigation," Sustainability, vol. 15, pp. 1-28, July 2023.

[2] A. Yakubov, W. Shbair, N. Khan, and R. State, “BlockPGP: A Blockchain-based framework for PGP Key Servers," International Journal of Networking and Computing, vol. 10, pp. 1-24, January 2020.

[3] G. S. Chhabra and D. S. Bajwa, “Review of the e-mail system, security protocols, and email forensics," International Journal of Computer Science and Communication Networks, vol. 5, pp. 201-211, January 2015.

[4] Asif Karim, Sami Azam, Bharanidharan Shanmugam and Krishnan Kannoorpatti. Efficient Clustering of Emails Into Spam and Ham: The Foundational Study of a Comprehensive Unsupervised Framework. *IEEE Access*, *8*: 154759-154788, 2020.

[5] Md. B. Hossain, M. Rahayu, Md. A. Ali, S. Huda, Y. Kodera, and Y. Nogami, “A Smart Contract Based Blockchain Approach Integrated with Elliptic Curve Cryptography for Secure Email Application,” *2023 Eleventh International Symposium on Computing and Networking Workshops (CANDARW)*, Matsue, Japan, pp. 195–201, Nov 2023.

[6] Md. B. Hossain, M. Rahayu, Md. A. Ali, S. Huda, Y. Kodera, and Y. Nogami, “A Blockchain-based Approach with zk-SNARKs for Secure Email Applications,” *Int. J. Networking Comput.*, vol. 14, no. 2, pp. 225–247, 2024.

[7] Md. B. Hossain, M. Rahayu, S. Huda, Md. A. Ali, Y. Kodera, and Y. Nogami, “A Blockchain-Based Approach for Secure Email Encryption with Variable ECC Key Lengths Selection,” *The 8th International Conference on Mobile Internet Security*, Sapporo, Japan, pp. 1–14, 2024.

[8] S. Garfinkel, *PGP: Pretty Good Privacy*. Sebastopol, CA, USA: O’Reilly Media, 1995.

[9] M. Banerjee, J. Lee, and K. W. R. Choo, “A blockchain future for internet of things security: a position paper," Digital Communications and Networks, vol. 4, no. 3, pp. 149-160, August 2018.

[10] M. Fartitchou, K. E. Makkaoui, N. Kannaouf, Z. E. Allali, “Security on Blockchain Technology," In Proceedings of International Conference on Advanced Communication Technologies and Networking (CommNet), vol. 1, pp. 1-7, September 2020.

[11] Z. Wang, H. Jin, W. Dai, K. K. R. Choo, and D. Zou, “Ethereum smart contract security research: survey and future research opportunitie," Frontiers of Computer Science, vol. 15, no. 2, pp. 1-18, October 2020.

[12] S. N. Khan, F. Loukil, C. G. Guegan, E. Benkhelifa, and A. B. Hani, “Blockchain smart contracts: Applications, challenges, and future trends," Peer-to-Peer Network Application, vol. 14, pp. 2901-2905, April 2021.

[13] D. P. Castillo, J. Bermejo, and F. Machio, “A Secure Email Solution Based on Blockchain," Blockchain and Applications, vol. 320, pp. 355-358, September 2021.

[14] V. Rudramalla and R. S. Prasad, “Secured Email Data Based on Blowfish with Blockchain Technology," Science, Technology and Development, vol. 8, pp. 456-464, November 2019.

[15] V. Dimitriadis, L. Maglaras, N. Polemi, I. Kantzavelou, and N. Ayres, “Uncuffed: A Blockchain-based Secure Messaging System," In Proceedings of the Pan-Hellenic Conference on Informatics (PCI), vol. 25, pp. 340-345, February 2022.

[16] H. S. Huang, T. S. Chang, and J. H. Wu, “A Secure File Sharing System Based on IPFS and Blockchain," In Proceedings of the 2nd International Electronics Communication Conference, vol. 20, pp. 96-100, August 2020.

[17] M. F. Hinarejos and J. Ferrer-Gomila, “A Solution for Secure Multi-Party Certified Electronic Mail Using Blockchain," IEEE access, vol. 8, pp. 102997-103006, May 2020.

[18] A. Varghese, “Email Verification Service using Blockchain," Technical Disclosure Commons, vol. 2468, pp. 1-8, September 2019.

[19] J. C. Gonzalez, V. G. Dıaz, E. R. Nunez-Valdez, A. G. Gomez, and R. G. Crespo, “Replacing email protocols with blockchain-based smart contracts," Cluster Computing, vol. 23, pp. 1795-1801, May 2020.

[20] D. Wilson and G. Ateniese, “From Pretty Good To Great: Enhancing PGP using Bitcoin and the Blockchain," In Proceedings of International Conference on Network and System Security, pp. 368-375, August 2015.

[21] Duane Wilson, and Giuseppe Ateniese. From Pretty Good To Great: Enhancing PGP using Bitcoin and the Blockchain. *In Proceedings of Inter-*

national Conference on Network and System Security, volume 1, pages 368-375, 2015.

[22] Akanksha Saini, Qingyi Zhu, Yong Xiang, Longxiang Gao, and Yushu Zhang. Smart- Contract-Based Access Control Framework for Cloud Smart Healthcare System. *IEEE Internet of Things Journal*, 8(7):5914-5925, 2021.

[23] Jiahui Huang, Teng Huang, Huanchun Wei, Jiehua Zhang, Hongyang Yan, Duncan S. Wong, and Haibo Hu. zkChain: A privacy-preserving model based on zk-SNARKs and hash chain for efficient transfer of assets. *Transactions on Emerging Telecommunications Technologies*, 2022(1):1-11, 2022.

[24] Xiaoqiang Sun, F. Richard Yu, Peng Zhang, Zhiwei Sun, Weixin Xie, and Xiang Peng. A Survey on Zero-Knowledge Proof in Blockchain. *IEEE Network*, 35(4):198-205, 2021.

[25] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin. Anonymous distributed Ecash from bitcoin. *In Proceedings 2013 IEEE symposium on security and privacy*, volume 1, pages 397-411, 2013.

[26] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. *In Proceedings 2014 IEEE symposium on security and privacy*, volume 1, pages 459-474, 2014.

[27] Zhangshuang Guan, Zhiguo Wan, Yang Yang, Yan Zhou, and Butian Huang. BlockMaze: An efficient privacy-preserving account-model

blockchain based on zk-SNARKs. *IIEEE Transactions on Dependable and Secure Computing*, 19(3):1446-1463, 2022.

[28] Zhiming Song, Guiwen Wang, Yimin Yu, and Taowei Chen. Digital identity verification and management system of blockchain-based verifiable certificate with the privacy protection of identity and behavior. *Security and Communication Networks*, 2022(1):1-24, 2022.

[29] Lei Xu, Nolan Shah, Lin Chen, Nour Diallo, Zhimin Gao, Yang Lu and Weidong Shi. Enabling the sharing economy: Privacy respecting contract based on public blockchain. *In Proceedings of the ACM workshop on blockchain, cryptocurrencies and contracts*, volume 1, pages 15-21, 2017.

[30] Dongkun Hou, Jie Zhang, Sida Huang, Zitian Peng, Jieming Ma, and Xiaohui Zhu. Privacy-preserving energy trading using blockchain and zero knowledge proof. *In Proceedings 2022 IEEE international conference on blockchain (blockchain)*, volume 1, pages 408-412, 2022.

[31] SoonHyeong Jeong and Byeongtae Ahn. Implementation of real estate contract system using zero knowledge proof algorithm based blockchain. *The Journal of Supercomputing*, 77(10):11881–11893, 2021.

[32] Keke Gai, Haokun Tang, Guangshun Li, Tianxiu Xie, Shuo Wang, Liehuang Zhu, and Kim-Kwang R. Choo. Blockchain-based privacy preserving positioning data sharing for IoT-enabled maritime transportation systems. *IEEE Transactions on Intelligent Transportation Systems*, 24(2):2344–2358, 2022.

[33] Ahmad Al-Jarrah1, Amer Albsharat and Mohammad Al-Jarrah. Word-based encryption algorithm using dictionary indexing with variable en-

cryption key length. *International Journal of Electrical and Computer Engineering (IJECE)*, *12(1)*: 669-683, 2022.

[34] Xing Zhang, Min Yang, Jian Chen, Tianning Li, and Changda Wang. RAB: A lightweight block cipher algorithm with variable key length. *Peer-to-Peer Networking and Applications*, *1*: 1-16, 2024.

[35] S. Sheikh and T. Banday, "Multi-Recipient E-mail Messages: Privacy Issues and Possible Solutions," *Advances in Electrical and Computer Engineering*, vol. 21, pp. 115–126, 2021.

[36] M. E. Cecchinato, A. Sellen, M. Shokouhi, and G. Smyth, "Finding Email in a Multi-Account, Multi-Device World," *Proc. SIGCHI Conf. Human Factors in Comput. Syst.*, pp. 1200–1210, May 2016, doi: 10.1145/2858036.2858473.

[37] Q. Xu, L. Zhang, and C. Tan, "Blockchain-based secure PGP key management for decentralized email encryption," *Int. J. Inf. Secur.*, vol. 19, pp. 403–417, 2020.

[38] Q. Xu, C. Tang, and J. Wang, "Multi-Receiver Authentication Scheme for General Access Structure," *IEEE Access*, vol. 8, pp. 21644–21652, Jan 2020.

[39] L. Deng, "Anonymous certificateless multi-receiver encryption scheme for smart community management systems," *Soft Comput.*, vol. 24, no. 1, pp. 281–292, Jan 2020.

[40] M. Fu, X. Gu, D. Wenhao, J. Lin, and H. Wang, "Secure Multi-receiver Communications: Models, Proofs, and Implementation," in *Proc. Int.*

Conf. Algorithms and Architectures for Parallel Processing, Springer, Cham, pp. 689–709, 2019.

[41] M. F. Hinarejos, F. J. Vico, and J. C. González, "A solution for secure certified electronic mail using blockchain as a secure message board," *IEEE Access*, vol. 8, pp. 123456–123467, 2020.

[42] O. Chamadoira, M. Rodríguez, I. Rodríguez, P. Fraga-Lamas, and T. M. Fernández-Caramés, "A blockchain-based secure email system," *IEEE Access*, vol. 8, pp. 8765–8780, 2020.

[43] C. Piedrahita, L. J. García Villalba, and E. Fernández-Medina, "A secure email architecture based on blockchain," *IEEE Access*, vol. 9, pp. 112233–112245, 2021.

[44] M. Yakubov, A. Dorri, S. S. Kanhere, and R. Jurdak, "A blockchain-based PKI management framework," *IEEE Access*, vol. 8, pp. 221560–221572, 2020.

[45] Y. Khalifa, F. El Jamiy, A. Barnawi, and M. Imran, "Secure and scalable email distribution using blockchain in permissioned ledgers," *IEEE Access*, vol. 12, pp. 14567–14579, 2024.

[46] R. Vadhava, A. Kumar, and S. Patel, "A blockchain-enabled secure group communication system for emails," *IEEE Access*, vol. 11, pp. 99876–99888, 2023.

[47] Y. Bao, L. Jiang, and Q. Liu, "Blockchain-based secure and efficient data sharing for IoT email services," *IEEE Access*, vol. 8, pp. 121212–121223, 2020.

[48] L. Yang, H. Zhang, and M. Li, "Certificateless ECC-based email encryption using blockchain for multi-recipient communication," *IEEE Access*, vol. 11, pp. 45678–45690, 2023.

[49] P. R. Zimmermann, *The Official PGP User's Guide*. Cambridge, MA, USA: MIT Press, 1995.

[50] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: `https://bitcoin.org/bitcoin.pdf`

[51] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "An overview of blockchain technology: Architecture, consensus, and future trends," in *Proc. IEEE Int. Congress on Big Data (BigData Congress)*, Honolulu, HI, USA, Jun. 2017, pp. 557–564.

[52] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," 2014. [Online]. Available: `https://ethereum.org/en/whitepaper/`

[53] A. Ghorbani, A. Afzali-Kusha, and Z. Navabi, "Secure email communication via blockchain and hybrid cryptography," *IEEE Access*, vol. 9, pp. 115714–115729, 2021.

[54] Zeli Wang, Hai Jin, Weiqi Dai, Kim-Kwang R. Choo, and Deqing Zou. Ethereum smart contract security research: survey and future research opportunities. *Frontiers of Computer Science*, 15(2):1-18, 2020.

[55] Shafaq N. Khan, Faiza Loukil, Chirine Ghedira-Guegan, Elhadj Benkhelifa, and Anoud Bani-Hani. Blockchain smart contracts: Applica-

tions, challenges, and future trends. *Peer-to-Peer Network Application*, 14(5):2901-2905, 2021.

[56] Jose C. Gonzalez, Vicente G. Diaz, Edward R. N. Valdez, Alberto Gomez, and Ruben G. Crespo. Replacing email protocols with blockchain-based smart contracts. *Cluster Computing*, 23(1):1795-1801, 2020.

[57] Diego Piedrahita, Javier Bermejo, and Francisco Machío. A Secure Email Solution Based on Blockchain. *Blockchain and Applications*, 320(1):355-358, 2021.

[58] Vasileios Dimitriadis, Leandros Maglaras, Nineta Polemi, Ioanna Kantzavelou, and Nick Ayres. Uncuffed: A Blockchain-based Secure Messaging System. *In Proceedings of the Pan-Hellenic Conference on Informatics (PCI)*, volume 25, pages 340-345, 2021.

[59] Hsiao-Shan Huang, Tian-Sheuan Chang, and Jhih-Yi Wu. A Secure File Sharing System Based on IPFS and Blockchain. *In Proceedings of the 2nd International Electronics Communication Conferenc*, volume 20, pages 96-100, 2020.

[60] Dindayal Mahto and Dilip Kumar Yadav. RSA and ECC: A Comparative Analysis. *International Journal of Applied Engineering Research*, *12(19)*: 9053-9061, 2017.

[61] Faiza Benmenzer and Rachid Beghdad. Combining Elliptic Curve Cryptography and Blockchain Technology to Secure Data Storage in Cloud Environments. *International Journal of Information Security and Privacy*, *16(1)*: 1-20, 2021.

[62] Rajat Verma, Namrata Dhanda, and Vishal Nagar. Application of Truffle Suite in a Blockchain Environment. *Proceedings of Third International Conference on Computing, Communications, and Cyber-Security*, pages 693-702. Springer, 2023.

[63] A. Saini, Q. Zhu, Y. Xiang, L. Gao, and Y. Zhang, "A smart-contract-based access control framework for cloud smart healthcare system," *IEEE Internet of Things Journal*, vol. 8, pp. 5914–5925, Apr. 2021.

[64] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *Proc. IEEE Symp. Security and Privacy (SP)*, San Jose, CA, USA, pp. 839–858, May 2016.

[65] G. Zyskind, O. Nathan, and A. Pentland, "Decentralizing privacy: Using blockchain to protect personal data," in *Proc. IEEE Security and Privacy Workshops (SPW)*, San Jose, CA, USA, pp. 180–184, May 2015.

[66] ZoKrates Team, "ZoKrates: A toolbox for zkSNARKs on Ethereum," [Online]. Available: `https://zokrates.github.io/`

[67] S. Zhang, X. Xie, J. Wang, and F. Li, "zk-rollups: Scaling and privacy on public blockchains," *arXiv preprint arXiv:2001.03095*, Jan. 2020.

[68] Alexander Yakubov, Wazen Shbair, Nida Khan, and Radu State. BlockPGP: A Blockchain-based framework for PGP Key Servers. *International Journal of Networking and Computing*, 10(1):1-24, 2020.

[69] E. Altulaihan, A. Alismail, M. M. H. Rahman, and A. A. Ibrahim, "Email security issues, tools, and techniques used in investigation," *Sustainability*, vol. 15, no. 13, pp. 1–28, Jul. 2023.

[70] J. Benaloh, "RSA and beyond: Public-key cryptography in the real world," in *Proc. IEEE Symp. Security and Privacy (SP)*, Oakland, CA, USA, 2006.

[71] J. Lopez and R. Dahab, "An overview of elliptic curve cryptography," *University of Campinas*, Technical Report, 2010.

[72] Md. B. Hossain, M. Rahayu, Md. A. Ali, S. Huda, Y. Kodera, and Y. Nogami, "Case-driven analysis of ECC key length variations in secure key exchange," submitted to *IEEE Access*, 2024.

[73] Markus Schäffer, Monika Di Angelo and Gernot Salzer. Performance and Scalability of Private Ethereum Blockchains. *International Conference on Business Process Management*, pages 103-118, 2019.

[74] C.-I. Fan, P.-J. Tsai, J.-J. Huang, and W.-T. Chen, "Anonymous Multi-receiver Certificate-Based Encryption," *Proc. Int. Conf. Cyber-Enabled Distributed Comput. Knowl. Discovery (CyberC)*, pp. 19–26, Oct. 2013.

[75] D. Hinarejos, D. Megías, and A. M. Ferrer, "Blockchain-based key distribution and privacy-preserving email system," in *Proc. IEEE Intl. Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 1179–1186, 2020.

[76] X. Yang, M. Xie, Q. Liu, and H. Guo, "Certificateless multi-recipient encryption scheme for blockchain-based group email systems," *IEEE Access*, vol. 11, pp. 47222–47235, 2023.

[77] A. Shamir, "Identity-based cryptosystems and signature schemes," in *Advances in Cryptology*, vol. 196, G. R. Blakley and D. Chaum, Eds. Springer, 1985, pp. 47–53.

[78] R. Xu, L. Zhang, Z. Gao, Z. Qin, L. Shao, and Z. Zhang, "Blockchain-enabled efficient certificate issuance and revocation for secure vehicular communications," *IEEE Transactions on Vehicular Technology*, vol. 70, no. 5, pp. 4094–4106, 2021.

[79] B. K. Panigrahi, S. Mandal, and A. K. Samantaray, "DPoS-based blockchain framework for distributed PKI and trust management in IoT," *IEEE Access*, vol. 10, pp. 44860–44872, 2022.

[80] X. Li, W. Zhang, X. Chen, and Y. Han, "Blockchain-based efficient group key distribution for IoT security," *IEEE Internet of Things Journal*, vol. 9, no. 13, pp. 10758–10770, 2022.

[81] X. Wu, W. Zhan, J. Li, and H. Xu, "CRT-based secret key sharing mechanism on blockchain," *Computers, Materials & Continua*, vol. 78, pp. 1563–1578, 2024.

[82] R. Halder, A. Dey, M. Alazab, and G. Srivastava, "Secure web of trust with blockchain-based PKI for digital certificates," *IEEE Internet of Things Journal*, early access, 2024.

[83] S. Ahmed, S. W. Kim, and J.-H. Cho, "Decentralized access control with blockchain-based key management for cloud storage security," *IEEE Access*, vol. 11, pp. 32456–32470, 2023.

[84] J. Ni, G. Fang, Y. Zhao, J. Ren, L. Chen, and Y. Ren, "Distributed group key management based on blockchain," *Electronics*, vol. 13, p. 2216, 2024.

[85] S. T. H. Rizvi, Z. N. Shukur, and R. A. Saeed, "An ECC-based secure key sharing scheme for IoT using blockchain," *Sensors*, vol. 22, p. 5722, 2022.

[86] N. Patel and K. Kotecha, "Decentralized PGP key management using blockchain for secure email systems," *Symmetry*, vol. 15, p. 981, 2023.

[87] A. N. Shakor, K. N. Qureshi, and J. A. Alzubi, "A secure and dynamic AES key management framework using ECC and blockchain for cloud environments," *Future Internet*, vol. 16, p. 146, 2024.

[88] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts (SoK)," in *Proc. Int. Conf. Principles of Security and Trust*, Uppsala, Sweden, 2017, pp. 164–186.