PROTECTING DATA PRIVACY WITH DECENTRALIZED SELF-EMERGING DATA RELEASE SYSTEMS

by

Chao Li

B.S., Dalian University of Technology, China, 2012

B.S., University of Edinburgh, UK, 2012

M.S., Imperial College London, UK, 2013

Submitted to the Graduate Faculty of

the School of Computing and Information in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2019

UNIVERSITY OF PITTSBURGH

SCHOOL OF COMPUTING AND INFORMATION

This dissertation was presented

by

Chao Li

It was defended on

April 10, 2019

and approved by

PROTECTING DATA PRIVACY WITH DECENTRALIZED SELF-EMERGING DATA RELEASE SYSTEMS

Chao Li, PhD

University of Pittsburgh, 2019

In the age of Big Data, releasing private data at a future point in time is critical for various applications. Such self-emerging data release requires the data to be protected until a prescribed data release time and be automatically released to the target recipient at the release time. While straight-forward centralized approaches such as cloud storage services may provide a simple way to implement self-emerging data release, unfortunately, they are limited to a single point of trust and involves a single point of control.

This dissertation proposes new decentralized designs of self-emerging data release systems using large-scale peer-to-peer (P2P) networks as the underlying infrastructure to eliminate a single point of trust or control. The first part of the dissertation presents the design of decentralized self-emerging data release systems using two different P2P network infrastructures, namely Distributed Hash Table (DHT) and blockchain. The second part of this dissertation proposes new mechanisms for supporting two key functionalities of self-emerging data release, namely (i) enabling the release of self-emerging data to blockchain-based smart contracts for facilitating a wide range of decentralized applications and (ii) supporting a cost-effective gradual release of self-emerging data in the decentralized infrastructure. We believe that the outcome of this dissertation would contribute to the development of decentralized security primitives and protocols in the context of timed release of private data.

Keywords: data privacy, decentralization, timed release, blockchain, DHT, smart contract.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

I would like to express my gratitude to many people who have been playing indispensable roles during the entire journey of my PhD study.

Firstly and foremost, I am deeply grateful to my advisor Prof. Balaji Palanisamy. During the past five years, he has been patiently mentoring, directing and supporting my research work. He has set an example of excellence as a researcher, mentor and instructor.

I would also like to appreciate the help from my committee members, Prof. James Joshi, Prof. Prashant Krishnamurthy and Prof. Wei Gao, for their insightful suggestions and valuable guidance through my dissertation study.

Great thanks also goes to my fellow colleagues and my friends in Pittsburgh, for all the happy times in and out of the lab during the past five years.

Finally, I would like to give my endless gratitude to my parents and my fiancee, whose love and support have always been the greatest inspiration for me in my pursuit for betterment.

1.0 INTRODUCTION

In the age of Big Data, releasing private data at a future point in time is critical for various applications. Such private data release requires the data to be protected until a prescribed data release time and be automatically released to the target recipient at the release time. The hidden private data appears to emerge by itself to the data recipient at the release time without any assistance from the data sender, and thus has been referred to as self-emerging data [81, 82, 83, 84].

Such self-emerging data is widely found in practice. Examples include secure auction systems (bidding information needs protection until all bids arrive), copyrights-aware data publishing (data is automatically released when the copyright expires), secure voting mechanisms (votes are not allowed to be accessed until the end of the polling process), and post/tweet scheduler (schedule content to automatically post at optimal times). In the above examples, self-emerging data is released in an all-or-nothing manner, indicating that the complete information carried by the hidden data is revealed at a single release time. Self-emerging data may also get gradually released through multiple release times, allowing the carried information to be gradually revealed over time. Examples include data of individuals with privacy requirements that relax over time [77]. For instance, personal data of individuals (e.g., location trajectory patterns, shopping patterns, travel history) collected during their lifetime may be sensitive during the childhood and youth life of an individual, however, the sensitivity of such data may decrease as the individual ages and may drop significantly after the end of the individuals life and a few decades after the end of the individuals life.

Centralized storage systems such as cloud storage services [1, 7, 8] may provide a simple and straight-forward approach for implementing self-emerging data release. The storage service provider may simply keep the sensitive data until the prescribed release time and make it available at the release time. However, such a centralized approach significantly limits the data protection to a single point of trust and a single point of control. Even in cases when the service providers are trustworthy, such centralized models lead to channels of attacks beyond the control of service providers for an adversary to breach the security and privacy of the data. It includes insider attacks [34, 98], external attacks on the centralized data infrastructure, malware and large-scale denial-of-service attacks [3, 10]. In 2014, 28% of the respondents of the US State of Cybercrime Survey [34] reported being victims of insider attacks and 32% reported that insider attacks were more damaging than attacks performed by outsiders.

One possible method for reducing risks encountered by the use of a single centralized service provider is to replace the single service provider with multiple service providers and apply techniques such as secret sharing scheme [114] to make the multiple service providers jointly implement self-emerging data release. However, as long as the identities of the service providers are public, adversaries may still easily locate the service providers and try to compromise them through various types of attacks. As a result, the attack resilience of using multiple service providers highly depends on the number of involved service providers, while an increased number of service providers usually also make the cost of running the service get higher.

Motivated by the aforementioned discussion, this dissertation proposes new decentralized designs of self-emerging data release systems. Inspired by BitTorrent [25] and Tor [123], we observe that the properties of large-scale peer-to-peer (P2P) networks help resolve the challenges encountered by the previous methods. In a P2P network, each peer is able to work as a service provider of releasing self-emerging data. Then, the service providers of a specific service request could be a small group of randomly selected peers, whose identities are hard to be disclosed due to the pseudonymity and anonymity of P2P networks. Thus, attacks targeting the service providers of a specific service request, as launched in the previous methods where the identities of service providers are public, can hardly be successfully carried out in the context of P2P networks because locating these service providers is extremely difficult considering the size of large-scale P2P networks.

In this dissertation, we study the use of two well-known types of P2P networks, namely

Distributed Hash Table (DHT) [91, 119] and blockchain [100, 128], as the underlying infrastructure of designing self-emerging data release systems.

The dissertation starts from investigating approaches to implement self-emerging data release using P2P networks running DHT protocols. The classical DHT protocols such as Chord [119] and Kademlia [91] have been widely used to enable efficient lookup among peers of a P2P network. The protocols require each peer to maintain links to a few neighbors (maximum $O(\log\ n)$ neighbors in a network of size n), so a message corresponding to a given key can be routed within $O(\log\ n)$ hops from any peer in the network to a peer closest to the key. With such protocols, self-emerging data can get secretly routed from the sender to the recipient along a pre-determined pseudo-random routing path so that the data can be recovered exactly at the release time by the recipient. However, we observe that the use of a DHT infrastructure may lead to new challenges that are specific to P2P networks, including churn (i.e. nodes join and leave the P2P network) and new attacks relevant to the Sybil attack [48]. To resolve these challenges under the assumption that DHT peers may behave either honestly or maliciously, we propose to split self-emerging data into multiple fragments using erasure coding [126] and leverage the increased redundancy to make the data resist against the aforementioned threats.

After studying the use of a DHT infrastructure, the dissertation employs a blockchain infrastructure [128] that offers more robust and attractive features including decentralized democratized trust and native cryptocurrencies with monetary value. A blockchain is a distributed ledger, which publicly records data as a chain of blocks with each block containing the hash of its previous block. To falsify data in a blockchain, adversaries must hold enormous resources (e.g., computation power in Proof-of-Work consensus protocol [100], amount of stake in Proof-of-Stake consensus protocol [74]) that can compete with the sum of resources owned by the rest peers in the network. The great difficulty in successfully launching such attacks solidifies peers' confidence that the data recorded in a blockchain is verifiable and permanent, thus creating a decentralized trust among the mutually distrusting peers. This decentralized trust, fueled by the monetary value of native cryptocurrencies, provides new possibilities of resolving the challenges of churn and attacks in the new context of blockchain infrastructure. Instead of requiring the majority of peers to perform honestly as assumed in the context of DHT infrastructure, the blockchain infrastructure allows us to assume that all the peers are adversaries with rationality [47, 61, 62, 103], which is more in line with practical market rules. The protocols of implementing self-emerging data release in the blockchain infrastructure can then be programmed as an enforceable smart contract [128] through which each peer serving for a specific service request needs to pay a certain amount of cryptocurrency as the security deposit. In this way, we are able to make rational peers always choose to honestly comply with the protocols, instead of tending to perform any misbehavior violating the protocols, as such misbehaviors will make their security deposit get confiscated.

Having demonstrated feasibility of releasing self-emerging data in the blockchain infrastructure, our further research about the blockchain infrastructure shows that it is possible to release self-emerging data not only to the accounts belonging to peers, but also to the accounts belonging to smart contracts. The rapid development of blockchain techniques has made smart contracts get widely adopted as the back-end logic for running the emerging decentralized applications (DAPP) such as decentralized voting and bidding systems [2, 92]. A mechanism that supports releasing self-emerging data to smart contracts can thus facilitate a wide range of decentralized applications, allowing users to schedule their target smart contracts to be automatically executed at future points of time, without revealing their private input data (i.e., self-emerging data) before the expected execution time. However, unlike accounts belonging to peers, the smart contract accounts are passive and transparent, making the previous solutions not applicable. To overcome this difficulty, we design a new mechanism that makes self-emerging data get released to a proxy contract deployed by the user, which then automatically calls the target smart contract on behalf of the user. The mechanism jointly applies techniques of data redundancy and cryptocurrency-driven enforcement and can handle rational adversaries and malicious adversaries altogether.

Finally, having explored the system design using different types of P2P network infrastructures and the release of self-emerging data to different types of recipients, the dissertation investigates the possible options of inputting the self-emerging data to the designed system under different circumstances. Specifically, we find that in the circumstance of gradually releasing self-merging data through multiple release times, for the purpose of reducing the cost of maintaining multiple snapshots of large-size private data, it would be more desirable to perturb the data in a reversible way using perturbation keys so that a single maintained snapshot is capable of revealing multiple levels of information at multiple prescribed release times. Unfortunately, existing privacy-preserving data perturbation mechanisms [38, 49, 54, 71, 73, 85] do not meet our requirements because the randomness involved in these mechanisms makes the perturbed data hard to be recovered. Therefore, we propose a new set of mechanisms that applies perturbation keys as the seeds of a generator of pseudo-randomness and replaces any randomness involved in the conventional data perturbation mechanisms with such pseudo-randomness so that the same keys, upon being released by the designed system, could be used by the recipients to reverse the perturbation process and reduce the perturbation level. We demonstrate the effectiveness of the proposed mechanisms in two representative scenarios of gradually releasing self-emerging data, namely association data disclosure and location data disclosure.

In the rest of this chapter, we first outline the key research tasks of this dissertation and then briefly present the organization of the rest chapters.

1.1 OVERVIEW OF RESEARCH TASKS

This dissertation includes three research components and has four research tasks. The three research components are shown in Figure 1 as *Infrastructure*, *Output* and *Input*.

Infrastructure: The *Infrastructure* research component refers to the study of using different types of large-scale P2P networks as the underlying infrastructure of designing decentralized self-emerging data release systems. Depending on the features of the underlying P2P network infrastructure, the system design may encounter different challenges and require different solutions. Both DHT and blockchain have been widely used for establishing P2P networks with scale, geographic distribution, and decentralization. For example, the Vuze DHT network contains over 1 million nodes [56] and the Ethereum blockchain network consists of more than 10,000 nodes distributed all over the world [4]. However, DHT and blockchain involve several key similarities and differences:

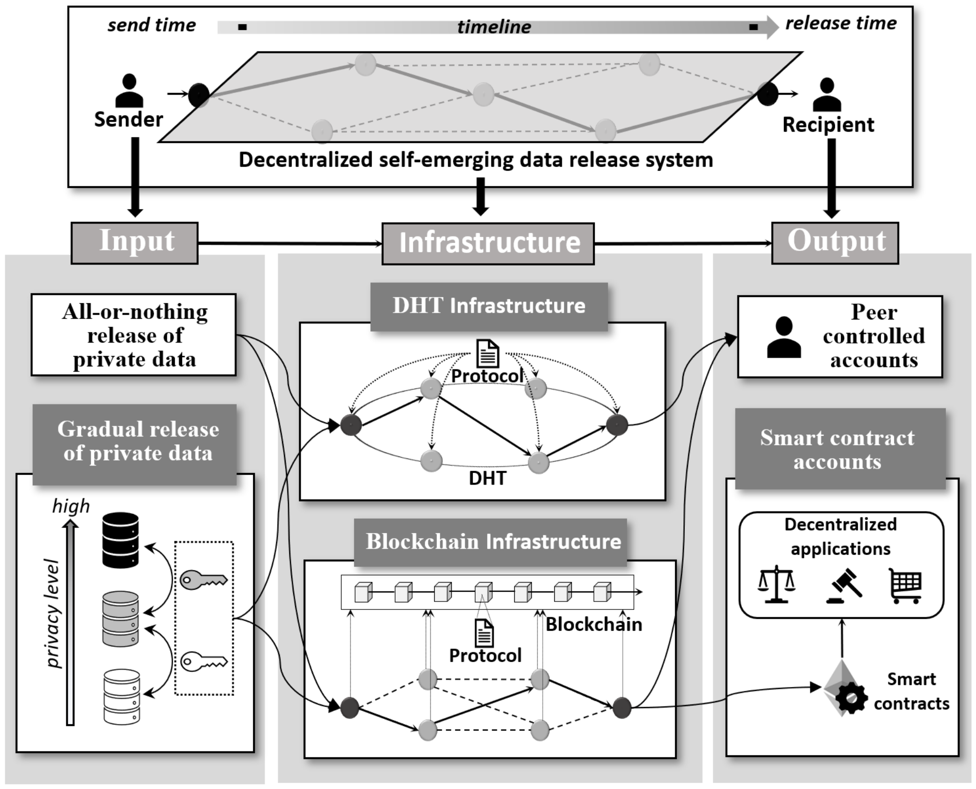


Figure 1: An overview of research tasks

- *Design objectives*: In short, DHT is a lookup service designed for enabling efficient communication among nodes in a P2P network while blockchain is a distributed ledger managed by the entire P2P network to create a distributed trust. For more technical details about DHT and blockchain, please refer to Appendix A and Appendix B, respectively.
- *Storage*: Both DHT nodes and blockchain nodes can store data locally. However, data stored in the distributed ledger of blockchain is publicly revealed to all blockchain nodes. Therefore, special attention needs to be taken to avoid storing private data publicly on a blockchain.
- *Communication*: Both DHT nodes and blockchain nodes can build private channels with other nodes. However, any data associated with the blockchain is publicly revealed to all blockchain nodes. Examples include inputs and outputs of calling a function within a smart contract. Therefore, to leverage the decentralized trust offered by blockchain, we need to pay attention to any direct or indirect interaction with the blockchain, even if we don't intend to store data on it.

- *Cryptocurrency*: Unlike DHT networks, blockchain networks such as Bitcoin [100] and Ethereum [128] offer native cryptocurrencies (i.e., Bitcoin and Ether) that can be leveraged as monetary incentives and security deposits in protocol design. Because of this, protocols designed for the DHT-based decentralized self-emerging data release system mainly rely on increased data redundancy to prevent the hidden data from being stolen by adversaries before the prescribed release time, whereas in the blockchain networks, we can additionally leverage the enforcement driven by the monetary incentive and penalty to assist the protocol design and system development.

After analyzing the *Infrastructure* research component, we have figured out that both the DHT infrastructure and the blockchain infrastructure are qualified P2P network infrastructure of designing decentralized self-emerging data release systems, so we set the research task ***T-1*** and ***T-2*** to investigate the two options, respectively.

Output: The *Output* research component refers to the study of releasing self-emerging data to different types of recipients. Protocols designed for releasing self-emerging data to a certain type of recipients may not be applicable to other types of recipients. The DHT infrastructure only involves a single type of recipients while the blockchain infrastructure involves two types of recipients. In the DHT infrastructure, recipients simply refer to the peers of the DHT network, who need to control DHT nodes to actively collect the self-emerging data at the release time. However, in the blockchain infrastructure supporting smart contracts (e.g., Ethereum), it is possible to release self-emerging data to two types of recipients that run External Owned Account (EOA) or Contract Account (CA), respectively. To be brief, an EOA account is controlled by a peer of the blockchain network through a pair of public/private keys while a CA account is controlled by a smart contract. Recently, smart contracts have been widely used for running the back-end logic for the emerging decentralized applications (DAPP). Supporting the release of self-emerging data to smart contracts can thus facilitate a wide range of decentralized applications. There are two main differences between EOA and CA accounts that are relevant to the design of decentralized self-emerging data release system:

- *Transparency*: A peer controlling an EOA account in a blockchain network has similar abilities of a peer controlling a DHT node in a DHT network, meaning that the peer can keep the received self-emerging data in secret by obtaining it from private channels and storing it locally. In contrast, a smart contract controlling a CA account is a piece of transparent program code recorded in the blockchain, meaning that the received self-emerging data must also be recorded in the blockchain and thus get publicly revealed to the entire network.

- *Passiveness*: A peer controlling an EOA account can actively participate in the self-emerging data release protocol (e.g., initiate a conversation with other accounts) while a smart contract controlling a CA account can only passively wait for transactions sent by other accounts to invoke its inside code (i.e., listen to incoming transactions).

After exploring the *output* research component, we have figured out the differences of releasing self-emerging data to different types of recipients and the need of new approaches that support releasing self-emerging data to the smart contract accounts. Therefore, we set the research task ***T-3*** to resolve this problem.

Input: The *Input* research component refers to the study of the possible options of inputting the self-emerging data to the designed system. Under different circumstances, the self-emerging data may be expected to be released in a single time (*all-or-nothing release*) or multiple times (*gradual release*). Generally speaking, there are three options to input self-emerging data to the designed system:

- *Plaintext*: As the basic option, the sender of private data can directly input the plaintext of private data to the system. Since the data needs to be stored by the P2P nodes and also routed among the nodes, such a straightforward solution may result in a high cost of storing and transferring large-size private data (e.g., a healthcare dataset), which in turn affects both the security and scalability of the self-emerging data release system.
- *Encryption key*: To eliminate the drawbacks of using plaintext, one option is to encrypt the private data with a key and only input the encryption key to the system. Such an encryption-based scheme can be adopted in both *all-or-nothing release* and *gradual release*. Specifically, in the *all-or-nothing release*, private data only needs to be released for once, so only a single snapshot of the encrypted data needs to be maintained by either

data sender or recipient to make the data available at the release time. However, in the *gradual release*, to gradually increase the utility of the disclosed information over time, multiple snapshots of the encrypted data needs to be maintained with each snapshot corresponding to a different level of utility, which may result in a high cost of storing and maintaining large-size private data.

- *Perturbation key*: The last option is to perturb the private data with privacy-preserving data perturbation mechanisms [38, 49, 54, 71, 73, 85] to change the utility level of the information carried by the perturbed data. Since the perturbation level can be gradually decreased to increase the utility level over time, the perturbation-based solution seems to be a proper way of overcoming the issues of high storage/maintenance cost in the encryption-based solution. Unfortunately, existing privacy-preserving data perturbation mechanisms do not meet the requirements of designing decentralized self-emerging data release system for two reasons: (1) data perturbed through existing schemes cannot be de-perturbed to recover data utility; (2) existing perturbation schemes are not designed to be implemented through keys.

After analyzing the *Input* research component, we have seen the lack of proper solutions for supporting the gradual release of self-emerging data with low cost, so we determine to develop such solutions in the last research task ***T-4***.

In summary, we make the following contributions in this dissertation:

- First, we propose new designs of self-emerging data release using two different types of large-scale P2P networks as the underlying infrastructure.
- Second, we propose and develop solutions for supporting the release of self-emerging data to different types of recipients in P2P networks that facilitate both traditional centralized applications and emerging decentralized applications.
- Finally, we propose and design cost-effective techniques for gradually releasing self-emerging data by using perturbation keys as inputs. We develop a suite of techniques for supporting cost-effective gradual release of two representative types of private data, namely association data and location data.

1.2 CHAPTERS OVERVIEW

The rest of the dissertation is organized as follow: Chapter 2 provides literature review. Then, in Chapter 3 and Chapter 4, the DHT infrastructure and blockchain infrastructure are investigated, respectively. After that, in Chapter 5 and Chapter 6, we discuss the techniques for supporting the release of self-emerging data to smart contracts and supporting cost-effective gradual release of self-emerging data, respectively. Finally, we conclude and present future directions in Chapter 7.

2.0 LITERATURE REVIEW

In this chapter, we provide the literature review. Specifically, in Section 2.1, we provide a review of the literature about timed-release of private data to demonstrate the need for decentralized solutions of releasing private data in future. Then, in Section 2.2, we review recent work of enforcing behaviors performed by the participants of secure multi-party computation protocols using cryptocurrencies, which inspires the use of cryptocurrency-driven enforcement in our design of blockchain-based self-emerging data release system. Finally, in Section 2.3, we review the representative privacy-preserving data perturbation mechanisms to illustrate that the existing techniques fail to meet the requirements of gradually releasing self-emerging data with low cost.

2.1 TIMED-RELEASE OF PRIVATE DATA

The problem of revealing private data only after a certain time in the future has been researched for more than two decades. The problem was first described by May as timed-release encryption (TRE) in 1992 [90] and has intrigued many researchers in the field of cryptography since then. Existing work on this topic can be divided into three categories, namely *time-lock puzzle*, *time server* and *reference time clock*.

Time-lock puzzle: The first category of existing solutions was designed to make data recipients solve a mathematical puzzle, called time-lock puzzle, before reading the messages [24, 28, 109]. The time-lock puzzle can only be solved with sequential operations, thus making multiple computers no better than a single computer. In addition, since the time-lock puzzle scheme requires no third party, there is no single point of trust problem. However, the time-lock puzzle scheme suffers from two key drawbacks. First, due to increasing advancements in computing hardware and hardware performance, the time taken by such puzzle computation is not determinate and hence these solutions cannot tackle the situations that demand the data be released with a precise release time. Second, the puzzle computation is associated with a significant computation cost. Incurring such high computation costs for a large big data infrastructure does not lead to a scalable cost-effective solution.

Time server: The second category of existing solutions relies on a third party, also known as a time server, to release the protected information at the release time in future. The information, sometimes called time trapdoors, can be used by recipients to decrypt the encrypted message [26, 35, 40, 72, 76, 97, 109]. Initially, the third party was designed to actively interact with data sources and data recipients to complete the process [44, 90, 96, 109]. Although the interactive third party scheme can release data at an accurate release time, the known identities of the data sources and recipients may cause security issues. Because of this, researchers have focused on developing timed-release encryption (TRE) based on non-interactive time server. In 2003, Mont *et al.*[97] proposed the non-interactive TRE model based on quadratic residues (QR-TRE). However, the confidentiality of private data sending through their system highly relied on the trustworthiness of the time server because the time server can decrypt the data before the release time. In 2005, Chan and Blake [26] proposed a scalable, server-passive, user-anonymous TRE scheme based on bilinear pairing, which only asked the time server to be curious. Based on their work, the formal model of TRE was proposed by Cathalo *et al.* [32]. In 2007, more efficient TRE schemes were proposed [68, 35], which significantly reduced the operation cost and also allowed replacing a single time server with multiple servers. In 2008, the non-interactive TRE was formally defined in [40]. After that, researchers focused on developing variations of the standard TRE model to extend its range of application. For example, the TRE model with pre-open capability was proposed in [99, 76], which allowed the encrypted information to be decrypted before the release time in some emergent situations. The TRE model supporting one-to-many service was proposed in [50], which allowed the encrypted information to be decrypted by multiple recipients at the release time. Besides, in standard TRE, the single time trapdoor sent by the time server may be easily lost or missed by the recipient. To solve this, time-specific encryption (TSE) [106, 72], as an extension of TRE, was proposed to split time to slices so that any time trapdoor released during one time slice can be used to decrypt the information. Although the efficiency and flexibility of time-server-based approaches have been constantly improved, the time server in this model has to be trusted to not collude with recipients so that encrypted messages cannot be entered before release time. This restriction makes this set of solutions involve a single point of trust.

Reference time clock: The third category of existing solutions uses blockchain [100]. The difficulty of PoW (proof-of-work) can be diversely adjusted to change the average generation time of each block to the desired value, which makes blockchain to be a reference time clock with correctness guaranteed by the distributed network. Therefore, by combining witness encryption [55] with blockchain [69, 87], one can leverage the computation power of PoW in blockchain to decrypt a message after a certain number of new blocks have been generated. However, the current implementation of witness encryption is far from practical, which requires an astronomical decryption time estimated to be 2^{100} seconds [87].

To sum up, there is a need for a scalable and cost-effective solution for releasing private data to future, which should not involve a single point of trust. This dissertation aims at filling this gap by designing decentralized self-emerging data release systems using the P2P network infrastructure.

2.2 CRYPTOCURRENCY-DRIVEN ENFORCEMENT

The idea of using cryptocurrency to enforce participants of a protocol to perform desirable behaviors was first proposed in 2014 by Andrychowicz *et al.* [16], who designed a timed commitment protocol using Bitcoin [100] for the purpose of resolving the fair Secure Multi-party Computation (SMC) problem more efficiently. After that, extensive follow-ups [15, 20, 94] have further improved the efficiency of resolving fair SMC problems with blockchain and the effectiveness of the cryptocurrency-driven enforcement in protocol design has been widely recognized. Since the blockchain-based systems designed in Chapter 4 and Chapter 5 of this dissertation mainly rely on the cryptocurrency-driven enforcement inspired by the previous research of the blockchain-based SMC to make the designed systems resist against possible misbehaviors violating the protocols, in this subsection, we review the existing work relevant to the blockchain-based SMC. Before going to the details, we first present the differences between the blockchain-based SMC and the traditional SMC.

The traditional SMC, originally proposed by Yao [131] and Goldreich *et al.* [60], allows multiple mutually distrusting parties to obtain the output of a function using their private data as function input without needing a trusted third party. According to [104], the computation is performed in a way that 1) the output is correct and 2) cheating parties will not be able to learn any information about the honest parties' inputs. Unlike the traditional SMC, the blockchain-based SMC allows the inputs to be revealed and focuses more on the fairness among the parties [16, 20, 15, 94]. For example, in the coin tossing problem [27, 16] where two parties (say Alice and Bob) want to jointly generate a value that has equal probability to be 0 or 1 (i.e., a random bit), Alice and Bob can send a randomly selected bit (b_A and b_B) to each other, so that the output of function $b = b_A \oplus b_B$ will become a shared random bit when at least one of b_A and b_B is random. To make the above protocol fair, b_A and b_B should be received simultaneously. Otherwise, in case that Bob receives b_A first, he can select b_B based on the value of b_A to get desired b. The solution for resolving such fairness problem is called commitment-based SMC protocol [27], which usually consists of two phases, namely *Commit* and *Open*. During the *Commit* phase, each participant should make the hash of a secret $h(s)$ public while keeping the secret s unknown to other participants. Then, during the *Open* phase, each participant should disclose the secret s. One fundamental limitation of the commitment-based SMC protocol is the lack of enforcement. In the coin tossing example, Alice and Bob may make a bet. Alice agrees to pay Bob 1 USD when $b = 0$ and Bob agrees to pay Alice 1 USD when $b = 1$. However, during the *Open* phase, when Bob sees b_A and computes $b = 1$, he may abort the protocol by rejecting disclosing b_B. Even if Bob discloses b_B and loses the bet, he may still reject to pay 1 USD to Alice. In both these situations, the protocol becomes unfair to Alice.

In 2014, Andrychowicz *et al.* [15] proposed to use Bitcoin [100] to design a timed commitment protocol for the fair lottery, which is actually an implementation of the fair SMC. In brief, during the *Commit* phase, the committer should create a *Commit* transaction with

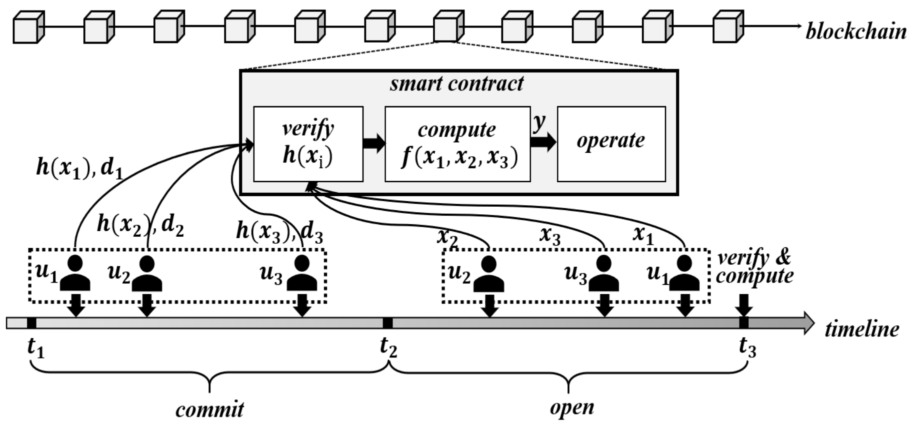


Figure 2: Commitment-based secure computation using Ethereum

both a monetary deposit and the hash of his secret in it. The committer should also send a *PayDeposit* transaction to the recipient, which contains both a timed lock and the signature of the committer. Then, before the end of *Open* phase, the committer can create an *Open* transaction with the secret to get back the deposit, otherwise, after the deadline, the recipient can use the *PayDeposit* transaction to get the deposit paid by the committer. This bitcoin-based protocol forces the committer to disclose the secret for not losing the deposit. It can also force the committer to respect the function result by automatically operating the committer's digital asset (i.e., Bitcoin) based on the function result. Recently, Miller *et al.* [94] proved that commitment-based SMC protocols designed over Ethereum can offer better performance because Ethereum begins with a 'Turing-complete' language for general-purpose use. We show an example of the commitment-based secure computation using Ethereum in Figure 2. In the example, during the *Commit* phase $[t_1, t_2]$, the three users u_1, u_2, u_3 should send hashed secrets and deposits to the smart contract. Then, during the *Open* phase $[t_2, t_3]$, the three users should send their secrets to the smart contract. Finally, at the end of *Open* phase, namely t_3, the smart contract will verify the secrets received during the *Open* phase with the hashed secrets received during the *Commit* phase, compute the function results and execute pre-determined operations based on the computation results. If a secret fails to pass the verification, the corresponding deposit will be confiscated.

2.3 PRIVACY-PRESERVING DATA PERTURBATION

The problem of privacy-preserving data perturbation has been studied extensively in the framework of statistical databases. Samarati and Sweeney [112],[121] introduced the k-anonymity approach which has led to some new techniques and definitions such as l-diversity [88] and t-closeness [85]. There had been some work on anonymizing graph datasets with the goal of publishing statistical information without revealing information of individual records. Backstrom et al. [18] show that in fully censored graphs where identifiers are removed, a large enough known subgraph can be located in the overall graph with high probability. Ghinita et al. present an anonymization scheme for anonymizing sparse high-dimensional data using permutation-based methods [59] by considering that sensitive attributes are rare and at most one sensitive attribute is present in each group. The safe grouping techniques proposed in [22, 42] consider the scenario of retaining graph structure but aim at protecting privacy when labeled graphs are released.

Based on the concept of differential privacy[49], there had been many work focused on publishing sensitive datasets through differential privacy constraints [38, 54, 124]. Differential privacy had also been applied to protecting sensitive information in graph datasets such that the released information does not reveal the presence of a sensitive element [43, 71, 111]. Recent work had focused on publishing graph datasets through differential privacy constraints so that the published graph maintains as many structural properties as possible while providing the required privacy [111].

The problem has also been researched extensively in the area of location data disclosure. The representative location data perturbation schemes includes dummies [75], spatial location cloaking [19, 39, 58, 70, 95, 130] and landmark objects [67]. Also, recent work has studied the location privacy problem by perturbing the location information based on differential privacy constraints prior to disclosure [14, 31].

In the past, based on k-anonymity [122], there have been many work using spatial location cloaking to protect location privacy. *CliqueCloak* algorithm proposed in 2004 considered the individual user's personalized privacy requirement for the first time [57]. A grid-based cloaking framework, *Casper* further extended this model with a privacy-aware query processor [95]. Subsequently, a directed-graph based cloaking algorithm was proposed to improve the success rate of anonymization [130] and the Hilbert Cloak algorithm uses a Hilbert curve to fill the whole area and track users [58]. While these techniques were designed for mobile users traveling on Euclidean space, recent work has considered the location cloaking problem under a constrained road network model [125, 132].

We find that these existing techniques cannot effectively support the gradual release of private data with low cost, so we propose to design a new set of mechanisms that can make private data get gradually released through the decentralized self-emerging data release system in a cost-effective approach. Details about this will be presented in Chapter 6.

3.0 SELF-EMERGING DATA RELEASE USING DISTRIBUTED HASH TABLES

In this chapter, we explore the design of a self-emerging data release system using large-scale Distributed Hash Table (DHT) [119] networks as the underlying infrastructure. The research of this chapter corresponds to the research task *T-1*. Before going to the details, we briefly answer the key research questions of accomplishing this research task.

Research goals: The goal of the system designed in this chapter is to ensure private data to be protected until a release time set by a DHT node (i.e., sender) and also enable the self-emergence of the private data to another DHT node (i.e., recipient) at the release time.

Properties of DHT nodes: In the system designed in this chapter, all participants of the self-emerging data release, including data senders, data recipients and service providers, are peers of a DHT network, who need to run DHT nodes to take any action. For short, DHT allows nodes in a P2P network to efficiently communicate and transfer data with other nodes. A DHT node has the ability to communicate with other nodes through private channels established with basic cryptographic techniques and also the ability to locally store data received from other nodes. Besides, a DHT node has the freedom to choose to join and leave the network at any time.

Adversary models: We note that the design objectives can be significantly challenged by adversaries controlling a sizable proportion of the DHT network. When a sufficient number of DHT nodes has been compromised by an adversary, the adversary can either release the hidden data before the prescribed release time (*release-ahead attack*) or destroy the hidden data altogether (*drop attack*). These two specific attacks in combination with the traditional churn issues [120] in DHTs constitute significant challenges to the design of the system. Thus, ensuring high resilience to churn and to release-ahead attacks and drop attacks is also a central objective of the system design. In this chapter, we divide DHT nodes into two categories, namely honest nodes and malicious nodes. In short, we assume that honest nodes may join and leave the network at any time but they will follow the designed protocols as expected when they are in the network. In addition, data locally stored at honest nodes is unavailable to the adversary. In contrast, malicious nodes are the ones controlled by the adversary, so these nodes can arbitrarily violate the designed protocols and data locally stored at these nodes is known by the adversary.

Technical approaches: The reliability of the self-emerging data release system established over the DHT infrastructure primarily replies on the increased redundancy of the private data. DHT nodes can freely join and leave the P2P network and there is no general and effective approach to enforce behaviours performed by DHT nodes. As a result, requesting a single DHT node to store the private data is not a robust way of hiding private data in the DHT network because the data will get lost once that DHT node leaves the network. Therefore, to make data survive in the highly dynamic DHT network, data needs to be replicated to make the replicas stored at different nodes so that the recipient can still receive the self-emerging data at the release time even if a fraction of the nodes storing the replicas have left the network or have been compromised by adversaries. By jointly considering challenges of churn and attacks, the protocols designed in this chapter leverage erasure coding [126] to split private data into a group of fragments and make the fragments keep moving among DHT nodes without sticking to fixed positions.

Evaluation: We conduct extensive experimental evaluation of the proposed protocol using Overlay Weaver DHT toolkit [115] and the results demonstrate that the proposed erasure-coding-based schemes provide high resilience to both release-ahead and drop attacks as well as to the churn issues in DHT.

In the rest of this chapter, we first provide the system overview and analyze security challenges in Section 3.1. Then, we present the self-emerging data release protocol in detail in Section 3.2. After that, we experimentally evaluate the proposed solution in Section 3.3. Finally, we summarize this chapter in Section 3.4.

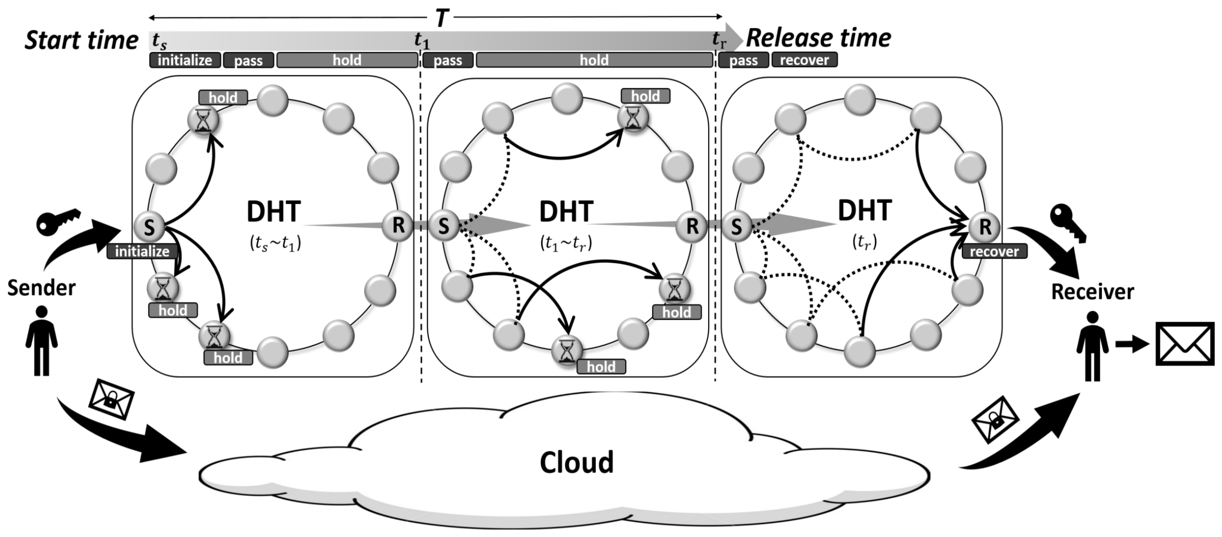


Figure 3: DHT-based decentralized self-emerging data release system

3.1 SYSTEM OVERVIEW

In this section, we first present the DHT-based decentralized self-emerging data release system and then discuss the main security challenges, including the adversary models and churn.

3.1.1 DHT-based decentralized self-emerging data release system

There are four major entities in the DHT-based self-emerging data release system, namely the *data senders*, the *data recipients*, the *DHT network* and the *cloud*, as shown in Figure 3. As the owner of private data, the data senders want to protect the data stored in the cloud from being accessed until a pre-defined data release time. The data, however, needs to be accessible to the recipients after the release time. As discussed in Section 1.1, the data senders may use a secret key to encrypt/perturb their data before uploading to the cloud and use the DHT network to make the secret key 'disappear' before the future data release time. They then need the secret key automatically appears to the recipients at the release time, allowing the recipients to recover the protected data.

If we denote the start time as t_s and the expected future release time as t_r, we can express the entire time period that the data owner wants to make the secret key disappear as T. It is referred to as the emerging time period as the key is emerging from the DHT to the intended recipient during T. At start time t_s, the sender process her data with a secret key and sends the encrypted/perturbed data and the key to the cloud and the DHT network, respectively (shown as 'initialize' in Figure 3). During the emerging time period T, the secret key will be packaged, split into multiple fragments and routed among the nodes in the DHT. Each fragment, after being stored by a node in the DHT for a limited time period (shown as 'hold' in Figure 3), will be sent to the next node and gets routed towards the recipient along a carefully designed path. At the release time t_r, the recipient can collect the fragments from the DHT network to recover the secret key (shown as 'recover'). The recipient can then download the encrypted/perturbed data from the storage cloud and recover the original data using the obtained key. It is easy to see that the DHT network takes the core role in the system. Next, we analyze the challenges and attacks that lead to the compromise of the hidden secret key in the DHT network.

3.1.2 Adversary models

Based on the objective of the adversary, we define two attack models, namely the *release-ahead attack* aiming to extract the key from DHT network before the release time t_r and the *drop attack* aiming to prevent the key from being recovered by the legitimate recipient after t_r. For both the attack models, the adversary needs to control a fraction of DHT nodes which can be realized through Sybil attack [48], Eclipse attack [116] or collusion with other adversaries. We divide the entire DHT nodes into two categories, namely honest nodes always following the designed protocol as expected and malicious nodes controlled by the adversaries. Specifically, we use p to denote the fraction of DHT nodes in the network controlled by all the adversaries and we assume these adversaries can collude without any restriction. In other words, we can imagine a single adversary that owns all the malicious nodes in the network to launch either release ahead attack or drop attack.

In release ahead attack, the adversary, who may be the legitimate recipient, aims to extract the secret key from the DHT network to recover the original private data before the release time t_r. The adversary can collect the fragments of the key from controlled DHT nodes (i.e., malicious nodes) if the packages were ever stored at any malicious node. This type of attack is effective in many scenarios that require timed release of self-emerging data. For instance, in an online exam scenario, if the adversary can obtain the exam questions before other participants, he can gain more time to answer the questions and affect the fairness of the exam.

In drop attack, the adversary aims to prevent the secret key from being received by the legitimate recipient to recover the private data at release time t_r. That is, if any malicious node receives a fragment of the secret key before release time, the malicious node can refuse following the designed protocol to further route that fragment to the recipient. A successful drop attack can make the encrypted/perturbed data become inaccessible even after the release time t_r. In those time-sensitive scenarios, this means the scheduled activity has to be canceled. For example, the online exam cannot be started. In addition, since the key is lost, unless the adversary releases those dropped fragments again, the private data can never be recovered.

3.1.3 Churn impact

The data storage in DHT networks always suffers from the churn issue [120], namely the phenomenon that nodes frequently join and leave the network. The churn impact to our system can be summarized as short-term and long-term impacts. The short-term impact is caused by the transiently left nodes. The nodes leave the network for a short time period, so their ID and responsibilities have not been transferred to other nodes. This may block the routing of fragments of secret key in the network for a short time period but its effect is limited. However, the long-term impact is vital to our system. A node is 'dead' when it leaves the network for a long time and its ID and responsibility in the secret key routing is deprived. Because of the death of the nodes, the fragments of the secret key stored by these nodes are also lost. Even if the fragments are replicated to other nodes, we note that the new nodes also have probability p (the fraction of nodes controlled by the adversary) to be malicious, thus significantly increasing the chance of the adversaries to obtain the fragments and recover the secret key. For example, in the case that a fragment is replicated to three nodes and two of them are dead and then replaced by new nodes during the emerging time period T, the probability for the adversary to get the fragment increases from $1-(1-p)^3$ to $1-(1-p)^5$. Therefore, the conventional replication [126] is not adopted.

Motivated by these challenges, we propose the self-emerging data release protocols for the DHT network to carefully package and route the fragments of secret key during the emerging time period T to handle both the attack and churn.

3.2 SELF-EMERGING DATA RELEASE PROTOCOLS

In this section, we present the proposed self-emerging data release protocols in detail. The protocols consist of three components, namely *routing path construction*, *initial package generation* and *package routing*, to be implemented in a sequential order. Specifically, at start time t_s, the sender enters DHT network as a node. It first locally determines routing path pattern based on the adopted pattern construction scheme and pseudo-randomly select DHT IDs to fill in the pattern. Then, based on the pattern and selected IDs, the sender node locally generates the initial data packages. Finally, it sends the initial packages out and the packages will be routed and processed along the paths to deliver the secret key to the recipient at the release time t_r.

Based on the adopted routing path construction schemes, we propose three protocols and all of them are based on the erasure coding mechanism [126]. As a common mechanism to protect data, erasure coding [126] can divide a data package to m fragments and re-code them into n fragments so that the package can be recovered from any m fragments ($m \leq n$). We start from the *one-hop path pattern* scheme, which applies erasure coding to establish multiple one-hope paths between sender and recipient to guarantee attack resilience. Then, by taking dead nodes (churn) into account, we propose the *adjusted one-hop path pattern* scheme to make it resilient to churn issues by estimating the number of dead nodes and adjusting the parameters of erasure coding based on the estimation. After that, by dividing the entire emerging time period $T = t_s - t_r$ into several shorter time periods, we propose the *multi-hop path pattern* scheme to iteratively implement the erasure coding mechanism to route the fragments of secret key so that the loss of fragments during each shorter time period can be suppressed and the lost fragments can be recovered by multiple usages of erasure coding. To compare the schemes in terms of attack resilience, we measure the release-ahead attack resilience, R_r as the probability that an adversary fails to restore the secret key before the legitimate release time t_r, and drop attack resilience, R_d as the probability that an adversary fails to prevent the secret key from being restored by the recipient at the release time t_r. Specifically, we desire $R_r = R_d$ because we expect that the proposed protocol has both good release-ahead attack resilience and good drop attack resilience without compromising either of them and making the protocol vulnerable. Next, we present details of the three protocols in turn.

3.2.1 One-hop scheme

The one-hop path pattern scheme applies the erasure coding to split the secret key into n fragments and send each of them through a one-hop path to the recipient to allow at most $n - m$ of the fragment transmissions to be unsuccessful. In other words, n holder nodes are applied to store the n fragments for the entire emerging time period T. We name all the nodes selected by the sender to form the path pattern as holder nodes. Each holder node receives fragment(s) from its predecessor(s), holds(stores) the fragments for a time period (in this scheme the entire emerging time period T) and sends the fragments to its successor(s) after that time period.

Routing path construction: To construct the one-hop path pattern, we need to determine the total number of fragments, n and the minimum (threshold) number of fragments to restore the secret key, m. Given the maximum available nodes that can be used to form the pattern (namely the limited recourse), N, we can calculate the value of n and m to maximize the attack resilience through Algorithm 1.

If an adversary controls at least m holder nodes, the secret key will be directly restored at start time and the release-ahead attack is successful. If the adversary controls at least $n - m + 1$ holder nodes, the recipient will fail to receive at least m fragments to restore

```
Algorithm 1: One-hop path pattern
  Input  : Maximum available node number $N$.
  Output: Total fragment number $n$, threshold fragment number $m$.
1 $m = \lfloor \frac{N+1}{2} \rfloor$;
2 $n = 2m - 1$;
```

the secret key at release time t_r, which makes drop attack successful. Therefore, by setting $n - m + 1 = m$, namely $n = 2m - 1$, we get equivalent release-ahead attack resilience R_r and drop attack resilience R_d.

Lemma 1. *In one-hop scheme, with $R_r = R_d$, a larger n makes attack resilience R_r and R_d higher.*

Proof. From [110], we can get:

$$\frac{n}{m} = \left(\frac{\sigma\sqrt{\frac{p(1-p)}{m}} + \sqrt{\frac{\sigma^2 p(1-p)}{m} + 4(1-p)}}{2(1-p)}\right)^2 \tag{3.1}$$

where p denotes the probability that a random DHT node is malicious and σ is positively proportional to R_d. To get $R_r = R_d$ so that the system has good attack resilience towards both the two attack models, we need $n = 2m - 1$, namely $\frac{n}{m} = \frac{2m-1}{m} \approx 2$. Therefore, in equation 3.1, with fixed value of $\frac{n}{m}$ and p, a larger m makes σ larger and therefore R_d higher. Since $n \approx 2m$ and $R_r = R_d$, we can conclude that larger n makes R_r and R_d higher. □

Therefore, given the limited available nodes N to form the pattern, we need to maximize n to maximize the attack resilience, so we set $m = \lfloor \frac{N+1}{2} \rfloor$ (line 1) to maximize m as an integer and then get $n = 2m - 1$ (line 2).

Initial package generation: The sender generates n initial data packages as the n fragments of secret key.

Package routing: At start time t_s, the sender node sends the n fragments to the n holder nodes. The n holder nodes hold the fragments for the entire emerging time period T. At the release time $t_r = t_s + T$, the holder nodes send the fragments to the recipient node.

Security analysis: The release-ahead attack resilience is $R_r = 1 - \sum_{i=m}^{n} \binom{n}{i} p^i (1-p)^{n-i}$ and drop attack resilience is $R_d = 1 - \sum_{i=n-m+1}^{n} \binom{n}{i} p^i (1-p)^{n-i}$, as the success rate of both the

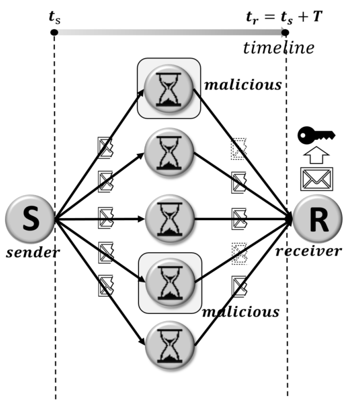


Figure 4: One-hop scheme

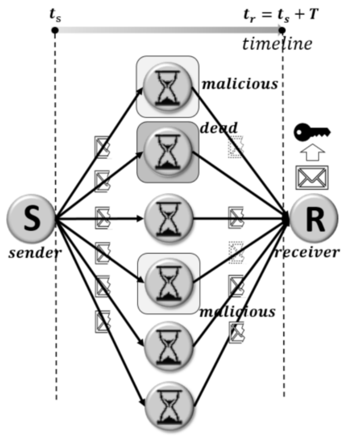


Figure 5: Adjusted one-hop scheme

attack models follows binomial distribution. In the example shown in Figure 4 with $n = 5$ and $m = 3$, two holder nodes are malicious. In release-ahead attack, the two malicious holder nodes can get two fragments, which is less than m to restore the encryption key package. In the drop attack, the two malicious holder nodes can drop two fragments, but the recipient can still get 3 (i.e., m) fragments to restore the encryption key package at release time t_r.

In this simple one-hop scheme, the impact of churn is not taken into account. Next, to handle the churn issues, we propose an adjusted one-hop scheme.

3.2.2 Adjusted one-hop scheme

If the emerging time period T becomes longer, more holder nodes may become dead due to churn, which makes their stored fragments get lost. We can approach this problem in two ways. The first solution is to generate a new fragment whenever one existing fragment is lost. However, to generate the new fragment, at least m living fragments have to be collected by one DHT node to restore the secret key and re-generate the n fragments through erasure coding. This means the secret key has to be known by one node, which significantly increases the success rate of release-ahead attack because this node has probability p to be malicious. Therefore, we decide to take the second solution, namely estimating the number of dead holder nodes as d and reserve some fragments at the beginning for these estimated dead holder nodes by adjusting m and n.

Algorithm 2: Adjusted one-hop path pattern

```
  Input  : Maximum available node number $N$, emerging time period $T$.
  Output: Total fragment number $n$, threshold fragment number $m$.
1 $n = N$;
2 $p_{dead} = 1 - e^{-\frac{1}{\lambda}T}$;
3 $d = p_{dead} * n$;
4 for $m = 1$ to $n$ do
5     $Dif = |\sum_{i=m}^{n} \binom{n}{i} p^i (1-p)^{n-i} - \sum_{i=n-d-m+1}^{n-d} \binom{n-d}{i} p^i (1-p)^{n-d-i}|$
6 end
7 $m =$ the value between 1 and $n$ that minimizes $Dif$.
```

Routing path construction: As suggested by [23], the node death in DHT network can be expressed by a decay pattern, namely the exponential distribution. We can then estimate the percentage of dead holder nodes after the emerging time period T to be $p_{dead} = 1 - e^{-\frac{1}{\lambda}T}$, where λ is the average DHT node lifetime. (In [23], λ is set to 3 years, but it can be changed for different DHT networks.) Therefore, among the n total holder nodes, the number of dead holder nodes should be $d = p_{dead} * n$, which makes the number of living holder nodes to be $n - d$. To do drop attack, the adversary should drop at least $n - d - m + 1$ fragments among the $n - d$ living fragments to make the recipient can at most obtain $m - 1$ fragments at the release time t_r and therefore failing to recover the secret key. The probability of this, which follows the binomial distribution, can be calculated by $P_d = \sum_{i=n-d-m+1}^{n-d} \binom{n-d}{i} p^i (1-p)^{n-d-i}$. However, the dead nodes have no influence on the release-ahead attack because the adversary can collect the fragments at the beginning of the process before any node death. The probability for the adversary to collect at least m fragments from the total n fragments to restore the encryption key package is $P_r = \sum_{i=m}^{n} \binom{n}{i} p^i (1-p)^{n-i}$. Given the maximum available nodes N and emerging time period T, we can calculate m and n through Algorithm 2. We first set $n = N$ (line 1) to maximize attack resilience(the proof is similar to that of lemma 1), and then estimate the number of dead nodes (line 2-3). After that, we try to find the value of m between 1 and n to make $P_d = P_r$ so that $R_d = R_r$ (line 4-7).

Initial package generation: We use the same approach as the one-hope scheme.

Package routing: We use the same approach as the one-hope scheme.

Security analysis: In the example shown in Figure 5, we have the number of maximum available nodes $N = 6$ and an estimated $p_{dead} = \frac{1}{6}$. Therefore, we can calculate $n = N = 6$,

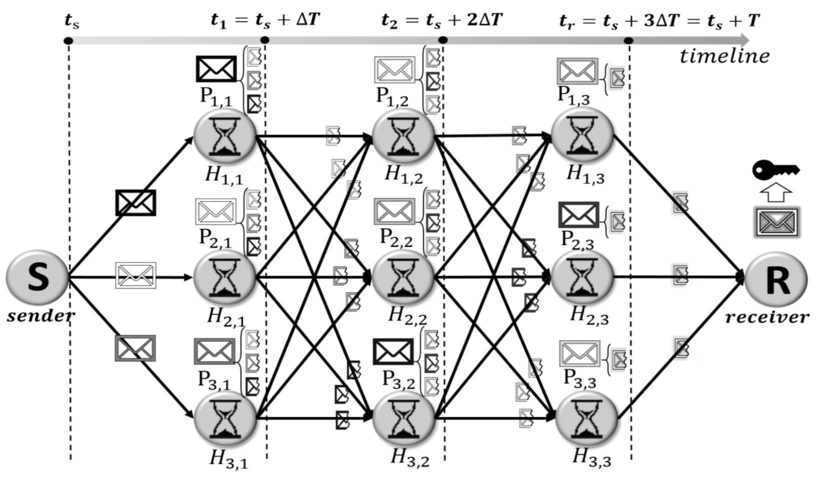


Figure 6: Multi-hop scheme

$d = 1$ and $m = 3$ from Algorithm 2. If there are two malicious holder nodes and one dead holder nodes after the emerging time period T, the adversary will fail to restore the encryption key package with $2 < m$ obtained fragments and the recipient can successfully recover the encryption key package with $3 = m$ received fragments.

Although the adjusted one-hop scheme is resilient to both attack resilience and churn, it has two security problems when the emerging time period T is large. To better express T in time scale, we represent T to be α times of average DHT node lifetime, λ. We believe this is reasonable because different DHT networks may have different average DHT node lifetime. First, when emerging time period T is large, the percentage (p_{dead}) of nodes to be dead may be quite large. For example, if we set $p_{dead} = 0.99 = 1 - e^{-\frac{1}{\lambda}T}$, we can get $\alpha = 4.6$, which means 99% fragments will be lost due to churn after 4.6 times of average DHT node lifetime, λ. In such cases, the adjusted one-hop scheme fails to make attack resilience high. Another issue in this context is the error of estimating the number of dead nodes. An implicit assumption for the adjusted one-hop scheme to be successful is the high accuracy of the estimation result. Unfortunately, in practice, the real number of dead nodes does not always match with the estimation and the estimation error becomes larger for longer emerging time period T. To handle these two issues, we propose the multi-hop scheme.

Algorithm 3: Multi-hop path pattern

```
Input  : Maximum available node number $N$, emerging time period $T$, DHT node average
         lifetime $\lambda$.
Output: Total fragment number $n$, threshold fragment number $m$, number of groups of $n$
         holder nodes $l$.
1  for $l = 1$ to $\lfloor 5(\frac{T}{\lambda}+1)2^{\lg N-3} \rfloor$ do
2      $n = \lfloor \frac{N}{l} \rfloor$;
3      $p_{dead} = 1 - e^{-\frac{T}{\lambda l}}$;
4      $d = p_{dead} * n$;
5      for $m = 1$ to $n$ do
6          $Dif = |\sum_{i=m}^{n} \binom{n}{i} p^i (1-p)^{n-i} - \sum_{i=n-d-m+1}^{n-d} \binom{n-d}{i} p^i (1-p)^{n-d-i}|$
7      end
8      $m$ = the value between 1 and $n$ that minimizes $Dif$;
9      $R_r = (1 - \sum_{i=m}^{n} \binom{n}{i} p^i (1-p)^{n-i})^l$;
10     $R_d = (1 - \sum_{i=n-d-m+1}^{n-d} \binom{n-d}{i} p^i (1-p)^{n-d-i})^l$;
11 end
12 The selected $(l, n, m)$ maximizes $min(R_r, R_d)$.
```

3.2.3 Multi-hop scheme

Instead of leveraging a single set of nodes (Figure 5) to hold the fragments during the entire T, we now arrange multiple sets of nodes (Figure 6) to carry the fragments in relay from the sender to the recipient. Also, the single usage of the erasure coding is now extended to a nested usage so that the old fragments can be merged at each set of nodes to generate new fragments and the reduced number of alive fragments can be replenished during each re-generation. Specifically, we divide the entire long emerging time period T to l pieces of short time periods, namely $T = l * \triangle T$. To form the path pattern, we need l sets of nodes to carry the fragments in relay and each set to take charge of the fragments for $\triangle T$, namely $(i-1)\triangle T \leq t < i\triangle T$, where $i \in [1, l]$. Each set contains n nodes, so the entire number of nodes to form the path pattern is $n * l$, which should be pseudo-randomly selected by the sender node in a non-repeated way. At the start of the i^{th} short time period, namely the time $t = (i-1)\triangle T$, each node in the i^{th} set receives one fragment from each node in the $(i-1)^{th}$ set. Ideally, the number of received fragments should be n. However, some fragments may be lost due to drop attack (the $(i-1)^{th}$ set has malicious nodes) or churn (some nodes in the $(i-1)^{th}$ set become dead during the $(i-1)^{th}$ short time period). If the number of received fragments is at least m, the node in the i^{th} set can still successfully merge the received fragments to get the one generating them through erasure coding (called parent fragment). This parent fragment consists of the n new fragments and the IDs of the nodes in the $(i+1)^{th}$ group. At the end of the i^{th} short time period, namely the time $t = i\triangle T$, the n new fragments are sent to the n nodes in the $(i+1)^{th}$ group. The whole process is then repeated during the next short time period $\triangle T$. This routing scheme has two advantages. First, since $\triangle T < T$, the lost fragments during $\triangle T$ is much fewer than the lost ones during T. Second, by iteratively implementing erasure coding, each group of n nodes can recover the lost fragments caused by drop attack or churn during the previous short time period.

Routing path construction: Besides the values of m and n for erasure coding, the multi-hop scheme also needs to decide the value of l, namely the number of short time period $\triangle T = \frac{T}{l}$, also as the number of sets of nodes. The multi-hop path pattern algorithm is shown as Algorithm 3. Assume we have determined l, since the number of maximum available nodes is $N = n * l$, we can get $n = \lfloor \frac{N}{l} \rfloor$ (line 2). Then, we estimate the dead nodes d during the short time period $\triangle = \frac{T}{l}$ (line 3-4) and find the value for m (line 5-8, same as Algorithm 2). We can then calculate the attack resilience R_r and R_d for each value of l. Our objective is to find the value of l that maximize $min(R_r, R_d)$ from the range $[1\ ,\ \lfloor 5(\frac{T}{\lambda}+1)2^{\lg N-3} \rfloor]$, where the upper bound of the range is estimated through simulation that will be presented in Section 3.3.

Initial package generation: The sender node should first run algorithm 3 to determine l, n, m and pseudo-randomly choose non-repeated IDs for the selected nodes. After that, the sender node can pretend to be the recipient node that has recovered the secret key. Then, the sender node can split the secret key into n fragments through erasure coding and assume these n fragments are the parent fragments maintained by the nodes in the last set (l^{th} set), called l^{th} parent fragments. Next, the sender node should split each of the l^{th} parent fragments into another n fragments received by the node in the 1^{st} set from the n nodes in the $(l-1)^{th}$ set. At this stage, the sender node should have n^2 fragments because there are n nodes in the $(l-1)^{th}$ set and each of them sends n fragments to the l^{th} set of nodes (as shown in Figure 6). The n fragments for each node in the $(l-1)^{th}$ set can be merged to generate the parent fragment maintained by it so that the sender only need to keep the

$(l-1)^{th}$ parent fragments to save space. By repeating this procedure, the sender node can get the $(l-2)^{th}$ parent fragments, $(l-3)^{th}$ parent fragments... and finally the 1^{st} parent fragments maintained by the 1^{st} set of nodes, which are actually the initial packages sent from the sender node to them.

Package routing: Figure 6 shows an example of multi-hop scheme with $l = 3$ and $n = 3$. At start time t_s, the sender node sends three initial packages to the three 1^{st} group nodes. Each 1^{st} group node gets three contained fragments from the initial package, stores them for a short time period $\triangle T$ and send the three fragments to the three 2^{nd} group nodes at $t_1 = t_s + \triangle T$. Then, each 2^{nd} group node restores a parent fragment from the received fragments, gets three contained fragments, holds them for $\triangle T$ and sends them to the three 3^{rd} group nodes at $t_2 = t_s + 2\triangle T$. Finally, each 3^{rd} group node restores a package from the received fragments, gets the secret key fragment, holds it for $\triangle T$ and sends it to the recipient node at t_r to make the recipient restore the secret key.

Security analysis: The multi-hop scheme has better performance when the emerging time T is large. When the Algorithm 3 gives $l = 1$, the output pattern is same as the one generated by the adjusted one-hop scheme. Therefore, we can consider the adjusted one-hop scheme to be a special case of the multiple-hop scheme.

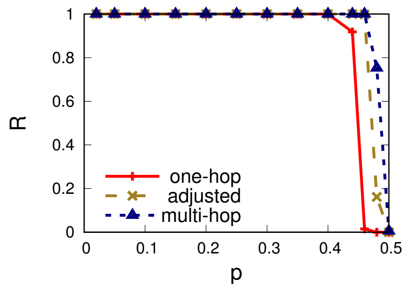
Next, we evaluate the proposed protocols through experimental evaluation.

3.3 EXPERIMENTAL EVALUATION

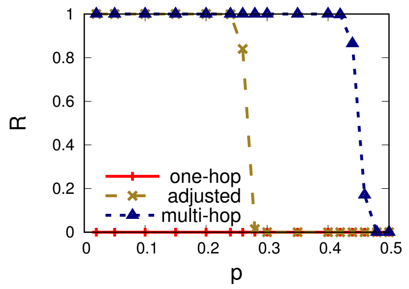
In this section, we experimentally evaluate the performance and security offered by the proposed protocols. Before reporting our results, we first present our experimental setup.

3.3.1 Experimental setup

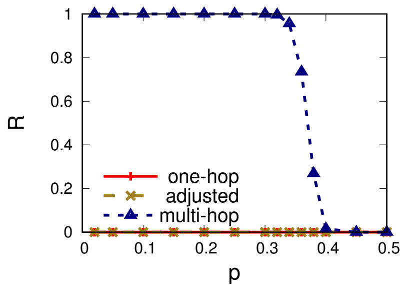
We use an Intel Core i7 PC with 16GB RAM to simulate the protocols through the Java-based DHT toolkit *Overlay Weaver*. We invoke at most 10000 DHT node instances and repeat each experiment for 1000 times to show the average results. To evaluate the attack resilience,



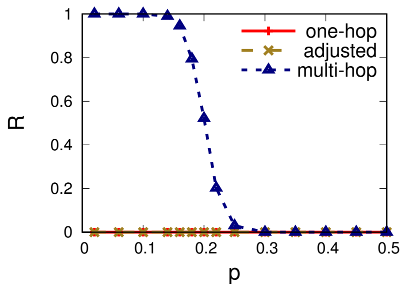
(a) $T = 0.1\lambda$



(b) $T = \lambda$



(c) $T = 10\lambda$



(d) $T = 100\lambda$

Figure 7: Varying emerging time period T

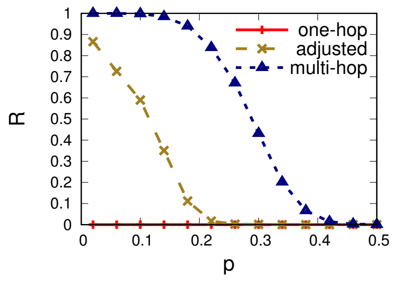
we mark a DHT node as malicious with probability p. To evaluate the churn resilience, we set a lifetime for each DHT node, which follows exponential distribution suggested by [23].

3.3.2 Experimental results

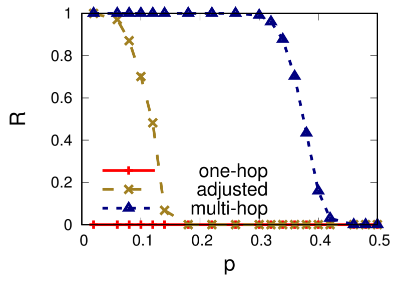
In our experiments, we first evaluate the impact of varying emerging time period T to the performance and security of the three protocols with one-hop scheme (one-hop), adjusted one-hop scheme (adjusted) and multi-hop scheme (multi-hop) respectively. Then, we evaluate the impact of the maximum available nodes N, namely the limited resource to construct the path pattern, to the protocols. Finally, we discuss the selection of the upper bound of l range in Algorithm 3.

The first set of experiments evaluates the protocols with varying emerging time period T. The objective of the protocols is to hold and hide the secret key in the DHT network for the emerging time period T. Therefore, the value of T is an important factor to evaluate it. If the protocol can only effectively work for short T, we do not find its performance to be good enough to satisfy long emerging times. A longer emerging time period T may result in more dead nodes, which requires the protocol to be both churn-resilient and attack-resilient. To comprehensively understand the performance of the protocols, we measure their attack resilience with four representative value of T, namely short emerging time period $T = 0.1\lambda$, medium emerging time period $T = \lambda$, long emerging time period $T = 10\lambda$ and very long time period $T = 100\lambda$, where λ denotes the average lifetime of DHT nodes (e.g. three years in [23]). Since we equally treat the release ahead attack resilience R_r and drop attack resilience R_d, the measured $R = R_r = R_d$. The maximum available nodes N is fixed to 10000. In Figure 7(a), we measure attack resilience R with varying p for the short emerging time period $T = 0.1\lambda$. All the three protocols show good R. Even the protocol based on one-hop scheme, which is most susceptible to T, can maintain quite high R when $p \leq 0.44$. However, even for short T with little churn impact, we can see the performance of the multi-hop scheme is the best. In Figure 7(b), the emerging time period is 10 times than the previous one, which makes the churn impact start to be strong. For the one-hop and adjusted one-hop schemes, since $p_{dead} = 1 - e^{-1} = 0.63$, 63% of the n fragments has been lost due to churn. As can be seen, since the one-hop scheme does not adjust n and m for this, its R directly drops to 0. In contrast, the adjusted one-hop scheme adjusts the value of m by estimating the number of dead nodes and its R is still good before $p = 0.26$. Compared with these two schemes, the performance of the multi-hop scheme is much better. The reason for that is the partitioning of the entire emerging time period T. By dividing it into multiple small pieces, the number of dead nodes during each small time period can be reduced and the lost fragments can also be recovered by the iterative erasure coding. In Figure 7(c), the emerging time period T is further increased by 10 times. Such a long T makes nearly 100% nodes to be dead for the two one-hop schemes and results in their $R = 0$. In contrast, although the multi-hop scheme is also affected, it still keeps $R > 0.99$ when $p \leq 0.32$. Finally, even for the very long T in Figure 7(d), the multi-hop scheme can still maintain $R > 0.99$ when $p \leq 0.14$. As can be seen, all the three schemes work well for short T. The adjusted one-hop scheme can keep good performance for medium T, but only the multi-hop scheme can work well even for long and very long T.

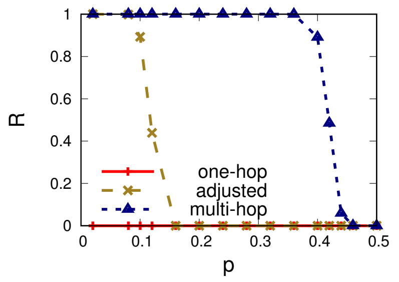
The second set of experiments evaluates protocols with maximum total available node



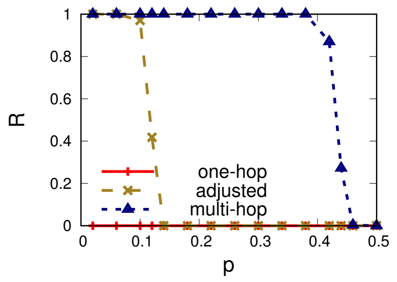
(a) $N = 100$



(b) $N = 1000$



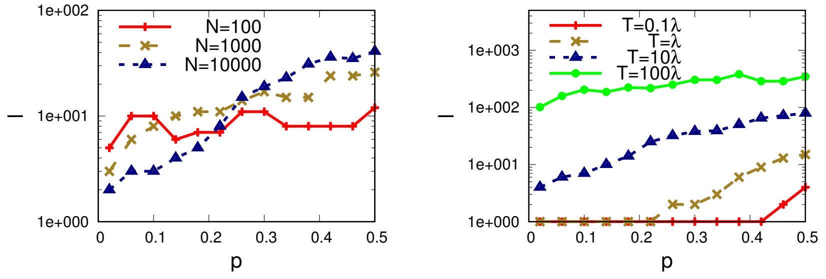
(c) $N = 5000$



(d) $N = 10000$

Figure 8: Varying path construction resource N

N to build the path pattern, namely the path construction resource. Our default choice of N is 10000, which means the routing path pattern is constructed by 10000 DHT nodes. This is acceptable for large-scale DHT network. However, in practice, if a DHT network is not big enough to support $N = 10000$, we want to understand the impact of reduced N to the performance of protocols by reducing N to 5000, 1000, and 100. For this set of experiments, we set the emerging time period $T = 2\lambda$, which has made the attack resilience of the one-hop attack drop to 0 even for $N = 10000$, so we mainly focus on the performance of adjusted one-hop scheme and multi-hop scheme. In Figure 8(a), N is reduced to 100, which means the routing path pattern is formed by at most 100 DHT nodes. We can find that the attack resilience R of one-hop scheme rapidly drops from 0.866 for $p = 0.02$ to 0.422 for $p = 0.12$. In contrast, the attack resilience R of multi-hop scheme keeps higher than 0.99 before $p = 0.14$ and drops lower than 0.5 when p is around 0.29. We can conclude that the multi-hop scheme can still effectively work for small path pattern with $N = 100$ when the probability of node to be malicious p is not high but the adjusted one-hop scheme does not



(a) Varying N (b) Varying T

Figure 9: l upper bound selection

work well. In Figure 8(b), we increase N by 10 times. Both the two schemes have improved attack resilience. Specifically, the adjusted one-hop scheme can make $R > 0.9$ for $p < 0.08$ and $R > 0.5$ for $p < 0.13$, and the multi-hop scheme can make $R > 0.99$ for $p < 0.30$ and $R > 0.5$ for $p < 0.38$. Then, we further increase N by 5 times to 5000 in Figure 8(c). The adjusted one-hop scheme can make $R > 0.9$ for $p < 0.10$ and $R > 0.5$ for $p < 0.12$, and the multi-hop scheme can make $R > 0.99$ for $p < 0.38$ and $R > 0.5$ for $p < 0.42$. We find that the reduction way of the attack resilience R of the adjusted one-hop scheme along the increasing p changes from a smooth manner to a steep manner from $N = 1000$ to $N = 5000$. We can consider $p = 0.12$ as the threshold. When N is small, the R value for $p < 0.12$ gradually drops from 1 to 0.5 and the R for $p > 0.12$ gradually drops from 0.5 to 0. In contrast, when N is large, the R keeps close to 1 for $p < 0.12$ and suddenly drops to almost 0 after $p = 0.12$. In Figure 8(d), the attack resilience of adjusted one-hop scheme has little change. The multi-hop scheme slightly increases $R > 0.99$ for $p < 0.40$. As can be seen, the value of N can be reduced to 5000 from 10000 without losing big performance.

The goal of the third set of experiments is to reasonably bound the selection of l in the Algorithm 3 because we have proved the multi-hop scheme is the most effective one and the unnecessary loops in l selection can drop the performance of Algorithm 3. In Figure 9(a), we fix $T = 2\lambda$ and measure value of l with varying p for $N = 100, 1000, 10000$. We can see that the upper bound of l happens at large p and changes from 12 to 26 to 41 for $N = 10^2, 10^3, 10^4$ respectively so that we can roughly summarize the increment to be twice when N increases from 10^i to 10^{i+1}. In Figure 9(b), we fix $N = 10000$ and measure value of l with varying p for $T = 0.1\lambda, \lambda, 10\lambda, 100\lambda$. We can find the upper bound of l also happens at large p and can be bounded by $\frac{10T}{\lambda} + 10$. We can then combine the finding from the two figures to conclude that $(\frac{10T}{\lambda} + 10) * 2^{\frac{\lg N}{\lg 10000}} = (\frac{5T}{\lambda} + 5) * 2^{\lg N - 3}$.

3.4 SUMMARY AND DISCUSSION

In this chapter, we propose the design of a decentralized self-emerging data release system using large-scale Distributed Hash Table (DHT) networks as the underlying infrastructure. The proposed schemes allow the data sender to securely hide the secret keys of the private data stored in clouds such that the data becomes available at the defined release time but remains unavailable prior to the release time. We present a suite of routing path construction schemes for securely storing and routing secret key in DHT networks that prevent an adversary from inferring the secret key prior to the release time (*release-ahead attack*) or from destroying the key altogether (*drop attack*). Our experimental evaluation using Overlay Weaver DHT emulator toolkit demonstrates that the proposed schemes are resilient to both *release-ahead attack* and *drop attack* as well as to attacks that arise due to traditional churn issues in DHT networks.

Due to the design objectives of the DHT protocols, it is hard to enforce behaviors performed by DHT nodes to make the nodes honestly perform desired behaviors, so the DHT-based decentralized self-emerging data release system proposed in this chapter mainly relies on the redundancy and recovery mechanisms to make the system resist against attacks and churn, which usually requires complex routing paths composed of hundreds of DHT nodes. In the next chapter, we examine another P2P network infrastructure of designing decentralized self-emerging data release systems, which leverages the decentralized consensus and the native cryptocurrency of blockchains to enhance protocol enforceability, thus reducing the number of nodes required for safely routing the self-emerging data.

4.0 SELF-EMERGING DATA RELEASE USING ETHEREUM BLOCKCHAIN NETWORK

In this chapter, we study the ways of designing a self-emerging data release system using the Ethereum blockchain network [128]. Our research in this chapter corresponds to the research task *T-2*. Before presenting the details, it is important to first explain the key research questions of accomplishing this research task when the underlying P2P network infrastructure has been changed from a DHT infrastructure to a blockchain infrastructure.

Research goals: Similar to the design objectives in Chapter 3, the goal of the system designed in this chapter is to keep the private data protected until a prescribed data release time and get automatically released to the legitimate recipient at the release time, even if the data sender goes offline. However, as we have discussed in section 1.1, unlike a recipient in DHT networks that simply implies a DHT node, a recipient of self-emerging data in a blockchain network, such as Ethereum, may imply two types of accounts, namely an External Owned Account (EOA) controlled by a peer of the blockchain network through a private/public key pair or a Contract Account (CA) controlled by a smart contract. Due to the different properties associated with the two types of recipients, such as the abilities of actively participating in the protocol or locally storing data, the underlying protocols designed for the self-emerging data release system may also become different. Therefore, it is important to emphasize that our study in this chapter focuses on the scenario that both the data sender and recipient of the self-emerging data are EOA accounts. In other words, self-emerging data is released to peers controlling EOA accounts. In the rest of this chapter, for ease of presentation, we denote the data sender and recipient as S and R while the rest of EOA accounts as Ps representing peers in the Ethereum network. The details of releasing self-emerging data to CA accounts, which could support privacy-preserving timed execution of functions in smart contracts, will be presented in the next chapter.

Properties of peers in Ethereum: A peer in Ethereum can create EOA accounts to interact with the blockchain and other peers. Similar to a peer running DHT nodes, a peer in Ethereum also has the ability of storing data locally and communicating with other nodes privately and can freely join and leave the network. Specifically, two peers in Ethereum can establish a private channel through the Whisper protocol [13], where a sender encrypts a message with recipient's public key and a recipient decrypts the message with the corresponding private key.

Adversary models: Instead of following the assumption made in Chapter 3 that the peers (i.e., DHT nodes or EOA accounts) are either honest or malicious, the unique feature of blockchains produced by the blending of smart contracts and cryptocurrency allows us to assume that all the peers in Ethereum are adversaries with rationality. In DHT, due to the lack of trust among nodes, there is no way that one node can enforce another node to do anything. However, in Ethereum blockchain, since the decentralized trust has been established through blockchain, each smart contract can be considered as a (virtual) trusted third party. Then, by asking each participant of a smart contract to deposit a certain amount of cryptocurrency as the security deposit to the contract, any fraudulent or dishonest behavior that violates the agreements recorded in the contract will make the violator be monetarily penalized, which incentivizes all rational participants to honestly follow the contract. In this chapter, we leverage such financial incentive and penalty to revisit the countermeasures against undesirable misbehaviors of service providers.

Technical approaches: Our smart contract implementation recruits a set of Ethereum peers to jointly follow the proposed self-emerging data release service protocol allowing the participating peers to earn the remuneration paid by the service users. Meanwhile, the recruited peers need to pay security deposits so that any detected misbehaviors can result in the deposits being confiscated. We model the problem as an extensive-form game with imperfect information to protect against *post-facto attacks* including *drop attack* and *release-ahead attack*. Through careful design of the smart contract based on game theory, we demonstrate that the best choice of any rational Ethereum peer in the proposed technique is to always honestly follow the correct protocol.

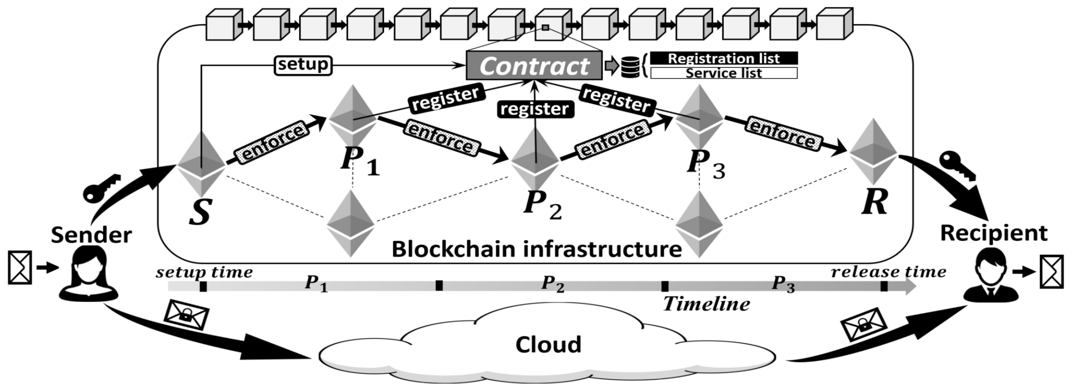


Figure 10: Blockchain-based decentralized self-emerging data release system

Evaluation: We validate the efficacy and attack-resilience of the proposed techniques through rigorous analysis and experimental evaluation on the Ethereum official test network. The experiments demonstrate the low monetary cost and the low time overhead associated with the proposed approach and validate its guaranteed security properties.

In the rest of this chapter, we first introduce the self-emerging data release system established over the blockchain infrastructure in section 4.1. Then, in section 4.2, we present the self-emerging data release service protocol in detail. In section 4.3, we implement and evaluate the proposed protocol on the Ethereum official test network. Finally, we summarize this chapter in section 4.4.

4.1 SYSTEM OVERVIEW

In this section, we present an overview of the proposed self-emerging data release system using the blockchain infrastructure and we introduce the key ideas behind the proposed self-emerging data release service protocol.

4.1.1 Self-emerging data release system using Ethereum blockchain infrastructure

The proposed blockchain-based decentralized self-emerging data release system consists of four key entities (Figure 10) namely data senders, data receivers, a cloud storage and the blockchain infrastructure enabling the self-emerging data release service.

Data sender (S): Data senders have private data to be released to data recipients at a future point in time. At *setup time* t_s, a data sender encrypts/perturbs the private data using a secret key, sends the encrypted/perturbed private data to a cloud storage system and sends the encrypted secret key into the blockchain infrastructure for timed release at the expected release time.

Data recipient (R): Data recipients receive the private data at the expected data release time. While the encrypted/perturbed private data can be downloaded from the cloud storage at any time, the secret key from the blockchain infrastructure can be released to data recipients only at the *release time* t_r determined by data senders.

Cloud: A cloud storage is used as a medium for data senders to transfer the encrypted/perturbed private data to data recipients.

Blockchain infrastructure: The blockchain infrastructure forms the core component of the self-emerging data release system. It implements the protocols necessary for offering self-emerging data release services to data senders.

4.1.2 Self-emerging data release service protocol

The proposed timed data release protocol implemented on the blockchain peer-to-peer network splits a long storage time duration into a series of successive shorter time durations, each of which is handled by a different peer on the blockchain network as the encrypted secret key gets routed on the blockchain network from time t_s to t_r. Thus, the encrypted secret key is routed from the sender to recipient through a routing path formed by multiple peers of the network, each of which stores the encrypted key for a short time window. In the example shown in Figure 10, the storage time duration $[t_s, t_r]$ is split into three fractions and the encrypted secret key is passed from sender S to recipient R through a routing path formed by peers P_1, P_2 and P_3. The proposed protocol enables such a routing scheme through onion routing [46] that requires the sender to first encrypt the secret key using the public key of the recipient and then iteratively form layers of encryption using the public keys of the selected peers on the routing path. As a result, each peer on the routing path decrypts one layer of the encryption of the secret key using their private keys before forwarding it to the subsequent peers on the path until it reaches the recipient who decrypts the final layer of the encryption to obtain the key in plain text.

The protocol provides incentive to the participating peers by requiring the data senders to pay remunerations to the peers for obtaining the store and forward services from them to route the encrypted key along. To participate in the contract, the protocol requires the peers to pay security deposits so that any misbehaviors detected in the protocol will result in their deposits being confiscated.

The protocol satisfies two key requirements in order to be effective in practice:

- First, it ensures credibility so that senders, recipients and peers are guaranteed that they all see the same protocol when they participate in the service. We implement the service protocol using the Ethereum smart contract platform [128] which ensures that when smart contracts get deployed into the blockchain infrastructure, the protocol can be recorded in the blockchain and be available to the public and becomes nearly tamper-proof unless someone controls a majority of computation power of the distributed network [4].
- Second, the protocol needs to be enforceable so that peers are guaranteed to receive remunerations for honestly performing the agreed services while being penalized for any misbehavior or failure to render the promised service. The terms and conditions of the protocol implemented as determinate logics in our approach pass the ownership of money to the smart contract such that it ensures that the only way to receive payment from the smart contract is to trigger the contract with a satisfied condition dictated in the protocol.

The proposed protocol consists of four key components. We introduce them briefly here and present their detailed design in section 4.2.

Peer registration: At any point in time, a new peer P can register by paying a security deposit to join a contract by adding into the *registration list* maintained by the contract.

This process makes the entire network learn that the peer has registered and can provide services during its prescribed working times. For example, in Figure 10, we find that P_1, P_2 and P_3 have been registered before the setup time t_s.

Service setup: At any point in time, a sender S can pay remunerations and submit peers selected from the registration list to a contract C and set up a self-emerging data release service. This process makes the service to be recorded by a *service list* maintained by the contract, C. In Figure 10, we find that sender S requests a service at t_s with selected peers P_1, P_2 and P_3.

Service enforcement: After a service has been set up, the participants, namely sender S, recipient R and peers, Ps should follow the protocol honestly in order to render the service successfully. Behaviors violating the protocol will lead to service failure and such misbehaviors are detected and penalized by the contract. In Figure 10, the process of routing the encrypted secret key from S to R through the path formed by the three peers is enforced by the contract C through paying remunerations for honest behaviors while confiscating deposits for misbehaviors detected by C.

Reporting mechanism: To effectively detect misbehaviors in the protocol implemented in the smart contract, the reporting mechanism incentives peers to report misbehaviors by announcing an award in the contract.

4.1.3 Attack models

In this chapter, we model adversaries with rationality and consider two key *post-facto attack* models, namely *drop attack* and *release-ahead attack*.

Rational adversaries: Recently, it has been widely recognized that assuming an adversary to be semi-honest or malicious is either too weak or too strong in many practical cases and hence modeling adversaries with rationality [47, 61, 62, 103] is a relevant choice in several attack scenarios. Informally, a semi-honest adversary follows the prescribed protocol but tries to glean more information from available intermediate results while a malicious adversary can take any action for launching attacks [64, 133]. A rational adversary lies in the middle of the two types. That is, rational adversaries are self-interest-driven, they choose to violate protocols, such as colluding with other parties, only when doing so brings them a higher profit. In this chapter, in order to design our system with strong and practical security guarantees, we model all involved participants, namely S, R and P, to be rational adversaries without assuming any of them to be honest.

Post-facto attacks: The system targets post-facto attacks. That is, it defends the private data against future attacks launched after the private data has been sent into the system. By allowing senders to declare a registration deadline t_d earlier than t_s and only select peers registered before this deadline, adversaries who decide to attack Alice's data after observing the start of self-emergence of Alice's data from the blockchain cannot make his newly registered peer be selected. Therefore, the primary way to launch attacks is to bribe the peers having Alice's data.

Drop attack: A drop attack happens when the encrypted secret key fails to reach the recipient R at release time t_r. For example, in Figure 10, after receiving the encrypted secret key from peer P_2, peer P_3 may decide to destroy it. A rational adversary may launch a drop attack for getting profit. In post-facto attacks, drop attacks primarily occur through peer bribery. As we have modeled both adversaries and collaborating peers, P to be rational, a drop attack happens only when the adversary gets higher profit from the drop attack than the paid bribery and when P receives higher bribery than the drop penalty. To break the win-win situation, we carefully design the detection mechanism in section 4.2.3 to make drop attacks detectable and to allow the reporting mechanism in section 4.2.4 to distinguish and penalize the adversaries. In addition, by modeling the protocol as an extensive-form game with imperfect information [79], we demonstrate that drop attack can be entirely prevented in our rational model.

Release-ahead attack: In release-ahead attacks, an adversary aims to obtain the secret key before the actual release time t_r and earn a profit by utilizing the data prior to the release time. In Figure 10, peer P_3 can launch a release-ahead attack by releasing the encrypted secret key to recipient R before t_r. Similar to drop attacks, release-ahead attacks primarily happen through peer bribery in post-facto attacks. However, unlike drop attacks that can be detected, a release-ahead attack happens secretly as peers on the path can share stored data to any party without leaving a mark. Our proposed techniques handle this challenge by designing a reporting mechanism to model the release-ahead attack as an extensive-form game with imperfect information (section 4.2.4). It makes rational adversaries choose to never launch release-ahead attacks as the game ensures that the best choice of any rational Ethereum peer is to always honestly follow the correct protocol.

4.1.4 Assumptions

We make the following key assumptions in this chapter:

- We assume that the monetary value of the private data within a service request is bounded by the highest deposit paid by the registered service providers, which represents the real-time solvency of the system.

- Similar to Bitcoin [100], we assume that the pseudonymity offered by Ethereum to the network peers is not adequate [93]. Although techniques such as mixing [29] have been proposed, it is still unclear whether the identification of peers can be adequately protected in such large-scale P2P networks. Therefore, we assume that adversaries can freely communicate with any Ethereum peers and we assume no protection of pseudonymity or anonymity.

- We also assume that an adversary and the peers in communication do not trust each other as otherwise their cost-free collusion violates the rationality assumption of all parties.

- Our system employs the Whisper protocol [13] to enable communication between two Ethereum peers. We assume that a private channel generated using the Whisper protocol between any two Ethereum peers is secure.

- Finally, we assume that the number of registered peers is adequate for providing the required service. We assume that there are at least two different available registered peers at any moment for each service request.

4.2 SELF-EMERGING DATA RELEASE SERVICE PROTOCOL

In this section, we present the proposed service protocol organized along four subsections, each of which discusses a key component of the protocol.

4.2.1 Peer registration

In this subsection, we present the first part of the protocol, *peer registration*, designed for allowing peers to make themselves known to the network. After presenting the protocol in Table 1, we discuss the peer working window and deposit management in more detail. To set up self-emerging data release services, a prerequisite is to have a platform for making peers Ps and data senders Ss know each other. Since peers and senders have no trust in each other, instead of a face-to-face negotiation, they need to transfer their information (peer working window T^w and sender storage window T^s) and money (remuneration and deposit) to the decentralized smart contract C and treat C as a trusted intermediary to put the deal through. A new peer registers by sending their working windows, public keys and deposit to join the contract C. This information is recorded in the registration list maintained by C.

Peer working windows: As discussed in Section 4.1.2, the proposed service protocol splits a long storage time duration, T^s into a series of successive shorter time durations, each of which is handled by a different peer during its working window, T^w, as the encrypted secret key gets routed on the blockchain network. Figure 11 shows an example representing T^ws as horizontal segments in a coordinate frame with timeline and peer indexes as x and y axes respectively. Here, the segment at the bottom-left corner represents a working window $[t_1, t_2]$ belonging to P_i.

Deposit management mechanism: The proposed protocol uses deposits as a mechanism to penalize peer misbehaviors in order to prevent drop and release-ahead attacks. Senders may want to pay more for getting a higher deposit from peers as guarantees of their behaviors to send private data with higher monetary value v. To support such requirements, we design a dynamic deposit management mechanism that incorporates deposit with two states: frozen and unfrozen. One can imagine that each peer has a deposit account in contract C. The

Table 1: Peer registration

Peer registration protocol

1. To be registered, each peer must submit a set of future working windows T^ws and a public key to contract C. It must also pay a deposit to contract C as assurance of no misbehavior while providing future services.
2. Each peer agrees to complete any assigned jobs.
3. Each peer agrees to allow the contract to freeze a part of its deposit for an assigned job until the job is completed.
4. Each peer agrees to renew the public key for each job.
5. Each peer can modify working windows T^ws and the unfrozen deposit at any time, but jobs assigned before modification should still be completed.

deposit account is opened after registration and its balance is denoted as d^a. Initially, d^a is unfrozen. Later, data senders can calculate the amount of deposit they want from peers, denoted as d^s, based on the monetary value of the private data v. Then, during service setup, senders should only select peers from the registration list with at least d^s unfrozen deposit in their accounts. The amount of d^s deposit, once being verified by contract C, will be frozen from accounts of selected peers until the end of their services. At any time, each peer can only manage its unfrozen deposit in the account as the ownership of the unfrozen part has been temporarily transferred to contract C. In this way, the designed deposit management mechanism encourages peers with secure storage environment to keep a high deposit balance so that they can get jobs requiring higher deposit d^s to earn more payments by taking higher risk.

4.2.2 Service setup

Next, we present the second part of the protocol, namely *service setup*, designed for allowing senders to select peers from the registration list based on their requirements and set up the

Table 2: Service setup

Service setup protocol

1. Before setup time t_s, senders compute the remuneration $\widehat{r}$ and deposit d^s required by this service and then locally run the peer selection algorithm to select peers from the registration list satisfying their requirements.
2. At setup time t_s, senders submit service information including selected peers to contract C. Also, both sender S and recipient R should pay $p > d^s + \widehat{r}$ to contract C.
3. Upon receiving a setup request, contract C calculates remuneration $\widehat{r}$ and deposit d^s of this service, then:
   a. If $p > d^s + \widehat{r}$ and each selected peer has unfrozen deposit higher than d^s, C will approve the setup, freeze d^s of selected peers and refund $p - d^s - \widehat{r}$ to S, $p - d^s$ to R.
   b. Otherwise, C will reject the setup and refund p to S, R.

service with contract C after paying remunerations. We first present the protocol description for service setup in Table 2 and then illustrate the remuneration computation and peer selection algorithm in detail.

Remuneration computation: The total remuneration $\widehat{r}$ paid by the sender consists of two parts $\widehat{r_c}$ and $\widehat{r_s}$. The $\widehat{r_c}$ component is charged to compensate the cost of peers for invoking functions of contract C during the service, so $\widehat{r_c} = kr_c$ for k selected peers. The $\widehat{r_s}$ component is charged to reward peers for storing the secret key, so it should be higher for longer storage time $|T^s|$. Meanwhile, to encourage more peers to serve for long-term storage, senders should be charged more for a later storage hour closer to release time t_r than an earlier one closer to setup time t_s. Therefore, if we represent the charge of i^{th} storage hour as $\triangle r_s^i$ and set the first hour charge as $\triangle r_s^1$, by setting per hour increment of $\triangle r_s^i$ as α, we get $\triangle r_s^i = \triangle r_s^{i-1} + \alpha$, which further gives $\widehat{r_s} = \frac{|T^s|[\triangle r_s^1 + \triangle r_s^1 + (|T^s|-1)\alpha]}{2} = |T^s|\triangle r_s^1 + \frac{|T^s|(|T^s|-1)}{2}\alpha$. Additionally, S should be charged more for a higher monetary value of private data v as an incentive to make peers maintain higher balance in deposit accounts, so we consider the above $\widehat{r_c}$ and $\widehat{r_s}$ as the charging standard when $v = \triangle v$ (e.g., $\triangle v = \$100$) and adjust the final $\widehat{r}$ based on that. To sum up, a sender should pay remuneration $\widehat{r} = (\lceil \frac{v}{\triangle v} \rceil)^{\beta}[kr_c + |T^s|\triangle r_s^1 + \frac{|T^s|(|T^s|-1)}{2}\alpha]$ in total and a peer serving for i^{th} to j^{th} hours in T^s should be paid $r = (\lceil \frac{v}{\triangle v} \rceil)^{\beta}[r_c + (j-i)\triangle r_s^1 + \frac{i+j-2}{2}\alpha]$, where $\alpha > 0$ and $\beta > 1$.

Peer selection: The peer selection algorithm has two objectives, namely (i) minimizing remunerations paid by senders and (ii) maximizing the expected profit made by the peers. To realize the first objective, we note that the only way to reduce remuneration $\widehat{r}$ is to make k smaller, namely selecting fewer peers for a service, which does not impact the expected profit r earned by selected peers as $\widehat{r_s}$ is fixed. For achieving the second objective, we need the algorithm to always pick earlier hours in peer working windows T^ws first so that deposit d^s can be unfrozen as soon as possible. For example, the algorithm needs to pick just one hour from a ten-hour T^w for a one-hour service. By picking the last hour in T^w, that peer only receives a one-hour profit because deposit d^s has to be frozen for the entire ten hours. In contrast, by picking the first hour, since deposit d^s will be unfrozen after one hour, the door for accepting new jobs is reopened and that peer can make a ten-hour profit in the best case. We design a greedy algorithm to achieve both of these objectives simultaneously. By decomposing the peer selection problem into a series of subproblems, we define each subproblem as 'given all peer working windows T^ws covering an input time point, output the T^w that makes the total number of selected peers minimum'. Once a T^w is selected, its beginning time t_b is then used as the input time point of the subsequent subproblem to select the next peer. Intuitively, in a subproblem, the greedy choice is to pick the T^w with earliest t_b. Therefore, we have:

Lemma 2. *The greedy algorithm that always picks T^w with earliest t_b minimizes the number of selected P for a service.*

Proof. Let us consider that the peer selection problem is decomposed into n rounds of continuous subproblems. If an algorithm falls behind the greedy algorithm in round i, then the only way for this algorithm to catch up with the greedy algorithm at round $i+1$ will be to select the greedy choice of round $i+1$ in round $i+1$, but this can at most make its performance same as the greedy algorithm. □

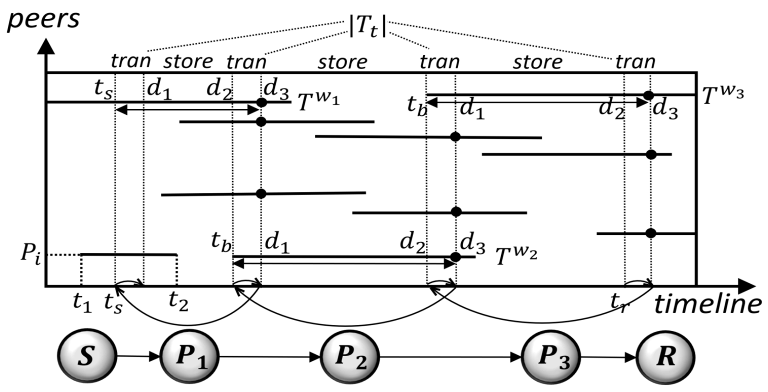


Figure 11: Peer selection

We demonstrate an example of the peer selection process in Figure 11. Instead of release time t_r, the algorithm takes $t_r + |T_t|$ as the input time point of the first-round subproblem as we need to leave a short time period $|T_t|$ for data transfer between each pair of adjacent peers on path. Therefore, in the example with path $S \rightarrow P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow R$, we need four $|T_t|$s. In the first round, there are three available peer working windows T^ws covering $t_r + |T_t|$ and obviously T^{w_3}, due to its earliest begin time t_b among the three, is the greedy choice. As a result, we select P_3 as the last peer on path and set $T^{w_3}.t_b + |T_i|$ as input of the second-round subproblem. We then get T^{w_2} as second-round greedy choice, so we select P_2 and set $T^{w_2}.t_b + |T_i|$ as input of the third-round subproblem, which gives T^{w_1} as third-round greedy choice to pick P_1. This is the end of peer selection process as T^{w_1} has already covered setup time.

The pseudo-code of the peer selection algorithm is shown in Algorithm 4 (we assume peers have passed registration deadline check and balance check). The peer selection problem is decomposed into a series of subproblems. For each subproblem (loop 2-15), the algorithm traverses all available peer working windows T^ws (loop 3-8) to find the ones satisfying the conditions that: 1) it covers the input time point of this round (line 4); 2) it has earlier t_b than the ones that have been traversed (line 5); 3) the peer has not been selected for this service (line 5). After an eligible T^w is found, the greedy choice for this round is updated (line 6). Finally, the end of the traversal gives the greedy choice for the current round subproblem.

Algorithm 4: Peer selection algorithm

```
Input  : Registered peer working window set with enough unfrozen deposit $\widehat{T^w}$, requested sender
         storage window $T^s = [t_s, t_r]$, transfer time period $|T_t|$.
Output: Selected peer working window list $\widetilde{T^w}$.
Initialize $t_{cur} = t_r$, $t_{pre} = t_r$, $T^w_{sel}$;
while $t_{pre} > t_s$ do
    for each $T^w \in \widehat{T^w}$ do
        if $T^w.t_b < t_{cur} + |T_t|$ & $T^w.t_e > t_{cur} + |T_t|$ &
          $T^w.t_b < t_{pre} + |T_t|$ & $nonRepeat(T^w)$ then
            $T^w_{sel} = T^w$; $t_{pre} = T^w.t_b$;
        end
    end
    if $nonRepeat(T^w_{sel})$ then
        $\widetilde{T^w} \leftarrow T^w_{sel}$; $t_{cur} = t_{pre}$;
    end
    else
        Fail;
    end
end
```

If the greedy choice is different from that of the last round, the algorithm approves it and starts the next round (line 9-11). Otherwise, the algorithm fails to find available Ps for this service and returns $False$. The complexity of this algorithm is $O(|\widehat{T^w}||\widetilde{T^w}|)$.

4.2.3 Service enforcement

The third component of the protocol deals with *service enforcement* that specifies the behaviors that should be followed by the sender S, recipient R and peers, Ps during the service process to render the service successful. The protocol sets deadlines for each behavior and treats any missing behavior as a drop attack to enable drop attacks behavior to be detectable. Next, we present the protocol in Table 3 with a discussion on the designed behaviors. We then model the protocol as an extensive-form game with imperfect information to prove that any rational participating peer will always follow the protocol honestly.

Whisper key submission: Our system employs the Whisper protocol [13] to transfer secret keys between any two Ethereum peers by building private channels with symmetrical whisper keys. Specifically, the first peer should encrypt its whisper key with the public key of the second peer and submit it to contract C so that only the second peer can get the

Table 3: Service enforcement

Service enforcement protocol

1. Before time $t_s + |T_t|$, the sender must submit hashes of certificates and the encrypted whisper key to contract C. It must also encrypt the secret key using public keys of selected peers and transfer it to the first peer.
2. Each selected peer must decrypt one layer of the received encrypted secret key, submit the obtained certificate to contract C and verify the behavior of previous participants before its first deadline d_1. It must submit encrypted whisper key to contract C before its second deadline d_2 and transfer the secret key to the next peer before its third deadline d_3.
3. Before time $t_r + |T_t|$, the recipient must first decrypt the last layer of the encrypted secret key to submit the obtained certificate to contract C and then verify the behavior of both the previous participants and the recipient itself.
4. If any verification launched by a peer (or recipient) in term 2 (or 3) gives $False$, C should immediately terminate the service and judge the last participant on the path that fails to pass the verification to be guilty. Then, C should refund deposit d^s to all innocent participants, pay remuneration r to innocent peers and issue confiscated d^s and unused r to sender.
5. If a verification gives $Ture$, contract C should refund deposit d^s and pay remuneration r to all participants that have already honestly finished their job before their deadlines.

whisper key and set up the channel.

Certificate: We design certificates for detecting drop attacks. For each peer and recipient, we need the sender to secretly generate a unique certificate and package it along with the corresponding layer of the encrypted secret key. Therefore, upon decrypting the received encrypted secret key with the private key, the peer (or recipient) will get the unique certificate. The peer (or recipient) then should submit the certificate to contract C. If the hash of the submitted certificate is same as the one submitted by the sender, the correct reception of encrypted secret key can be proved. Otherwise, a drop attack is detected. However, with certificates, we can only detect that a drop attack has happened between two adjacent peers. It is hard to further figure out which of the two peers launched the attack as the channel between them is private. We will discuss how to handle such a dispute in Section 4.2.4.

Verification: We design verification as a function of contract C for enforcing submission of whisper keys and certificates. A missing whisper key or certificate, both causing a drop attack, cannot be automatically detected by contract C. Here, we need the verification function to be triggered by Ethereum peers to check whether the submissions have been made on time. If all the submissions have been correctly made until the time of verification, the function returns a $True$. Otherwise, it returns a $False$. For each self-emerging data release service, multiple verifications are required to detect a drop attack in a timely manner so that the service can be terminated on time and deposits of innocent peers can be unfrozen quickly. We carefully design the protocol as an extensive-form game with imperfect information to prove that any rational participant in this game will always choose to submit both whisper key and certificate on time.

The game induced by the protocol: We model the protocol as an extensive-form game with imperfect information [79], which can be represented as a game tree in Figure 12. For ease of explanation, the example only has one peer P between sender S and recipient R on path, however, the services with more peers follow the same result. The game has three players $\{S, P, R\}$. Its basic actions are (whisper key and/or certificate) submission (s) and verification (v), so the action set is $\{s, v, \bar{s}, \bar{v}, sv, s\bar{v}, \bar{s}v, \bar{s}\bar{v}\}$, where $\bar{s}$ and $\bar{v}$ represent no submission and no verification respectively and $sv, s\bar{v}, \bar{s}v, \bar{s}\bar{v}$ stand for the combinations. The game tree consists of choice nodes $\{n_0, ..., n_{14}\}$ and terminal nodes $\{n_{15}, ..., n_{30}\}$. At the

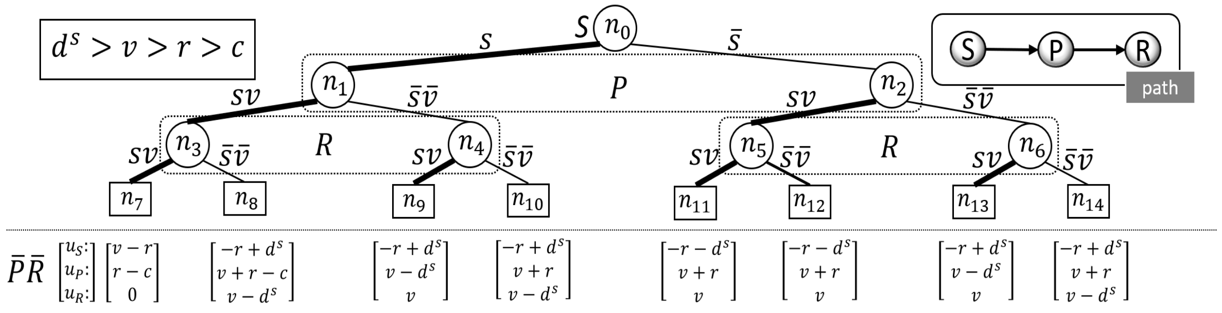


Figure 12: Game tree induced by service enforcement protocol

beginning of the game, sender S ($\{n_0\}$) can choose either to submit whisper key or not by taking one action from $\{s, \bar{s}\}$. Then, the game moves to peer P ($\{n_1, n_2\}$), who has no idea about the choice made by sender S (imperfect information). The peer P should choose one action from $\{sv, s\bar{v}, \bar{s}v, \bar{s}\bar{v}\}$, namely four combinations of doing submission and verification or not, but we argue that $s\bar{v}$ and $\bar{s}v$ can be omitted. The reason is that a peer P choosing $s\bar{v}$ gets same payoff as choosing sv if no previous player has chosen $\bar{s}$ as there is no need of verification in this situation. In contrast, if there is at least one previous player who has chosen $\bar{s}$, the payoff by choosing $s\bar{v}$ is equal to that of choosing $\bar{s}\bar{v}$ as there is no need of submission when a drop attack has been launched earlier. As a result, $s\bar{v}$ can be replaced by sv and $\bar{s}\bar{v}$, and it is also true for $\bar{s}v$. Finally, the game goes to the turn of recipient R ($\{n_3, n_4\}, \{n_5, n_6\}$), who has no idea of the action taken by sender S and peer P. Similar to P, recipient R should choose one action from $\{sv, \bar{s}\bar{v}\}$, but s here only means the certificate submission as it is the last peer on the path.

We now analyze the payoffs shown under the terminal nodes, where u_S, u_P and u_R represent payoff of sender S, peer P and recipient R respectively. The payoffs have uncertainty. Most peers on the path, by dropping the encrypted secret key, can only save a service cost c, but some peers can get an additional profit no more than the monetary value of the private data v (for ease of presentation, we represent it as v in this game). Therefore, it is uncertain whether peer P and receipt R can get the additional benefit v from dropping the package. To model this uncertainty, we use P and R to represent the ones only targeting at c and

$\bar{P}$ and $\bar{R}$ to represent the ones also targeting at v. By considering this uncertainty, this game can actually be modeled as a more sophisticated Bayesian game [17]. However, we find that the four situations in this game ($\{PR, P\bar{R}, \bar{P}R, \bar{P}\bar{R}\}$) can reach the same Nash equilibrium [101] and therefore, for ease of explanation, we will only analyze the situation that both peer P and recipient R can get additional benefit v, namely $\bar{P}\bar{R}$.

In $\bar{P}\bar{R}$, we will show that if deposit $d^s > v$ is satisfied, then the best choice of each player is to do both submission and verification on time. We start from analyzing the choice of recipient R between sv and $\bar{s}\bar{v}$ at the last step of this game. At n_3, by choosing sv, R gets 0 at n_7, which is higher than $u_R = v - d^s$ at n_8 if $\bar{s}\bar{v}$ is chosen and $d^s > v$ is satisfied. By further checking n_4 to n_6, we can find sv always brings u_R no less than u_R from $\bar{s}\bar{v}$, which proves that sv dominates $\bar{s}\bar{v}$ and R should always choose sv no matter how the game has been played before. Following the same rule, peer P should always choose sv at $\{n_1, n_2\}$ if $d^s > v - (r - c)$ is satisfied. Since we need $r > c$ to make Ps get positive profit from the service, $d^s > v - (r - c)$ can be automatically satisfied when $d^s > v$. Finally, with the same rule, sender S should always choose s at n_0.

In game theory, if by taking a strategy, a player can make the expected payoff no less than that induced by taking any other strategy no matter what strategies are taken by other players, this strategy will become his or her best response. If all the players are taking their best responses, the game will reach a Nash equilibrium [101]. Nash equilibrium is the most important solution concept in game theory, which describes a situation that every player chooses the best response and no one can make payoff higher by changing strategy if no one else changes strategy. In this game, the Nash equilibrium is reached when all the players follow the bold edges, which results in all rational players, whether they are sender, recipient or peers, choosing to honestly obey the protocol.

4.2.4 Reporting mechanism

In this subsection, we present the last part of the protocol in Table 4, namely *reporting* designed for handling both release-ahead attacks and the dispute of drop attacks that are hard to be detected by *service enforcement* protocol.

Table 4: Reporting

Reporting protocol
1. Any peer can report a release-ahead attack to contract C with evidence before t_r to earn an award a.
2. Any peer on the path can report a dispute of drop attack between a suspect (the peer before this reporter on path) and the reporter to contract C before deposit d^s of the suspect is unfrozen to earn an award a.

Release-ahead attack: As discussed in Section 4.1.3, it is very difficult to detect a secret release attack made by peers on the path. We design a reporting mechanism to enable a release-ahead attack to be reported with evidence by adversaries themselves. The evidence should include a message and the message signed by the private key of the disloyal peer, which has been released by the peer to the adversary. Then, contract C can verify the correctness of the private key with the public key of that peer. If the private key is proved to be the one of this peer, the adversary will get an award, a from contract C while the peer will lose its deposit d^s. This reporting mechanism is an effective way to prevent release-ahead attacks as long as both adversary and the peer are rational. In the game between them, the best response of the adversary is to always report the peer to earn the award a from contract C without any penalty. Based on this knowledge, the best response of any peer on the path is to never accept bribery. Therefore, the Nash equilibrium of this game makes such a release-ahead attack never happen.

Dispute of drop attack: As discussed in Section 4.2.3, drop attacks cannot be solely prevented by verification. After a drop attack is detected between two adjacent peers on the path when the second peer between the two fails to submit the correct certificate, it is hard to figure out which peer actually launched it. It can be either launched by the first peer by not sending the correct encrypted secret key to the second peer or by the second peer by maliciously denying the reception of the encrypted secret key. In addition, it can be launched by the sender S by submitting fake hashes of certificates to contract C at the very beginning. To solve it, we allow the second account to report the dispute. Upon receiving the report, contract C should confiscate deposit d^s of the three participants and send back an award a to the second peer. Again, this anti-intuitive reporting mechanism is an effective way to prevent drop attack dispute by making the three participants as a community of interests as long as these accounts are rational. In this game, when there is a drop attack, the second peer has the dominant action to always report the dispute because it will lose part of its deposit $d^s - a$ by reporting it but lose the entire deposit d^s due to the missing certificate by not reporting it. With this knowledge, the best response of the first peer and sender is to never launch a drop attack because otherwise they will lose the entire deposit $d^s > v$ due to the report. Finally, given the best response of the first peer and sender, if $d^s > v + a$ is satisfied, the best response of the second peer is also to never launch a drop attack because otherwise it will lose $d^s - a > v$ due to the report. As a result, the Nash equilibrium is reached when all of them choose to never launch a drop attack.

4.3 IMPLEMENTATION

In this section, we present the implementation of the proposed self-emerging data release smart contract and experimentally evaluate its performance and security.

4.3.1 Implementation

We first introduce the implementation setup and then present the functions created in the smart contract and demonstrate how they work in practice. Finally, we present the test instance for our experimental evaluation.

Setup: We programmed the smart contract in the contract-oriented programming language *Solidity* [11], deployed it to the Ethereum official test network *rinkeby* [9] and tested it with Ethereum official Go implementation *Geth* [6]. We used the *SolRsaVerify* contract [12] to verify signatures in the release reporting mechanism. We ran our experiments on an Intel Core i7 2.70GHz PC with 16GB RAM.

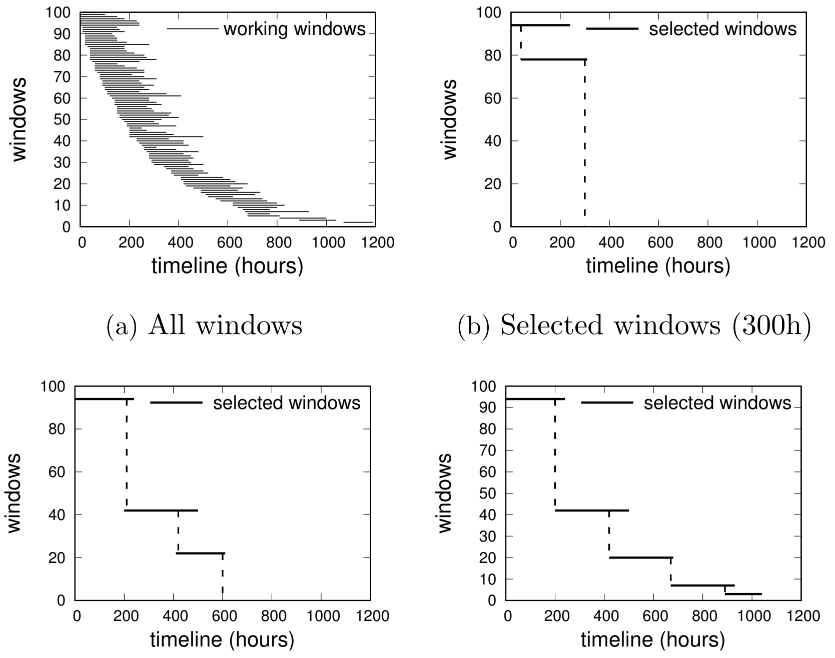
Table 5: Summary of functions in the smart contract

Sections	**Invokers**	**Functions**	**Purposes**
Register	P	newPeer	register a new Peer
	P	updateBalance	update deposit balance
	P	updateWindow	update working windows
	P	updatePubKey	update public keys
Setup	S	senderSign	sender signs the contract
	R	recipientSign	recipient signs the contract
	S	setup	setup the service
Enforce	S	setCert	submit hashes of certificates
	P,R	verifyCert	verify received certificates
	P	setWhisperKey	submit encrypted whisper keys
	P,R	verification	do verification
Report	Any	releaseReport	report a release-ahead attack
	Any	releaseAward	get award for reporting release
	P,R	dropReport	report a drop attack
	P,R	dropAward	get award for reporting drop

Contract functions: We design the smart contract to include 15 main functions for supporting the four parts of protocol presented in Section 4.2. The functions are organized as follows: The functions are shown in Table 5 with their respective invokers and purposes. For example, function *newPeer()* is designed to be invoked by peers during registration phase for being registered into the list.

- **Registration**: Peers (Ps) can first invoke *newPeer()* to be registered and recorded into the peer list and then manage their unfrozen deposit balance, working windows and public keys through the other three functions.
- **Setup**: A sender (S) should download the peer list, locally run peer selection algorithm to select peers from the list and estimate remuneration r. Then, S should sign the contract through *senderSign()* and also inform the recipient (R) to sign it through *recipientSign()*. Finally, S should invoke *setup()* to complete service setup and the smart contract (C) will freeze deposit d^s of each selected P after verifying payments of S and R and record the service information into the service list.
- **Enforce**: At the beginning of a service, S should invoke *setCert()* to submit hashes of certificates to C. Then, during the service process, *verifyCert()* is invoked by Ps and R to submit certifications, *setWhisperKey()* is invoked by Ps to submit encrypted whisper key and *verification()* is invoked by Ps and R to do verification.
- **Report**: Any Ethereum peer can invoke *releaseReport()* to report a release-ahead attack and get award through *releaseAward()* after the report has been verified to be correct. Similarly, P and R on path can report a drop attack through *dropReport()* and get award through *dropAward()*.

Test instance: For testing purpose, we generated 100 Ethereum accounts to be registered as peers. Each peer offers one working window represented as a horizontal segment in Figure 13(a). We design an input parameter *Time* to simulate the time during testing. As can be seen, the 100 working windows are distributed in the future 1200 hours. Their start times follow an exponential distribution with a mean of 300 hours while their lengths follow a normal distribution with a mean of 15 hours and a standard deviation of 5 hours. The reason is that we believe more peers may want to serve in the nearer future due to its lower



(a) All windows (b) Selected windows (300h)

(c) Selected windows (600h) (d) Selected windows (1000h)

Figure 13: Peer selection

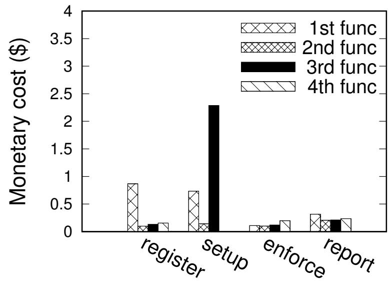
uncertainty. From Figure 13(b) to 13(d), we show the results of peer selection algorithm for sending the private data to 300, 600 and 1000 hours in the future by selecting two, three and five peers respectively. The storage on each selected P, upon hitting the dotted line, will be transferred to the next P. In all cases, storage on each P starts from the beginning of its window, which signifies the design goal of the peer selection algorithm.

4.3.2 Experimental evaluation

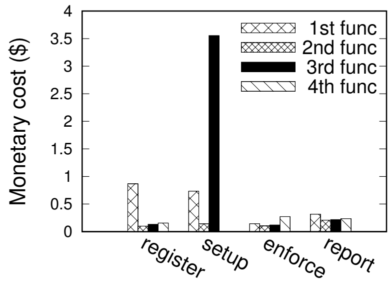
We use the presented test instance to experimentally evaluate the performance and security of the smart contract. We begin by first evaluating the monetary cost and time overhead of the functions and then test the contract in different conditions including drop attack and release-ahead attack scenarios.

Monetary cost: The monetary costs of functions in Table 5 for the three-peer case in Figure 13(c) are shown in Figure 14(a). The results shown represent the maximum possible monetary costs for invoking the functions. For ease of presentation, results are grouped into four clusters. Each cluster represents a protocol subsection and contains three or four functions following their order in Table 5. In Ethereum, each function call will cost some gases if it changes the state of contract. Therefore, the raw data measured here is the gas cost of each function, which is then transferred to cost in \$ based on 1 gas = $1.0371979124 \times 10^{-8}$ ETH and 1 ETH = \$300 as of date, 10/29/2017 [5]. As can be seen, most functions cost very little. Specifically, among the fifteen functions, eight cost lower than \$0.2 and twelve cost lower than \$0.3. The remaining three functions are *newPeer()* (\$0.86), *senderSign()* (\$0.73) and *setup()* (\$2.29). They cost higher as data is stored into the registration list and service list in C through the three functions. However, since each P only calls *newPeer()* for once during registration and each S only calls *senderSign()* and *setup()* once during service setup, these costs are quite acceptable in practice. Thus, in case of three selected Ps, a self-emerging data release service costs \$5.07 in total, including \$3.33 cost incurred by S, \$0.44 cost incurred by each P and \$0.41 cost incurred by R. To study the scalability of the self-emerging data smart contract, we measured the monetary costs of the functions for the five-peer case in Figure 13(d) as Figure 14(b). Compared with Figure 14(a), only costs of three functions *setup()*, *setCert()* and *verification()* are increased as a higher number of selected Ps requires more data to be stored in data list with more certificates and more rounds of verifications. However, the increments of *setCert()* and *verification()* are quite small and the increment of *setup()* from \$2.29 to \$3.56 is not a drastic overhead for storing the private data for a longer duration of 1000 hours.

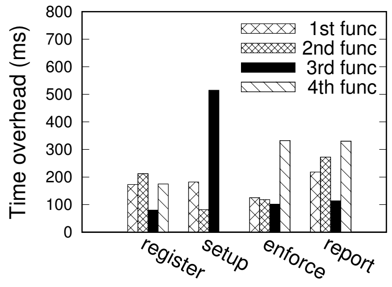
Time overhead: The time overheads of functions in Table 5 for the three-peer case in Figure 13(c) are shown in Figure 14(c). All results are averaged for 100 tests. Among the fifteen functions, nine spent 0-200ms, three spent 200-300ms and two spent 300-400ms. The *setup()* function spent the maximum time of 515ms due to the large amount of service data for storing. Again, we tested the five-peer case in Figure 13(d) and showed the results in Figure 14(d). The two more selected peers make the same three functions *setup()*, *setCert()* and *verification()* spend more time for the same reasons. Here again, the increments are quite acceptable. In addition, we tested the time overhead of the peer selection algorithm, which shows that the algorithm is quite efficient by spending less than 20ms for even a peer list with 1000 working windows.



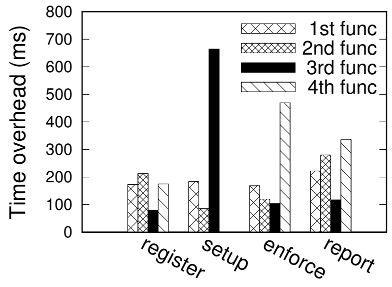
(a) Monetary cost (3 peers)



(b) Monetary cost (5 peers)



(c) Time overhead (3 peers)



(d) Time overhead (5 peers)

Figure 14: Performance evaluation

Table 6: Security evaluation

Cond	**S**	**P1**	**P2**	**P3**	**P4**	**P5**	**R**
1.	5	5	5	5	5	5	5
2.	7.872	5.010	5.017	5.026	5.035	5.040	5
3.	8.489	5.010	4.4	5.026	5.035	5.040	5
4.	8.212	5.310	5.017	5.026	5.035	4.4	5
5.	1.347	5.010	5.017	5.026	4.4	1.7	5

Security evaluation: Finally, we evaluate the security protection offered by the smart contract by testing the results of a self-emerging data release service in different conditions when the S, R and Ps engage in suspicious behaviors, shown in Table 6. The test is based on the five-peer case in Figure 13(d) and the parameters about remuneration are set as $\alpha = 0.000012$ ETH, $\beta = 1.1$, $\triangle r_s^1 = 0.000001$ ETH, $\triangle v = 1$ ETH, $r_t = 0.002$ ETH respectively. The parameter setting can be adjusted, but it should not make the remuneration too low as in that case, one may not be incentivized to freeze \$1000 for half a year for earning a meager payment of \$0.1. In addition, we set $d^s = 1.2v$ and award $a = 0.1v$.

- Condition 1: Before the service, S, R and the five Ps all hold 5 available ETH. Then, S wants to send a secret key with its monetary value $v = 3$ ETH.

- Condition 2: If all the participants follow the protocol honestly, S can earn 2.872 ETH from the 3 ETH v after paying 0.128 ETH to Ps. Each P can earn its remuneration based on the length of its service time as well as the distance of its service from the setup time t_r. As can be seen, the P_5 offering service for 890h-1000h earns much more than P_1 serving for the first 240 hours.

- Condition 3: If P_2 does not submit its whisper key or certificate on time, its confiscated deposit $d_s = 3.6$ ETH will make its final payoff to be $5 - 3.6 + 3 = 4.4$ ETH.

- Condition 4: If P_5 releases its data to P_2, P_2 can report it to earn the 0.3 ETH award, which will make P_5 get $5 - 3.6 + 3 = 4.4$ ETH payoff.

- Condition 5: If P_4 does not send the secret key to P_5 through the private channel, P_5 can report this drop dispute, which will make P_4 get 4.4 ETH payoff. Without reporting it to earn the 0.3 ETH award, P_5 can only get $5 - 3.6 = 1.4$ ETH payoff due to the failure of certificate submission.

As can be seen, in conditions 3 to 5, adversaries with misbehavior only get 4.4 ETH payoff, which makes them lose 0.6 ETH. Therefore, a rational Ethereum peer should always choose to honestly follow the protocol resulting in condition 2.

4.4 SUMMARY AND DISCUSSION

In this chapter, we develop decentralized techniques for supporting self-emerging data release using smart contracts in the Ethereum blockchain network. Our proposed service protocol implemented as smart contracts is immutable in the Ethereum blockchain. The credibility and enforceability of the protocol are guaranteed through a careful design based on extensive-form games with imperfect information to prevent possible post-facto misbehaviors including drop attacks and release-ahead attacks. We develop the smart contract using *Solidity* and implement the system on the Ethereum official test network.

The scenario focused by this chapter requires both the data sender and recipient to be peers controlling External Owned Accounts (EOAs) through private/public key pairs. In the next chapter, we will discuss a more challenging scenario, namely releasing self-emerging data to a Contract Account (CA) controlled by a smart contract, where the recipient is unable to locally store data or privately communicate with other accounts.

5.0 PRIVACY-PRESERVING TIMED EXECUTION OF SMART CONTRACTS

In the previous chapter, we were focusing on releasing self-emerging data to peers who control EOA accounts through private/public key pairs. In this chapter, we examine a more challenging scenario that corresponds to the research task *T-4* proposed in Section 1.1, namely releasing self-emerging data to a passive smart contract that has already been deployed by peers to a CA account. In short, mechanisms satisfying such requirements can facilitate a wide range of decentralized applications, allowing users to schedule their target smart contracts to be automatically executed at future points of time, without revealing their private input data (i.e., self-emerging data) before the expected execution time. Next, before going to the details, we first present the motivations that stimulate our study of releasing self-emerging data to smart contracts. We then present other key research questions associated with the objectives of this chapter.

Motivations and objectives: Recent implementations of blockchain-based smart contract platforms, such as Ethereum [128] and NEO [102], have attracted a large number of developers to build decentralized applications using smart contracts that avoid the need of a centralized server to manage and maintain the data [2, 92, 94]. The market cap for the leading smart contract platform, Ethereum, peaked at $134 billion [52] in 2018 and thousands of decentralized applications, ranging from social networks to financial software, have been developed over Ethereum [118]. The Smart Contracts market is estimated to grow at a CAGR of 32% during the period 2017 to 2023 [117]. A decentralized application may involve one or more smart contracts and each smart contract may contain multiple functions that need to be invoked by application users through transactions. For instance, a sealed-bid auction smart contract [129] requires bidders to reveal their sealed bids by invoking a function (e.g., a *reveal() function*) during a time window. Similarly, a voting smart contract [92] requires voters to publish their votes using a *vote()* function during the voting time window. Each called function in a smart contract is executed by the entire blockchain network. Since both function code and function inputs (i.e., bid or vote) are available on the blockchain, the function outputs are deterministic and their correctness can be verified by the network, thus cutting out centralized middlemen or intermediaries for running these functions [78]. A key fundamental limitation of existing smart contract platforms is the lack of support for users to schedule timed execution of transactions such that their target functions can be invoked at a later time, even when the users go offline. For example, if Bob plans to take a week off work and could not respond to an auction or voting mechanism implemented on Ethereum during the prescribed time windows, he needs a mechanism to schedule these timed transactions by automatically invoking *reveal()* and *vote()* during the time windows. Here, the inputs to these functions namely the bids and the votes are extremely sensitive and need to be securely protected until the prescribed time windows even when Bob is offline. Therefore, we need a two-stage mechanism, namely (1) protecting the inputs of a function (i.e., self-emerging data) before a prescribed execution time and (2) automatically releasing these inputs to the contract address (CA) of target smart contract (i.e., auction smart contract) at the prescribed execution time to make the function get executed by miners.

Challenges: It is easy to implement such a mechanism in centralized application environments as function inputs can be stored by centralized servers and function execution can be triggered by centralized servers at prescribed execution time. For instance, Boomerang [30] allows users of Gmail to schedule their emails to be sent when users have no connection with the Internet. Similarly, Postfity [108] helps users to schedule messages to be posted onto many centralized social networks. However, both the two stages of the aforementioned mechanism is hard to be designed in decentralized platforms such as Ethereum. First, to guarantee verifiability of function outputs, function inputs need to be put onto the blockchain and as a result, both function inputs and outputs become public to all peers at the time the schedule is initialized, thus leading to privacy risks with the input data. Second, when a transaction invoking a function is deployed into the network, the invoked function is executed immediately, which makes it difficult to support timed execution when the user has already gone offline.

Properties of contract accounts (CAs) in Ethereum: Unlike accounts controlled by peers (i.e., DHT nodes, Ethereum EOAs), a contract account (CA) in Ethereum is passive and transparent. The execution of any function of deployed smart contracts must be invoked through either transactions sent by EOAs or messages sent from CAs. All these transactions/messages, as well as function inputs inside them, are publicly recorded by the Ethereum blockchain, which makes the function outputs deterministic because all miners can execute the function with the same inputs and gets the same outputs.

Adversary models: In this chapter, we further increase the strength of adopted adversary models for the purpose of designing more robust countermeasures against potential attacks. In the previous Chapter 4, we have assumed that all the peers are adversaries with rationality. In the first part of security analysis in this chapter, we follow the same assumption made in Chapter 4 and assume that all EOAs are rational adversaries but not malicious. Then, in the second part of security analysis in this chapter, we further assume that there exists a malicious (or irrational) adversary targeting a specific service request while all other EOAs not owned by this malicious adversary are still rational adversaries.

Technical approaches: In this chapter, we observe that instead of requesting an EOA account to invoke the target function at the release time, it is more effective to invoke the target function through a message sent by a smart contract. This is because smart contracts are trustworthy while peers running EOA accounts may perform undesirable misbehaviors. Therefore, we design a new mechanism that makes self-emerging data get released to a proxy contract deployed by the user of decentralized applications and then make the proxy contract automatically call the target function on behalf of the user. The mechanism jointly applies techniques of data redundancy and cryptocurrency-driven enforcement and can handle rational adversaries and malicious adversaries altogether. The proposed mechanism does not reveal function inputs (i.e., self-emerging data) before the execution time window selected by the user, as the function inputs are privately maintained by a set of trustees randomly selected from the network and released only during the execution time window. The function inputs are protected through secret share [114] and multi-layer encryption [45] and possible misbehaviors of the trustees are made detectable and verifiable through a suit of misbehavior report mechanisms implemented in the Ethereum Smart Contracts and any verified misbehavior incurs the monetary penalty on the violator.

Evaluation: We implement the proposed approach using the contract-oriented programming language *Solidity* [11] and test it on the Ethereum official test network *rinkeby* [9] with Ethereum official Go implementation *Geth* [6]. Our implementation and experimental evaluation that the proposed approach is effective and the protocol has a low gas cost and time overhead associated with it.

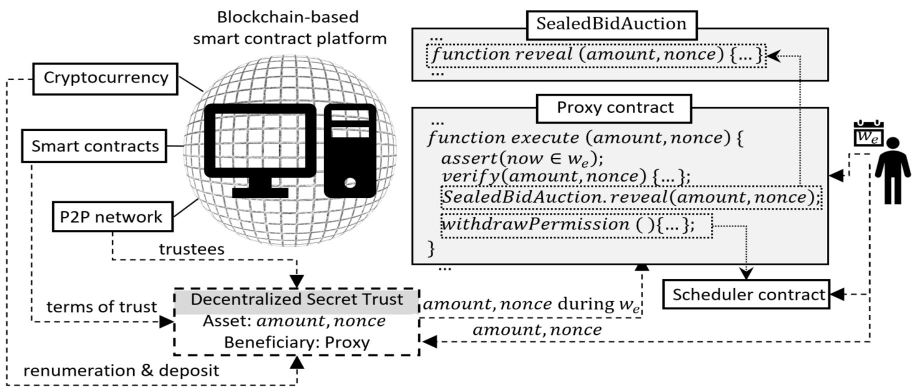
In the rest of this chapter, we first introduce the timed execution in Ethereum in Section 5.1 and then present the designed protocols in detail in Section 5.2. After providing security analysis in Section 5.3, we implement and evaluate the proposed protocol on the Ethereum official test network in Section 5.4. Finally, we summarize this chapter in Section 5.5.

5.1 OVERVIEW OF TIMED EXECUTION IN ETHEREUM

In this section, we first describe the challenges involved in implementing timed execution of smart contracts over Ethereum. We then present the key ideas behind the proposed solution and introduce the organization of the proposed protocol and discuss the security challenges and potential attacks encountered in the proposed approach.

5.1.1 Problem statement

The Ethereum blockchain platform [128] can be viewed as a giant global computer as shown in Figure 15. If a user creates a EOA and uses the EOA to send a transaction with inputs x_1 and x_2 to call function $f(x_1, x_2)$ of a smart contract C at time t_1, function $f(x_1, x_2)$ will be executed instantly and the inputs x_1 and x_2 will be made public. This is acceptable if the user just wants to reveal x_1 and x_2 at time t_1. However, if the user needs to reveal x_1 and x_2 during a future execution time window w_e, sending the transaction at t_1 will not work. For example, Bob may want to make function $reveal(amount, nonce)$ of a sealed-bid auction smart contract [129] be executed during a future execution time window w_e. Then, sending



Blockchain-based smart contract platform
Cryptocurrency
Smart contracts
P2P network
trustees
terms of trust
renumeration & deposit
Decentralized Secret Trust
Asset: amount, nonce
Beneficiary: Proxy
amount, nonce during w_e
amount, nonce
SealedBidAuction
function reveal (amount, nonce) {…}
Proxy contract
function execute (amount, nonce) {
assert(now ∈ w_e);
verify(amount, nonce) {…};
SealedBidAuction.reveal(amount, nonce);
withdrawPermission (){…};
}
Scheduler contract
w_e

Figure 15: At time t_1, Bob wants to schedule function *reveal(amount,nonce)* in contract *SealedBidAuction* [129] to be executed during a future time window w_e

the transaction out at t_1 will make his bid value be known to all other bidders immediately, which violates his privacy requirements.

5.1.2 Privacy-preserving timed execution

To support privacy-preserving timed execution of smart contracts, the transaction calling function $f(x_1, x_2)$ must be sent during the prescribed execution time window w_e while inputs x_1 and x_2 should not be revealed before w_e. Our proposed protocol for supporting privacy-preserving timed execution is implemented as two smart contracts, namely a unique scheduler contract C_s managing all schedule requests of users in Ethereum and a proxy contract C_p deployed by each user having a schedule request. At the time of setting a timed execution, the protocol requires the user to (1) store schedule information, including a cryptographic *keccak-256* hash [21] of function inputs x_1 and x_2 to the scheduler contract C_s, (2) deploy a proxy contract C_p and (3) employ a group of EOAs as trustees. The main functionality of the proxy contract C_p is implemented through a function *execute*() in it. Once C_p receives a transaction during w_e with the desired inputs x_1 and x_2 verified through their hashes in scheduler contract C_s, the function *execute*() will immediately send a message calling the target function $f(x_1, x_2)$ with inputs x_1 and x_2. The trustees are in charge of storing inputs x_1 and x_2 off the blockchain before the execution time window w_e and they send a transaction with the inputs to the proxy contract C_p during w_e. The terms of the decentralized secret trust created by the user as a settlor, namely what the trustees can or cannot do, are programmed as functions in smart contracts C_s and C_p. Each trustee needs to pay a security deposit d (i.e., Ether) to the scheduler contract C_s and any detectable misbehavior of this trustee makes the deposit be confiscated. The security deposit serves as an economic deterrence model for enforcing behaviors of peers in the blockchain network [16, 94]. Finally, after the trustees have sent a transaction with inputs x_1 and x_2 to the proxy contract C_p during w_e, they can withdraw both their deposit and remuneration paid by the user from the scheduler contract C_s. In the example of Figure 15, at t_1, Bob stores hash of inputs *amount* and *nonce* to C_s, deploys C_p and employs a group of trustees. These trustees, after signing an agreement with Bob, are in charge of revealing the asset *amount* and *nonce* to the beneficiary, proxy contract C_p, during w_e. During the execution time window w_e, after the trustees have sent a transaction with inputs *amount* and *nonce* to C_p, the function *execute*() in C_p can trigger *reveal*() in the *SealedBidAuction* contract through *SealedBidAuction.reveal*() and also unlock trustees' deposit and remuneration in C_s through *withdrawPermission*().

5.1.3 Protocol overview

The proposed protocol consists of four components:

Trustee application: At any point in time, an EOA can apply to C_s for getting added into a trustee candidate pool maintained by C_s by submitting its working time window and paying a security deposit. During the working time window, the EOA should be able to connect with Ethereum to send transactions to the proxy contract C_p. In the example shown in Figure 16, we notice that ten EOAs joined the pool. The public pool then makes the entire network learn that this EOA can provide services during its declared working times.

User schedule: During setup time window w_s, a user can schedule a transaction by registering the schedule to scheduler contract C_s, deploying a proxy contract C_p, and secretly selecting trustees from the pool. The selected trustees should keep the function inputs privately before the execution time window w_e while revealing them during w_e to make the target function be executed. In Figure 16, during setup window w_s, the user informed the schedule with the scheduler contract C_s and deployed the proxy contract C_p. Then, the user randomly selected three EOAs from the pool as trustees and signed agreements with the trustees through private channels created by the whisper protocol [13]. Any data exchanged through the whisper channels are encrypted and can only be viewed by the data sender and data recipient.

Function Execution: During execution time window w_e, the selected trustees submit the function inputs to the proxy contract C_p through transactions, which triggers C_p to verify correctness of function inputs with C_s and then call the scheduled function in the target contract C_t. In Figure 16, during w_e, the trustees submitted stored data to proxy contract C_p. After verifying the received data with the hashes stored in scheduler contract C_s, C_p called the function in C_t.

Misbehavior report: During the entire process, trustees may perform several types of misbehaviors violating the protocol, such as secretly disclosing stored data before w_e or rejecting to submit stored data during w_e. To tackle these issues, the protocol involves several misbehavior report mechanisms that allow any witness of a misbehavior to report it to the scheduler contract C_s and earn a component of the deposit paid by the suspect trustee once the report is verified to be true.

5.1.4 Security challenges and attack models

The proposed mechanism encounters several critical security challenges, which can be roughly classified using two attack models.

Time difference attacks: The time difference attack happens when an adversary aims at obtaining the function inputs at a time point t_d earlier than the execution time window w_e so that he can leverage the time difference between t_d and w_e to achieve his purpose. There are three key methods to launch a time difference attack.

- **Malicious trustee**: An adversary may choose to create a suite of EOAs and make all these EOAs join the pool so that once some of these EOAs are selected as trustees, the adversary is able to obtain the stored data directly. To protect the system from malicious

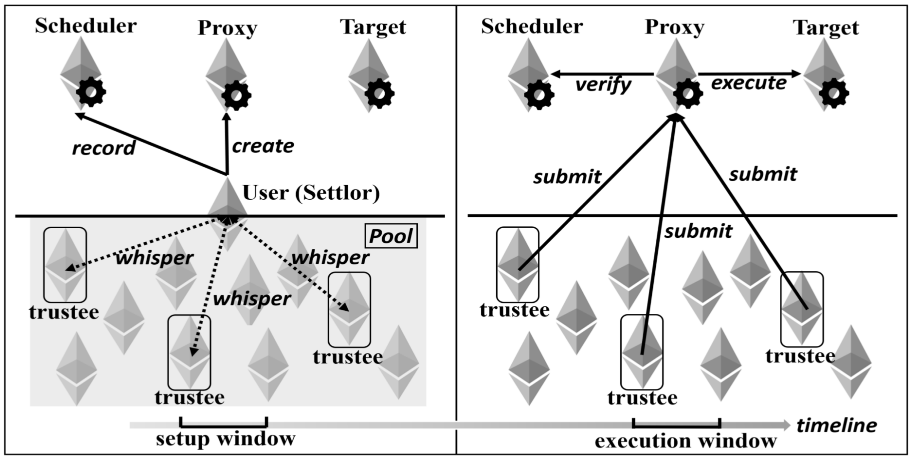


Figure 16: Protocol overview

trustees, the protocol employs both secret share [114] and multi-layer encryption [45] in *user schedule* component of the protocol.

- **Trustee identity disclosure**: In *user schedule* component of the protocol, trustees are secretly selected by user U. Therefore, from the perspective of EOAs besides the selected trustees and user U, all EOAs in network with working time windows satisfying U's requirement have equal chance to be selected by U, thus protecting the identifications of selected trustees with highest entropy and uncertainty. However, a trustee, after being selected, may maliciously announce its identity to the public to seek trade with potential adversaries about the stored data. To prevent such misbehavior, the proposed protocol employs a trustee identity disclosure report mechanism in *misbehavior report* component of the protocol, which forces a trustee to disclose its identity with the sacrifice of the confiscation of its security deposit.
- **Advance disclosure**: A trustee may choose to voluntarily disclose the stored data to the entire network without seeking bribery. To penalize such misbehavior, an advance disclosure report mechanism is employed in the *misbehavior report* component, which makes any trustee disclosing its stored data in advance lose its security deposit.

Execution failure attack: The execution failure attack happens when an adversary aims at making the execution of the target function fail during the execution time window w_e. There are two key methods to launch this attack.

- **Absent trustee**: A trustee may become absent during the execution time window w_e, which makes its stored data get lost. To prevent this type of misbehavior, the *user schedule* component of protocol requires each selected trustee to provide a signature, which will only be revealed along with the function inputs during w_e. Therefore, before w_e, the identities of trustees are kept secret. In contrast, during w_e, the identities become public so that any present trustee can report an absent trustee through the absent trustee report mechanism in the *misbehavior report* component of protocol, which penalizes any absent trustee by confiscating its security deposit.
- **Fake submission**: A trustee may submit fake stored data to the proxy contract C_p during w_e, which may cause the restoration of the function inputs to fail. The protocol handles this type of misbehavior using the fake submission report mechanism in the *misbehavior report* component of protocol, which confiscates violator's security deposit if its submission is proved to be fake.

5.2 PROTOCOL DESCRIPTION

In this section, we present the proposed protocol organized along the four components introduced in Section 5.1.3.

5.2.1 Trustee application

The first component *trustee application* allows EOAs that want to earn remuneration through the trustee job to register to the scheduler contract C_s and make their information public. We present the *trustee application* protocol in Table 7, which consists of three key steps.

Step 1: Each trustee candidate should be a newly generated EOA, which only has an amount of Ether (the cryptocurrency in Ethereum) that will be submitted to the scheduler contract C_s as security deposit d in step 2. No additional Ether should be left because we

Table 7: Trustee application

Trustee application protocol

Input: scheduler contract C_s

Apply:

1. An Ethereum node creates a new EOA.
2. This EOA applies to the scheduler contract C_s for being added into the trustee candidate pool by submitting a public key, a whisper key, working time window, a security deposit and a beneficiary address.
3. The scheduler contract C_s verifies the application and accept the application if all required data has been submitted.

will need the account to make its account private key public during execution time window w_e.

Step 2-3: An EOA should apply for the trustee candidate by sending a transaction to C_s with the five listed information.

- The public key will later be used by user U in step 8 of *user schedule* component to generate *onions* [45]. Here, the term *onion* refers to the output of iteratively encrypting data with multiple public keys.
- The whisper key will later be used by user U in *user schedule* component to establish private channel with this EOA through whisper protocol [13].
- The working time window will be used by user U in step 6 and 10 of *user schedule* component to select trustees satisfying U's requirements (i.e., execution time window).
- The security deposit is a fixed amount of Ether hard-coded in scheduler contract C_s. Once being submitted to C_s, the deposit can only be withdrawn at the end of EOA's working time window, if there is no misbehavior reported through report mechanisms in *misbehavior report* components.
- Finally, the protocol needs the EOA to make its account private key public in *function*

execution component, so the beneficiary address will be the address of a safe EOA to receive deposit and remuneration withdraw.

5.2.2 User schedule

The second component *user schedule* prescribes how a user should set a schedule through three key operations, namely deploying a proxy contract (step 3), registering the schedule information to scheduler contract C_s (step 4) and implementing a two-round trustee selection (step 5-13). We present this component in Table 8. For the illustration of the protocol in step 5 to 13, we will use the example shown in Figure 17.

Step 2: The total remuneration that should be paid by user U is $r = nlr_t + r_e$, where r_d is a fixed per trustee remuneration hard-coded in C_s and r_e is a fixed amount of reward hard-coded in C_m paying to the first trustee calling *execute*() in C_t during w_e. Both r_d and r_e can only be withdrawn by trustees after the end of execution time window w_e.

Step 4: After the schedule has been registered in C_s, the on-chain schedule information cannot be modified. Therefore, the information can be used by trustee candidates later in step 7 and 11 to verify the information transmitted through off-chain whisper channels from user U.

Step 5: The Shamir secret sharing scheme [114] with parameter (m, n) can split the *key* to n *shares*. Later, any m *shares* among the n can be combined to restore the *key* while even $m - 1$ *shares* fail to do it. Therefore, even if some *shares* are compromised, the compromised *shares* may be insufficient to restore the *key* before execution window w_e while the rest *shares* may still be sufficient to restore the *key* during w_e. In the example of Figure 16, we set $(m, n) = (2, 3)$, so three *shares* are generated from *key* after splitting.

Step 6-13: The design of two-round trustee selection implements the decentralized secret trust. The trustees selected in the first round should agree the user encrypt the *shares* with their public keys for multiple layers so that the *shares* become *onions* [45] and harder to be compromised. Then, the trustees selected in the second round should take charge of storing these *onions*. Later, during w_e, once both the private keys of the first-round trustees and *onions* stored by the second-round trustees are made public, the *key* can be restored to

Table 8: User schedule

User schedule protocol

Input: scheduler contract C_s, target contract C_t

Initialization:

1. User U decides function inputs IN, execution time window w_e, secret sharing parameters (m, n), number of layers l, a 256-bit secret key *key* and a 256-bit random number R_U.
2. User U computes the remuneration r.
3. User U deploys proxy contract C_p to the network.
4. User U registers the schedule to scheduler contract C_s with $(w_e, m, n, l, C_p^{addr}, r)$ and receive a schedule ID *sid*.
5. User U splits *key* to n *shares* through (m, n) secret sharing.

First-round trustee selection:

6. User U randomly selects $n(l-1)$ trustees and sends each trustee a (sid, tid), where *tid* refers to a non-repeated ID in the range of $[0, n(l-1))$ assigned to the trustee.
7. Each selected trustee T then does the following:
   - 7.1. Verify $(U^{addr}, sid, tid, w_e, r)$ with C_s.
   - 7.2. Generate a 256-bit random number R_T.
   - 7.3. Take keccak256 hash $h(T^{addr}, R_T)$.
   - 7.4. Sign $(U^{addr}, sid, tid, h(T^{addr}, R_T))$ with T's private key, which gives signature $vrs = (v, r, s)$.
   - 7.5. Send $h(T^{addr}, R_T)$ and vrs back to U.
8. User U encrypts *shares* to *onions* with public keys of selected trustees.
9. User U takes keccak256 hash $h(onion)$ of each *onion* and submits the hash values to C_s.

Second-round trustee selection:

10. User U randomly selects n trustees and sends each trustee a $(sid, tid, onion)$, where *tid* is non-repeated in $[n(l-1), nl)$.
11. Each selected trustee T follows step 7, but in addition verifying received *onion* with $h(onion)$ in C_s.

Ciphertext and hash disclosure:

12. User U encrypts (IN, vrs, R_U) with *key* and make $E(key, (IN, vrs, R_U))$ public.
13. User U submits keccak256 hash $h(IN, R_U)$ and each trustee's $h(T^{addr}, R_T)$ to C_s.

decrypt the function inputs. The process offers following additional security features:

- The identities of selected trustees are kept private. In these steps, each trustee only communicates with the user through a whisper channel and all information that needs to be made public are announced by the user (step 9,12,13). Therefore, the identity of each trustee is only known to the user. This feature helps in suppressing collusion among trustees.
- The identities of selected trustees are verifiable and only the trustees can pass the verification. To be verified as a specific trustee, both the trustee's address T^{addr} and the nonce R_T need to be submitted to C_s and their hash should match with the one submitted by user in step 13. Since R_T is created by the trustee, only the trustee has the ability to pass the verification. This feature also helps in suppressing collusion among trustees. We will discuss it in detail later in *misbehavior report* component.
- The identities of selected trustees are undeniable. The user has signatures of the trustees (step 7,11) and the encrypted signatures are made public in step 12. Therefore, once *key* is restored during w_e, the decrypted signatures can reveal the identities of all trustees. This feature helps in detecting absent trustees who disappear during w_e.
- The trustees are also protected against malicious users. It may be insecure to only allow users to publicly speak. A malicious user may fabricate information and make trustees lose security deposit. To protect trustees from such users. Once a user has registered a schedule in step 4, the submitted information cannot be changed. Then, in step 7 and 11, each trustee can check the information before sending a signature to the user. This is also the main reason that we need two rounds. In step 11, the second-round trustees should first verify the correctness of the onions with the hash submitted by the user in step 9 and then provide signatures.

In the example of Figure 17, six trustees (T_1-T_6) are selected by user U in the first round and their six public keys encrypt each of the three *shares* with two layers, thus turning the *shares* into two-layer *onions*. Then, three trustees (T_7-T_9) are selected by user U in the second round to store the three *onions*. Finally, U ends the schedule by making the ciphertext public and submitting all hash values to C_s.

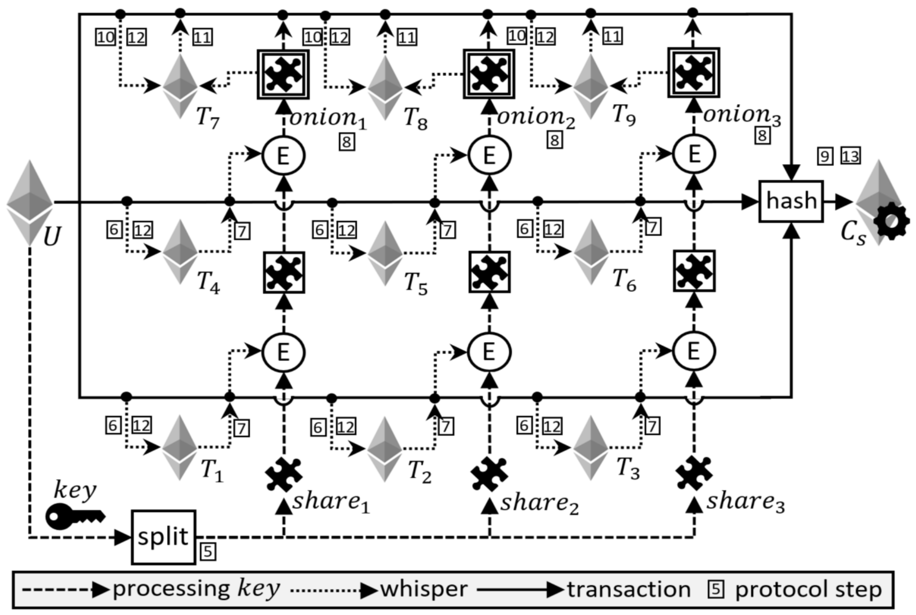


Figure 17: User schedule example

Table 9: Function execution

Function execution protocol

Input: scheduler contract C_s

Submission (first half of w_e):

1. Each trustee T verifies its identity with $h(T^{addr}, R_T)$ by submitting R_T to C_s.
2. Each trustee T submits *onion* or its private key to C_s, where *onion* should be verified with $h(onion)$.

Execution (second half of w_e):

3. Any trustee T can get *shares* by decrypting *onions* with the private keys.
4. Any trustee T can get *key* by combing any m *shares*.
5. Any trustee T can get (IN, vrs, R_U) by doing $D(key, E(key, (IN, vrs, R_U)))$.
6. Any trustee T can submit (IN, R_U) to proxy contract C_p, where (IN, R_U) can be verified with $h(IN, R_U)$ in C_s and the correct function inputs IN will trigger C_p to call the target contract C_t.

5.2.3 Function Execution

The third component of the protocol, *function execution* indicates how the trustees selected in *user schedule* component should collaboratively reveal the function inputs during execution window w_e and send a transaction with the function inputs to the proxy contract C_p through two phases, namely *submission* (step 1-2) and *execution* (step 3-6). We present the third component in Table 9.

Step 1-2: The *submission* phase indicates the first half of execution window w_e, during which the protocol requires first-round and second-round trustees to submit their private keys and stored *onions*, respectively. To submit either a private key or an *onion*, a trustee should also provide the nonce R_T generated in step 7 and 11 of *user schedule* so that its identity can be verified with $h(T^{addr}, R_T)$.

Step 3-6: The *execution* phase refers to the second half of execution window w_e. Since both *onions* and private keys have been submitted, during this phase, any verified trustee should be able to turn *onions* back to *shares*. Then, based on Shamir secret sharing scheme, any m shares can be combined to restore the *key* created by user S in step 1 of *user schedule*. After getting the *key*, any trustee is able to decrypt the encrypted (IN, vrs, R_U). Finally, before the end of w_e, a verified trustee, after obtaining function inputs IN and nonce R_U, should send proxy contract C_p a transaction with both IN and R_U. Then, C_p will immediately verify received IN and R_U with $h(IN, R_U)$ in scheduler contract C_s. If both of them are correct, C_p immediately send a message with IN to the target contract C_t to call the scheduled function.

5.2.4 Misbehavior report

The *misbehavior report* represents the final component of the protocol and involves four types of misbehaviors that will result in the violator's security deposit being confiscated. All these misbehaviors are witnessable and the protocol rewards the reporter of a misbehavior a component of the violator's security deposit as an incentive while sending the rest of the violator's security deposit to the user. We present this final component in Table 10.

Trustee identity disclosure report: This report mechanism is designed to handle the

Table 10: Misbehavior report

Misbehavior report protocol

Input: scheduler contract C_s

Trustee identity disclosure report:

1. Before the start of execution time w_e, any EOA can report a trustee identity disclosure misbehavior by submitting the nonce R_T of the violator to scheduler contract C_s.
2. If $h(T^{addr}, R_T)$ using the submitted R_T is same as the one in C_s, the misbehavior is verified.

Advance disclosure report:

3. Before the start of execution time w_e, any EOA can report an advance disclosure misbehavior by submitting the private key belonging to the violator to scheduler contract C_s.
4. If the public key derived from that private key is same as the violator's public key in C_s, the misbehavior is verified.

Absent trustee report:

5. After step 5 in *function execution*, any trustee can report an absent trustee misbehavior to scheduler contract C_s by submitting the signature vrs of the absent trustee.
6. The address of the violator can be derived through $T = sigVerify((U^{addr}, sid, tid, h(T^{addr}, R_T)), vrs)$.

Fake submission report:

7. After step 2 in *function execution*, any trustee can report a fake submission misbehavior to scheduler contract C_s if the trustee finds a submitted private key is incorrect.
8. If the public key derived from that private key is different from violator's public key in C_s, the misbehavior is verified.

trustee identity disclosure misbehavior presented in Section 5.1.4. Before the start of execution window w_e, a trustee may choose to reveal its identity to seek collusion. To prove its identity, the violator has to reveal the nonce R_T created by itself in step 7/11 of *user schedule* so that its identity can become verifiable through $h(T^{addr}, R_T)$ in C_s. However, with this report mechanism, any EOA, after knowing R_T before w_e, can report it to C_s to earn reward.

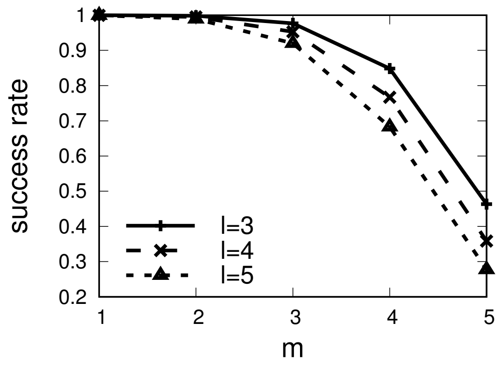
Advance disclosure report: The advance disclosure misbehavior introduced in the Section 5.1.4 can be handled using this report mechanism. Before the start of w_e, a round-one trustee may choose to disclose its private key, which may help an adversary to decrypt *onions* to *shares*, restore *key* and obtain IN before the start of w_e. However, with this report mechanism, any EOA, after knowing violator's private key before w_e, can betray the violator by reporting it to C_s.

Absent trustee report: This report mechanism handles the absent trustee misbehavior described in Section 5.1.4. Any trustee may become absent during w_e, thus increasing the failure chance of schedule. With this report mechanism, any trustee, after obtaining signatures of all other trustees in step 5 of *function execution*, can locally verify attendance of all other trustees, thus being able to report absent trustees to C_s.

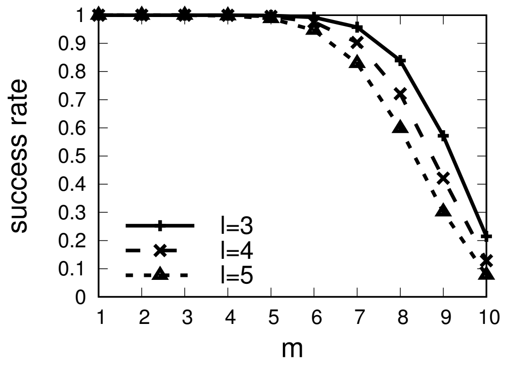
Fake submission report: Finally, the design of fake submission report aims at dealing with the fake submission misbehavior presented in Section 5.1.4. In step 2 of *function execution*, a submitted private key may not be the right one. Any trustee can locally verify a private key submitted by a suspect trustee through deriving the corresponding public key from the private key and comparing it with the public key submitted by that suspect trustee during *trustee application*, thus becoming able to report violators to C_s.

5.3 SECURITY ANALYSIS

Next, we analyze the security guarantees of the proposed approach by modeling adversaries in two different categories, namely rational adversaries and malicious adversaries. Specifically, in Section 5.3.1, we start by assuming that all EOAs are rational adversaries but not



(a) $n = 5$



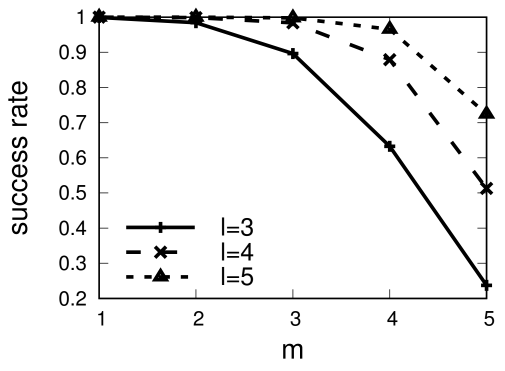
(b) $n = 10$

Figure 18: Schedule success rate when 5% of trustees perform misbehaviors inadvertently

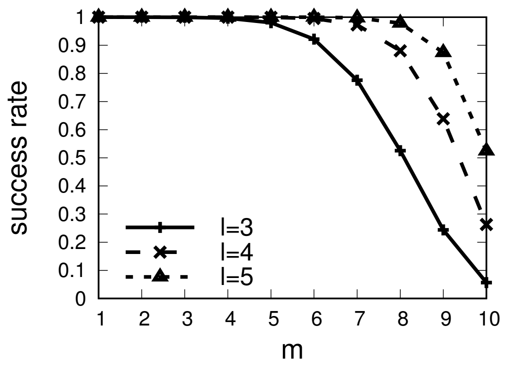
malicious. Then, in Section 5.3.2, we further assume that there exists a malicious adversary targeting user U's private data while all other EOAs not owned by this malicious adversary are still rational adversaries.

5.3.1 Rational adversary

We have introduced the properties of rational adversaries in Chapter 4. In this chapter, we start by assuming that all EOAs in the network are rational adversaries but no one is malicious. Without countermeasures, such rational adversaries, after being selected by user as trustees, may perform four types of misbehaviors introduced in Section 5.1.3, including *trustee identity disclosure*, *advance disclosure*, *absent trustee* and *fake submission*. As per the four misbehavior report mechanisms designed in *misbehavior report* component, as long as the *key* can be restored during the execution time window w_e, any of the four types of misbehaviors will lead to confiscation of the violator's deposit. To prevent restoration of the *key* so that misbehaviors can be performed in free, a certain fraction of trustees must collude to not submit their stored data (i.e., *onion* or private key) together. However, due to trustee identity disclosure report mechanism in *misbehavior report*, revealing trustee identity to other EOAs means losing deposit, so such a collusion will not happen among rational



(a) $n = 5$



(b) $n = 10$

Figure 19: Schedule success rate when 50% of trustees are malicious

adversaries. Therefore, when there is no malicious adversary, rational adversaries will never voluntarily perform misbehaviors.

It is possible that a rational adversary performs misbehaviors inadvertently, such as forgetting providing the service or losing EOA's private key. Such kinds of inadvertent misbehaviors lead to same results of intentionally performing *absent trustee* misbehavior. If we denote the percentage of EOAs performing inadvertent misbehaviors as p_{IM}, the success rate of a schedule with parameters (l, m, n) will be computed through the Cumulative Distribution Function of Binomial distribution, namely $SR = 1 - \sum_{i=n-m+1}^{n} \binom{n}{i} P^i (1-P)^{n-i}$, where $P = 1-(1-p_{IM})^l$ represents the probability that one *share* is lost. In Figure 18, we present the computed schedule success rate when 5% of trustees perform misbehaviors inadvertently. Specifically, in Figure 18(a), by fixing n to 5 and changing m from 1 to 5, it shows that a smaller m, namely lower threshold for restoring *key*, performs higher resistance against inadvertent misbehaviors. By further changing l from 3 to 5, we can find that a smaller l offers better resistance against inadvertent misbehaviors. Then, in Figure 18(b), n is increased to 10. The increment of n enhances the resistance against inadvertent misbehaviors when m and l do not change. Thus, larger l and n while smaller m help maintaining high resistance against inadvertent misbehaviors.

5.3.2 Malicious adversary

We next assume that there exists a malicious adversary aiming at attacking a specific user U while the rest of EOAs are rational adversaries. The malicious adversary may choose to launch either a time difference attack or an execution failure attack. There are two approaches to launch the two types of attacks, namely trustee bribery and Sybil attack [48]. Through trustee bribery, the malicious adversary can deploy a smart contract with a fund larger than the security deposit d and use this smart contract as bait to bribe a trustee, even if the trustee's identity is not known. For example, to obtain a specific trustee's private key for launching a time difference attack, the smart contract can be set with a condition 'If any EOA in the network can submit the private key owned by the trustee who is in charge of (U^{addr}, sid, tid) to the bribery contract before U's execution time window, the EOA can withdraw the fund stored in bribery contract.' Since the fund in bribery contract is larger than the security deposit, a rational trustee may choose to reveal the private key to the bribery contract to increase its profit. Besides, the malicious adversary can create an arbitrary amount of EOAs and make all these EOAs join the trustee candidate pool. This attack approach was named Sybil attack [48].

We now analyze the cost to make either a time difference attack or an execution failure attack successful.

Lemma 3. *To launch a successful execution failure attack, a malicious adversary needs to spend at least $(n - m + 1)d$.*

Proof. To launch a successful execution failure attack, a malicious adversary should aim at impeding the restoration of *key* at execution time, which means at least $n - m + 1$ *shares* should be dropped. The drop of a single *share* may be implemented by either a rational trustee bribed by the malicious adversary or a trustee directly controlled by the malicious adversary through Sybil attack. However, in both the two conditions, due to the existence of the misbehavior report mechanisms, the drop of a single *share* will cost security deposit d, so the cost of dropping $n - m + 1$ *shares* will be at least $(n - m + 1)d$. □

Lemma 4. *A malicious adversary needs to spend at least* $m(l-1)d$ *to bribe trustees for making a time difference attack successful.*

Proof. To bribe trustees for making a time difference attack successful, a malicious adversary should aim at restoring *key* before execution time window w_e, which means at least m *shares* should be obtained before w_e. To obtain a single *share*, the malicious adversary needs to deploy $l-1$ bribery smart contracts to collecting private keys from $l-1$ different trustees, which, due to the existence of the misbehavior report mechanisms, will cost at least $(l-1)d$. Therefore, the cost of obtaining m *shares* before w_e will be at least $m(l-1)d$. $\square$

Lemma 5. *Through Sybil attacks [48], the expected value of security deposit that a malicious adversary needs to pay to launch a successful time difference attack is* $(l-2)vd$*, where* v *denotes the number of trustee candidates available to user* U *that are not controlled by this malicious adversary.*

Proof. The situation refers to the *malicious trustee* method introduced in Section 5.1.4, where the trustee candidates available to user U during *user schedule* component can be divided into two parts. By denoting the number of rational candidates not controlled by the malicious adversary as v and the number of malicious candidates injected by the malicious adversary as x, we get $p_M = \frac{x}{x+v} \to x = \frac{vp_M}{1-p_M}$, where p_M denotes the percentage of malicious candidates. To obtain a single *share*, all the $l-1$ trustees providing private keys for encrypting this *share* to an *onion* should be selected from malicious candidates, which has the probability p_M^{l-1}. Since there are n shares in total, the overall process can be viewed as a Binomial distribution $B(n, p_M^{l-1})$ with mean np_M^{l-1}. Then, the expected amount of security deposit $\widehat{d}$ that should be paid by the malicious adversary to make $np_M^{l-1} = m$ can be computed with $\frac{\widehat{d}}{xd} = \frac{m}{np_M^{l-1}}$, which makes $\widehat{d} = x \cdot \frac{dm}{np_M^{l-1}} = \frac{vp_M}{1-p_M} \cdot \frac{dm}{np_M^{l-1}} = \frac{vdm}{n} \cdot \frac{p_M^{2-l}}{1-p_M}$. Since the malicious adversary cannot control (v, d, m, n), to minimize $\widehat{d}$, we set $f(p_M) = \frac{vdm}{n} \cdot \frac{p_M^{2-l}}{1-p_M}$ and compute $f'(p_M) = 0$, which gives $\frac{(2-l)p_M^{1-l}}{1-p_M} + \frac{p_M^{2-l}}{(1-p_M)^2} = 0 \to p_M = \frac{l-2}{l-1}$ Therefore, when $\widehat{d}$ is minimized: $x = v \cdot \frac{l-2}{l-1} \cdot (l-1) = (l-2)v \to \widehat{d}_{min} = (l-2)vd$ $\square$

For example, when $v = 10000$, $l = 4$ and $d = \$100$, $\widehat{d}_{min}$ will be four million dollars. In Figure 19, we present the computed schedule success rate when 50% of trustees are controlled by a malicious adversary who aims at launching a time difference attack. As can be seen, smaller m while larger n and l help enhancing the resistance against time difference attacks performed through Sybil attack.

5.4 IMPLEMENTATION

In this section, we present the implementation of the proposed protocol and discuss the experimental evaluation of the proposed mechanism in Ethereum.

5.4.1 Implementation of protocol

We first introduce the implementation setup and then present both key off-chain functions in node.js and on-chain functions in *Solidity* [11] and demonstrate how they work in practice. After that, we present two test instances used in our experimental evaluation.

Setup: We programmed the smart contracts in *Solidity* [11], the most commonly used smart contract programming language, deployed them to the Ethereum official test network *rinkeby* [9] and tested them with Ethereum official Go implementation *Geth* [6]. Our experiments are performed on an Intel Core i7 2.70GHz PC with 16GB RAM.

Implemented functions: The protocol primarily relies on 6 off-chain functions shown in Table 11 and 15 on-chain functions shown in Table 12. In both the tables, we show the components and steps where each function works in protocol. For example, function *share*() is used in step 5 of *user schedule* component to split *key* to n *shares* using Shamir secret sharing [114].

- ***Trustee application***: Any EOA in the network can invoke *newCandidate()* to join the trustee candidate pool maintained by scheduler contract C_s.
- ***User schedule***: Any EOA can invoke *newUser()* to be recorded as a user and then set up new schedule through *newSchedule()*. Then, during whisper communication with trustees, $h(onion)$ should be submitted to C_s through *setOnion()* while $h(T^{addr}, R_T)$ and $h(IN, R_U)$ should be submitted to C_s through *setTrustee()*. Meanwhile, the generation of

Table 11: Key off-chain functions in node.js, *share()* and *combine()* are in secrets.js [113], *ecsign()* is in ethereumjs-util [53], *encrypt()* and *decrypt()* are in eth-ecies [51], *soliditySha3()* is in web3-utils [127]

Component	**Step**	**Function**	**Purpose**
Schedule	5	share	split *key* to *shares*
	7,11	ecsign	sign data with private key
	8	encrypt	encrypt *shares* to *onions*
	9,13	soliditySha3	compute keccak256 hash
Execute	3	combine	combine *shares* to *key*
	4	decrypt	decrypt *onions* to *shares*

shares, signatures, *onions* and hash values are completed by *share()*, *ecsign()*, *encrypt()*, *soliditySha3()* in node.js, respectively.

- ***Function execution***: A trustee can submit private key and *onion* through *submitPrivkey()* and *submitOnion()*, respectively. Then, after decrypting *onions* to *shares* through *decrypt()* and combining *shares* to *key* through *combine()*, any trustee has the ability to make the target function be executed through *execute()*. Finally, after the execution window is over, trustees can withdraw deposit and remuneration through *withdrawD()* and *withdrawR()*, respectively.
- ***Misbehavior report***: The four types of report mechanisms are implemented by *identityReport()*, *advanceReport()*, *absentReport()* and *fakeReport()*, respectively. Then, after the execution window is over, reporters can withdraw reward through *withdrawA()*.

Test instance: We design two test instances A and B as in Table 13: Instance A employs 15 trustees while instance B employs 40 trustees. As a result, instance B has higher schedule success rate under both 5% inadvertent misbehaviors (IM) and 50% malicious (M) trustees. Besides, based on Lemma 1 (L1), Lemma 2 (L2) and Lemma 3 (L3), the cost of malicious

Table 12: Key on-chain functions in solidity, the three colored functions are in proxy contract C_p, the rest of the functions are in scheduler contract C_s

Component	**Step**	**Function**	**Purpose**
Apply	2,3	newCandidate	join candidate pool
Schedule	4	newUser	register as a new user
	4	newSchedule	initialize a new schedule
	9	setOnion	submit hashes of onions
	13	setTrustee	submit hashes of trustees
Execute	1,2	submitPrivkey	submit private key
	1,2	submitOnion	submit onion
	6	execute	execute the target contract
	7	withdrawD	withdraw security deposit
	7	withdrawR	withdraw remuneration
Report	1,2	identityReport	report identity disclosure
	3,4	advanceReport	report advance disclosure
	5,6	absentReport	report absent trustee
	7,8	fakeReport	report fake submission
	2,4,6,8	withdrawA	withdraw report award

Table 13: Test instances

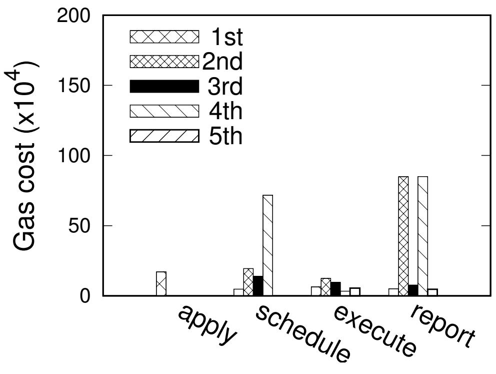
Instance	**l,m,n**	**5% IM**	**50% M**	**L1**	**L2**	**L3**
A	3,2,5	99.82%	98.44%	4d	4d	vd
B	4,4,10	99.95%	99.99%	7d	12d	2vd

adversaries in instance B is higher than that in instance A. However, since instance B requires more trustees in instance A, user U's cost in instance B is also higher than that in instance A, which is the price of stronger security guarantee. In both instance A and B, we use the *SealedBidAuction* contract [129] as the target contract C_t and we assumed user's goal was to schedule a transaction calling function $reveal(amount, nonce)$. Specifically, we designed an input parameter *time* to simulate the time during testing.

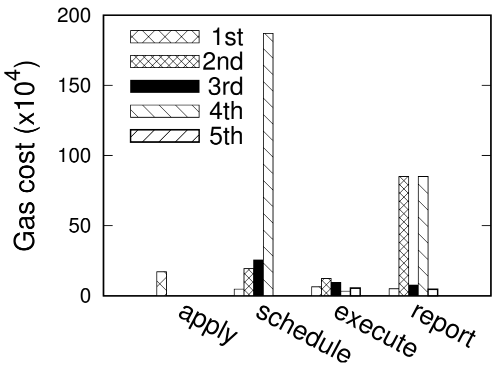
5.4.2 Experimental evaluation

We use the presented test instances to experimentally evaluate the performance of the smart contracts, namely the gas cost and time overhead of each function presented in Table 12.

Gas cost: Gas is spent in Ethereum for deploying smart contracts or calling functions. The gas costs of functions in Table 12 for instance A and B are shown in Figure 20(a) and Figure 20(b), respectively. For ease of presentation, results are grouped into four clusters. Each cluster represents a protocol component and contains a group of functions following their order in Table 12. As can be seen, most functions cost very little. Specifically, among the fifteen functions, eight cost lower than 10^5 gas and eleven cost lower than 2×10^5 gas. Among the rest four functions, both *advanceReport()* and *fakeReport()* cost around 8.5×10^5 because the two functions need to derive public key from private key on chain. Gas costs of the last two functions, namely $setOnion()$ and $setTrustee()$, change with n and nl, respectively. From instance A to B, l increases from 3 to 4 and n increases from 5 to 10. As a result, gas cost of $setOnion()$ increases from 1.40×10^5 to 2.55×10^5 and gas cost of



(a) Instance A



(b) Instance B

Figure 20: Gas cost

settrustee() increases from 7.17×10^5 to 1.87×10^6.

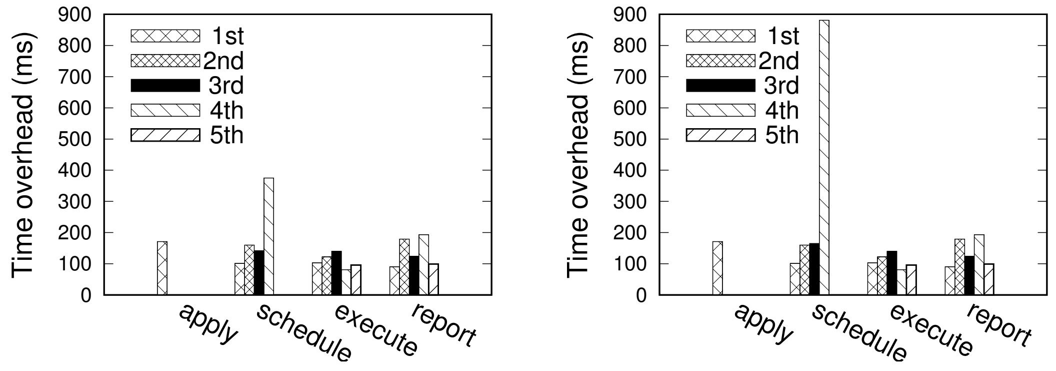
To complete a schedule, some functions need to be invoked for multiple times. In Table 14, we show the number of times that each function needs to be invoked in a single schedule when there is no report needed:

Besides, the gas cost of deploying proxy contract C_p is about 1.33×10^6. Therefore, the total gas costs of instance A and B are 7.60×10^6 and 1.72×10^7, respectively. Both gas price and Ether price keeps dramatically swinging [5]. For example, based on prices of date 12/5/2016, instance A and B cost \$1.2 and \$2.72, respectively. However, based

Table 14: Call count of functions in a single schedule

Function	**No.**	**Function**	**No.**	**Function**	**No.**
newCandidate	nl	setTrustee	1	execute	1
newSschedule	1	submitPrivkey	$n(l-1)$	withdrawD	nl
setOnion	1	submitOnion	n	withdrawR	nl

on prices of date 10/29/2017, the two instances cost \$22.8 and \$51.6, respectively. As can be seen, the monetary cost of a timed-execution service is highly influenced by the fluctuation of cryptocurrency market, which may be a common limitation of cryptocurrency-based applications.



(a) Instance A (b) Instance B

Figure 21: Time overhead

Time overhead: The time overheads of functions in Table 12 for instance A and B are shown in Figure 21(a) and Figure 21(b), respectively. All results are averaged for 100 tests. Among the fifteen functions, fourteen functions spend 0-200ms. It is the function *setTrustee()* that spends more time to record information of all the trustees to the blockchain. Specifically, *setTrustee()* spends 375ms for instance A while 881ms for instance B as there are more trustees in instance B.

5.5 SUMMARY AND DISCUSSION

In this chapter, we develop a new decentralized privacy-preserving timed execution mechanism that allows users of Ethereum-based decentralized applications to schedule timed transactions without revealing sensitive inputs before an execution time window chosen by the users. The proposed approach involves no centralized party and allows users to go offline at their discretion after scheduling a timed transaction. The timed execution mechanism protects the sensitive inputs by employing a set of trustees from the decentralized blockchain network to enable the inputs to be revealed only during the execution time. We implement the proposed approach using *Solidity* and evaluate the system on the Ethereum official test network. Our rigorous theoretical analysis and extensive experiments validate the security properties and demonstrate the low gas cost and low time overhead associated with the proposed approach.

Among the three research components proposed in Chapter 1.1, namely *Infrastructure*, *Output* and *Input*, we have discussed *Infrastructure* in Chapter 3 and Chapter 4 and also *Output* in this chapter. In the next chapter, we will examine the last component *Input* and look for a solution that can support the cost-effective gradual release of self-emerging data.

6.0 GRADUAL RELEASE OF PRIVATE DATA OVER TIME

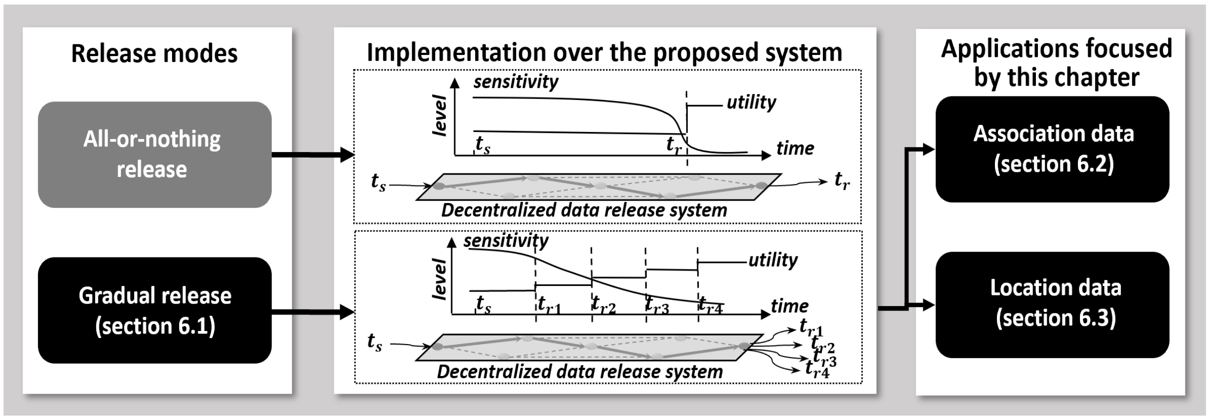

Release modes
All-or-nothing release
Gradual release (section 6.1)
Implementation over the proposed system
sensitivity
utility
level
time
t_s
t_r
Decentralized data release system
t_{r1}
t_{r2}
t_{r3}
t_{r4}
Applications focused by this chapter
Association data (section 6.2)
Location data (section 6.3)

Figure 22: All-or-nothing release and gradual release

We have investigated decentralized mechanisms of releasing self-emerging data with DHT infrastructure and blockchain infrastructure respectively in Chapter 3 and Chapter 4 and have also examined approaches of outputting self-emerging data to smart contracts in Chapter 5. In this chapter, we explore the research task *T-4*, namely the last research task proposed in this dissertation, which aims at developing techniques to support gradual release of self-emerging data in a cost-effective way. As shown in Figure 22, depending on how the data sensitivity changes over time, an application requiring to release self-emerging data may choose between two schemes:

- *All-or-nothing release*: It is used when data sensitivity suddenly drops at a future time point, allowing the data user (i.e., recipient) gets nothing useful before the time while learning what he expects only after the time. Data is released for a single time.
- *Gradual release*: It is used when data sensitivity gradually reduces over time, allowing the data utility to keep increasing along with the continuously dropping data sensitivity.

Data is released for multiple times.

We notice that all-or-nothing release is a special case of gradual release, so we only focus on the more challenging gradual release of private data in the rest of this chapter.

Next, depending on the ways of inputting private data to the designed systems, the cost of gradual release using the systems designed in the previous chapters could be quite different. As we have discussed in Section 1.1, an application can choose among three options to input the private data into the system, namely the *plaintext*, its *encryption key* or its *perturbation key*. Obviously, simply inputting the *plaintext* of private data will result in both high storage cost and high communication cost when the size of private data is large (e.g., a healthcare dataset). Therefore, in the rest of this chapter, we first explore the rest two options in Section 6.1, namely *encryption key* and *perturbation key*, to determine a cost-effective approach for the gradual release of private data. After that, in Section 6.2 and Section 6.3, we develop techniques of implementing the cost-effective approach in two representative scenarios, namely association data disclosure and location data disclosure, respectively. Finally, we summarize this chapter in Section 6.4.

6.1 COST-EFFECTIVE GRADUAL RELEASE OF PRIVATE DATA

In this section, we first present an approach of using *encryption keys* for gradual release of private data and then discuss the way of reducing the cost by replacing *encryption keys* with *perturbation keys*.

The approach of using *encryption keys* for gradual release is shown in Figure 23. In this approach, at the initial time point t_A, the data owner (i.e., sender) can operate a specific privacy-preserving data perturbation technique (e.g., [36, 37, 38, 49, 54, 71, 73, 85, 66]) over the private data for multiple times so that multiple snapshots of the private data with different perturbation levels (and thus different utility levels) can be generated. Then, after encrypting all the snapshots with different encryption keys (denoted as $EKey$), the encryption keys (except $EKey_1$) and encrypted snapshots should be sent into the decentralized self-emerging data release system and a cloud storage platform respectively. Specifically,

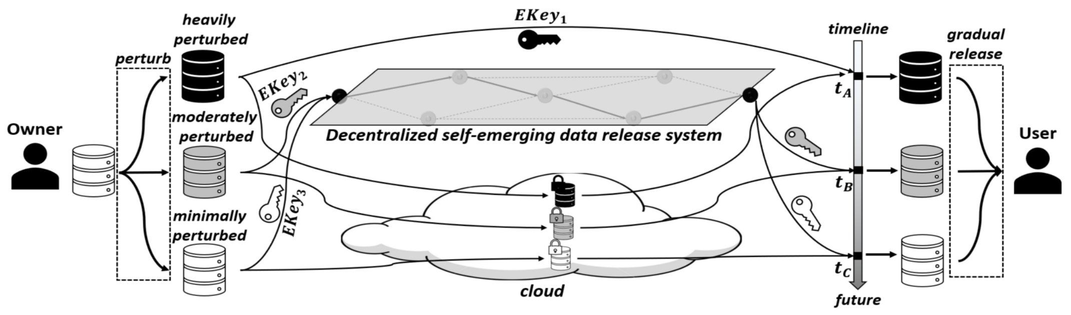


Figure 23: Using *encryption keys* for gradual release of private data

$EKey_1$ can be directly used by the data user (i.e., recipient) to get the most heavily perturbed snapshot. After that, the data sensitivity may keep dropping as time goes by. At a future time point t_B, the encryption key $EKey_2$ will be released by the decentralized self-emerging data release system, allowing the data user to decrypt the moderately perturbed snapshot in cloud and thus obtaining more useful information from it. Similarly, at an even remoter time point t_C, the released encryption key $EKey_3$ will allow the user to gain further information from the decrypted minimally perturbed snapshot.

However, we find that a major limitation of using *encryption keys* is the cost for storing multiple encrypted snapshots of the dataset. In case that the Amazon S3 cloud storage service is used (0.023 USD/GB per month), to release one 100GB snapshot per month for one year (i.e. first month: release one snapshot, store the rest eleven snapshots; second month: release one snapshot, store the rest ten snapshots...), the storage cost will be about 150 USD. We believe such a high storage cost is unnecessary, so we further proposed the cost-effective approach shown in Figure 24. The key idea behind this approach is to develop a new class of reversible perturbation techniques that can use *perturbation keys* (denoted as $PKey$) to pseudo-randomly perturb data so that these keys, once being released in future, can be used to directly de-perturb the single snapshot held by the user to reduce its perturbation level. We say the perturbation level of such kind of snapshots is reversible because it can be reduced by *perturbation keys* in future. In Figure 24, at t_A, with the reversible perturbation techniques, the *perturbation key* $PKey_2$ pseudo-randomly perturbs the minimally perturbed snapshot to the moderately perturbed snapshot. Then, $PKey_1$ further pseudo-randomly perturbs the moderately perturbed snapshot to the heavily perturbed snapshot. At this

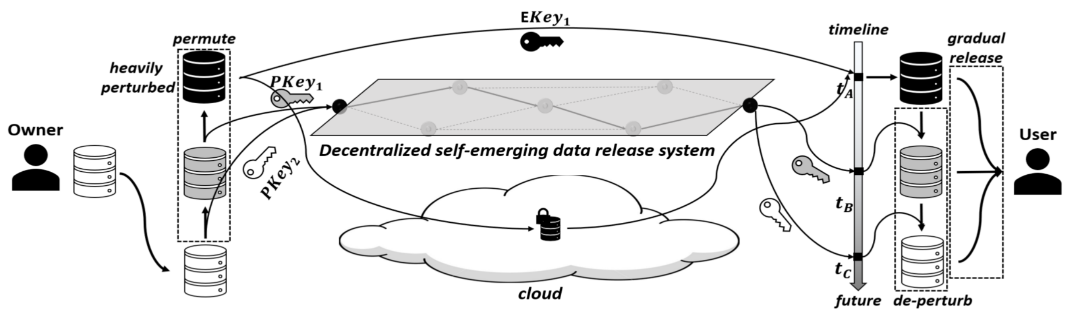


Figure 24: Cost-effective gradual release of private data by using *perturbation keys*

phase, all the snapshots except the most heavily perturbed one can be deleted and only the encrypted heavily perturbed snapshot should be stored in cloud. Also, the data owner should send the *perturbation keys* into the decentralized self-emerging data release system while sending the encryption key of the heavily perturbed snapshot directly to the data user. After that, at future time point t_B, with the released $PKey_1$, the user can de-perturb the heavily perturbed snapshot to the moderately perturbed snapshot. Similarly, at t_C, the minimally perturbed snapshot can be recovered from the moderately perturbed snapshot using the released $PKey_2$. Compared with the approach of using *encryption keys*, the cost-effective approach only needs to maintain one snapshot as the 'seed' of all other snapshots, thus significantly reducing the cost.

In the next two sections, we present techniques of gradually releasing two types of commonly used data, namely association data (Section 6.2) and location data (Section 6.3), using the cost-effective approach presented in this section. At the core of these techniques is the use of pseudo-randomness created by *perturbation keys* in the privacy-preserving data perturbation mechanisms. In order to making the process of data perturbation reversible, we propose to apply *perturbation keys* as the seeds of a generator of pseudo-randomness and replace any randomness involved in conventional data perturbation mechanisms with such pseudo-randomness so that the same keys, upon being released by the designed system, could be used by the recipients of self-emerging data to reverse the perturbation process and reduce the perturbation level. In both the two following sections, we first introduce the relevant concepts and then present details of using *perturbation keys* to develop privacy-preserving data perturbation mechanisms that support the cost-effective approach of gradually releasing data from the designed system. Finally, we experimental evaluate the designed approaches.

6.2 GRADUAL RELEASE OF ASSOCIATION DATA

In this section, we present details of gradually releasing association data from the designed systems using *perturbation keys*. Private association data is usually published via privacy-preserving data perturbation schemes, where the raw data is perturbed to meet the privacy requirements before the data is published. However, conventional privacy-preserving data perturbation schemes have focused on publishing a single snapshot of a dataset that offers a fixed privacy level, thus failing to support the gradual release of private data [49, 54, 71, 73]. In order to applying the cost-effective approach to the gradual release of association data, we develop a set of techniques of multi-level reversible association data perturbation that use *perturbation keys* to control the sequential generation of multiple snapshots of the perturbed data to offer multi-level access based on privacy levels, thus allowing only the *perturbation keys* to be sent into the self-emerging data release system and only a single snapshot of perturbed dataset to be maintained. Extensive experiments on real association dataset show that our techniques are efficient and scalable.

6.2.1 Overview of Concepts and Models

In this subsection, we first model bipartite association graphs and introduce the definitions of differential privacy. We then model the multilevel reversible association data privacy.

Bipartite Association Graph Model: Private data in real world often arises in the form of associations between entities such as the drugs purchased by patients in a pharmacy store or the movies rated by viewers in a movie rating database or the products purchased by buyers in an online shopping website [33, 63, 65]. Such associations are best captured as bipartite association graphs with nodes representing the entities (e.g., drugs and patients) and the edges correspond to the associations between them (e.g., Patient Bob purchased the Insulin drug). A bipartite graph can be represented as $BG = (V, W, E)$, which consists of $m = |V|$ nodes of a first type, $n = |W|$ of a second type and a set of edges $E \subseteq V \times W$.

Thus, a bipartite graph can represent a set of two-node pairings, where a two-node pairing (a, b) represents an edge between node $a \in V$ and node $b \in W$.

Differential privacy: Differential privacy [49] is a state-of-the-art privacy paradigm that makes conservative assumptions about the adversary's background knowledge and protects a single individual's information in a dataset by considering adjacent datasets which differ only in one record. The conventional (individual) differential privacy protects the inference of a single individual's information in a dataset. For example, in a bipartite graph representing the associations between drugs and patients, such a single individual's protection may correspond to the inference of the graph edge representing a patient (e.g., 'Alice') purchasing a drug (e.g.,'Citalo'). For the purpose of protecting sensitive information of a group of individuals (e.g., the total number of cancer medicines purchased by patients in a specific neighborhood), differential privacy can be further extended to support group data protection based on the notion of group differential privacy [105]. Group differential privacy protects sensitive aggregate information about groups of records using higher noise injection and perturbation. When records of a dataset are grouped into larger groups, the transformed dataset will provide coarser aggregate information and the privacy offered by group differential privacy will be stronger. Therefore, by grouping the records of a dataset into multiple granularity levels, different privacy levels can be offered by implementing group differential privacy at different granularity levels in the dataset. In this section, we employ both individual and group differential privacy to provide multi-level disclosure of the association data using a single instance of the perturbed dataset. For more details about individual and group differential privacy, please refer to [105].

Multilevel reversible association data privacy: We would like techniques developed in this section to support the multi-level reversible association data privacy, which can be viewed as a sequence of permutation and noise injection steps. Figure 25 illustrates the process with an example bipartite graph where the original bipartite graph is shown as snapshot S_0, which consists of eight left (patient) nodes denoted by *PID*, eight right (medicine) nodes denoted by *MID* and eleven edges representing which medicine was purchased by which patient. To protect group differential privacy, the bipartite graph is first partitioned into multiple levels of subgraphs representing different granularity levels based on a taxonomy tree or some

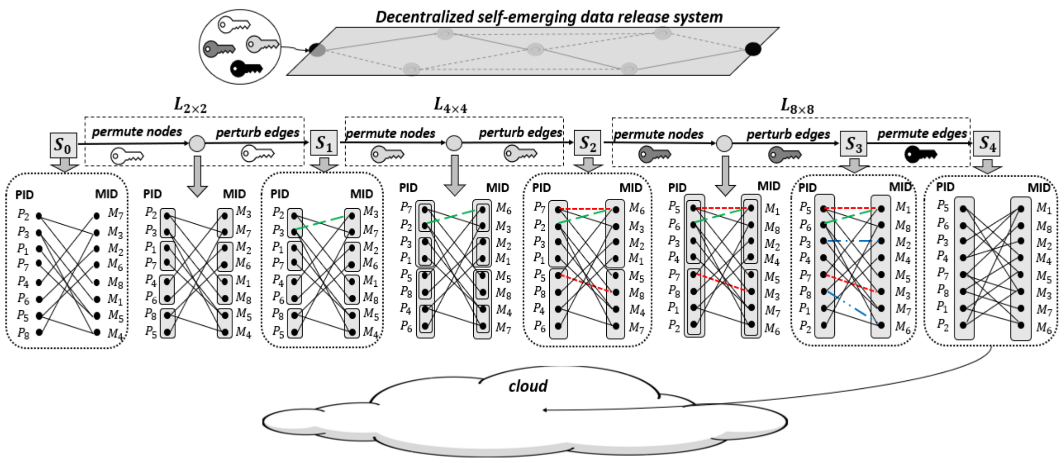


Figure 25: Multilevel reversible association data privacy

granular subgraph generation techniques such as the one presented in [105]. In the example of Figure 25 , at level $L_{2\times2}$, both the left and right nodes are grouped into groups of two nodes and thus it generates sixteen subgraphs. Similarly, at level $L_{4\times4}$ and level $L_{8\times8}$, nodes are grouped into four subgraphs and a single graph, respectively. Based on the partitioning, dataset owners (i.e., senders of self-emerging data) can choose to make a lightly perturbed snapshot, S_1 at $L_{2\times2}$, a moderately perturbed snapshot, S_2 at $L_{4\times4}$ and a heavily perturbed snapshot, S_3 at $L_{8\times8}$. To generate each perturbed snapshot mentioned above, we propose to implement one step of (node) permutation followed by one step of (edge) perturbation. The purpose of node permutation is to ensure information generalization. For example, at snapshot S_0, left nodes P_2, P_3 and right nodes M_7, M_3 form a subgraph contained by $L_{2\times2}$, which has a single edge (P_2, M_3). Without node permutation, specific information in S_0, such as the edge (P_2, M_3) that indicates P_2 purchased M_3, can be viewed by users who only have privilege to view S_1 to learn generalized information about subgraphs at $L_{2\times2}$. In contrast, by permuting P_2, P_3 and also by permuting M_7, M_3 within their size-two groups, the label M_3 is swapped with M_7. Thus, instead of edge (P_2, M_3), a fake edge (P_2, M_7) indicating incorrect specific information is contained in S_1, whereas generalized information about the subgraph is still maintained in S_1. This process is followed by the edge perturbation process which aims to prevent specific information to be inferred from the generalized information in the exposed snapshot. For example, after node shuffling, the subgraph between P_2, P_3 and M_3, M_7 shows generalized information that one patient between P_2 and P_3 has purchased one medicine between M_3 and M_7. It has four possibilities, namely (P_2, M_3), (P_2, M_7), (P_3, M_3) and (P_3, M_7). An adversary with some background knowledge may infer that (P_2, M_7), (P_3, M_3) and (P_3, M_7) cannot exist and therefore will be able to conclude the existence of edge (P_2, M_3) from the generalized information. To address this concern, edge perturbation can be used to perturb the edges of each subgraph based on randomized differential privacy mechanisms (e.g., Laplace Mechanism [49]). In the example, users receiving S_1 can also view the injected edge (P_3, M_3) guaranteeing differential privacy, thus feeling uncertain to conclude the existence of (P_2, M_3). After the two steps, S_1 can be generated, which reveals generalized information about subgraphs at $L_{2\times2}$ while protecting individual information in S_0. Similarly, S_2 and S_3 reveal $L_{4\times4}$ information while protecting specific information of $L_{2\times2}$ and $L_{8\times8}$ information while protecting information of $L_{4\times4}$, respectively. At the end of the encoding phase, if S_3 still contains sensitive information about $L_{8\times8}$ that the data owner is not willing to share to all possible users, edge permutation can be executed over S_3 to permute all the edges in S_3 so that the obtained S_4 is safe for disclosure.

6.2.2 Reversible association data perturbation

To generate snapshot S_i from S_{i-1} as shown in Figure 25, a *perturbation key* is used to first pseudo-randomly permute the two sides of nodes of each subgraph and then pseudo-randomly perturb edges within each subgraph. Also, edge permutation at the last step can be pseudo-randomly implemented using a *perturbation key*. Next, we show how to use *perturbation keys* to perform edge perturbation, node permutation and edge permutation so that legitimate users (i.e., recipients of self-emerging data) can use *perturbation keys* to reverse S_4 to any previous snapshots (e.g., S_0, S_1, S_2, S_3) containing finer information. The pseudo-codes of edge perturbation, node permutation and edge permutation are presented in Appendix C.

Reversible edge perturbation: For each subgraph, the reversible edge perturbation step first uses the *perturbation key* to pseudo-randomly sample a noise using the Laplace Mechanism [49]. Specifically, during noise injection, the number of injected edges is sampled from Laplace pseudo-random value generator with mean 0, variance $\triangle f/\epsilon$ and seed K, where $\triangle f$ and ϵ denote the sensitivity and budget of Laplace Mechanism [49] and K refers to the *perturbation key*. After this process, legitimate users can receive the *perturbation key* to reversibly remove the injected noise. With the same seed K, same n can be generated, which can then select and remove the same sequence of edges.

Reversible node permutation: For each subgraph, the reversible node permutation step uses the *perturbation key* to pseudo-randomly shuffle node labels (e.g., PID, MID) during the encoding phase and later uses the same key to recover their order during the decoding process. During the encoding phase, the *perturbation key* is used as a seed of the pseudo-random stream generator to generate a sequence of pseudo-random numbers, which is then used to shuffle the left nodes and right notes of the subgraph. Specifically, each pseudo-random number swaps two left (right) nodes. The first node between the two is selected from top to bottom along with its position while the second node is pseudo-randomly selected by the pseudo-random number using modular arithmetic. At the end of encoding phase, both left nodes and right nodes have been shuffled in a reversible manner. Later, during decoding phase, given the same key, the same sequence of pseudo-random numbers can be obtained to recover left nodes and right nodes of the subgraph. Instead of starting from top to bottom, the decoding process starts from bottom to top with a reverse order so that operations implemented during encoding can be reversibly implemented during decoding, which results in the recovery of the original subgraph. In Figure 26, the labels of the nodes are permuted through reversible node permutation while the edges are permuted through reversible edge permutation (to be discussed later). In the example, Alice uses a *perturbation key* as a seed to the pseudo-random stream generator to get a sequence of pseudo-random numbers R, where the first six pseudo-random numbers (assumed to be $[35, 18, 46, 12, 27, 57]$) and second six pseudo-random numbers (assumed to be $[7, 18, 24, 29, 62, 67]$) in R are used to shuffle the left and right nodes of the bipartite graph respectively. Then, the first pseudo-random number $R_1 = 35$ swaps P_2 and P_3, followed by 18 swapping P_3 and P_6, 46 swapping P_4 and P_5, 12 swapping P_6 and P_5, 27 swapping P_4 and P_6, 57 swapping P_2 and P_4. As a result, left nodes in the left bipartite graph are permuted to the order in the right bipartite graph. Later, in Figure 27, Bob gets the *permutation key* from Alice and uses the key as a seed and generates the same R as generated by Alice. Among the first six pseudo-random numbers $[35, 18, 46, 12, 27, 57]$, $R_6 = 57$ is first picked to swap P_2 and P_4, followed by 27 swapping P_4

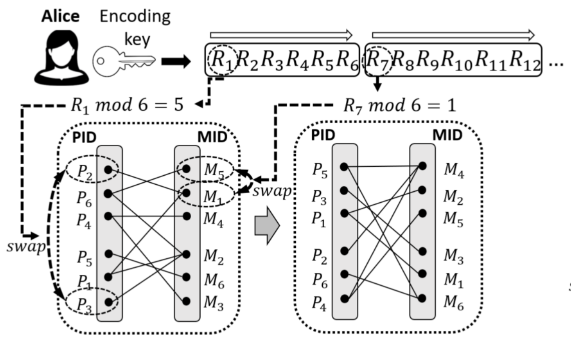

Alice
Encoding key
$R_1 R_2 R_3 R_4 R_5 R_6$ $R_7 R_8 R_9 R_{10} R_{11} R_{12}$ ...
R_1 mod 6 = 5
R_7 mod 6 = 1
PID
MID
swap
swap

Figure 26: Encoding

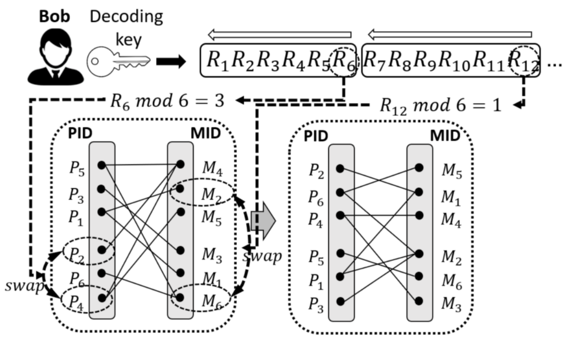

Bob
Decoding key
$R_1 R_2 R_3 R_4 R_5 R_6$ $R_7 R_8 R_9 R_{10} R_{11} R_{12}$ ...
R_6 mod 6 = 3
R_{12} mod 6 = 1
PID
MID
swap
swap

Figure 27: Decoding

and P_6, 12 to swapping P_6 and so on. Therefore, the original order of the left nodes can be recovered.

Reversible edge permutation: Edge permutation is implemented as the last step in the encoding phase and therefore it represents the first step during the decoding phase. The edges of the bipartite graph are represented using an adjacency matrix. For example, the adjacency matrix of the left bipartite graph in Figure 26 is shown as the matrix E below, where the first row represents that P_2 is linked with M_1. Here, the edges can be shuffled by simply permuting the adjacency matrix.

$$E = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \quad \overline{E} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Similar to node permutation, in both the encoding and decoding phase, the same sequence of pseudo-random numbers can be obtained through the same *perturbation key*. Then, given the adjacency matrix of size $|V||W|$, we use the first $|V||W|$ pseudo-random numbers to perform $|V||W|$ rounds of swap operation. Each time, the first edge is selected based on a fixed order (top to bottom and left to right during encoding phase, right to left and bottom to top during decoding phase) and the second edge is pseudo-randomly selected by the pseudo-random number using modular arithmetic. In this way, by reversibly performing the swap operation during the decoding phase, the original order of the edges can be recovered. In Figure 26, if the first and second pseudo-random numbers generated by a key are 53 and 71, we first use 53 to swap $E[\lfloor \frac{0}{6} \rfloor][0 \ mod \ 6] = E[0][0]$ and $E[\lfloor \frac{53 \ mod \ 36}{6} \rfloor][(53 \ mod \ 36) \ mod \ 6] = E[2][5]$. Then, we use 71 to swap $E[0][1]$ and $E[5][5]$. By repeating this for all the 36 pseudo-random numbers, the adjacency matrix can be transformed as $\overline{E}$ to represent the right bipartite graph in Figure 26.

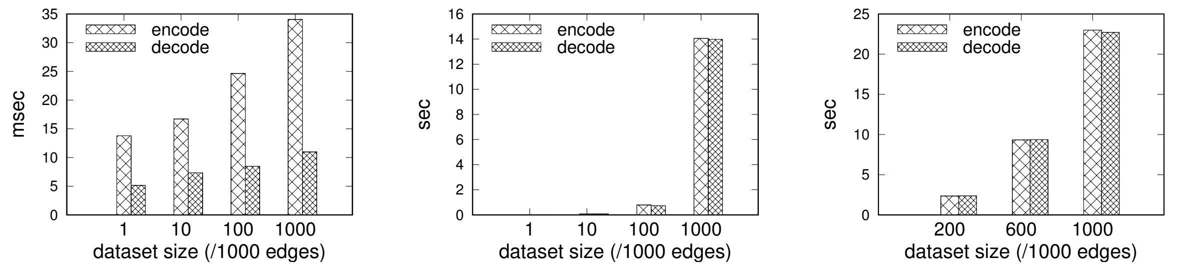
6.2.3 Experimental Evaluation

In this subsection, we present the experimental study on the performance of the proposed reversible data perturbation techniques. We first briefly describe the experimental setup.

Experimental setup: Our experiment setup was implemented in Java with an Intel Core i7 2.70GHz 16GB RAM PC. The bipartite graph dataset used in this work is the MovieLens dataset [63] which consists of 6,040 users (left nodes), 3,706 movies (right nodes) and 1,000,209 edges describing rating of movies made by users.

Experimental results: The experimental results are organized into two parts. First, we evaluate the performance efficiency of the three key components of the reversible perturbation process separately, namely edge perturbation, node permutation and edge permutation. Then, we integrate the three components and evaluate the performance of the complete reversible data perturbation process. In our experiments, we generate three granularity levels and evaluate the time and space consumption for each granularity level during encoding and decoding phases.

The first set of experiments evaluates the performance efficiency of edge perturbation, node permutation and edge permutation separately. We evaluate the scalability of these algorithms by varying the size of dataset and we measure the time taken for their execution both in encoding and decoding phases. In Figure 28(a), edge perturbation is evaluated. The dataset size is changed from one thousand edges to one million edges. Specifically, the one-million-edge dataset represents the entire MovieLens dataset. The results show that both noise injection (encode) and removal (decode) processes have significantly low time consumption cost and demonstrate high scalability. Even when the dataset size increases 1000 times, the time consumption increases only by a factor of 2. For a dataset with one

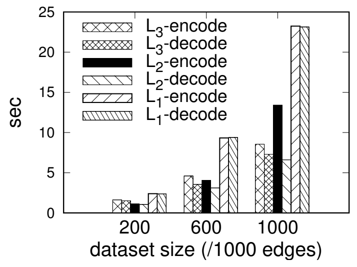


(a) edge perturbation (b) node permutation (c) edge permutation

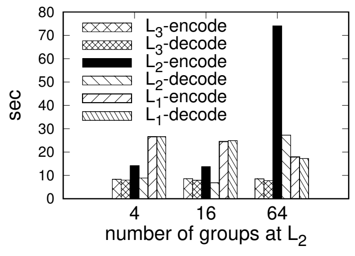
Figure 28: Algorithm performance

million edges, the noise injection and removal processes cost only about 35ms and 10ms respectively. Here, compared with noise injection, the noise removal process usually has a lower time consumption. This is because the process of noise removal employs some meta data information attached to the perturbed dataset which significantly accelerates its speed of execution. Next, in Figure 28(b), we evaluate the node permutation process with the same experiment setting. Unlike edge perturbation, although the time consumption of node permutation is significantly small for small datasets, it becomes acceptably larger for the one-million-edge dataset, which is about 14s. Finally, in Figure 28(c), we measure the time taken by the edge permutation process using dataset sizes that vary from 0.2 million edges to 1 million edges. The results show that the time taken by the process for the one-million-edge dataset is about 23s, which is quite acceptable as the edge shuffling process is only required to be run once during the entire process.

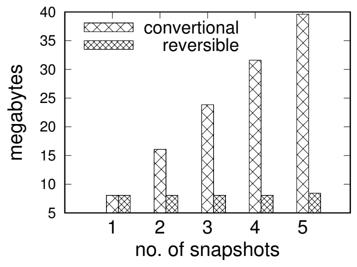
The second set of experiments evaluates the performance of the multiple levels of perturbation during the process. In this part, we processed the dataset to generate three granularity levels of subgraphs, denoted as L_1, L_2 and L_3 respectively. We applied the *DiffPar* partitioning algorithm [105] to generate the granularity levels. The algorithm runs several rounds of specializations and each specialization can partition a bipartite graph into four non-overlapping subgraphs. Therefore, after n rounds of specializations, the original bipartite graph has been partitioned to 4^n non-overlapped subgraphs. In this experiment, we use the MovieLens dataset and we consider the entire graph as level L_1, the 16 (4^2) subgraphs generated by two specializations as level L_2 and the 256 (4^4) subgraphs generated by four specializations as level L_3. For ease of understanding, L_1, L_2 and L_3 can be considered to



(a) time consumption w/ varying size



(b) time consumption w/ varying partitioning



(c) comparison of storage cost

Figure 29: Multi-level performance

roughly correspond with $L_{8\times8}$, $L_{4\times4}$ and $L_{2\times2}$ in the example of Figure 25. In Figure 29(a), we evaluate the encoding and decoding time for each granularity level when the dataset size is varied from 0.2 million edges to 1 million edges. As can be seen, as the dataset size increases, the time taken by all the three granularity levels also show a reasonable increase. Level L_3 needs to run edge perturbation and node permutation over the 256 subgraphs. Due to the very small subgraph size and the low sensitivity for protecting differential privacy for individual edges, L_3 has the lowest time consumption. At level L_2, although the number of subgraphs reduces to 16, the corresponding increase in subgraph size makes its time consumption higher than that of L_3 for large dataset size. Finally, the time consumption of level L_1 is dominated by edge permutation, which follows the same trend as shown in Figure 28(c). In Figure 29(b), we fix the dataset size as one million edges while changing the number of subgraphs at level L_2 from 16 to 4 and 64. This change at L_2, as shown in Figure 29(b), has no influence on the results of L_3. The reduction from 16 to 4 makes an increase for both L_2 and L_1 while the increase from 16 to 64 makes results at L_2 significantly increased and results at L_1 obviously decreased. These results show that instead of the average size of subgraphs, the time consumptions of granularity levels are much more sensitive to the amount of the injected noises. Finally, in Figure 29(c), we compare the storage cost required by conventional framework and reversible framework. Based on Figure 25, three granularity levels can generate at most five snapshots. As can be seen, using the conventional framework, the storage cost is linearly increased with the number of generated snapshots as data owner needs to store all of them. However, the proposed reversible framework efficiently employs the use of *perturbation keys* to allow all snapshots to be recovered from a single published snapshot protected with the highest privacy level. Thus, the data owner only needs to store one snapshot. The size of the *perturbation keys* and the stored metadata for noise injection have little influence on the overall storage cost.

6.3 GRADUAL RELEASE OF LOCATION DATA

In this section, we present details of gradually releasing location data using *perturbation keys*. Location anonymization refers to the process of perturbing the exact location of users as a spatially cloaked region such that a user's location becomes indistinguishable from the location of a set of other users. However, conventional location anonymization techniques [36, 37, 66] are developed as unidirectional and irreversible techniques which fail to support the cost-effective gradual release of privacy data shown in Figure 24. Therefore, we present *ReverseCloak*, a new class of reversible spatial cloaking mechanisms that effectively provides multi-level location privacy protection, allowing de-anonymization of the cloaking region when corresponding *perturbation keys* are released to the data users in future through the self-emerging data release system. Extensive experiments on real road networks show that our techniques are efficient and scalable.

6.3.1 Overview of Concepts and Models

In this subsection, we first describe the road network model used to capture the mobility features of mobile users [41, 80, 125]. Then, we present the concept of location anonymization. Finally, we define the multilevel reversible location privacy problem.

Road network model: We model the road network as a graph $G = (\nu_G, \varepsilon_G)$, where ν_G represents the set of junctions and ε_G represents the set of road segments. A junction is defined as the crossover point of any two roads or the end of a road segment. A road segment is defined as the direct road connecting any two adjacent junctions. Each segment is uniquely determined by the two junctions associated with it while each junction is associated with one or more adjacent road segments. In the road network, each mobile user is assumed to move along the segments and change direction only at junctions. A user may send her true location information with the anonymization requirements to a trusted anonymization server which then transforms this raw location information into a cloaking region that meets the required privacy levels.

Location anonymization: We consider two kinds of privacy requirements arising in a road network namely *location k-anonymity* and *segment l-diversity.* The *k-anonymity* requirement ensures that the exposed location of a user indistinguishable from the location information of at least $k - 1$ other users. However, satisfying *location k-anonymity* alone may not be sufficient to protect the location privacy of the user in cases when there are homogeneity attacks [89]. For instance, if all the k users contained in a *k-anonymized* spatial region are present in a single physical location, such as a hospital, then even though there are k users in the cloaked region, an adversary observing the region can still infer the actual location of the subject with high certainty. To protect against such scenarios, the notion of *location l-diversity* has been introduced [86, 89]. A cloaked location satisfies *segment l-diversity* [125] if the cloaked region not only includes k distinct users but also contains l well represented road segments. Therefore, from an attacker's perspective, a cloaking area with more segments increases the difficulty to track a user and hence ensuring a larger l-diversity provides higher location privacy. In a personalized location privacy model, for each location anonymization request, the level of k-anonymity, δ_k, and l-diversity, δ_l, are given by the user in a customizable manner. These two parameters together decide the anonymization level. In addition, in order to maintain the QoS above a certain level, user needs to set anonymization restrictions to cloaking spatial area, namely the spatial tolerance σ_s, indicating the maximum acceptable cloaking spatial area.

Multilevel reversible location privacy: We would like techniques developed in this section to support multi-level location privacy in data gradual release scenarios. In such cases, the location privacy of users is protected under multiple privacy levels, with higher anonymity levels to be maintained in recent future and lower privacy levels to be maintained in remote future. In the multi-level reversible location privacy framework, a trusted anonymizer obtains the raw location information from the mobile clients with the user-defined profile. With

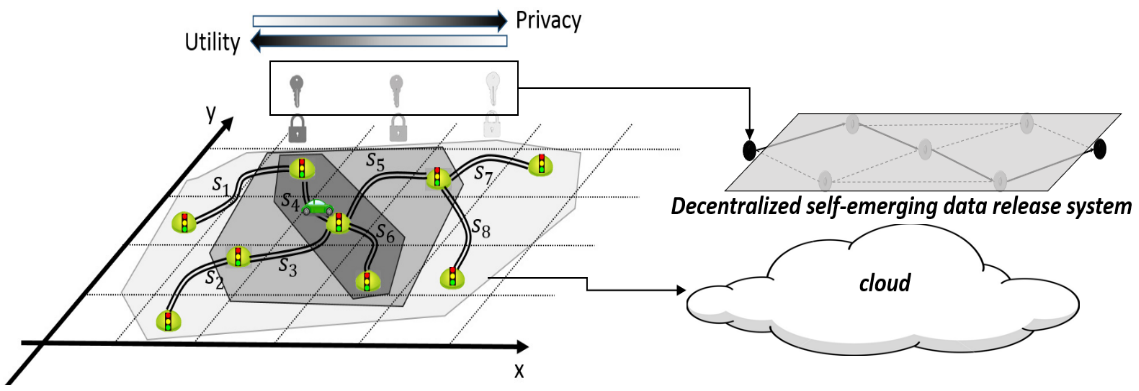


Figure 30: Multilevel reversible location anonymization

the multi-level privacy model, the user-defined profile consists of the privacy requirements for each privacy level, L^i, except L^0 referring to a cloaking region with only the segment of actual user. Accordingly, the user-defined privacy profile is represented by (δ_k^i, δ_l^i), where $1 \leq i \leq N-1$ and N denotes the number of privacy levels. In addition, each privacy level, L^i is associated with a shared perturbation key, Key^i, which is used to drive the anonymization process for that privacy level. Therefore, with access to the *perturbation key* of a particular privacy level, users of the cloaked location can selectively de-anonymize the cloaked region to reduce privacy levels to obtain finer location information. A detailed example of a four level case is shown in Figure 30. The segment s_4 contains the actual user of level, L^0. Using the *perturbation key* Key^1, s_6 is added to reach the privacy level, δ_k^1, δ_l^1 of L^1. Then, Key^2 is used further to extend the cloaking region to meet δ_k^2, δ_l^2 of level L^2 by adding segments $\{s_3, s_5\}$. Finally, $\{s_1, s_2, s_7, s_8\}$ are added using the perturbation key, Key^3 to reach the highest privacy level, L^3. Based on the cost-effective gradual release approach described in Figure 24, the keys should be sent into the decentralized self-emerging data release system while only the largest cloaking region $\{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8\}$ need to be stored in cloud. Later, when the cloaked location information needs to be reduced in privacy levels, it can be done using the perturbation keys. For instance, for accessing the information at the lower privilege level, L^2, Key^3 can be used to exactly identify and remove the segments $\{s_1, s_2, s_7, s_8\}$ from the cloaking region to reduce to the cloaked region corresponding to level, L^2. Similarly, using both Key^3 and Key^2, the segments $\{s_1, s_2, s_7, s_8, s_3, s_5\}$ can be identified and removed from the cloaking region to reduce to level, L^1. Therefore, by merely managing the shared *perturbation keys* among the location users at different privilege levels, the whole process protects location privacy under multiple discrete levels as customized in the user-defined privacy profile.

6.3.2 Reversible Location Cloaking

In this subsection, we present the proposed *ReverseCloak* mechanisms that support multi-level location cloaking over road networks. We propose two algorithms, namely reversible global expansion (RGE) and reversible pre-assignment-based local expansion (RPLE). The pseudo-codes of the two algorithms are presented in Appendix D.

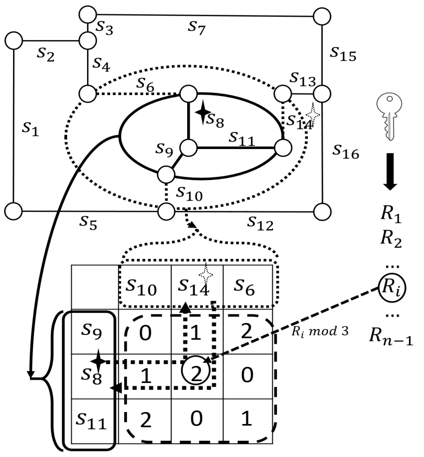


Figure 31: Reversible global expansion

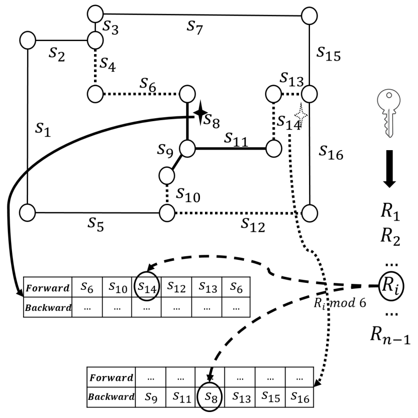
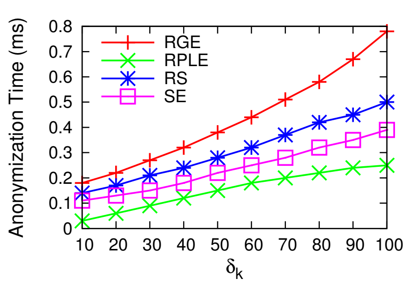


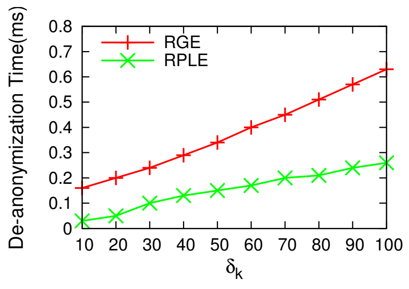
Figure 32: Reversible pre-assignment-based local expansion

In reversible location cloaking, the anonymization and de-anonymization processes are considered as a continuous selection and removal of road segments on the geographic road map respectively. To ensure that the process is reversible, the segments are selected in a pseudo-random manner. Each road segment on the map is linked to several other segments, which are located close to it. Once a road segment S is selected during anonymization, the next selected road segment is from one of its linked segments. With a certain *perturbation key*, a fixed segment S' among them is deterministically selected. However, without the *perturbation key*, all its linked segments would have the same probability to be selected, thus making the selection process pseudo-random and making it impossible to reverse without possessing the *perturbation key*. Then, during the de-anonymization process, the newly selected segment S' maps to the previous road segment S using the *perturbation key*. The algorithms checks which road segment is linked with S' to narrow down the options and whether segment S' can be deterministically selected with the *perturbation key* if we assume a segment is S. A key challenge here is the 'collision' issue that could happen in the de-anonymization process. That is, we may find multiple road segments that meet the conditions to be the candidate of the previously chosen road segment. To address this issue, in RGE, for each road segment selection during anonymization, the links of previously selected segments are rebuilt on the fly to avoid collisions and optimize the selection based on the current state. In RPLE, prior to the anonymization process, all the road segments in the map are pre-assigned their links in a collision-free manner. As a result, RGE has larger anonymization runtime to build collision-free links on the fly but smaller memory requirement while RPLE has smaller anonymization runtime but requires larger memory space to store the collision-free links. Next, we review the process of RGE and RPLE with Figure 31 and Figure 32, respectively.

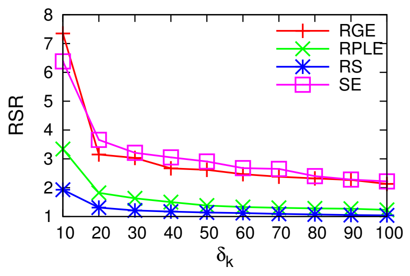
In both Figure 31 and Figure 32, the current cloaking region is $\{s_8, s_9, s_{11}\}$, where s_8 is the last selected segment, and the algorithms are selecting the next segment to be added into the cloaking region. In RGE (Figure 31), the three selected segments $\{s_8, s_9, s_{11}\}$ and the same number of non-selected nearby segments $\{s_6, s_{10}, s_{14}\}$ are taken to form a 3x3 square matrix, where the cells are filled with 0-2 in a way that each row/column has no repeated value. Assume that the pseudo-random number R_i generated through the *perturbation key* gives $R_i \ mod \ 3 = 2$, then s_{14} will be the next selected segment because only the cell $[s_8][s_{14}]$ has value 2 at row s_8. Later in de-anonymization, after removing s_{14}, the same matrix can be formed and the same *perturbation key* can give $R_i \ mod \ 3 = 2$. By looking at column s_{14}, since only the cell $[s_8][s_{14}]$ has value 2, the algorithm understands that s_8 should be the next removed segment. In this way, the reversibility can be established in a collision-free manner. Unlike RGE, in RPLE (Figure 32), prior to the anonymization process, the algorithm has generated one forward list and one backward list for each segment in the map. All the lists have the same length, which is six in the example. Assume that the pseudo-random number R_i gives $R_i \ mod \ 6 = 3$, then s_{14} will be the next selected segment because it is the third element in the forward list of s_8. Later in de-anonymization, since s_8 is also the third



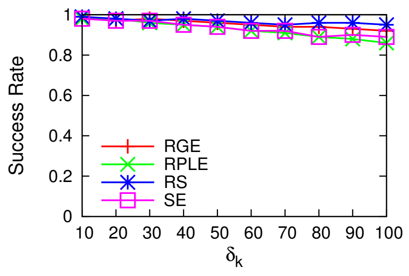
(a) Anonymization Time



(b) De-anonymization Time



(c) Relative Spatial Resolution



(d) Success Rate

Figure 33: Performance with Varying Anonymity Level

element in the backward list of s_{14}, with the same *perturbation key* giving $R_i \ mod \ 6 = 3$, the algorithm is able to remove s_8 after s_{14}. As can be seen, to establish reversibility in RPLE, s_{14} should be at the same position in the forward list of s_8 where s_8 is located in the backward list of s_{14}. With this objective, in RPLE, the two lists for all the road segments can be generated in a greedy manner.

6.3.3 Experimental Evaluation

In this subsection, we first present the experimental setup, and then evaluate the performance of proposed reversible global expansion and pre-assignment-based location expansion algorithms, compared with several existing cloaking algorithms.

Experimental setup: To simulate different anonymization schemes, we use GTMobiSim mobile trace generator for road network [107]. Our experiments were designed based on a real road network map of northwest part of Atlanta, involving 6979 junctions and 9187 segments, obtained from maps of National Mapping Division of the USGS. There are 10,000 cars randomly generated along the roads based on Gaussian distribution. Once a car is generated, the associated destination is also randomly chosen and the route selection is based on shortest path routing. In our experiments, four different anonymization schemes are implemented: Random Sampling (RS), Star-based road-network expansion(SE) [125], a candidate representative of existing road network-based expansion schemes, Reversible Global Expansion(RGE) and Reversible Pre-assignment-based Local Expansion(RPLE). The first two algorithms (RS and SE) are irreversible while the two algorithms proposed in this section (RGE and RPLE) are reversible support multi-level privacy control. All the schemes are implemented in Java with the help of GTMobiSim.

Experimental results: We evaluates the performance of the algorithms by varying the anonymity level δ_k as $\delta_k = 10i \ for \ i = 1, 2...10$. Here, the spatial tolerance, σ_s, is set as a function of the anonymity level, δ_k such that $\sigma_s = 400\sqrt{i} \ for \ i = 1, 2...10$, where the unit is meter. Therefore, the maximum allowable special region is a circular region with the user's actual location as the center and the spatial tolerance, σ_s as the radius. we also set 5% standard deviation for each σ_s and the segment diversity level δ_l is fixed to be 10. For this experiment, we consider only one privacy level and for the multi-level reversible techniques, this privacy requirement represents the privacy level of the least privileged user.

We compare the average anonymization time for the various approaches in Figure 33(a). In Figure 33(a), we find that RPLE is fastest in the anonymization phase among all the compared techniques. The reason is that the assignment of transition values in the RPLE scheme has been done *apriori* and at the time location cloaking, the transition graph is directly looked up as compared to dynamically computing it on the fly in the RGE approach. Also, for all the algorithms, the anonymization time is longer for larger δ_k as stricter privacy requirements result in cloaking areas with more segments and it therefore requires addition of more segments into the cloaking area.

Figure 33(b) shows the impact of varying δ_k on the de-anonymization time. Since only reversible algorithms can perform de-anonymization of the cloaked region, RS and SE are not considered for this experiment. For both RGE and RPLE, the variation trends of de-anonymization time are similar as the anonymization time in Figure 33(a) as the computational complexity of the de-anonymization process is similar to that of the anonymization process. In both anonymization and de-anonymization phases, RPLE is faster than RGE because RPLE prevents collision in a apriori manner through its intelligent pre-assignment of forward and backward transitions while RGE prevents collision by dynamically assigning the transition values during location cloaking.

Figure 33(c) displays the impact of changing the anonymity level, δ_k on relative spatial resolution (RSR) which is defined the ratio of the size of the obtained cloaking area to size of the maximum allowable spatial area, specified by the spatial tolerance level, d. Here, RS has the lowest relative spatial resolution (RSR) as its candidate expansion region covers all the segments within the maximum allowable spatial area, thus making the size of the cloaking area close to the maximum spatial area even when δ_k is small. we also find that the relative spatial resolution of SE and RGE is larger than RPLE as the cloaking segments in SE and RGE are selected from a global neighboring segment set, providing a tighter structure as compared to a local neighboring set in the RPLE approach.

In Figure 33(d), we compare the success rate of the anonymization process with varying δ_k value. The success rate represents the fraction of the cases where the cloaking algorithm is able to provide a cloaking region meeting the privacy requirements in terms of δ_k and δ_l. we find that all the algorithms have a high success rate indicating that most of the anonymization requests are cloaked successfully to meet the privacy requirements. we also find that for all the schemes, the success rate decreases slowly when δ_k becomes very large. This is because a larger δ_k requires a larger cloaking area, which is harder to be satisfied by a given spatial tolerance. However, we note that even for higher anonymity levels, such as $\delta_k = 100$, the success rates of both RGE and RPLE are higher and are close to 90%. RS keeps the highest success rate here as its failure occurs only when the total number of users within the maximum spatial area is smaller than δ_k. In fact, the success rate of the RS scheme defines the theoretical maximum success rate of the cloaking process for the given anonymization requests. we also note that SE and RGE have slightly higher success rate than RPLE as their cloaking regions have higher density and smaller size, thus being easier to meet the spatial tolerance requirement.

6.4 SUMMARY AND DISCUSSION

In this chapter, we propose the cost-effective approach for gradual release of self-emerging data that allows the perturbation level of private data to be gradually reduced over time by using the decentralized self-emerging data release system. We propose two representative applications that use the proposed cost-effective approach to gradually release association data and location data, respectively. For each of them, we present the adopted privacy paradigms and also develop a set of reversible perturbation techniques used for generating multiple reversible snapshots of the private data through *perturbation keys*. Extensive experiments show that the proposed techniques are effective and efficient.

So far, we have comprehensively discussed all the three research components (i.e., *Infrastructure*, *Output*, *Input*) and have resolved all the research tasks proposed in Section 1.1. In the next chapter, we will conclude this dissertation and present possible future directions.

7.0 CONCLUSION AND FUTURE DIRECTIONS

7.1 CONCLUSION

In this dissertation, we study new decentralized designs of self-emerging data release systems using large-scale peer-to-peer (P2P) networks as the underlying infrastructure. The first part of the dissertation presents the design of decentralized self-emerging data release systems using two different P2P network infrastructures, namely Distributed Hash Table (DHT) and blockchain. Specifically, our system designed in Chapter 3 leverages the efficient lookup service of DHT to establish a suite of routing path construction schemes for securely storing and routing the self-emerging data in DHT networks before a prescribed data release time. It demonstrates that increasing data redundancy is an effective approach of concealing data in the highly dynamic DHT network and protect the data from being stolen by adversaries. However, due to the lack of ways of enforcing behaviors performed by the peers in DHT, the DHT-based system usually needs complex routing paths constructed by hundreds of nodes to offer sufficient data redundancy, which may result in an unacceptable cost. To resolve this issue, our system designed in Chapter 4 leverages the decentralized trust and the native cryptocurrency offered by blockchain to enforce peers to honestly follow their agreements through cryptocurrency-driven monetary incentive and penalty. With the assumption that all peers are rational, the protocols carefully designed through game theory in Chapter 4 can make rational nodes choose to honestly comply with the protocols, instead of tending to perform any misbehavior violating the protocols, as such misbehaviors will make their deposit get confiscated.

The second part of this dissertation proposes new mechanisms for supporting two functionalities of self-emerging data release, namely supporting the release of self-emerging data to smart contracts and supporting the cost-effective gradual release of self-emerging data. The mechanism proposed in Chapter 5 enables releasing self-emerging data to smart contracts, thus facilitating a wide range of decentralized applications, allowing users of decentralized applications to schedule functions of smart contracts to be executed automatically at future points of time, without revealing the private input data before the expected function execution time. In Chapter 6, we analyze possible options of inputting the self-emerging data to the designed system and propose a cost-effective approach of using *perturbation keys* to gradually release self-emerging data over time. We develop a suite of mechanisms supporting cost-effective gradual release and demonstrate the effectiveness of the proposed mechanisms in two representative scenarios of gradually releasing self-emerging data, namely location data disclosure and association data disclosure.

7.2 FUTURE DIRECTIONS

We believe that the outcome of this dissertation would contribute to the development of decentralized security primitives and protocols in the context of timed release of private data. Next, we provide a brief list of possible future directions for our work.

- The protocols proposed in this dissertation are designed for making the self-emerging data get released at a release time prescribed by the data sender. In some cases, it may be hard for the data sender to clearly identify the expected release time. Instead, it may be desirable to release the data when a certain event happens. Therefore, one future direction is to extend the existing time-driven data release to event-driven data release for the purpose of increasing flexibility of releasing self-emerging data.
- In the systems designed in this dissertation, upon sending the self-emerging data out, the data senders cannot make any change to their service requests or their data. In some circumstances, senders may want to change their strategies of releasing their data or change their data after the data has been passed to service providers within the P2P network infrastructure. Therefore, additional protocols are desirable to make the systems

support such functionalities securely. Here we list a part of possible functionalities to be designed:

  - *Acceleration*: Senders can make the data get released earlier than the release time.
  - *Deceleration*: Senders can make the data get released later than the release time.
  - *Revocation*: Senders can revoke the self-emerging data before the release time.
  - *Redirection*: Senders can change the data recipient before the release time.
  - *Update*: Senders can update the self-emerging data before the release time.

- In Ethereum, any transaction creating new smart contracts or calling functions of existing smart contracts will spend gas, namely spending real money. As a result, services established over Ethereum is highly sensitive regarding gas cost because expensive services are hard to be widely accepted by users in practice. In the blockchain-based systems designed in this dissertation, we have paid attention to the gas consumption and have required only the hash values to be saved in the blockchain in most cases, so the gas cost corresponding to a specific service request is mainly relevant to the number of involved service providers. However, the proposed protocols do not differentiate old service providers from new service providers, failing to utilize the past performance of service providers for reducing service cost. Therefore, one future direction of this dissertation could be establishing a trust management mechanism that leverages the service history of each provider to compute a trust score and dynamically adjusts the number of involved service providers based on their trust scores, thus being able to reduce the number of selected service providers when most of them maintain high trust scores.

APPENDIX A

DISTRIBUTED HASH TABLE

In this section, we introduce Distributed Hash Table (DHT) with more details. The DHT technology refers to a class of protocols that provide lookup services for storing *(key,value)* pairs in an overlay network and also for efficiently retrieving the stored value associated with a given key. In a distributed network composing of N nodes, a straightforward approach of mapping data to the nodes is by leveraging the hash function and modular arithmetic, namely taking $hash(file)\ mod\ N$. However, in a distributed system, it is quite common that a node may have downtime and may join and leave the network frequently. In case of a new node joining the network, the number of nodes in the network increases from N to $N+1$, requesting the mapping between data and nodes to be re-computed through $hash(file)\ mod\ (N+1)$. As a result, many files that have been stored on the previous N node need to be relocated, resulting in significantly high traffic that may even block the network.

Consistent hashing: To resolve the aforementioned issue, we need technique to organize the nodes in a way that the amount of migrated data due to a single joint or left node can be minimized. Such a technique is named consistent hashing and has been adopted by most DHT protocols as a fundamental building block [91, 119]. As illustrated in Figure 34, with consistent hashing, the space that the hash function can map is fixed to $[0, 2^n - 1]$ and is organized as a ring. Each node in the network should choose a n-bit ID that has the same length of each lookup key associated with a file, so both node IDs and lookup keys occupy the same ring-shaped space. In the example of Figure 34, four nodes are mapped to a hash ring

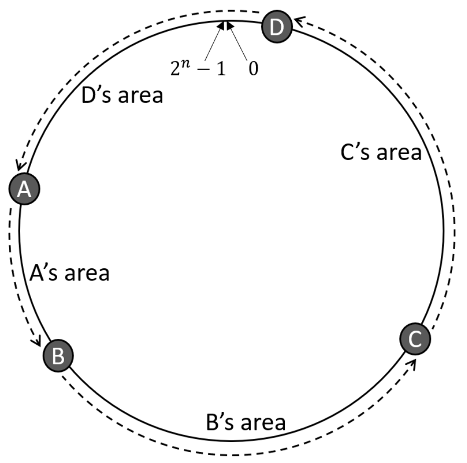


Figure 34: Consistent hashing

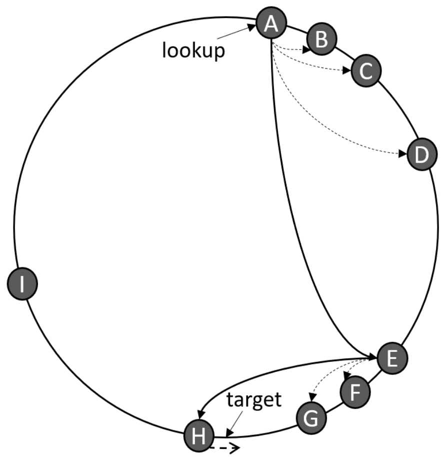


Figure 35: Routing protocol in Chord

with a fixed size of $[0, 2^n - 1]$, partitioning the entire ring into four non-overlapped sections, namely (A, B), (B, C), (C, D) and (D, A). Then, when data is mapped to a position of the ring according to its n-bit lookup key, the node taking charge of storing the data should be the one that has the closest ID to the data position in the counterclockwise direction of the ring. For example, in Figure 34, node A is responsible for storing data falling within the range (A, B), and node B is responsible for storing data falling within the range of (B, C). Compared with the straightforward approach, the most significant advantage of the consistent hashing is that a node joining or leaving the network does not affect data mapped to the entire ring-shaped space, but only affects data allocated to a single section of the ring as well as nodes associated with this section. For instance, if node A leaves the network, all the data falling within the range of (A, B) should then be re-assigned to node D, which only affects the data falling within the range of (A, B). In another example where a new node, say node E, joins the network and locates between node A and node B, only a part of data stored in node A needs to be transferred to node E.

Routing in DHT: In a naive routing protocol, a node receiving a lookup query first checks the local storage. If the data is not locally stored, the node then forwards the query to its neighboring node on the ring in the counterclockwise direction. The process will be repeated until the data is found. With such a naive routing protocol, in the worst case, the query time complexity is $O(N)$, where N denotes the network size. For example, in Figure 34, to retrieve a lookup key K stored at node C from node D, the query has to be forwarded through a path $D \to A \to B \to C$. In order to improving the efficiency of lookup queries, it would be necessary to request each node to maintain some routing information regarding the mapping between lookup keys and DHT nodes. For instance, if we request node D in Figure 34 to store the fact that 'lookup key K is stored at node C', then node D will be able to directly forward a query of key K to node C. In practice, different DHT protocols may implement the above strategy in different approaches. In this dissertation, we introduce the routing protocol used by Chord [119]. Chord requests each node to maintain a finger table. The entry i in the finger table of node j is the first node that succeeds or equals $j + 2^i$. For example, if a node has ID $j = 1$, then its finger table should consist of the IP addresses of nodes associated with the following IDs: $1 + 2^0 = 2$, $1 + 2^1 = 3$, $1 + 2^2 = 5$, $1 + 2^3 = 9$ and so forth. Then, upon receiving a query of key K, a node in Chord always forwards the query to the node in its finger table with the closest ID to K. To describe this greedy process, we present an example shown in Figure 35. In the beginning, node A receives a lookup query of key K from a client and finds K is not locally stored. Node A then checks its finger table and finds that K is larger than even the largest node ID, say node E, in its finger table, so node A forwards the query to node E. After that, node E also finds K is not locally stored but it then finds that K is very close to node H stored in its finger table, so node E forwards the query to node H. Finally, node H finds the file associated with K from the local memory and sends the file back to the client through the path $H \to E \to A$. To sum up, Chord requests each node to maintain a finger table of degree $O(\log N)$ for the purpose of reducing the query time complexity from $O(N)$ to $O(\log N)$.

APPENDIX B

BLOCKCHAIN AND SMART CONTRACT

In this section, we introduce the blockchain and smart contracts with more details. A blockchain represents a decentralized and distributed public digital ledger that guarantees that the records stored in it cannot be tampered without compromising a majority of nodes in the network. It was first conceptualized by a person known as Satoshi Nakamoto in 2008 as the underlying technology of a cryptocurrency named Bitcoin [100]. Since then, the growth of Bitcoin and the emerging follow-up cryptocurrencies have positioned blockchain as a promising solution for creating trust in a decentralized environment.

The Blockchain technology is an elegant combination of cryptography and game theory. It first relies on solid cryptographic techniques such as hash function, digital signature and Merkle tree to offer the cryptocurrency with mathematically provable security, thus gathering investment from its believers and placing a monetary value on the cryptocurrency. It then leverages the monetary value of the cryptocurrency to incentivize members of the entire P2P network to compete with each other for positions that can receive rewards of cryptocurrency for updating the ledger on behalf of the entire network. In the Proof-of-Work (PoW) consensus protocol of Bitcoin, members must spend computational resources to solve a mathematical problem and only the winner can update the blockchain. As a result, to falsify the ledger, an attacker must own a huge amount of computational power that can defeat the sum of the power of the entire P2P network, which is extremely difficult in practice. Besides, even if there is a strong attacker who has the power of falsifying the ledger, using the power to gather more rewards of cryptocurrency from the mathematical

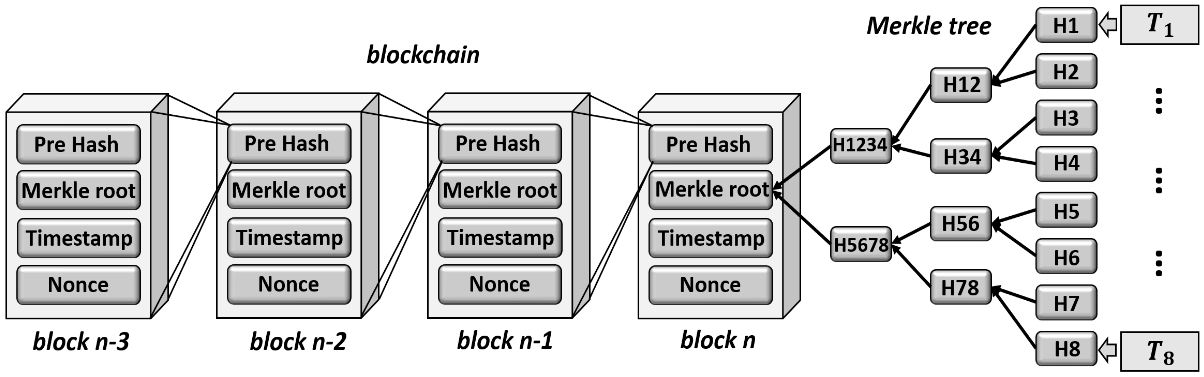


Figure 36: Blockchain structure in Bitcoin

competition may be a better choice than attacking the blockchain. Once the attack happens, investors will lose their confidence in the security of blockchain and the monetary value of the cryptocurrency will significantly drop, which also reduces the value of cryptocurrency owned by the attacker. From the perspective of game theory, this fact may make a rational self-interested attacker choose to honestly obey the rules of a blockchain for the purpose of pursuing higher profit.

Blockchain structure: Most blockchains follow a chain-of-block structure to organize data. In Figure 36, we show a chain of four blocks with block IDs $n-3$ to n. In the Bitcoin blockchain network, each account (i.e., a peer of the P2P network) owns a unique pair of public/private keys and also a unique account address derived from the public key. To transfer Bitcoin from one account to another, the sender account needs to create and broadcast a transaction that first applies sender's signatures of private key to claim ownership of the transferred Bitcoin and then uses recipient's public key to declare the ownership shift. Later when the recipient wants to spend the received Bitcoin, he or she can easily apply the account private key to prove the ownership. In the Bitcoin network, accounts trying to obtain cryptocurrency rewards are called miners. Each miner keeps collecting transactions created by other accounts and package them into the body of a block. Meanwhile, each miner keeps computing the answer to a mathematical challenge. Once a miner obtains the answer, this miner will generate a header for the current block, which consists of four main components:

- *Pre hash*: The previous hash refers to the cryptographic hash of the header of the previous block. For example, a miner that is producing block n should compute the hash of the header of block $n-1$. Due to the use of previous hash, the separated blocks are chained together so that falsification of any block header on the chain becomes verifiable.
- *Merkle root*: Merkle root is the root of a Merkle tree that is used for efficiently verifying the integrity of any transaction packaged at the body of the block. In the example of Figure 36, the body of block n contains eight transactions, namely T_1 to T_8. Then, the hashes of the eight transactions, H_1 to H_8, are iteratively grouped and hashed in a way that $H12 = hash(H1|H2)$, $H1234 = hash(H12|H34)$ and finally $root = hash(H1234|H5678)$. With Merkle root, fueled by pre-hash, falsification of any transaction of any block becomes verifiable.
- *Timestamp*: Timestamp refers to the time point when the block header is generated.
- *Nonce*: Nonce refers to a value relevant to the difficulty of the mathematical challenge. In Bitcoin, the average time of extending the blockchain by one new block is expected to be ten minutes, so nonce is used to dynamically adjust the difficulty of the challenge based on the recent computational power of the whole network for the purpose of making the block generation time stable.

Smart contract: In the leading smart contract platform Ethereum [128], there are two types of accounts, namely External Owned Accounts (EOAs) controlled by peers through pairs of private/public keys and Contract Accounts (CAs) assigned to smart contracts. A peer of Ethereum should first create an EOA with a pair of keys and then deploy smart contracts from the EOA, resulting in the creation of CAs associated with the smart contracts. A smart contract in Ethereum refers to a piece of program code that usually consists of multiple functions, a few parameters and perhaps some modifiers. After programming a smart contract in a language such as *Solidity* [11], a peer can compile the contract to get its *bytecode* and Application Binary Interface (*ABI*) and can send a contract creation transaction to the Ethereum network with *bytecode* and (optional) *ABI*. Upon receiving the transaction, miners will include the *bytecode* into the next block, meaning that a new smart contract has been created, whose CA can be deterministically computed from the address of its creator and a nonce. Each CA can be viewed as a small decentralized server that can act based on the functions in its contract and can store data (e.g., cryptocurrency) allowed by its contract. However, CAs are passive, meaning that execution of any function of deployed smart contracts must be invoked through either transactions sent by EOAs or messages sent from CAs. As a result, the transactions/messages, as well as function inputs inside them, are all recorded by the Ethereum blockchain, which makes the function outputs deterministic because all miners can execute the function with the same inputs and gets the same outputs. It is worth noting that a peer needs to pay Gas [128] for either deploying a new smart contract or calling a function of existing smart contracts in Ethereum. Gas can be exchanged with Ether, the cryptocurrency used in Ethereum, and Ether can be exchanged with real money.

```
contract betting {
  function deposit() payable public participantOnly;
  function reveal() public participantOnly;
  function reassign() public participantOnly;
}
```

Algorithm B.1: A simplified betting contract

We now illustrate smart contracts with a simplified betting contract presented in Algorithm B.1, where Alice and Bob decide to bet on a topic with cryptocurrency they have. The betting contract consists of three functions. Alice and Bob can first make deposits (i.e., the cryptocurrency Ether) to the contract (i.e., CA of the betting contract) through *deposit()*, then invoke *reveal()* after a certain temporal threshold to reveal the result and finally reassign the cryptocurrency locked in the contract based on the result by calling *reassign()*.

APPENDIX C

REVERSIBLE ASSOCIATION DATA PERTURBATION

In this section, we present the pseudo-codes of the algorithms proposed for the reversible association data perturbation, including the pseudo-codes of reversible edge perturbation, reversible node permutation and reversible edge permutation.

C.1 REVERSIBLE EDGE PERTURBATION

Algorithm 5: Noise injection

```
Input  : Bipartite graph $BG = (V, W, E)$, sensitivity $\triangle f$, budget $\epsilon$, key $K$.
Output: Perturbed bipartite graph $\widetilde{BG}$.
1  $n = \lfloor LaplaceRandom(0, \triangle f/\epsilon, K) \rfloor$;
2  Initialize counter $c = 0$, index $i = 0$, new edge recorder $\overline{NE}$, skipped index recorder $SI$;
3  while $c < n$ do
4      $ne = (rand(2i, K)\ mod\ |V|, rand(2i+1, K)\ mod\ |W|)$;
5      if $ne \notin E \cup \overline{NE}$ then
6          $\overline{NE} \leftarrow ne$; $c++$;
7      end
8      else
9          $\overline{SI} \leftarrow i$;
10     end
11     i++;
12 end
13 $\widetilde{BG} = (V, W, E \cup \overline{NE})$;
```

When the sampled noise is positive, the procedures of noise injection and noise removal are performed as shown in Algorithm 5 and Algorithm 6 respectively. During each loop (line

```
Algorithm 6: Noise removal
    Input   : Perturbed bipartite graph $\widetilde{BG}$, sensitivity $\triangle f$, budget $\epsilon$, key $K$, skipped index recorder $SI$.
    Output: Bipartite graph $BG$.
1 $n = \lfloor LaplaceRandom(0, \triangle f/\epsilon, K) \rfloor$;
2 Initialize index $i = 0$;
3 while $i < n + |SI|$ && $i \notin SI$ do
4     $re = (rand(2i, K)\ mod\ |V|, rand(2i + 1, K)\ mod\ |W|)$;
5     Remove edge $re$ from $\widetilde{BG}$;
6     i++;
7 end
8 $BG = \widetilde{BG}$;
```

3-12), two pseudo-random numbers are used to select one left node and one right node from the subgraph to form a new edge ne (line 4). If ne is not an existing edge, its selection will be confirmed (line 5-7); otherwise, this iteration will be skipped to avoid collision and this skipped index will be recorded into a list that will be attached with the key to be used during the decoding process later (line 8-10). The algorithm complexity is $O(n)$. Later in noise removal, with the same seed K, same n can be generated (line 1), which can then select and remove the same sequence of edges with assistance of SI (line 3-7). The complexity of this algorithm is $O(n + |SI|)$. However, when noises are negative, instead of using $|SI|$ to record the skipped iterations, we need to record all removed edges using the perturbation key.

C.2 REVERSIBLE NODE PERMUTATION

```
Algorithm 7: Node permutation: encoding
    Input   : Bipartite graph $BG = (V, W, E)$, key $K$.
    Output: Permuted bipartite graph $\overline{BG}$.
1 $R = PseudoRandom(K)$;
2 for $i = 0; i < |V|; i++$ do
3     Swap node $V[i]$ and node $V[R[i]\ mod\ |V|]$;
4 end
5 for $i = |V|; i < |V| + |W|; i++$ do
6     Swap node $W[i - |V|]$ and node $W[R[i]\ mod\ |W|]$;
7 end
```

We show the encoding phase and decoding phase in Algorithm 7 and Algorithm 8 respectively. During the encoding phase, the perturbation key is used as a seed to generate a se-

Algorithm 8: Node permutation: decoding

```
Input  : Permuted bipartite graph $\overline{BG} = (V, W, E)$, key $K$.
Output: Bipartite graph $BG$.
1 $R = PseudoRandom(K)$;
2 for $i = |V| - 1; i \geq 0; i - -$ do
3     Swap node $V[i]$ and node $V[R[i]\ mod\ |V|]$;
4 end
5 for $i = |V| + |W| - 1; i \geq |V|; i - -$ do
6     Swap node $W[i - |V|]$ and node $W[R[i]\ mod\ |W|]$;
7 end
```

quence of pseudo-random numbers denoted as R (line 1). Then, the first $|V|$ pseudo-random numbers in R are used to shuffle the left nodes in BG (line 2-4) while the pseudo-random numbers generated later, namely $|W|$ are used to shuffle the right nodes (line 5-7). Each pseudo-random number swaps two left (right) nodes. At the end of Algorithm 7, both left nodes and right nodes are shuffled in a reversible manner. Later, during decoding phase, given the same key, the same R can be obtained (line 1). The same two groups of pseudo-random numbers in R are used to recover left nodes (line 2-4) and right nodes (line 5-7) respectively. Here, both the algorithms have a complexity of $O(|V| + |W|)$.

C.3 REVERSIBLE EDGE PERMUTATION

Algorithm 9: Edge permutation: encoding

```
Input  : Bipartite graph $BG = (V, W, E[|V|][|W|])$, key $K$.
Output: Permuted bipartite graph $\overline{BG}$.
1 $R = PseudoRandom(K)$;
2 for $i = 0; i < |V||W|; i + +$ do
3     Swap edge $E[\lfloor \frac{i}{|W|} \rfloor][i\ mod\ |W|]$ and edge $E[\lfloor \frac{R[i]\ mod\ |V||W|}{|W|} \rfloor][R[i]\ mod\ |V||W|)\ mod\ |W|]$;
4 end
```

The encoding and decoding parts are shown in Algorithm 9 and Algorithm 10 respectively. In both the algorithms, same R can be obtained through the perturbation key (line 1). Then, we use the first $|V||W|$ pseudo-random numbers in R to perform $|V||W|$ rounds of swap operation (line 2-4). In this way, by reversibly performing the swap operation during the decoding phase, the original order of the edges can be recovered. Here, the algorithms

Algorithm 10: Edge permutation: decoding

```
Input  : Permuted bipartite graph $\overline{BG} = (V, W, E[|V|][|W|])$, key $K$.
Output: Bipartite graph $BG$.
1 $R = PseudoRandom(K)$;
2 for $i = |V||W| - 1; i \geq 0; i - -$ do
3     Swap edge $E[\lfloor \frac{i}{|W|} \rfloor][i\ mod\ |W|]$ and edge $E[\lfloor \frac{R[i]\ mod\ |V||W|}{|W|} \rfloor][R[i]\ mod\ |V||W|)\ mod\ |W|]$;
4 end
```

have a complexity of $O(|V||W|)$.

APPENDIX D

REVERSIBLE LOCATION DATA PERTURBATION

In this section, we present the pseudo-codes of the algorithms proposed for the reversible location data perturbation, including the pseudo-codes of reversible global expansion and reversible pre-assignment-based local expansion.

D.1 REVERSIBLE GLOBAL EXPANSION

The reversible global expansion (RGE) algorithm is shown as Algorithm 11. To perform the i^{th} forward transition in the anonymization process (loop 3 to 25), the selected segments and the candidate segments form a table that contains $|CloakA|$ rows and $|CanA|$ columns. (line 4 to 6). In the table, each transition value is assigned to one forward transition and its corresponding backward transition simultaneously so that these two transitions have same ID. The transition value in table cell (i, j) associated with i^{th} row and j^{th} column is computed by $((i-1)+(j-1))\ mod\ |CanA|$. Since the transition values for these potential forward transitions are different, the key can distinguish them clearly and select a unique forward transition from them (line 7 to 12). Here we note that the perturbation key can uniquely choose one backward transition. The key is used to generate a sequence of pseudo-random numbers and each pseudo-random number controls the selection of one transition. The i^{th} pseudo-random number, denoted by R_i, is responsible for both the i^{th} forward transition and $\{n-i\}^{th}$ backward transition. Therefore, for the i^{th} forward transition and $\{n-i\}^{th}$

backward transition, the same value can be uniquely determined by the pseudo-random number and the current cloaking region. This value, called picked value, can be calculated by $p_i = R_i \ mod \ |A|$ and it is used to select the transition with the transition value same as the picked value (line 13 to 19). After updating $CloakA$ and $CloakU$ (line 20), the algorithm stops when required δ_k and δ_l are met (line 22 to 24).

Algorithm 11: RGE

```
Input  : Road network graph $G$, original segment $s_u$, perturbation key $K_s$, user defined $\delta_k$, $\delta_l$, $\sigma_s$.
Output: A cloaking area $CloakA$ and a set of users $CloakU$.
Initially, $CloakA = \{s_u\}$, $CloakU = \{users\ on\ s_u\}$;
$currentSeg = s_u$;
while $dist(currentSeg, s_u) \leq \sigma_s$ do
    for $j = 1$ to $|CloakA|$ do
        Add the next neighboring segment to candidate set $CanA$;
    end
    Sort $CloakA$, $CanA$;
    $row$ = index of $currentSeg$ in $CloakA$;
    for $j = 0$ to $|CanA| - 1$ do
        $column = j$;
        $transitionValue[row, column] = (row + column)\ mod\ |CanA|$;
    end
    $R = PseudoRandomNext(K_s)$;
    $pickValue = R\ mod\ |CanA|$;
    for $j = 0$ to $|CanA| - 1$ do
        if $transitionValue[row, j] == pickValue$ then
            $nextSeg = CanA[j]$;
        end
    end
    Update $CloakA$ and $CloakU$ with $nextSeg$;
    $currentSeg = nextSeg$;
    if $|CloakA| \geq \delta_l$ and $|CloakU| \geq \delta_k$ then
        Return $CloakA$, $CloakU$;
    end
end
```

D.2 REVERSIBLE PRE-ASSIGNMENT-BASED LOCAL EXPANSION

The reversible pre-assignment-based local expansion (RPLE) algorithm is shown as Algorithm 12. In the RPLE algorithm, both the two transition tables contain $E \times \mathcal{T}$ empty cells initially (line 2), where $E = |\varepsilon_G|$ (line 1 to 2). Each of them has E rows and each row, which stands for a segment within G, has one forward transition list and one backward transition list with size $\mathcal{T}$. Since there is a one-to-one correspondence between the two tables, once we fill the forward transition table, the backward transition table is automatically filled. For each segment s in G (loop 3 to 25), it first establishes the neighboring list by calculating the n-hop neighboring segment. Like RGE, the calculation of n-hop neighboring segment can be done separately. After that, from line 7 to 24, it updates the forward and backward transition tables together. For each segment s in G, it tries each neighboring segment from its corresponding neighboring list, and treats that segment as potential segment s_p to form a transition relationship (line 8). Then, it begins to update the forward transition list of s and backward transition list of s_p.

Algorithm 12: RPLE(Pre-assignment)

```
Input  : Road network graph $G$, original segment $s_u$, temporal key $K_t$, spatial key $K_s$, transition
         list length $\mathcal{T}$, user defined $\delta_k$, $\delta_l$, $\sigma_t$.
Output: forward transition table $FT$, backward transition table $BT$.
1  $E$ = No. of segments in $G$;
2  Initially, the $E \times \mathcal{T}$ $FT$ and $BT$ are empty;
3  for each segment $s$ in $G$ do
4      for $i = 1$ to $E$ do
5          Add next neighboring segment to the neighboring list $NL$;
6      end
7      for $i = 0$ to $E - 1$ do
8          Potential segment $s_p = NL[i]$;
9          Initialize $empFT$ and $empBT$ with size $\mathcal{T}$;
10         for $j = 0$ to $\mathcal{T} - 1$ do
11             if $FT[s][j]$ is empty then
12                 Put $j$ to $empFT$;
13             end
14             if $BT[s_p][j]$ is empty then
15                 Put $j$ to $empBT$;
16             end
17         end
18         $emp = empFT \cap empBT$;
19         if $emp \mathrel{!}= \varnothing$ then
20             $selPosition = emp[0]$;
21             $FT[s][selPosition] = s_p$;
22             $BT[s_p][selPosition] = s$;
23         end
24     end
25 end
```

Since the two transition tables are filled gradually, the algorithm first checks the available positions of row s in forward table and row s_p in backward table (line 10 to 17), then takes the intersection that gives the empty position shared by the two transition lists (line 18) and finally choose the left most shared available position to put s_p in the row s in forward table and s in the row s_p in backward table (line 19 to 23). Therefore, by checking the same position of the two rows, we can do the transitions between s and s_p in anonymization and de-anonymization phases.

APPENDIX E

PUBLICATION LIST

Papers contributing to this dissertation:

- Chao Li and Balaji Palanisamy, “Reversible Spatio-temporal Perturbation for Protecting Location Privacy”, Elsevier Computer Communications, 2019.
- Chao Li and Balaji Palanisamy, “Decentralized Privacy-preserving Timed Execution in Blockchain-based Smart Contract Platforms, in IEEE HiPC, 2018.
- Chao Li and Balaji Palanisamy, “Decentralized Release of Self-emerging Data using Smart Contracts”, in IEEE SRDS, 2018.
- Chao Li, Balaji Palanisamy and Prashant Krishnamurthy, “Reversible Data Perturbation Techniques for Multi-level Privacy-preserving Data Publication”, in BigData Congress, 2018.
- Balaji Palanisamy, Chao Li and Prashant Krishnamurthy, “Group Privacy-aware Disclosure of Association Graph Data”, in IEEE Big Data, 2017.
- Chao Li and Balaji Palanisamy, “Emerge: Self-emerging Data Release using Cloud Data Storage”, in IEEE Cloud, 2017.
- Chao Li and Balaji Palanisamy, “Timed-release of Self-emerging Data using Distributed Hash Tables”, in IEEE ICDCS, 2017.
- Chao Li, Balaji Palanisamy, Aravind A. Kalaivanan and Sriram Raghunathan, “Reverse-Cloak: A Reversible Multi-level Location Privacy Protection System”, in IEEE ICDCS, 2017. [demo]

- Balaji Palanisamy, Chao Li and Prashant Krishnamurthy, “Group Differential Privacy-preserving Disclosure of Multi-level Association Graphs”, in IEEE ICDCS, 2017. [poster]
- Chao Li and Balaji Palanisamy, “De-anonymizable Location Cloaking for Privacy controlled Mobile Systems”, in NSS, 2015.
- Chao Li and Balaji Palanisamy, “ReverseCloak: Protecting Multi-level Location Privacy over Road Networks”, in ACM CIKM, 2015.

Other papers during my PhD study:

- Chao Li and Balaji Palanisamy, “Incentivized Blockchain-based Social Media Platforms: A Case Study of Steemit”, ACM Web Science, 2019.
- Runhua Xu, James Joshi and Chao Li, “CryptoNN: Training Neural Networks over Encrypted Data”, IEEE ICDCS, 2019.
- Chao Li, Balaji Palanisamy and Runhua Xu, “Scalable and Privacy-preserving Design of On/Off-chain Smart Contracts”, in BlockDM, 2019.
- Chao Li and Balaji Palanisamy, “Privacy in Internet of Things: from Principles to Technologies”, IEEE Internet of Things Journal, 2019.
- Lei Jin, Chao Li, Balaji Palanisamy and James Joshi, “k-Trustee: Location Injection Attack-resilient Anonymization for Location Privacy”, Elsevier Computers & Security, 2018.
- Chao Li, Balaji Palanisamy and James Joshi, “Differentially Private Trajectory Analysis for Points-of-Interest Recommendation”, in IEEE BigData Congress, 2017.
- Chao Li, Balaji Palanisamy and James Joshi, “SocialMix: Supporting Privacy-aware Trusted Social Networking Services”, in IEEE ICWS, 2016.

BIBLIOGRAPHY

[1] Amazon simple storage service (s3):. https://aws.amazon.com/s3/.

[2] Auctionhouse. http://auctionhouse.dappbench.com/.

[3] Aws best practices for ddos resiliency:. https://d0.awsstatic.com/whitepapers/DDoS_White_Paper_June2015.pdf.

[4] Ethernodes: The ethereum node explorer. https://www.ethernodes.org/network/1.

[5] Etherscan: gas price. https://etherscan.io/chart/gasprice.

[6] Geth: Official go implementation of the ethereum protocol. https://github.com/ethereum/go-ethereum.

[7] Google cloud storage:. https://cloud.google.com/storage/.

[8] Microsoft azure storage:. https://azure.microsoft.com/en-us/services/storage/.

[9] Rinkeby: Ethereum official testnet. https://www.rinkeby.io/#stats.

[10] Semantec report: The continued rise of ddos attacks:. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the-continued-rise-of-ddos-attacks.pdf.

[11] The solidity contract-oriented programming language. https://github.com/ethereum/solidity.

[12] Solrsaverify: Verification of rsa sha256 pkcs1.5 signatues. https://github.com/adriamb/SolRsaVerify.

[13] Whisper. https://github.com/ethereum/wiki/wiki/Whisper.

[14] Miguel E Andrés, Nicolás E Bordenabe, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. Geo-indistinguishability: Differential privacy for location-based systems. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 901–914. ACM, 2013.

[15] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. Fair two-party computations via bitcoin deposits. In *International Conference on Financial Cryptography and Data Security*, pages 105–121. Springer, 2014.

[16] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 443–458. IEEE, 2014.

[17] Walter Armbruster and Werner Böge. Bayesian game theory. *Game theory and related topics*, 17:28, 1979.

[18] Lars Backstrom, Cynthia Dwork, and Jon Kleinberg. Wherefore art thou r3579x?: anonymized social networks, hidden patterns, and structural steganography. In *Proceedings of the 16th international conference on World Wide Web*, pages 181–190. ACM, 2007.

[19] Bhuvan Bamba, Ling Liu, Peter Pesti, and Ting Wang. Supporting anonymous location queries in mobile environments with privacygrid. In *Proceedings of the 17th international conference on World Wide Web*, pages 237–246. ACM, 2008.

[20] Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In *International Cryptology Conference*, pages 421–439. Springer, 2014.

[21] Guido Bertoni et al. The keccak sha-3 submission. *Submission to NIST (Round 3)*, 6(7):16, 2011.

[22] Smriti Bhagat, Graham Cormode, Balachander Krishnamurthy, and Divesh Srivastava. Class-based graph anonymization for social network data. *Proceedings of the VLDB Endowment*, 2(1):766–777, 2009.

[23] Ranjita Bhagwan, David Moore, Stefan Savage, and Geoffrey M Voelker. Replication strategies for highly available peer-to-peer storage. In *Future directions in distributed computing*, pages 153–158. Springer, 2003.

[24] Nir Bitansky, Shafi Goldwasser, Abhishek Jain, Omer Paneth, Vinod Vaikuntanathan, and Brent Waters. Time-lock puzzles from randomized encodings. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, pages 345–356. ACM, 2016.

[25] Bittorrent. https://www.bittorrent.com/.

[26] Ian F Blake and Aldar C-F Chan. Scalable, server-passive, user-anonymous timed release public key encryption from bilinear pairing. *IACR Cryptology ePrint Archive*, 2004:211, 2004.

[27] Manuel Blum. Coin flipping by telephone a protocol for solving impossible problems. *ACM SIGACT News*, 15(1):23–27, 1983.

[28] Dan Boneh and Moni Naor. Timed commitments. In *Advances in CryptologyCrypto 2000*, pages 236–254. Springer, 2000.

[29] Joseph Bonneau, Arvind Narayanan, Andrew Miller, Jeremy Clark, Joshua A Kroll, and Edward W Felten. Mixcoin: Anonymity for bitcoin with accountable mixes. In *International Conference on Financial Cryptography and Data Security*, pages 486–504. Springer, 2014.

[30] Boomerang. https://www.boomeranggmail.com/.

[31] Nicolás E Bordenabe, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. Optimal geo-indistinguishable mechanisms for location privacy. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 251–262. ACM, 2014.

[32] Julien Cathalo, Benoît Libert, and Jean-Jacques Quisquater. Efficient and non-interactive timed-release encryption. In *ICICS*, volume 3783, pages 291–303. Springer, 2005.

[33] O. Celma. *Music Recommendation and Discovery in the Long Tail*. Springer, 2010.

[34] CERT Insider Threat Center. Us state of cybercrime survey (2014), 2014.

[35] Konstantinos Chalkias, Dimitrios Hristu-Varsakelis, and George Stephanides. Improved anonymous timed-release encryption. *Computer Security–ESORICS 2007*, pages 311–326, 2007.

[36] Rui Chen, Gergely Acs, and Claude Castelluccia. Differentially private sequential data publication via variable-length n-grams. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 638–649. ACM, 2012.

[37] Rui Chen, Benjamin Fung, Bipin C Desai, and Nériah M Sossou. Differentially private transit data publication: a case study on the montreal transportation system. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 213–221. ACM, 2012.

[38] Rui Chen, Noman Mohammed, Benjamin CM Fung, Bipin C Desai, and Li Xiong. Publishing set-valued data via differential privacy. *Proceedings of the VLDB Endowment*, 4(11):1087–1098, 2011.

[39] Reynold Cheng, Yu Zhang, Elisa Bertino, and Sunil Prabhakar. Preserving user location privacy in mobile data management infrastructures. In *International Workshop on Privacy Enhancing Technologies*, pages 393–412. Springer, 2006.

[40] Jung Hee Cheon, Nicholas Hopper, Yongdae Kim, and Ivan Osipkov. Provably secure timed-release public key encryption. *ACM Transactions on Information and System Security (TISSEC)*, 11(2):4, 2008.

[41] Chi-Yin Chow and Mohamed F Mokbel. Privacy in location-based services: a system architecture perspective. *Sigspatial Special*, 1(2):23–27, 2009.

[42] Graham Cormode, Divesh Srivastava, Ting Yu, and Qing Zhang. Anonymizing bipartite graph data using safe groupings. *Proceedings of the VLDB Endowment*, 1(1):833–844, 2008.

[43] Wei-Yen Day, Ninghui Li, and Min Lyu. Publishing graph degree distribution with node differential privacy. In *Proceedings of the 2016 International Conference on Management of Data*, pages 123–138. ACM, 2016.

[44] Giovanni Di Crescenzo, Rafail Ostrovsky, and Sivaramakrishnan Rajagopalan. Conditional oblivious transfer and timed-release encryption. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 74–89. Springer, 1999.

[45] Roger Dingledine et al. Tor: The second-generation onion router. Technical report, Naval Research Lab Washington DC, 2004.

[46] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, Naval Research Lab Washington DC, 2004.

[47] Changyu Dong, Yilei Wang, Amjad Aldweesh, Patrick McCorry, and Aad van Moorsel. Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing. *ACM CCS*, 2017.

[48] John R Douceur. The sybil attack. In *International Workshop on Peer-to-Peer Systems*, pages 251–260. Springer, 2002.

[49] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, volume 3876, pages 265–284. Springer, 2006.

[50] Keita Emura, Atsuko Miyaji, and Kazumasa Omote. A timed-release proxy re-encryption scheme. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 94(8):1682–1695, 2011.

[51] eth-ecies. https://github.com/libertylocked/eth-ecies.

[52] Ethereum market cap. https://coinmarketcap.com/currencies/ethereum/.

[53] ethereumjs-util. https://github.com/ethereumjs/ethereumjs-util.

[54] Arik Friedman and Assaf Schuster. Data mining with differential privacy. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 493–502. ACM, 2010.

[55] Sanjam Garg, Craig Gentry, Amit Sahai, and Brent Waters. Witness encryption and its applications. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 467–476. ACM, 2013.

[56] Roxana Geambasu, Tadayoshi Kohno, Amit A Levy, and Henry M Levy. Vanish: Increasing data privacy with self-destructing data. In *USENIX Security Symposium*, volume 9, 2009.

[57] Bugra Gedik and Ling Liu. A customizable k-anonymity model for protecting location privacy. Technical report, Georgia Institute of Technology, 2004.

[58] Gabriel Ghinita, Panos Kalnis, and Spiros Skiadopoulos. Prive: anonymous location-based queries in distributed mobile systems. In *Proceedings of the 16th international conference on World Wide Web*, pages 371–380. ACM, 2007.

[59] Gabriel Ghinita, Yufei Tao, and Panos Kalnis. On the anonymization of sparse high-dimensional data. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 715–724. Ieee, 2008.

[60] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229. ACM, 1987.

[61] Adam Groce and Jonathan Katz. Fair computation with rational players. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 81–98. Springer, 2012.

[62] Siyao Guo, Pavel Hubáček, Alon Rosen, and Margarita Vald. Rational sumchecks. In *Theory of Cryptography Conference*, pages 319–351. Springer, 2016.

[63] F Maxwell Harper and Joseph A Konstan. The movielens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 5(4):19, 2016.

[64] Carmit Hazay and Yehuda Lindell. A note on the relation between the definitions of security for semi-honest and malicious adversaries. *IACR Cryptology ePrint Archive*, 2010:551, 2010.

[65] Ruining He and Julian McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *Proceedings of the 25th International Conference on World Wide Web*, pages 507–517. International World Wide Web Conferences Steering Committee, 2016.

[66] Xi He, Graham Cormode, Ashwin Machanavajjhala, Cecilia M Procopiuc, and Divesh Srivastava. Dpt: differentially private trajectory synthesis using hierarchical reference systems. *Proceedings of the VLDB Endowment*, 8(11):1154–1165, 2015.

[67] Jason I Hong and James A Landay. An architecture for privacy-sensitive ubiquitous computing. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 177–189. ACM, 2004.

[68] Dimitrios Hristu-Varsakelis, Konstantinos Chalkias, and George Stephanides. Low-cost anonymous timed-release encryption. In *Information Assurance and Security, 2007. IAS 2007. Third International Symposium on*, pages 77–82. IEEE, 2007.

[69] Tibor Jager. How to build time-lock encryption. *IACR Cryptology ePrint Archive*, 2015:478, 2015.

[70] Panos Kalnis, Gabriel Ghinita, Kyriakos Mouratidis, and Dimitris Papadias. Preventing location-based identity inference in anonymous spatial queries. *IEEE transactions on knowledge and data engineering*, 19(12):1719–1733, 2007.

[71] Vishesh Karwa, Sofya Raskhodnikova, Adam Smith, and Grigory Yaroslavtsev. Private analysis of graph structure. *Proceedings of the VLDB Endowment*, 4(11):1146–1157, 2011.

[72] Kohei Kasamatsu, Takahiro Matsuda, Keita Emura, Nuttapong Attrapadung, Goichiro Hanaoka, and Hideki Imai. Time-specific encryption from forward-secure encryption. In *International Conference on Security and Cryptography for Networks*, pages 184–204. Springer, 2012.

[73] Shiva Prasad Kasiviswanathan, Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. Analyzing graphs with node differential privacy. In *Theory of Cryptography*, pages 457–476. Springer, 2013.

[74] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*, pages 357–388. Springer, 2017.

[75] Hidetoshi Kido, Yutaka Yanagisawa, and Tetsuji Satoh. Protection of location privacy using dummies for location-based services. In *Data Engineering Workshops, 2005. 21st International Conference on*, pages 1248–1248. IEEE, 2005.

[76] Ryo Kikuchi, Atsushi Fujioka, Yoshiaki Okamoto, and Taiichi Saito. Strong security notions for timed-release public-key encryption revisited. In *International Conference on Information Security and Cryptology*, pages 88–108. Springer, 2011.

[77] Fragkiskos Koufogiannis, Shuo Han, and George J Pappas. Gradually releasing private data under differential privacy. 2015.

[78] Kristian Lauslahti et al. Smart contracts–how will blockchain technology affect contractual practices? *Discussion paper*, 2017.

[79] Kevin Leyton-Brown and Yoav Shoham. Essentials of game theory: A concise multidisciplinary introduction. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 2(1):1–88, 2008.

[80] Chao Li and Balaji Palanisamy. Reversecloak: Protecting multi-level location privacy over road networks. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 673–682. ACM, 2015.

[81] Chao Li and Balaji Palanisamy. Emerge: Self-emerging data release using cloud data storage. In *Cloud Computing (CLOUD), 2017 IEEE 10th International Conference on*, pages 26–33. IEEE, 2017.

[82] Chao Li and Balaji Palanisamy. Timed-release of self-emerging data using distributed hash tables. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pages 2344–2351. IEEE, 2017.

[83] Chao Li and Balaji Palanisamy. Decentralized privacy-preserving timed execution in blockchain-based smart contract platforms. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, pages 265–274. IEEE, 2018.

[84] Chao Li and Balaji Palanisamy. Decentralized release of self-emerging data using smart contracts. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 213–220. IEEE, 2018.

[85] Ninghui Li, Tiancheng Li, and Suresh Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 106–115. IEEE, 2007.

[86] Ninghui Li, Tiancheng Li, and Suresh Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 106–115. IEEE, 2007.

[87] Jia Liu, Flavio Garcia, and Mark Ryan. Time-release protocol from bitcoin and witness encryption for sat. *IACR Cryptology ePrint Archive*, 2015:482, 2015.

[88] Ashwin Machanavajjhala, Johannes Gehrke, Daniel Kifer, and Muthuramakrishnan Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 24–24. IEEE, 2006.

[89] Ashwin Machanavajjhala, Johannes Gehrke, Daniel Kifer, and Muthuramakrishnan Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 24–24. IEEE, 2006.

[90] Timothy May. Timed-release crypto. *http://www. hks. net. cpunks/cpunks-0/1560. html*, 1992.

[91] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[92] Patrick McCorry, Siamak F Shahandashti, and Feng Hao. A smart contract for boardroom voting with maximum voter privacy. In *International Conference on Financial Cryptography and Data Security*, pages 357–375. Springer, 2017.

[93] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M Voelker, and Stefan Savage. A fistful of bitcoins: characterizing payments among men with no names. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 127–140. ACM, 2013.

[94] Andrew Miller and Iddo Bentov. Zero-collateral lotteries in bitcoin and ethereum. In *Security and Privacy Workshops (EuroS&PW), 2017 IEEE European Symposium on*, pages 4–13. IEEE, 2017.

[95] Mohamed F Mokbel, Chi-Yin Chow, and Walid G Aref. The new casper: Query processing for location services without compromising privacy. In *Proceedings of the 32nd international conference on Very large data bases*, pages 763–774. VLDB Endowment, 2006.

[96] Marco Casassa Mont, Keith Harrison, and Martin Sadler. The hp time vault service: Innovating the way confidential information is disclosed, at the right time. 2002.

[97] Marco Casassa Mont, Keith Harrison, and Martin Sadler. The hp time vault service: exploiting ibe for timed release of confidential information. In *Proceedings of the 12th international conference on World Wide Web*, pages 160–169. ACM, 2003.

[98] Andrew P Moore, Dawn M Cappelli, and Randall F Trzeciak. The big picture of insider it sabotage across us critical infrastructures. In *Insider Attack and Cyber Security*, pages 17–52. Springer, 2008.

[99] Yasumasa Nakai, Takahiro Matsuda, Wataru Kitada, and Kanta Matsuura. A generic construction of timed-release encryption with pre-open capability. In *International Workshop on Security*, pages 53–70. Springer, 2009.

[100] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[101] John F Nash et al. Equilibrium points in n-person games. *Proceedings of the national academy of sciences*, 36(1):48–49, 1950.

[102] Neo. https://neo.org/.

[103] Thanh Hong Nguyen, Rong Yang, Amos Azaria, Sarit Kraus, and Milind Tambe. Analyzing the effectiveness of adversary modeling in security games. In *AAAI*, 2013.

[104] Claudio Orlandi. Is multiparty computation any good in practice? In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 5848–5851. IEEE, 2011.

[105] Balaji Palanisamy, Chao Li, and Prashant Krishnamurthy. Group privacy-aware disclosure of association graph data. *IEEE Big Data*, 2017.

[106] Kenneth G Paterson and Elizabeth A Quaglia. Time-specific encryption. In *International Conference on Security and Cryptography for Networks*, pages 1–16. Springer, 2010.

[107] P Pesti, B Bamba, M Doo, L Liu, B Palanisamy, and M Weber. Gtmobisim: A mobile trace generator for road networks. *College of Computing, Georgia Inst. of Tech*, 2009.

[108] Postfity. https://postfity.com/.

[109] Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. 1996.

[110] Rodrigo Rodrigues and Barbara Liskov. High availability in dhts: Erasure coding vs. replication. In *International Workshop on Peer-to-Peer Systems*, pages 226–239. Springer, 2005.

[111] Alessandra Sala, Xiaohan Zhao, Christo Wilson, Haitao Zheng, and Ben Y Zhao. Sharing graphs using differentially private graph models. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 81–98. ACM, 2011.

[112] Pierangela Samarati. Protecting respondents identities in microdata release. *IEEE transactions on Knowledge and Data Engineering*, 13(6):1010–1027, 2001.

[113] secrets.js. https://github.com/grempe/secrets.js.

[114] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[115] Kazuyuki Shudo, Yoshio Tanaka, and Satoshi Sekiguchi. Overlay weaver: An overlay construction toolkit. *Computer Communications*, 31(2):402–412, 2008.

[116] Atul Singh et al. Eclipse attacks on overlay networks: Threats and defenses. In *In IEEE INFOCOM*. Citeseer, 2006.

[117] Smart contracts market research report global forecast to 2023. https://www.marketresearchfuture.com/reports/smart-contracts-market-4588.

[118] State of the dapps. https://www.stateofthedapps.com/.

[119] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[120] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 189–202. ACM, 2006.

[121] Latanya Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(05):557–570, 2002.

[122] Latanya Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(05):557–570, 2002.

[123] Tor. https://www.torproject.org/.

[124] Qian Wang, Yan Zhang, Xiao Lu, Zhibo Wang, Zhan Qin, and Kui Ren. Rescuedp: Real-time spatio-temporal crowd-sourced data publishing with differential privacy. In *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*, pages 1–9. IEEE, 2016.

[125] Ting Wang and Ling Liu. Privacy-aware mobile services over road networks. *Proceedings of the VLDB Endowment*, 2(1):1042–1053, 2009.

[126] Hakim Weatherspoon and John D Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *International Workshop on Peer-to-Peer Systems*, pages 328–337. Springer, 2002.

[127] web3-utils. https://www.npmjs.com/package/web3-utils.

[128] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.

[129] Writing a sealed-bid auction contract. https://programtheblockchain.com/posts/2018/03/27/writing-a-sealed-bid-auction-contract/.

[130] Zhen Xiao, Xiaofeng Meng, and Jianliang Xu. Quality aware privacy protection for location-based services. In *International Conference on Database Systems for Advanced Applications*, pages 434–446. Springer, 2007.

[131] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 162–167. IEEE, 1986.

[132] Bidi Ying and Dimitrios Makrakis. Protecting location privacy with clustering anonymization in vehicular networks. In *Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference on*, pages 305–310. IEEE, 2014.

[133] Wei-Wei Zhang and Ke-Jia Zhang. Cryptanalysis and improvement of the quantum private comparison protocol with semi-honest third party. *Quantum information processing*, 12(5):1981–1990, 2013.