ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica - Scienza e ingegneria

Corso di Laurea magistrale in Informatica

Privacy-Preserving Decentralized Crowdsensing: a Smart Contract-based approach

Relatore:
Chiar.mo Prof.
Stefano Ferretti

Correlatore:
Chiar.mo Prof.
Luca Bedogni

Presentata da:
Daniele Romanella

I sessione
Anno Accademico 2025/2026

Sommario

Il crowdsensing mobile consente la raccolta distribuita di dati geografici aggregando i contributi di un elevato numero di partecipanti. Questo paradigma si presta a scenari eterogenei come ad esempio: monitoraggio ambientale, analisi del traffico urbano, raccolta di dati epidemiologici. Scenari in cui la granularità spaziale e la scalabilità del numero di contributori sono requisiti primari.

Le soluzioni esistenti affidano il coordinamento della campagna a un server centralizzato, introducendo un unico punto di fiducia che può compromettere la riservatezza dei dati raccolti e la privacy dei partecipanti. Un server centralizzato può essere compromesso, può colludere con terze parti o essere soggetto a guasti, rendendo fragile l'intero sistema.

Questa tesi presenta il design, l'implementazione e la valutazione di un protocollo di crowdsensing decentralizzato e rispettoso della privacy, in cui il coordinamento è delegato a uno smart contract Ethereum.

Lo smart contract sostituisce il server fidato con un intermediario pubblicamente verificabile e resistente alle manomissioni: la sua logica è eseguita deterministicamente dall'Ethereum Virtual Machine (EVM), ogni nodo della rete ne mantiene una copia, e il comportamento del contratto non può essere alterato da alcuna singola parte dopo il deploy.

Il protocollo coinvolge tre attori principali. Il campaign initiator, `B` avvia la campagna, definisce i parametri spaziali e distribuisce le chiavi crittografiche. I sender raccolgono e inviano i dati geografici. I verifier, selezionati dal contratto su loro richiesta tra quelli idonei, validano le submission esprimendo un voto di approvazione o rifiuto. La partecipazione di sender e verifier è regolata da un meccanismo di cauzione tramite token ERC-20 personalizzato, `DST`: il deposito viene restituito al termine della campagna, unitamente al reward, solo ai partecipanti onesti.

La riservatezza dei dati è garantita attraverso crittografia ibrida tra i vari attori, mediati dallo smart contract. I dati cifrati vengono archiviati off-chain su IPFS; on-chain viene pubblicato esclusivamente il Content Identifier (CID) cifrato usando

la chiave K_{enc}.

L'area della campagna è suddivisa in una griglia spaziale di chunk: il contratto colloca on-chain l'origine geografica di ciascuna submission e limita il numero di dati accettati per chunk, prevenendo la sovra-rappresentazione delle zone ad alta densità di partecipanti.

Un'analisi di sicurezza e incentivi del protocollo base individua tre problemi aperti. Primo, un verifier malevolo può accedere ai dati in un qualunque momento a causa delle chiavi distribuite e sfruttando la visibilità della blockchain. Secondo, la distribuzione geografica dei reward non incentiva i sender a contribuire alle aree meno coperte, favorendo la concentrazione nelle zone già sature. Terzo, i verifier che esprimono voti errati non subiscono alcuna conseguenza economica, oltre alla perdita del deposito cauzionale.

Un secondo contratto Solidity, `PSC`, estende il protocollo base affrontando ciascuno dei tre problemi. Per il problema del verifier malevolo, `PSC` introduce un ulteriore strato crittografico: `B`, per ogni dato inviato crea una coppia di chiavi asimmetriche. Quindi `B` decifra il dato inviato, lo ri-cifra con la nuova coppia di chiavi per-dato e lo ricarica su IPFS; quando un verifier viene eletto, `B` gli consegna la relativa chiave di decrittazione cifrata sotto la chiave pubblica del verifier eletto, così che solo lui possa accedere al dato.

Per lo squilibrio geografico dei reward, `PSC` implementa una funzione di decadimento basata sul rapporto di saturazione del chunk: il reward per una submission verso un chunk già parzialmente saturo viene ridotto secondo la formula $(1-\alpha)^{1/2^k}$, dove α è il rapporto di saturazione e k è un parametro configurabile che controlla la ripidità del decadimento.

La radice frazionaria viene calcolata on-chain tramite il metodo babilonese iterativo, evitando operazioni in virgola mobile non supportate dall'EVM.

Per l'assenza di penalità, `PSC` traccia i voti corretti e scorretti di ciascun verifier: al superamento di una soglia configurabile di voti errati, il verifier viene rimosso automaticamente e, inoltre, il campaign initiator può espellere i verifier manualmente.

L'implementazione comprende i due contratti Solidity documentati, un SDK TypeScript che espone un'API per-attore e gestisce le operazioni crittografiche off-chain, e una suite di test completa sviluppata con Hardhat.

Le ottimizzazioni di gas documentate includono la gestione della pending list tramite swap-and-pop al posto dello shift lineare FIFO, la compressione delle struct in un singolo slot EVM da 32 byte, l'aritmetica `unchecked` nelle funzioni di reward, e l'eliminazione dei dati di enrollment tramite `delete` dopo il loro utilizzo.

La valutazione empirica copre tre dimensioni. L'analisi dei costi di gas su 16 scenari, con sender in $\{1, 10, 100, 1000\}$ e verifier in $\{2, 10, 100, 1000\}$, mostra che il contratto ottimizzato riduce i costi del verifier del 98% rispetto alla baseline FIFO. La baseline FIFO risulta non praticabile su reti Fusaka-era in tutti i 16 scenari, superando il limite per-transazione di 2^{24} gas introdotto dall'EIP-7825 di un fattore $3.4\times$. `PSC` aggiunge un overhead contenuto rispetto a PFC, con l'incremento più significativo sulla data submission dovuto al calcolo del reward decaduto.

Su reti L2 quali Arbitrum e Optimism, il costo di partecipazione diventa trascurabile per tutti i ruoli. L'analisi della saturazione dei blocchi sotto campagne concorrenti evidenzia una sezione critica sulla capacità del blocco: con quattro campagne parallele, il numero medio di blocchi consumati per campagna raggiunge $3.65\times$. La simulazione spaziale sul dataset di traiettorie taxi della città di Porto valida il meccanismo di decadimento del reward, mostrando come la funzione redistribuisca efficacemente gli incentivi verso le aree meno coperte. La copertura geografica satura attorno al 50% tra 500 e 1 000 sender, un limite attribuibile alla concentrazione geografica delle traiettorie nell'area urbana centrale piuttosto che al protocollo.

I risultati confermano la fattibilità del protocollo su reti L2 e su blockchain a basso costo del gas. Gli sviluppi futuri includono l'eliminazione della fiducia nel campaign initiator tramite Proxy Re-Encryption decentralizzata, l'estensione della verifica formale a scenari di enrollment disonesto, e la validazione su dataset con distribuzione geografica più uniforme.

Abstract

Mobile crowdsensing enables the collection of distributed geographic data by aggregating contributions from a large number of participants, each equipped with a sensing device. Existing solutions rely on a centralised server to coordinate the campaign, introducing a single point of trust that can compromise data confidentiality and participant privacy.

This thesis presents a privacy-preserving decentralised crowdsensing protocol in which coordination is delegated to an Ethereum smart contract, replacing the trusted server with a publicly verifiable and tamper-resistant intermediary. Data confidentiality is achieved through hybrid encryption: each sender encrypts their payload with a symmetric key that is in turn wrapped under an asymmetric key distributed by the campaign initiator, and stores the ciphertext on IPFS, publishing only the encrypted content identifier on-chain. Participation is incentivised through a custom ERC-20 token and a bond mechanism, while the campaign area is partitioned into a spatial grid of chunks so the contract records each submission's geographic origin on-chain and caps the number of submissions accepted per chunk, preventing over-representation of densely populated areas. A second contract extends the base protocol by addressing three open issues: verifier access to submitted data at any time through a shared decryption key and on-chain visibility, lack of geographic reward incentives, and absence of penalties for incorrect votes beyond bond loss.

An empirical evaluation covers gas consumption across multiple networks, block capacity contention under concurrent campaign execution, and spatial coverage on a real-world taxi trajectory dataset from the city of Porto. The results show that the optimised contract reduces verifier gas costs by approximately 98% relative to the FIFO baseline, and that the reward decay mechanism successfully redirects sender contributions towards underrepresented areas. On some Layer 2 networks, the cost of participation becomes negligible for all roles.

Contents

List of Tables

List of Figures

Listings

Chapter 0

Introduction

This thesis proposes a decentralised crowdsensing protocol [1; 2] implemented as an Ethereum smart contract [3; 4], in which data privacy is guaranteed through hybrid encryption and off-chain storage via IPFS [5], and participation is incentivized through an ERC-20 token [6] and a bond mechanism that discourages dishonest behaviour.

The design of the base contract builds on the smart-contract-coordinated crowdsensing scheme proposed by Bedogni and Ferretti [7]. Starting from that scheme, this thesis introduces three protocol extensions that address open privacy and incentive issues in the base design.

The protocol partitions the campaign area into a grid of spatial chunks, allowing the contract to record the geographic origin of each submission on-chain and to limit the number of data entries accepted per chunk, preventing the over-representation of densely populated areas.

Since a smart contract executes deterministically on the EVM and its code is publicly verifiable, it replaces the trusted server with a trustless intermediary whose behaviour cannot be altered by any single party [7; 8].

Despite these properties, a security and incentive analysis of the base protocol reveals several open issues: a malicious verifier can decrypt data from any sender, regardless of the slots it is elected to, through the shared decryption key and on-chain visibility, the geographic distribution of rewards does not incentivise senders to contribute to underrepresented areas, and verifiers who vote incorrectly face no penalty beyond forfeiting their bond. A second contract addresses each of these issues in turn, introducing a verifier-access mechanism based on an additional encryption layer that restricts each verifier to the data it is elected to verify, a reward

decay function that reduces the payout for submissions to already saturated chunks, and a penalty system that tracks incorrect votes and removes a verifier automatically once a configurable threshold is exceeded, or manually at the campaign initiator's discretion.

The main contributions of this thesis are:

- the design of a privacy-preserving crowdsensing protocol with a formal security analysis conducted using ProVerif [9] under the Dolev-Yao [10] attacker model;
- the implementation of two Solidity smart contracts with documented gas optimisations, including a swap-and-pop pending list management strategy justified empirically against a FIFO baseline;
- a TypeScript SDK that provides a per-actor API and handles off-chain cryptographic operations, serving as a reference implementation of the protocol;
- an empirical evaluation covering gas consumption across multiple networks, block capacity contention under concurrent campaigns, and spatial coverage and reward distribution on a real-world taxi trajectory dataset.

The remainder of this thesis is organised as follows. Chapter 1 introduces the background concepts required to understand the protocol. Chapter 2 presents the protocol design and its security analysis. Chapter 3 describes the smart contract implementation and the SDK layer. Chapter 4 reports the experimental evaluation. Chapter 5 summarises the contributions and outlines directions for future work.

Chapter 1

Background

This chapter introduces all the background concepts required to understand the protocol and the context in which it works.

1.1 Network architectures

Network architecture defines how nodes communicate and interact over a network. The most common architectural models are introduced in the following subsections.

1.1.1 Distributed systems

A distributed system is a collection of independent nodes that communicate and coordinate over a network to appear as a single coherent system [11]. This model enables horizontal scaling and eliminates single points of failure, making it suitable for high availability applications.

1.1.2 Client-Server

The client-server architecture is one of the most widely adopted architectural models [11]. It involves two types of nodes:

- **Server**: the node responsible for delivering content and services to clients, maintained by the service provider.
- **Client**: the node that communicates with the server to request a specific service or resource.

These two node types communicate with each other following a request-response model, where the client initiates the interaction and the server responds accordingly. Some protocols based on Client-Server architectures are:

- **HTTP**: Client sends an HTTP request and receives an HTTP Response [12]
- **DNS**: Client (Resolver) sends a DNS query to DNS server to resolve a name [13]
- **DHCP**: Client broadcasts a DHCP request over all the local network and if a DHCP server is listening it will handle the IP assignation [14]

1.1.3 Peer-to-peer

Peer-to-peer architectures are adopted in contexts where a single server is not able to guarantee a service, where the decentralization is a required property for the model or where the goal is to remove a single point of failure. [11]

In a peer-to-peer context all nodes act as both client and server, simultaneously. Each node is called `peer`, and it is able to request and provide resources to other peers.

In peer-to-peer protocols there is a challenge: peer discovery, the mechanism by which a node locates other peers to communicate with. A common approach is centralized trackers, where a server maintains a list of active peers, but this reintroduces a single point of failure. Distributed hash tables (DHT) solve this by distributing peer information across the network itself, removing any central dependency [15].

An example of a peer-to-peer protocol is BitTorrent [16], widely used for distributed file sharing. The peer-to-peer model also serves as the foundation for distributed storage systems such as IPFS, discussed in the following subsection.

1.1.4 IPFS

IPFS (InterPlanetary File System) is a peer-to-peer distributed file system that enables efficient and reliable data distribution among all nodes [5]. Unlike traditional location based addressing, each piece of content is identified by its cryptographic hash, known as a Content Identifier (CID). The protocol works in the following way: every node hosts and shares a portion of the data, when a client requests a specific CID, it uses the DHT to retrieve the data from the nodes that store it. In addition, IPFS is a versioned file system and it supports integration with decentralized applications, offering a storage layer for blockchains and Web3 ecosystems.

1.2 Blockchain

A blockchain is a growing distributed ledger that stores data in an ordered sequence of blocks [17]. Each block contains a set of transactions, the hash of the previous block, and a timestamp; this structure creates a chain. This chained structure makes the blockchain transactions resistant to alteration, because, when the transactions are registered, an adversary $\mathcal{A}$ would need to produce a block with the same hash, which is infeasible in probabilistic polynomial time [17].

1.2.1 Ethereum

Ethereum is an open source, decentralized blockchain platform created by Vitalik Buterin [18] in 2013. Bitcoin was primarily designed as a decentralized digital currency and distributed financial ledger, but Ethereum expanded the capabilities of blockchain technologies by introducing a programmable environment [18].

Smart Contracts

The core Ethereum architecture introduced the concept of *Smart Contracts*. These are deterministic and self-executing programs stored directly on the blockchain, where the terms of an agreement or the logic of an application are encoded into lines of code. When predetermined conditions are met, smart contracts execute automatically and autonomously, eliminating the need for trusted third-party intermediaries. Smart contracts are written in a Turing-complete programming language [18]. The execution of these contracts is handled by the *Ethereum Virtual Machine* (EVM).

The EVM is an isolated runtime environment that computes and maintains the global state of the Ethereum network. Every node participating in the network runs an instance of the EVM, ensuring that all state transitions are processed redundantly and validated by the network consensus algorithm. This distributed computation guarantees the integrity, immutability, and security of the system.

Gas

To allocate computational resources and mitigate malicious behaviors such as spam or infinite loops, Ethereum employs a pricing mechanism known as *Gas*. Every operation executed within the EVM requires a specific amount of computational effort, which must be compensated. Users initiate transactions and pay these gas fees using Wei.

$$1\text{ETH} = 10^{18}\text{wei}$$

Ether (ETH) is the network's native cryptocurrency, thereby incentivizing network validators to process and include their transactions in the next block.

Account types

The Ethereum state is managed through a system of accounts. There are two primary types of accounts: *Externally Owned Accounts* (EOAs), which are controlled by cryptographic private keys and typically managed by human users, and *Contract Accounts*, which are governed entirely by their underlying smart contract code. Historically, Ethereum relied on a Proof-of-Work (PoW) consensus mechanism to secure the network [18].

Proof-of-Stake and The Merge

Historically, Ethereum relied on a Proof-of-Work (PoW) consensus mechanism, where miners expended vast amounts of computational power to secure the network. However, on September 15, 2022, the network executed a major architectural upgrade known as `The Merge`. This event merged the original execution layer (Ethereum Mainnet) with a newly developed consensus layer (the Beacon Chain), effectively transitioning the entire network to a Proof-of-Stake (PoS) protocol [19]. Conceptually, PoS replaces energy-intensive mining with financial staking. To participate as a validator, a user must lock up (or `stake`) 32 Ether (ETH) into a designated smart contract. Instead of competing to solve arbitrary cryptographic puzzles, validators are pseudo-randomly selected by the protocol to propose new blocks or attest to the validity of blocks proposed by others. Honest participation is incentivized through block rewards and transaction fees. Conversely, malicious behavior or failure to fulfill validation duties is penalized through a mechanism called `slashing` which confiscates a portion of the validator's staked funds. The shift to PoS drastically reduced the network's energy consumption by approximately 99.98% [19]. For a decentralized crowdsensing system, this paradigm shift is of paramount importance. Under the previous PoW model, participating in the network required specialized, expensive hardware (such as ASICs or high-end GPUs) and incurred prohibitive electricity costs. In contrast, PoS significantly lowers the technical and financial barrier to running a node. In PoS, an adversary would need to control over 50% of the total staked ETH to compromise the network, which is economically prohibitive [20].

1.3 Crowdsensing

Crowdsensing, or Mobile Crowdsensing (MSC), is a technique that uses the collective sensing capabilities of individuals, who collect and share data about their surrounding environment, usually leveraging user's devices sensors [1]. The crowdsensing campaign can be participatory, in this case the users actively collect and upload data or opportunistic: in this case the data are collected and sent in background

without any effort form the user. There are several actors:

- Participants: People or devices which share data
- Requester: An entity that requires specific data and sets up the campaign
- Platform: A system that collects and processes data

Crowdsensing has been applied across several domains. In environmental monitoring, it enables large-scale data collection about air quality and pollution [21]. In traffic and transportation, it supports real-time infrastructure monitoring using sensors embedded in vehicles [22]. In healthcare, it has been used to model patient flows and support medical monitoring [23]. These applications share a common requirement: reliable, privacy-preserving data collection at scale [1].

1.3.1 Challenges

This approach presents several challenges:

- Data quality and data sparsity: data often suffers from noise, outliers and gaps because of uneven geographic distribution of participants in a certain area [1; 23].
- Privacy and Security: collecting data strictly related to real-world location threatens participants' privacy [1].
- Resource constraints: high-frequency sensing and measurements can drain devices' battery and consume a significant network bandwidth [23].
- System heterogeneity: since the crowdsensing campaign is made using the personal devices of the users, the system required to handle data from different devices [23].

Among these, data privacy and data quality are the primary challenges that the proposed protocol aims to address [1; 23].

1.4 Related work and state of the art

Mobile crowdsensing leverages the sensing capabilities of everyday mobile devices to collect data at scale, avoiding the cost of deploying dedicated sensor infrastructure [2].

Early MCS platforms adopted centralized architectures, where a trusted server mediates task assignment, data collection, and reward distribution. While effective, this

design introduces a single point of failure, requires participants to trust the platform operator with their data, and creates privacy vulnerabilities through exposed communication channels [3].

The integration of blockchain technology into MCS has been explored extensively as a means to decentralize platform control and remove the need for a trusted intermediary. Several surveys [3; 4] provide comprehensive overviews of this convergence. Smart contracts enable automated task management, verifiable reward distribution, and transparent data provenance without relying on a central authority. Systems such as CrowdBLPS [24] employ blockchain-based location-privacy preservation for crowdsensing, while ABCrowd [25] introduces reverse auctions as an incentive mechanism in a decentralized ecosystem.

BlockSense [26] proposes a dedicated blockchain for crowdsensing built on a Proof-of-Data consensus algorithm, using zk-SNARKs and homomorphic encryption to verify data quality. Chatzopoulos et al. [27] designed a spatial crowdsensing system where smart contracts manage both privacy and cost-optimal task allocation. More recently, Gigli et al. [8] presented an architectural vision for decentralizing MCS on distributed ledgers.

Despite the inherent pseudonymity of blockchain addresses, privacy remains a significant challenge in blockchain-based MCS. Transaction metadata, contribution patterns, and geolocation data can be exploited to de-anonymize participants, particularly in sparsely populated areas. Several approaches have been proposed to address this limitation. Zebralancer [28] employs zero-knowledge proofs to achieve worker anonymity on a public blockchain. CHASER [29] combines asymmetric encryption with zk-SNARKs to ensure both data confidentiality and participant anonymity. Other works leverage differential privacy [30] or linkable ring signatures [24] to protect location and identity information. contract-based protocol combining privacy-preserving data aggregation with quality-driven incentive mechanisms, implemented as a full-stack decentralized application on Ethereum. A common limitation of these approaches is that they focus primarily on anonymizing participant identity or the submitted data, without addressing the secure distribution of encryption keys among campaign actors.

Bedogni and Ferretti [7] proposed a system that directly addresses the relationship between anonymization and participation density. Their protocol uses a smart contract to coordinate crowdsensing campaigns, conditioning data acceptance on the presence of a sufficient number of participants within the same geographic area to form an adequate anonymization set. The smart contract manages encryption key distribution, data chunking into uniform sizes to prevent volume-based inference, and reward allocation through a market-driven mechanism. Their simulation re-

sults demonstrate that population density plays a critical role in campaign viability, with sparser areas requiring relaxed freshness constraints to reach the participation threshold. However, their work remains at the simulation level and does not provide a concrete smart contract implementation, nor does it detail the cryptographic key exchange protocol between campaign actors.

The use of smart contracts as a medium for cryptographic key exchange has received comparatively less attention. Muth and Tschorsch [31] proposed SmartDHX, a Diffie-Hellman key exchange scheme fully implemented as an Ethereum smart contract, providing asynchronous key agreement with message integrity and authenticity. Their approach executes the key exchange entirely on-chain, gaining verifiability at the cost of non-trivial gas consumption for cryptographic operations. Ruggeri et al. [32] extended the X3DH protocol onto a blockchain to eliminate the single point of failure of centralized key servers. Other works [33] have applied Elliptic Curve Diffie-Hellman in IoT blockchain contexts for sensor data exchange. However, these schemes target pairwise or group key agreement and do not address the specific workflow of a crowdsensing campaign, where an initiator must securely distribute distinct encryption and decryption keys to contributors and verifiers, respectively.

The protocol presented in this thesis builds upon the architectural vision of [7] and[8], extending it with a complete Solidity implementation and a detailed cryptographic key exchange protocol. Unlike SmartDHX and similar on-chain key agreement schemes, the proposed approach performs the asymmetric encryption (RSA-OAEP) off-chain and uses the smart contract solely as a secure transport channel for ciphered key material, avoiding the gas overhead of on-chain cryptographic computation. The smart contract integrates spatial privacy constraints through geographic chunking with configurable minimum participation thresholds, multi-verifier consensus with slot-based assignment, bond-based anti-fraud mechanisms, and an ERC-20 token incentive structure. Finally, a comprehensive gas consumption analysis and simulation framework provide empirical evidence of the protocol's economic viability an aspect often overlooked in prior privacy-focused MCS proposals.

Chapter 2

Protocol design

In this chapter the protocol design is presented. The protocol leverages smart contracts and asymmetric encryption to ensure confidentiality and data integrity. The protocol adopts a multi-layered encryption approach, performed off-chain, and uses the smart contract as a communication channel for the crowdsensing campaign. The key exchange is made to and from the campaign initiator (referred from now as "`B`"). `B` uses their private/public key pair for enhancing confidentiality, in this way, only `B` can access the other actors' public key, namely senders (`S`) and verifiers (`V`). This approach shares similarities with prior work on asymmetric key exchange mediated by smart contracts [31], although it differs substantially in both application domain and protocol design.

2.1 Actors

The protocol involves several actors that are described in the following subsections.

2.1.1 Campaign initiator

The campaign initiator (`B`) is the actor interested for collecting data for some reason. They define the geographic area where the smart contract works and set the smart contract hyperparameters, all configurable parameters are explained in Appendix A. In addition, they deposit the reward funds into the smart contract to ensure that funds are available and senders and verifiers can be compensated. The details of rewarding model are described in the 2.4.2 subsection.

2.1.2 Sender

Senders (hereafter referred to as "`S`") are the users who contribute to the crowdsensing campaign by submitting data retrieved from the environment where they are located. To contribute, senders, after the joining phase, upload data to IPFS and submit the encrypted CID through the smart contract.

2.1.3 Verifier

Verifiers (hereafter referred to as "`V`") are the users who contribute to the crowdsensing campaign by executing data quality checks and submitting the verification result through smart contract functions. To contribute, verifiers listen for events emitted by the smart contract. Upon receiving a `DataVerificationRequestedEvent`, they request verification authorization; if granted, they execute the verification and publish the result.

2.1.4 Smart Contract

Smart Contract (hereafter referred to as "`SC`") is the actor that mediates the interaction between all other actors. In addition, this actor handle the payment model and the bond mechanism, described in Sections 2.4.2 and 2.4.3. The smart contract acts as the sole communication channel between participants, eliminating direct interaction. It stores encrypted cryptographic material and data references on-chain, and enforces campaign lifecycle rules such as participation thresholds and automatic closure.

2.2 Chunking system

A fundamental challenge in implementing spatial crowdsensing on a smart contract is the lack of native floating-point support in the Solidity language [34]. Since geographic coordinates are inherently expressed as real numbers, a coordinate remapping strategy is required to represent them as integers without loss of precision. This section describes the spatial partitioning model adopted by the smart contract and the fixed-point arithmetic used to handle geographic coordinates.

2.2.1 Coordinate remapping

Since Solidity does not support floating-point arithmetic natively as stated before, geographic coordinates are transmitted by clients as a `GlobalPosition` struct composed of four fields: the integer and decimal parts of latitude, and the integer and

decimal parts of longitude,where the decimal component assumes a precision of up to seven digits after the decimal point.

To enable arithmetic operations within the EVM, coordinates are converted into a compact signed integer representation known as E7 format, where one unit corresponds to 10^{-7} degrees, providing a spatial resolution of approximately 1.1 cm. The conversion is performed by the `packCoordinates` function, which produces a `PackedPosition` struct containing two `int32` fields, `latE7` and `lonE7`.

For a non-negative integer part, the packed value is computed as:

$$\textit{(lon/lat)E7} = \textit{integer} \times 10^7 + \textit{decimal}$$

For a negative integer part, the decimal component is subtracted to correctly extend the magnitude:

$$\textit{(lon/lat)E7} = \textit{integer} \times 10^7 - \textit{decimal}$$

This representation preserves the full signed range of geographic coordinates without requiring any coordinate shifting, and fits within a 32-bit signed integer for the valid ranges of both latitude ($[-90°, +90°]$) and longitude ($[-180°, +180°]$) [35].

2.2.2 Spatial partitioning

The operational area of the smart contract is modelled as a square region subdivided into a uniform grid of $n \times n$ square chunks, where n denotes the number of chunks per side (`chunk4Side`). Each chunk represents a distinct geographic portion of the territory. The origin of the grid is located at the south-west corner of the area; latitude increases northward and longitude increases eastward from the origin. At deployment time, the end coordinate is derived by adding the total area extent to both axes of the packed origin:

$$\textit{end} = \textit{origin} + \texttt{chunkSize} \times \texttt{chunk4Side}$$

If no spatial subdivision is required, setting $n = 1$ yields a single chunk covering the entire operational area, preserving compatibility with deployments that do not use the chunking mechanism.

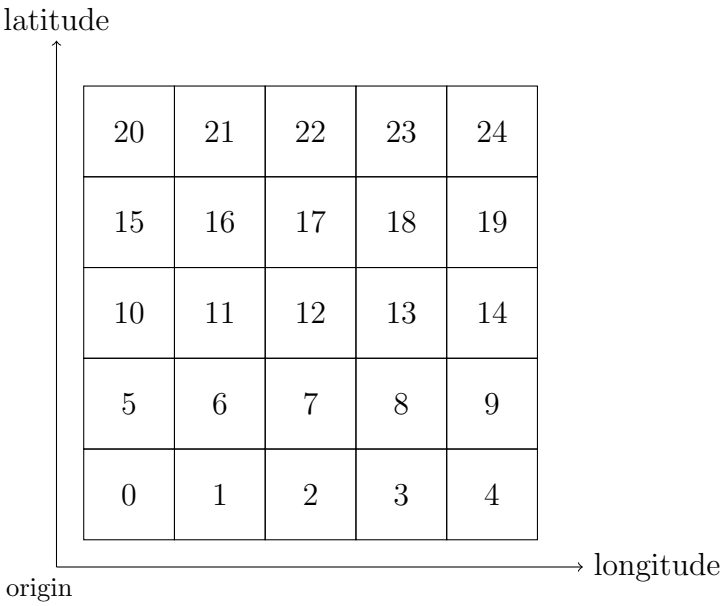

Figure 2.1: Chunk enumeration for a 5×5 grid (`chunk4Side` $= 5$). Index increases row by row from the southwest origin, consistent with the smart contract implementation.

2.2.3 Chunk indexing

Given a `PackedPosition`, its chunk coordinates are computed as:

$$\texttt{row} = \left\lfloor \frac{lat_{E7} - origin.lat_{E7}}{\texttt{chunkSize}} \right\rfloor, \qquad \texttt{col} = \left\lfloor \frac{lon_{E7} - origin.lon_{E7}}{\texttt{chunkSize}} \right\rfloor \tag{2.1}$$

Since latitude increases northward, row 0 corresponds to the southernmost strip; since longitude increases eastward, column 0 corresponds to the westernmost strip. Each chunk is then assigned a unique sequential index in the range $[0,\ n^2 - 1]$:

$$chunk\_index = \texttt{row} \times n + \texttt{col} \tag{2.2}$$

Where n is the number is `chunk4Side`, described in Table A.2.

This linear mapping establishes a direct correspondence between the two-dimensional grid coordinates and a one-dimensional array, enabling constant-time lookup of chunk data from its geographic position. A data submission is rejected if the packed coordinates fall outside the bounds $[origin, end)$ on either axis.

How chunks are numbered, as the latitude varies and as the longitude varies, is shown in Figure 2.1

2.3 Protocol workflow

The protocol workflow is described in the following subsections. Each phase is illustrated by a corresponding sequence diagram. The protocol design follows and extends the work presented in [7].

For clarity, the following notation is adopted throughout this chapter:

- $\text{Enc}(m, PK)$: encryption of message m using public key PK;
- $\text{Dec}(c, SK)$: decryption of ciphertext c using private key SK;
- (PK_X, SK_X): public and private key pair of actor X.

2.3.1 Initialization

Initialization phase consists of the following steps. The campaign initiator (`B`) deploys the smart contract, indicating all the parameters, described in Appendix A. Before the deployment, `B` generates its pair of private/public key (PK_B, SK_B) which is used for sender and verifiers enrollment (explained respectively in Section 2.3.2 and Section 2.3.3) and generates also an asymmetric key pair (K_{enc}, K_{dec}). After the deployment, the owner approves the transfer of a `totalDst` quantity of DST (described in 2.4.1) to the smart contract address, and then confirms the deposit. All interactions are shown in Figure 2.2

2.3.2 Sender enrollment

Any sender who wants to contribute to the crowdsensing campaign has to generate a private/public key pair. (PK_S, SK_S) Subsequently, `S` retrieves the public key of `B` from the smart contract. Then `S` computes:

$$c_1^S = \text{Enc}(PK_S, PK_B)$$

and sends c_1^S to the smart contract. Upon receiving the ciphertext, the smart contract emits an event that `B` monitors. When `B` detects the event, it retrieves the sender address and $c_1^S$from it.

Then `B` computes:

$$PK_S = \text{Dec}(c_1^S, SK_B)$$

At this point, `B` possesses `S`'s public key and can establish a secure channel. `B` computes:

$$c_2^S = \text{Enc}(K_{enc}, PK_S)$$

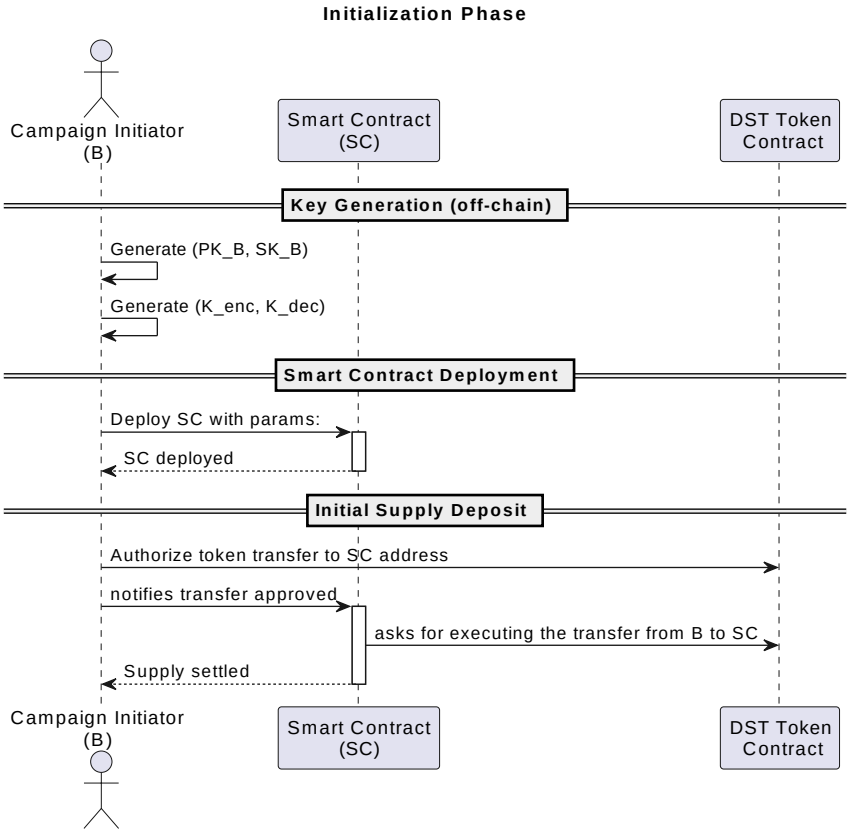

Figure 2.2: Initialization phase workflow

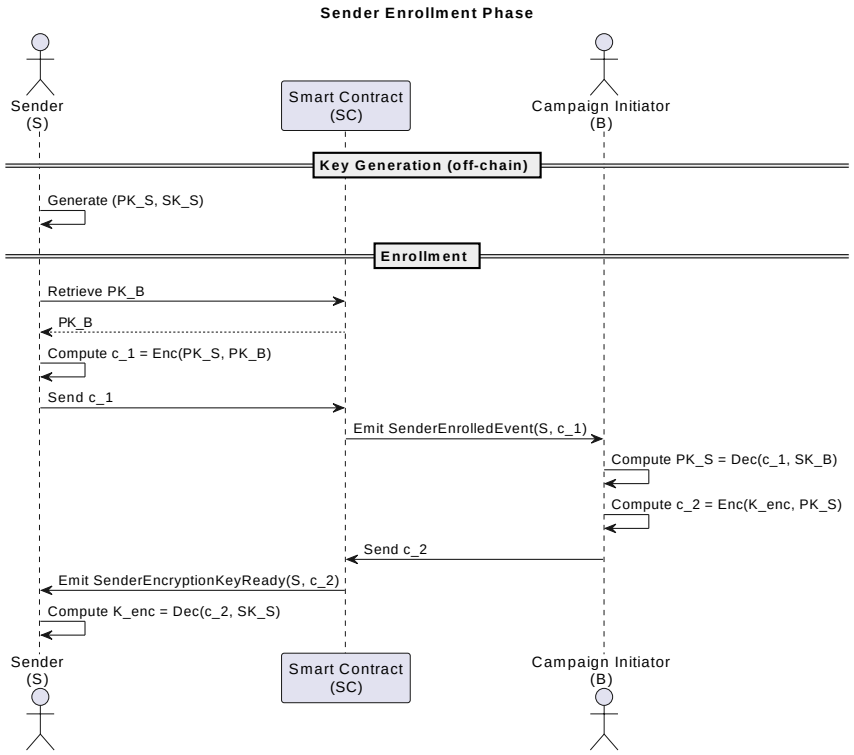

Figure 2.3: Sender enrollment phase workflow

And sends c_2^S to the smart contract which triggers an event to notify `S`. Finally, `S` retrieves c_2^S from the emitted event and computes

$$K_{enc} = \text{Dec}(c_2^S, SK_S)$$

This key will be used for CID encryption, as shown in Section 2.3.4. All interactions are shown in Figure 2.3

2.3.3 Verifier enrollment

Verifiers contribute to crowdsensing campaign by performing data quality checks on the data. Any verifier who wants to contribute to the crowdsensing campaign has to generate a private/public key pair (PK_V, SK_V). Subsequently, `V` retrieves the public key of `B` from the smart contract. Then `V` computes:

$$c_1^V = \text{Enc}(PK_V, PK_B)$$

then sends c_1^V to the smart contract.

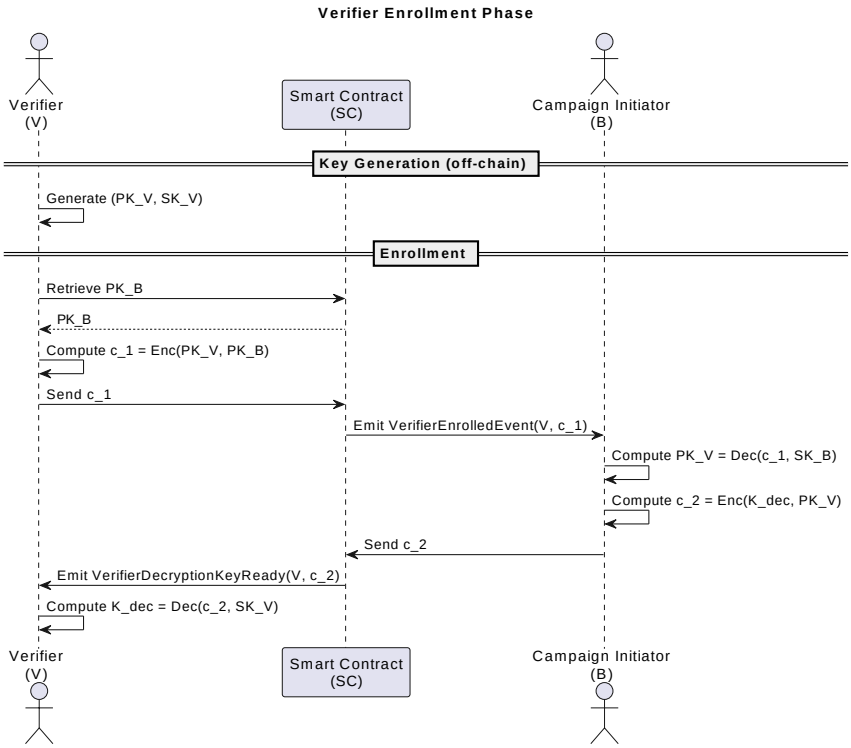

Figure 2.4: Verifier enrollment phase workflow

Upon receiving the ciphertext, the smart contract emits an event that `B` monitors. When `B` detects the event, it retrieves the verifier address and c_1^V from it.

Then `B` computes:

$$PK_V = \text{Dec}(c_1^V, SK_B)$$

At this point, `B` possesses `V`'s public key and can establish a secure channel. `B` computes:

$$c_2^V = \text{Enc}(K_{dec}, PK_V)$$

and sends c_2^V to the smart contract which triggers an event to notify `V`. Finally, `V` retrieves c_2^V from the emitted event and computes

$$K_{dec} = \text{Dec}(c_2^V, SK_V)$$

This key will be used for CID decryption, as shown in Section 2.3.5. All interactions are shown in Figure 2.4

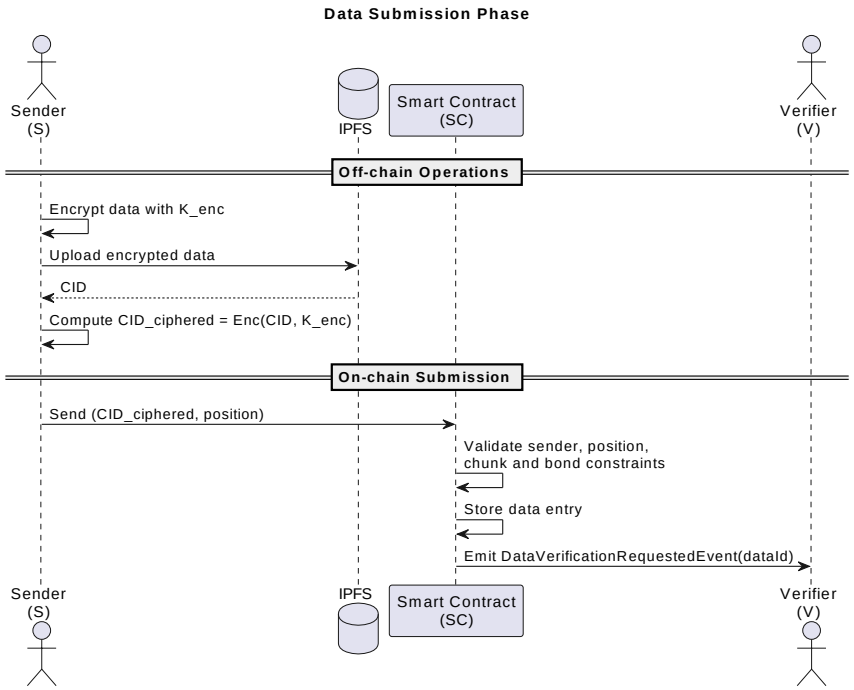

Figure 2.5: Data submission workflow

2.3.4 Data submission

Data submission is the main process of the protocol. It enables senders to contribute data references to the campaign. When `S` has data for the campaign, it encrypts data with K_{enc} key and uploads the encrypted data to IPFS. IPFS returns a Content Identifier (CID, see Section 1.1.4)

$$CID_{ciphered} = \text{Enc}(\text{CID}, K_{enc})$$

Then `S` sends $CID_{ciphered}$ to the smart contract along with the geographic position of the measurement. Once the data is registered, a specific event is emitted by the smart contract which is monitored by verifiers. This concludes the data submission phase and triggers the verification phase (described in Section 2.3.5). All interactions are shown in Figure 2.5

2.3.5 Verification

The verification phase is performed by verifiers, whose goal is to determine whether a given data entry meets the quality requirements of the campaign. To mitigate the risk of a single malicious or faulty verifier, each data entry is independently verified through multiple *verification slots*. The number of slots is conFigured to be odd, so

that a strict majority decision is always reachable and ties are structurally avoided. Slots are not statically assigned: each verifier must explicitly request authorization, and the smart contract elects it to an available slot only if the eligibility and cooldown constraints are satisfied.

A verifier `V` can request authorization either upon receiving a `DataVerificationRequestedEvent` or by polling autonomously. Please note that the smart contract is not constrained to return the data entry that triggered the event: any pending data entry eligible for verification can be returned.

The authorization request has two possible outcomes:

- The `VerifierElected` event is emitted, carrying among its parameters the ciphered CID and the address of the elected verifier;
- The authorization is denied, in which case `V` must wait before attempting again.

When `V` is elected, it can proceed with the verification. `V` computes:

$$CID = \mathrm{Dec}(CID_{ciphered}, K_{dec})$$

`V` then retrieves the encrypted data from IPFS using the CID and performs quality checks on it. After the checks, `V` submits the verification result to the smart contract.

As soon as a strict majority of slots agrees on a result (either positive or negative), the smart contract flags the data entry as `VALID` or `INVALID`, and the verification process concludes. Any remaining unfilled slots are no longer needed and are effectively discarded. If `V` cannot complete the task or misses the deadline, it can submit the special result `CANNOT_COMPLETE`; in this case, the verification slot is released and becomes available for reassignment. Once the data entry is flagged, payments and penalties are applied, as described in Section 2.4. All interactions are shown in Figure 2.6

The full data lifecycle is shown in Figure 2.7.

2.4 Incentive mechanisms

The incentive mechanism consists of two different approaches: rewarding actors to encourage honest behaviour and penalizing actors discouraging dishonest behaviours. These approaches are enforced through a rewarding system (Section 2.4.2) and a bond system (Section 2.4.3).

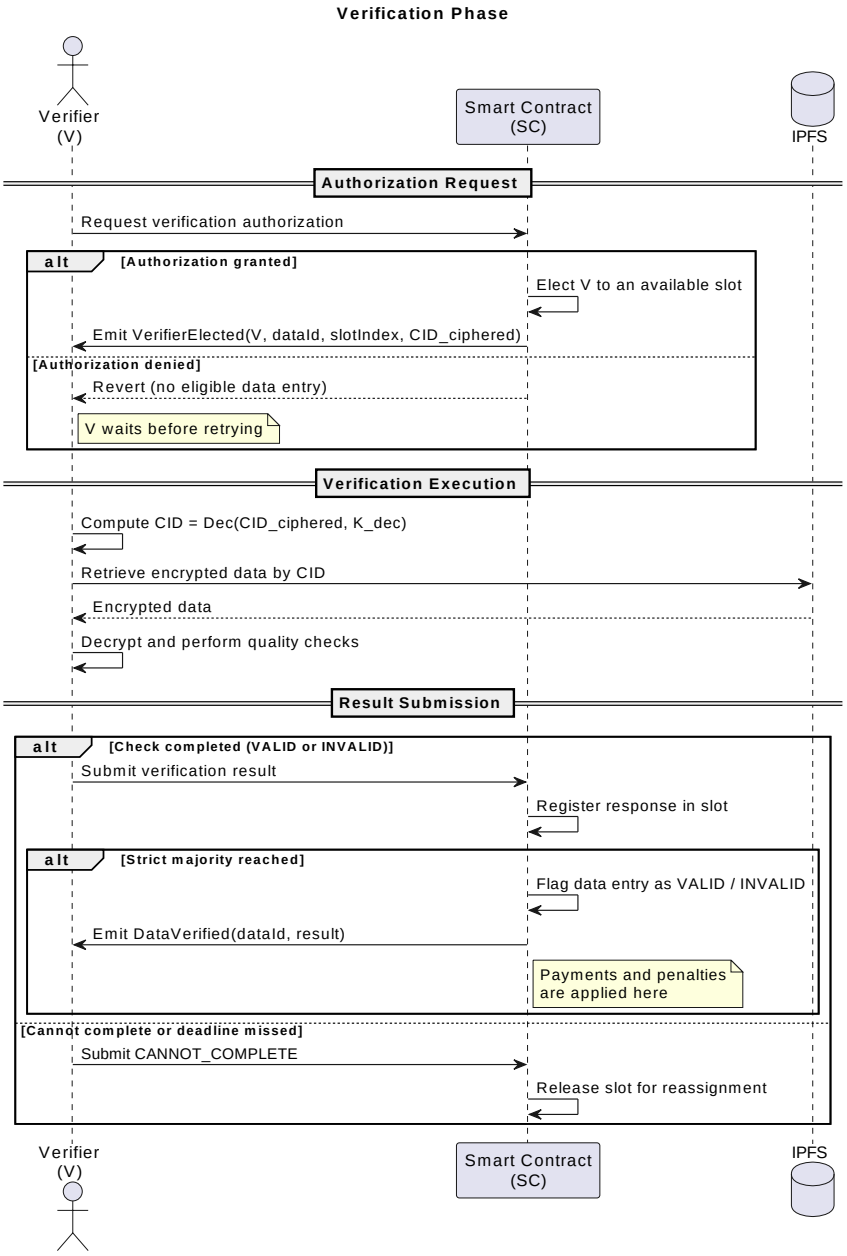

Figure 2.6: Verification process workflow

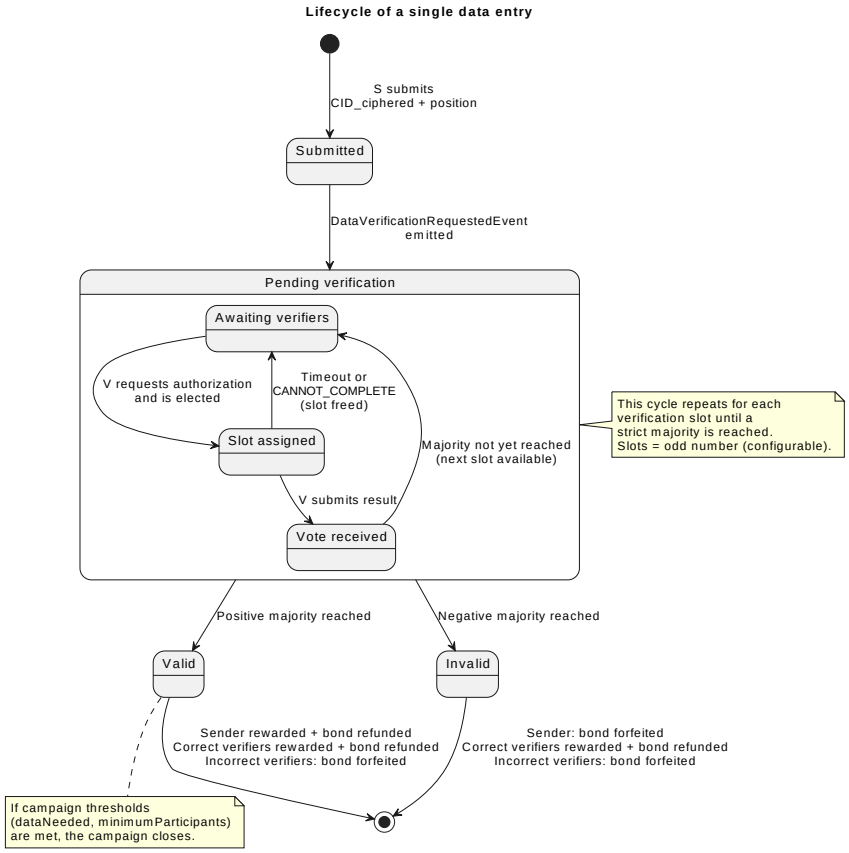

Figure 2.7: Lifecycle of a single data entry

2.4.1 Data Sent Token (DST)

Data Sent Token (DST) is a custom ERC-20 [6] token used to incentivize participation in the campaign, following the approach of reward-based crowdsensing incentive schemes such as [29].

2.4.2 Rewarding system

The rewarding system is configurable by `B`. The working process is quite straightforward: when a data entry is flagged as `VALID` or `INVALID` the rewarding process is triggered:

1. If the data is `VALID`, the smart contract rewards `S` with the conFigured payout. If the data is `INVALID`, `S` receives no payout.
2. Verifiers, on the other hand, are rewarded conditionally: only verifiers who voted according to the final verification result receive the conFigured payout. Verifiers who submitted a contradicting vote receive no reward.

2.4.3 Bond system

A bond is a refundable collateral, denominated in DST tokens, that senders and verifiers must deposit before contributing to the campaign. It serves as a commitment device: honest behaviour results in the bond being returned, while invalid contributions or incorrect verifications lead to its forfeit.

The bond system, like the rewarding one, is fully configurable by `B`, and operates alongside the rewarding system (Section 2.4.2). Bond tokens are refunded together with the associated reward, in order to minimize the number of transactions and the overall gas cost. Conversely, when a data entry is flagged as `INVALID`, the sender's bond is forfeited; similarly, verifiers whose vote contradicts the final verification result forfeit their bond.

The bond mechanism differs between senders and verifiers. Verifiers deposit a bond equal to `maxDataPerVerifierPerBond` $\times$ `dataVerifierBondPerData`, which covers multiple verifications; when the bond is insufficient, the transaction is reverted and the verifier can top it up and retry. Senders, on the other hand, are required to deposit `maxDataPerSender` $\times$ `dataSenderBondPerData` upfront. Since the bond decreases by `dataSenderBondPerData` for every submission and cannot be topped up, it effectively caps the number of contributions a sender can make per campaign.

For this reason, the bond payment constitutes an intermediate phase between enrollment and data submission. Bond payment is located in the complete workflow as

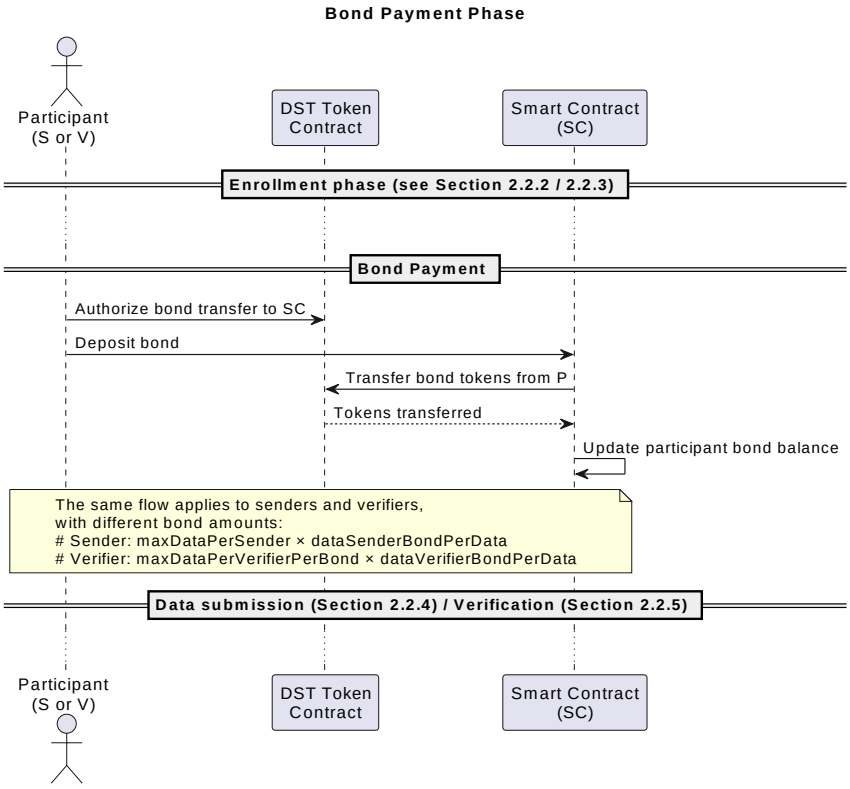

Figure 2.8: Paying bond workflow

shown in Figure 2.8. All parameters shown in this section are explained in Table A.1

The full campaign lifecycle is shown in Figure 2.9.

2.5 Security analysis and other issues

In this section the security properties of the protocol are analyzed and known limitations are discussed. The following sections discuss how to address the exposed issues and limitations of this protocol, or their extension.

2.5.1 Formal verification (ProVerif)

The presented protocol has been formally verified using ProVerif and is secure under the Dolev-Yao threat model [10; 9] if all the actors behave honestly: an external adversary $\mathcal{A}$ cannot recover the plaintext CID from $CID_{ciphered}$ and cannot decrypt the data stored on IPFS. Modeling and execution are described in the Appendix B.

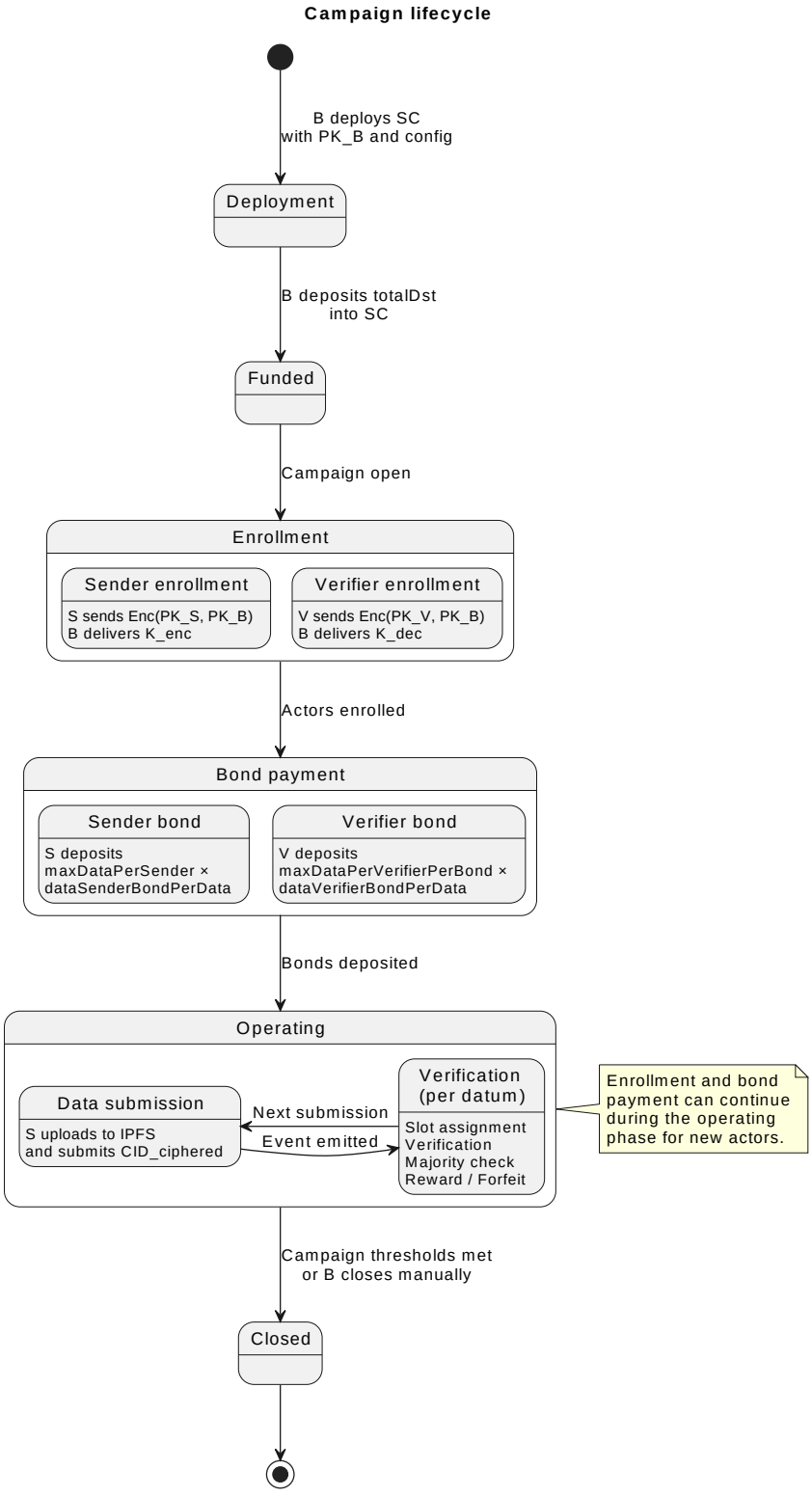

Figure 2.9: Campaign lifecycle

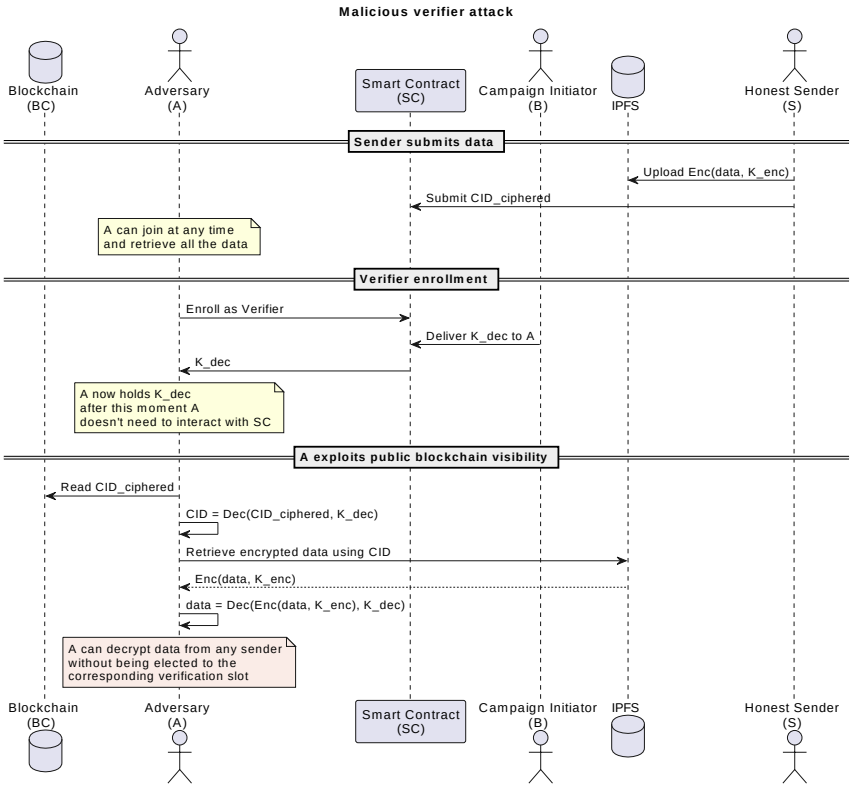

Figure 2.10: Malicious verifier attack in action

2.5.2 Malicious verifier attack

In the current protocol, the adversary $\mathcal{A}$ can join as a verifier and if the campaign initiator B accepts them as a verifier, then $\mathcal{A}$ is able to read all the data transmitted, regardless of which verifier is elected for a specific data entry. This follows from the public visibility of $CID_{ciphered}$ values on the blockchain and the shared K_{dec} among all enrolled verifiers. Since all transactions are publicly visible on the blockchain, $\mathcal{A}$ can observe every $CID_{ciphered}$ submitted by any sender and can decrypt $CID_{ciphered}$ and the corresponding data using K_{dec}. This issue requires that the senders must trust not only the campaign initiator, but also every enrolled verifier. The attack is shown in Figure 2.10.

2.5.3 Geographic incentive imbalance

The proposed protocol introduces a coordinate system and uses it to partition collected data into geographic chunks. Consider the following scenario: `B` starts a campaign in an area where some chunks cover urban zones and others cover rural

ones. Since urban areas have significantly more inhabitants than rural ones, it is reasonable to assume that data submissions concentrate in chunks covering cities. In the current protocol, the rewarding system offers no additional incentive to collect data in less populated areas. Furthermore, the privacy of senders in rural areas is more at risk due to lower population density, as fewer submissions per chunk make individual contributions easier to identify, a concern consistent with the broader spatio-temporal privacy risks in MCS [36]. The current protocol does not differentiate incentives by geographic density, leaving rural coverage and the associated privacy risk unaddressed. A possible extension could introduce density-aware reward multipliers, encouraging submissions in underrepresented chunks while discouraging over-contribution in saturated ones.

2.5.4 Centrality of B

The centrality of the campaign initiator can be considered a threat to the protocol, but it is also essential to its design: `B` funds the campaign by paying for data collection and the verification process. `B` has a direct economic interest in behaving honestly; moreover, the `totalDst` amount (described in Table A.3) is locked in the smart contract and cannot be withdrawn by `B`. However, if `B`'s private key SK_B is compromised, an adversary gains the ability to decrypt all actors' public keys and forge key deliveries, compromising the entire campaign. Distributing `B`'s role through techniques such as threshold cryptography [37] or multi-party key generation could mitigate this risk and is left as a possible future extension.

2.5.5 Address linkability

Senders interact with the smart contract through a single account and cannot change their address throughout the campaign. As a consequence, all submissions from the same sender are linkable: an observer can count the number of data entries a specific address has submitted and infer activity patterns. If a sender reaches the `maxDataPerSender` limit (described in Table A.1) and wishes to continue contributing, they must create a new account and deposit the bond again. The transfer of DST should be made with privacy-first currencies in DEX swappers for unlink the new account to the old one [36]. If the sender behaved honestly during the first submissions, the bond is fully refunded and can be reused for the new account. This trade-off was accepted deliberately: using a single address simplifies bond accounting and enables the `maxDataPerSender` cap, at the cost of submission linkability.

2.5.6 Verifier penalties

In the current protocol, the only penalty a verifier faces for incorrect behaviour is the loss of the deposited bond, which may be insufficient to deter sustained malicious activity. A possible improvement would introduce a reputation mechanism: past incorrect verifications should be recorded and forgiven only after n subsequent correct verifications, or the verifier should be removed after m accumulated incorrect verifications. This ensures, on the one hand, that the verifier maintains honest behaviour over time, and, on the other, that occasional mistakes do not lead to immediate removal.

Furthermore, a verification slot manipulation attack is possible. Since all verification results are publicly visible on the blockchain, an elected verifier can delay its submission and observe the votes of other verifiers. Once the majority outcome becomes apparent, the verifier submits a vote aligned with the majority just before the timeout, guaranteeing the reward without performing any actual data validation.

A robust solution would require a commit-reveal scheme [38; 39], where verifiers first commit a hash of their vote and later reveal the actual result. This would prevent verifiers from observing others' votes before submitting their own. However, implementing commit-reveal on-chain introduces additional gas costs and protocol complexity, and is left as a future extension.

2.5.7 Sybil attack

In the context of decentralized protocols, a Sybil attack occurs when a single adversary creates multiple identities to gain disproportionate influence over the system [40]. In this protocol, a Sybil attack targets the verifier role: an adversary could create enough identities to control multiple verification slots,with two possible goals: accessing all campaign data without legitimate participation, or accumulating a majority of slots to drain rewards by forcing arbitrary verification outcomes.

The Sybil attack is partially mitigated by two mechanisms:

- **Economic barrier**: each verifier identity must deposit a bond of $n \times$ `dataVerifierBondPerData` $\times$ `maxDataPerVerifierPerBond` DST, raising the cost of the attack proportionally to the number of identities created [27; 29];
- **Enrollment control**: all verifiers must be accepted by `B` during enrollment, providing an off-chain vetting opportunity.

However, these defenses may be insufficient against a well-funded adversary (e.g., a state actor) for whom the bond cost is negligible. In such a scenario, the primary defense remains `B`'s ability to vet verifier enrollment requests through off-chain identity

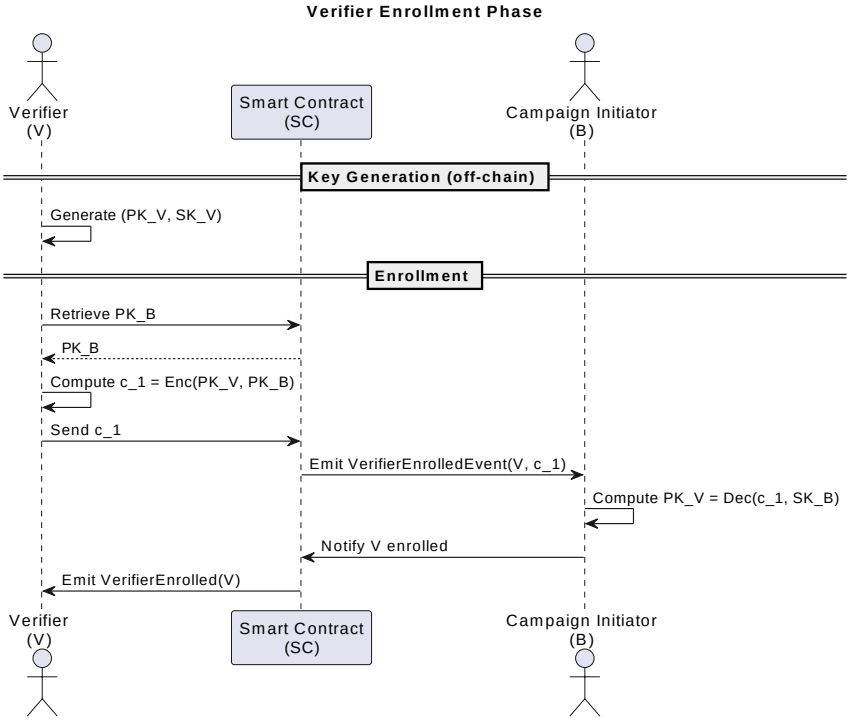

Figure 2.11: Novel verifier enrollment phase

verification mechanisms.

2.6 Addressing issues

In this section some solution at design-level are proposed to solve the issues described in Section 2.5.2, Section 2.5.3 and Section 2.5.6.

2.6.1 Addressing trusted verifier issue

From the sender's perspective, the proposed protocol requires trusting not only `B` but all current and future verifiers. This represents a strong discouragement for senders, as their data can be accessed by any enrolled verifier through the public blockchain, particularly if `B` accepts a malicious verifier. This issue can be addressed by modifying the verifier enrollment logic, as shown in Figure 2.11, and the data submission phase, as shown in Figure 2.12.

The idea is straightforward: during verifier enrollment, `B` retains the public key of every verifier locally. When a datum is submitted, `B` is notified through an

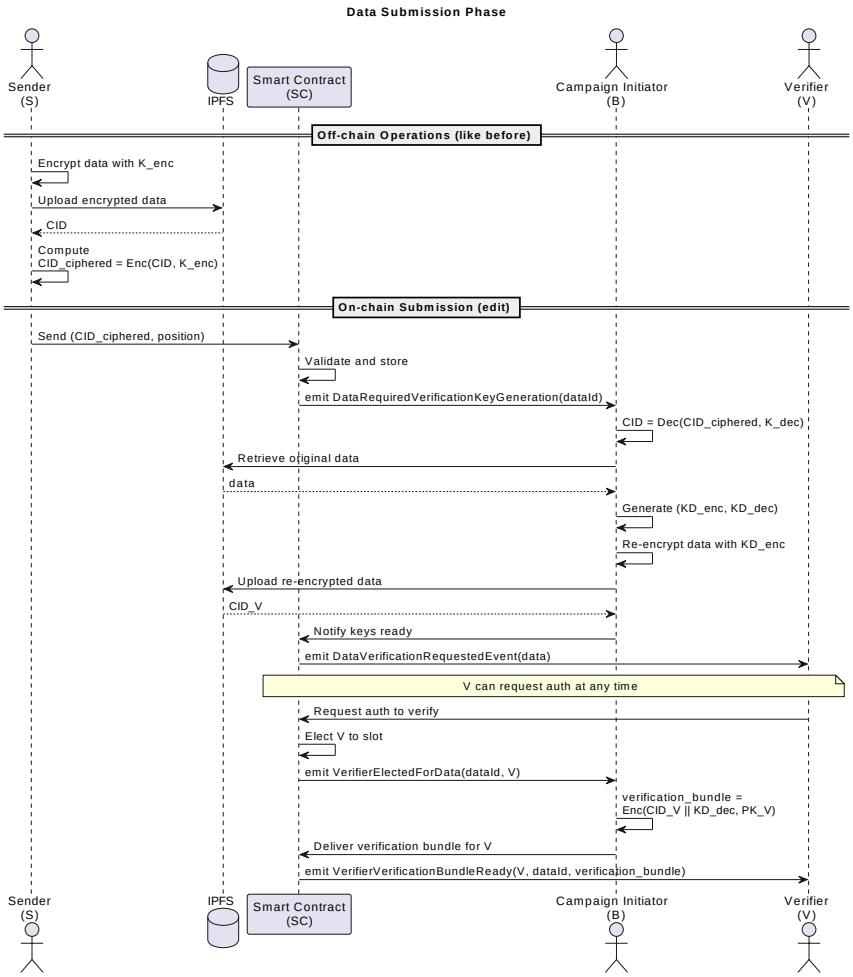

Figure 2.12: Data submission and novel verification workflow

event and generates a unique asymmetric key pair (KD_{enc}, KD_{dec}) for that data entry. Subsequently, `B` decrypts $CID_{ciphered}$ using K_{dec} to obtain the original CID, retrieves the encrypted data from IPFS, and decrypts it using K_{dec}. `B` then re-encrypts the data using KD_{enc} and uploads it to IPFS, obtaining a new content identifier CID_V. `B` concludes this step by notifying the smart contract that the per-datum key material is ready.

Once the smart contract is informed by `B`, it emits an event that verifiers monitor; upon receiving it, they can request verification authorization. When a verifier is elected to a slot, the smart contract notifies `B`, who constructs a `verification_bundle` containing CID_V and KD_{dec}, encrypted using the elected verifier's public key PK_V. Finally, `B` delivers the bundle to the smart contract, which emits an event notifying the elected verifier.

This extension eliminates the shared K_{dec} vulnerability: each verifier can only access data entries for which it has been specifically elected. However, this comes at the cost of increased workload for `B`, who must perform one re-encryption cycle per data entry, and additional IPFS storage, as each datum now exists in two encrypted forms.

2.6.2 Addressing geographical incentive issue

Section 2.2 describes the chunking system and Section 2.4 presents the incentive mechanisms to contribute to the campaign, but there is no incentive to contribute data to one chunk over another, with all the risks described in Section 2.5.3. The solution is to introduce a reward decay function that reduces the payout as a chunk approaches saturation, encouraging senders to target underrepresented areas.

Limited data per chunk If `maximumDataPerChunk` > 0, the saturation ratio is defined as follows:

$$\alpha = \frac{\texttt{chunkCount}}{\texttt{maximumDataPerChunk}} \tag{2.3}$$

where `maximumDataPerChunk` is defined in Table A.1 and `chunkCount` is the current number of data entries submitted within a chunk. The reward r is then calculated as:

$$r = \texttt{dataSenderPayout} \cdot (1 - \alpha)^{\frac{1}{2^k}} \tag{2.4}$$

where `dataSenderPayout` is defined in Table A.3 and k is the `rewardDecayFactor` as defined in Table A.5.

The parameter k controls the decay rate: higher values of k result in a gentler decay, allowing more submissions before the reward drops significantly, while lower values produce a steeper curve that penalizes contributions to already-populated chunks

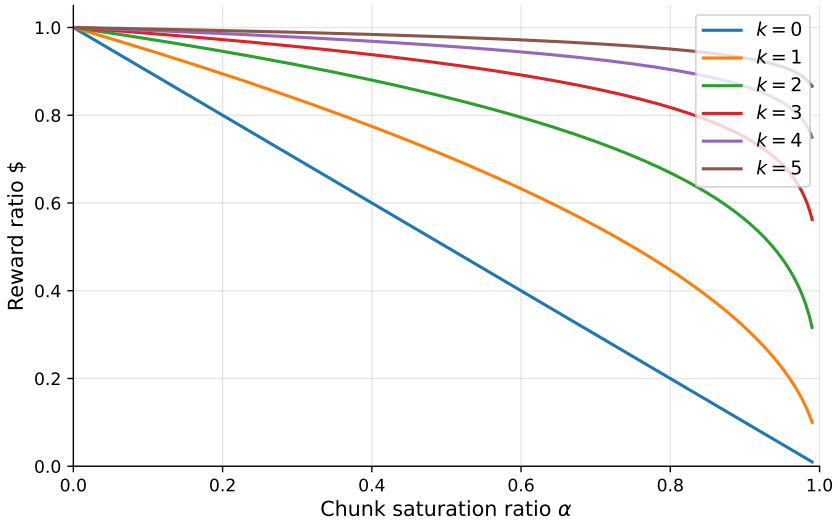

Figure 2.13: Reward decay as a function of chunk saturation ratio α for different values of k

more aggressively. When $\alpha = 1$, the chunk is saturated and no further submissions are accepted; this is enforced by a constraint in the smart contract.

If `maximumDataPerChunk` $= 0$, no per-chunk limit is enforced and the full reward is granted for every submission.

An example of the reward decay curve for different values of k is shown in Figure 2.13.

2.6.3 Addressing lack of verifier penalties

In the current protocol, verifiers whose submissions are incorrect only lose their `dataVerifierBondPerData`. However, this may not be sufficient to discourage sustained malicious behaviour over multiple verification cycles.

This issue can be addressed by introducing two mechanisms: an automatic removal system that tracks incorrect votes, and the ability for `B` to manually remove a verifier from the campaign. Two additional parameters are introduced:

- `kickVerifierAfterWrongVotes`: number of accumulated incorrect votes after which a verifier is automatically removed from the campaign;
- `pardonWrongVoteAfterRightVotes`: number of correct votes required to decrease the incorrect vote counter by one.

The forgiveness mechanism ensures that honest verifiers who occasionally make mistakes are not unfairly penalized over the course of a long campaign. When a verifier is removed, either automatically or by `B`, the remaining bond balance is returned to

the verifier. This design choice reflects the fact that removal itself is already a significant penalty, as the verifier loses access to all future rewards from the campaign.

Note that this mechanism does not address the slot manipulation attack described in Section 2.5.6, where a verifier deliberately delays submission to observe others' votes before responding. Since the adversary's votes are aligned with the majority, they are not counted as incorrect and the automatic removal is never triggered. Addressing this attack would require a commit-reveal scheme, which is left as a future extension.

All the solutions proposed in this section are implemented in an extended version of the smart contract, described in Chapter 3.

Chapter 3

Smart contract implementation

In the previous chapter the protocol was explained, some issues are identified, and mitigation strategies are proposed for the most critical ones. This chapter describes how the entire protocol is implemented. The development environment is made of Solidity on Ethereum [18] and Hardhat as the development and testing framework. In addition, Chapter also describes how `PrivacyFirstContract` and its extension, `PrivacySecondContract` interacts and which functions are overridden and the motivation behind every choice.

3.1 Architecture

The architecture is divided between on-chain operations and off chain ones, this distinction exists because a lot of actions are not executable on the blockchain, like IPFS upload/download and ciphers usage are not permitted on smart contacts due to the non-deterministic nature of operations such as encryption [34]. The off-chain operations are encapsulated in an SDK that allows actors to interact with the protocol without dealing directly with blockchain primitives. The structure is composed by two contracts: the base protocol inspired to [7] and another one that extends the base protocol with the mechanisms proposed in Section 2.6.

3.1.1 Contract hierarchy

The `PrivacySecondContract` extends `PrivacyFirstContract`, reusing its core logic without modification. The functionalities not covered by Section 2.6, such as sender enrollment, bond system, majority consensus, and lifecycle are inherited without modification. An overview of functions and more representative fields are shown in Figure 3.1.

Figure 3.1: Class hierarchy of the two smart contracts

There are four `internal` functions that have been overridden in the `PrivacySecondContract` to adapt them to the contract needs.

Since the extended contract needs to delivery verification bundles, to prevent duplicate delivery, a `_bundleDelivered` mapping is introduced. For this reason, once the bundle is sent the function `_emitVerifierElected` is overridden in `PrivacySecondContract` to reset the delivery state, allowing a new bundle to be delivered if the verifier slot is reassigned. This allows the usage of the same location of memory, the delivery of verification bundles is required in order to address the issue of verifiers trust, described in Section 2.6.1.

Since in Section 2.6.2 is described a mechanism to incentive data sensing in heterogeneous geographical areas, it is necessary to change the rewarding system. For this reason, the method `_rewardSender` is overridden and the only modification made is change the calculation of total payout. In the first version of the contract, the reward is always the full base payout, in the novel one, the contract reads the pre-computed reward from storage. The effective payout is calculated at the CID publish time, the contract execute the calculus in the `_computeDecayedReward` function. That function executes k times the sqrt of a value (as shown in equation 2.4), in *O(log n)* time using the babylonian method [41], to execute this operation, all values are scaled by a factor of `1E18` since the EVM, as noted in Section 2.2, is unable to execute floating point calculus.

The problem of missing verifiers penalties, in addition to bond loss, is discussed in Section 2.6.3. To address this issue, is required to modify how the smart contract collects the verifiers responses. For this reason, the functions `_rewardCorrectVerifiers` and `_tryRewardVerifier` are modified and the responses are saved in mappings that keeps the count of correct votes and wrong ones. These mappings are required to enforce the kick of systematically wrong voters, as proposed in Section 2.6.3.

The `PrivacySecondContract` overrides also public functions; the function used to publish decryption key (`publishVerifierDecryptionKey`) in the `PrivacyFirstContract` publishes the decryption key through the VerifierEnrolled event, using a key shared among all enrolled verifiers. In the novel contract, the function is overridden and publish the same event without the key, because, in order to avoid a single decryption key shared among all the verifiers, like described in 2.6.1. For these reasons, the function `publishIpfsCid`, is overridden and instead of adding the data in the pending data structure, the data entry requires that `B` to generate a per-data key pair (KD_{enc}, KD_{dec}), after which, `B` calls `notifyVerificationKeyReady` and the smart contract adds the data to pending data structure. The novel event `VerifierElectedForData` notifies `B` that a verifier has been elected, upon which `B` delivers a verification bundle via `deliverVerificationBundle`. Then a `VerifierVerificationBundleReady` is

emitted and the verifier can finally calculate and submit the verification result using `submitVerifyResult` overridden function that checks the verification bundle delivery.

In addition, two new functions are introduced: `getExpectedSenderReward` and `kickVerifier`, the first one is informative and useful for users to understand which payout they will receive if they send data for a specific chunk in that moment. The second one can be used from by campaign initiator `B` to exclude a verifier from the crowdsensing campaign. In this case the verifier's bond is returned, as the exclusion from the campaign already constitutes a sufficient penalty, consistently with the design choice described in Section 2.6.3.

A description of all overridden functions between the contracts is shown in Table 3.1

3.1.2 SDK layer

The SDK layer is responsible for all off-chain operations that the smart contracts cannot or should not perform. All the cryptographic operations cannot be executed on-chain because the smart contracts are deterministic, so, in the blockchain environment can exist an encryption scheme that is deterministic, because EVM is deterministic [34], so:

$$\texttt{ENC}_k(m) = c = \texttt{ENC}_k(m). \quad \forall m \tag{3.1}$$

Equation 3.1 shows that given the same message m, the same ciphertext c is always produced. Deterministic encryption scheme has been shown to be insecure [42] because if a sender wants to send to a specific receiver the same message two times an adversary $\mathcal{A}$ can understand that the message m was submitted twice because they have seen c flow two times on the network. Secure schemes such as RSA-OAEP and AES-GCM require a fresh random value per invocation [43; 44]; the EVM, having no trusted randomness source, cannot provide this. In addition, all the data sent to the smart contract, including the function call arguments, are visible and stored in the blockchain, so, if in a certain use-case is acceptable that $\texttt{ENC}_k$ is deterministic and the encryption were performed within the smart contract, the content of m is in any case stored and visible over the blockchain. Furthermore, performing cryptographic operations off-chain avoids the gas overhead that on-chain execution would introduce.

For this reason the SDK must be executed by the protocol actors. In addition, the proposed SDK exposes an interface that makes it easy for users to parrfc8017ticipate in the campaign and guarantees the correct execution of operations.

The UML class diagram of the SDK layer is shown in Figure 3.2.

Function	PFC	PSC	Reason
`publishVerifier-DecryptionKey`	Publishes decryption key on-chain via `VerifierEnrolled`	Emits `VerifierEnrolled` without key; key is generated off-chain	Trusted verifier flow (2.6.1)
`publishIpfsCid`	Adds data to pending; reward is fixed base payout	Stores data awaiting key generation; computes decayed reward	Trusted verifier flow and reward decay (2.6.1, 2.6.2)
`submitVerifyResult`	No prerequisite on bundle delivery	Requires `deliverVerificationBundle` to be called first	Trusted verifier flow (2.6.1)
`_emitVerifierElected`	Emits election event and assigns slot	Additionally resets `_bundleDelivered` flag	Allows bundle redelivery on slot reassignment
`_rewardSender`	Pays fixed base payout	Reads precomputed reward from `_computedSenderReward`	Reward decay (2.6.2)
`_rewardCorrectVerifiers` `_tryRewardVerifier`	Distributes rewards based on majority	Additionally records correct and wrong votes per verifier	Verifier penalties (2.6.3)

Table 3.1: Overridden functions between `PrivacyFirstContract` (PFC) and `PrivacySecondContract` (PSC)

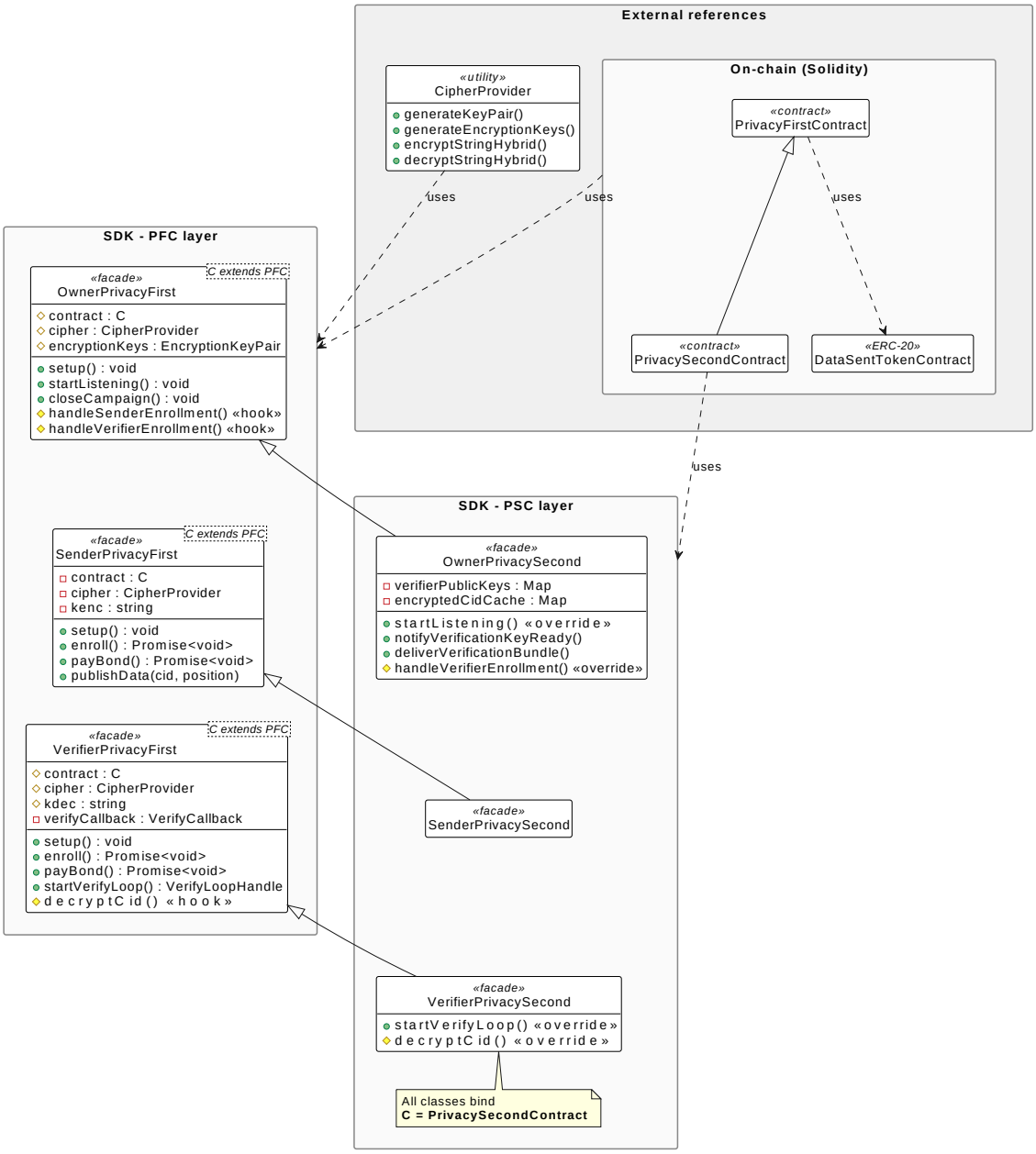

Figure 3.2: UML class diagram of SDK. The PrivacyFirstContract (PFC) defines the base facades together the template method hooks, the PrivacySecondContract (PSC) ovverddies the hooks to implement extensions described in Section 2.6

In the protocol there are, as before, three types of actors, described in Section 2.1. In the SDK architecture, there are three classes, once for each type of Actor.

The campaign initiator `B` can use two classes, `OwnerPrivacyFirst` and `OwnerPrivacySecond`, respectively for the first and the second version of the protocol. Through the `OwnerPrivacyFirst.setup` method, `B` can deposit the initial token supply and register all the listeners using `OwnerPrivacyFirst.startListening`. For the `OwnerPrivacySecond` contract, the method `startListening` is overridden enabling the registration of listeners for the new events. With event listeners `B` is able to handle enrollments automatically.

The sender `S` can use two classes, `SenderPrivacyFirst` and `SenderPrivacySecond`. The `SenderPrivacyFirst.setup` method allows `S` to enroll to the campaign and pay bond automatically, in addition, the method `SenderPrivacyFirst.publishData` automatically handles the encryption and publishes the ciphered CID. The `SenderPrivacySecond` was introduced only to pass the correct type to the parametric variable `C`. Since Hardhat generates contract types from the ABI rather than from a shared base class, `SenderPrivacySecond` does not extend `SenderPrivacyFirst`; however, TypeScript's structural typing allows it to satisfy the type parameter `C`, as both classes expose the same sender-facing interface.

The verifier `V` can use two classes, `VerifierPrivacyFirst` and `VerifierPrivacySecond`. The `VerifierPrivacyFirst.setup` method allows the verifier to enroll to the campaign and automatically handle the exchange of K_{dec} and pay bond. The novel class `VerifierPrivacySecond` overrides the `enroll` method, called by `setup`, and also overrides the `requestAuthToVerify` to listen for the new event `VerifierElectedForData`.

3.1.3 Off-chain cryptographic utility

The cryptographic utility follows a hybrid scheme: each CID is encrypted with a freshly generated symmetric key, and that key is in turn encrypted under the recipient's public key. Only the encrypted CIDs travel on the blockchain and the auxiliary public channel.

The symmetric layer uses `AES-256-GCM`, a NIST-recommended authenticated encryption scheme [44] that provides both confidentiality (via AES in CTR mode) and integrity/authenticity (via the GHASH authentication tag) in a single pass. A new symmetric key is sampled at every interaction, so compromise of one session does not affect any other.

The symmetric key is then wrapped with `RSA-OAEP`, the standard padding scheme for RSA encryption defined in PKCS#1 v2.2 [43]. Its predecessor, PKCS#1 v1.5, is vulnerable to chosen-ciphertext attacks, which OAEP addresses by introducing

randomized padding.

Finally, the proposed SDK demonstrates that the protocol is fully implementable by each actor. This is not a production-ready library but a reference implementation; as such, the keys live in the program's memory, for this reason, keys are not protected by a hardware security module.

3.2 Protocol phases implementation

This section discusses the main implementation decisions. All implementation choices are made in accordance with the protocol design described in Sections 2.2, 2.3, 2.4, 2.5 and 2.6.

Since the `PrivacySecondContract` (`PSC`) is an extension of `PrivacyFirstContract` (`PFC`), both contracts are presented. New and overridden methods are highlighted to show the extensions introduced, while methods inherited unchanged from the base protocol are presented for clarity.

3.2.1 Initialization and deployment

The campaign initiator `B` generates their own key pair (PK_B, SK_B) using the `cipher` in the SDK.

```
// ...
const ownerKeys = cipher.generateKeyPair();
// ...
```

Listing 3.1: B's key pair generation

Where `cipher` is an object used for all the off-chain cryptographic operations.

Afterward, `B` executes the contract deployment with the following code:

```
const privacyContract = <CF extends ContractFactory>.deploy(
    ownerKeys.publicKey,
    contractConfiguration,
    dataSentTokenAddress,
    defaultNovelParams,
);
```

Listing 3.2: `PrivacyContract` deployment

Where `CF` is a contract factory that can be `PrivacyFirstContract__factory` or `PrivacySecondContract__factory`, based on which version to deploy; the `ownerKeys.publicKey` is the `B`'s public key, which is required by the contract to settle it to

verifiers and senders; the contract configuration object (`contractConfiguration`) is described in Appendix A; the `dataSentTokenAddress` is also required for approving transfers from the contract account to the actors' accounts. Finally, the `defaultNovelParams` object is required only for the `PSC` deployment.

On the contract side, upon deployment, the constructor is executed and all configuration checks are performed

```
constructor(string memory ownerPubKey, Config memory config,
    address dstContractAddress) {
  require(config.rewardConfig.dataSenderPayout
    > config.rewardConfig.dataSenderBondPerData,
    "The␣senders␣payout␣isn't␣greater␣than␣senders␣bond");
  require(config.rewardConfig.dataVerifierPayout
   > config.rewardConfig.dataVerifierBondPerData,
   "The␣verifiers␣payout␣isn't␣greater␣than␣verifiers␣bond");
  require(config.originCoordinates.isValid(),
    "Origin␣coordinates␣should␣be␣valid");
  require(config.needs.checksRequired % 2 == 1,
    "Checks␣must␣be␣odd");
  require(config.chunkConfig.chunk4Side > 0,
    "chunk4Side␣must␣be␣greater␣than␣0");
  require(config.chunkConfig.chunkSize > 0,
    "chunkSize␣must␣be␣greater␣than␣0");
  require(config.needs.verifierCooldown > 0,
    "Cooldown␣should␣be␣positive");
    // setters ...
}
```

Listing 3.3: `PrivacyFirstContract` constructor

After the deployment, the classes `OwnerPrivacyFirst` (`OPF`) or `OwnerPrivacySecond` (`OPS`) can be instantiated.

```
    constructor(signer: HardhatEthersSigner,
    contract: C,
    dstToken: DataSentTokenContract,
    ownerKeys: KeyPair,
    cipher: CipherProvider) {
    //setters for all the params above
    this.encryptionKeys = cipher.generateEncryptionKeys();
}
```

Listing 3.4: `OPF`'s constructor, inherited by `OPS`

As shown in Listing 3.4 the only business logic implemented by the constructor is the auto-generation of encryption keys. Finally, the owner can invoke `OwnerPrivacy *.setup` defined in `OPF` as shown in Listing 3.5. The `payInitialSupply` method automatically handles the supply payout to the contract account; the event listeners are then set up by invoking `startListening`.

```
async setup() {
    //...
    this.payInitialSupply(config.rewardConfig.totalDst);
    this.startListening();
    //...
}
```

Listing 3.5: `OPF`'s setup method, inherited from `OPS`

The `startListening` method is defined in `OPF` and overridden in `OPS` to support the additional events in the second version of the protocol, code is shown in Listings 3.6 and 3.7.

```
startListening(): void {
//...
this.contract.on(
    this.contract.getEvent("SenderEnrolledEvent"),
    async (sender: string, key: string) => {
        this.handleSenderEnrollment(sender, key)})

this.contract.on(
    this.contract.getEvent("VerifierEnrolledEvent"),
    async (verifier: string, key: string) => {
       this.handleVerifierEnrollment(verifier, key)})

this.contract.on(
    this.contract.getEvent("DataVerified"),
    (dataId: bigint, result: boolean) => {
    this.handleDataVerificationCompleted(dataId, result)})
}
```

Listing 3.6: `OPF`'s startListening method

```
override startListening(): void {
//...
super.startListening();

this.contract.on(
```

```
6       this.contract.getEvent("
            DataRequiredVerificationKeyGeneration"),
7       async (dataId: bigint, event: any) => {
8           this.handleDataRequiresKey(Number(dataId), event)})
9
10  this.contract.on(
11      this.contract.getEvent("VerifierElectedForData"),
12      async (verifier: string, dataId: bigint) => {
13          this.handleVerifierElected(verifier, Number(dataId))
                })
14  }
```

Listing 3.7: OPS's startListening method, uses the inherited method from parent class

3.2.2 Sender enrollment

The interface for the Sender side in both implementations of the protocol is the same. For this reason the SDK class `SenderPrivacySecond` (SPS) has no overridden methods. The only role for the class is to give the right contract interface to the superclass.

Class binding implementation is shown in Listings 3.8 and 3.9. For this reason, in this section, all methods discussed are defined in class `SenderPrivacyFirst` (SPF).

```
1   class SenderPrivacySecond
2       extends SenderPrivacyFirst<PrivacySecondContract> {}
```

Listing 3.8: Full `SenderPrivacySecond` class implementation

```
1   class SenderPrivacyFirst
2       <C extends PrivacyFirstContract = PrivacyFirstContract> {
3       //...
4       private readonly contract: C;
5       //...
6       }
```

Listing 3.9: `SenderPrivacyFirst` class declaration

Before publishing data to IPFS, the sender needs to execute the code shown in Listing 3.10.

```
1   const sender = new SenderPrivacySecond
2       (_sender, contract, dst, cipher);
3   sender.setup();
```

Listing 3.10: `SenderPrivacySecond` class usage

Where `_sender` is HardhatSigner, `contract` is a reference to the deployed contract, `dst` is the reference to the `DataSentToken` contract and `cipher` is the cipher utility in the SDK.

The constructor generates the sender's key pair (PK_S, SK_S). The `setup` method calls the enroll method, and the `payBond` method that approves the `DST` transfer to the smart contract, after approval the sender calls the `senderPayBond` function of the Smart Contract that completes the transfer and updates the internal balance. The `enroll` method is implemented as shown in Listing 3.11

```
//After SenderEncryptionKeyReady listener subscription
const ownerPubKey: string = contract.getInitiatorPublicKey();
const encryptedPK = cipher.encryptStringHybrid(ownerPubKey,
   keys.publicKey); //encrypt PK_S using PK_B.
contract.publishEncryptedSenderPublicKey(JSON.stringify(
   encryptedPK)); //Publish encrypted PK_S
//wait for key from SenderEncryptionKeyReady emission (will
   be saved in encryptedKenc)
kenc = cipher.decryptStringHybrid(keys.privateKey, JSON.parse
   (encryptedKenc)); //Received K_enc
```

Listing 3.11: `SenderPrivacyFirst` enroll method

The sender `S` uses the `publishEncryptedSenderPublicKey`, the implementation is shown in Listing 3.12. The function emits the `SenderEnrolledEvent` event, which is handled by the campaign initiator as shown in Listing 3.6.

```
function publishEncryptedSenderPublicKey(string calldata
   encryptedPubKey) public {
  require(_campaignOpen, "Campaign is closed");
  require(!_senders[msg.sender],"Sender has already joined");
  emit SenderEnrolledEvent(msg.sender, encryptedPubKey);
}
```

Listing 3.12: publishEncryptedSenderPublicKey method, implemented in `PFC` and inherited from `PSC`

When `B` is triggered by the event, their SDK is triggered and the code handler is shown in Listing 3.13, basically, the campaign initiator decrypts the ciphertext to get PK_S. Finally, K_{enc} is encrypted using PK_S and sends it via the Smart Contract to `S` using `publishSenderEncryptionKey`.

```
handleSenderEnrollment(senderAddress: string,
   encryptedPubKeyJson: string) {
  const bundle = JSON.parse(encryptedPubKeyJson);
  const senderPubKey = cipher.decryptStringHybrid(keys.
     privateKey, bundle);
  const encryptedKenc = JSON.stringify(
    cipher.encryptStringHybrid(senderPubKey, encryptionKeys.
       encryptionKey));
  contract.publishSenderEncryptionKey(
    senderAddress, encryptedKenc);
}
```

Listing 3.13: B's callback function for sender key publishing

The `publishSenderEncryptionKey` call completes the registration and creates a struct to store the sender's information. Finally, the function emit the SenderEncryptionKeyReady event, which is awaited from the sender, like shown in Listing 3.11. The handler is quite simple, since the `SenderEncryptionKeyReady` has two parameters (senderAddress, encryptionKeyCiphertext), if `S` sees their address in senderAddress the handler is triggered and removed. The ciphertext of K_{enc} is returned; the sender then uses SK_S to decrypt and obtain K_{enc}.

With this implementation the sender enrollment reflects the design choices made in Section 2.3.2.

3.2.3 Verifier enrollment

This Section describes the verifier enrollment phase for the `PrivacyFirstContract`. The extended protocol version is discussed in Section 3.4.1 which implement the modifications in Section 2.6.1.

In the `PrivacyFirstContract` the implementation is very similar to the one realized for senders, shown in Section 3.2.2.

The verifiers can use the `VerifierPrivacyFirst` (`VPF`) class to join the crowdsensing campaign. The constructor of the class automatically generates (PK_V, SK_V) key pair and require the verifyCallback to execute on every data to check.

After the object initialization, the verifier has to call the `setup` method. The method calls the `enroll` method and `payBond`, which handles the bond payment, like the one implemented for the senders. The workflow is the same for senders but with different functions to invoke and event to wait for.

The general workflow is the following:

1. `V` register a listener for the `VerifierDecryptionKeyReady` event;
2. `V` retrieve PK_B which is embedded in the smart contract;
3. `V` encrypt PK_V using PK_B and sends it to the smart contract using the `publishEncryptedVerifierPublicKey` function;
4. The Smart Contract triggers via `VerifierEnrolledEvent` the campaign initiator which is listening for this event as shown in Listing 3.6;
5. `B` via the `handleVerifierEnrollment` handler, encrypts K_{dec} using PK_V, used for data decryption during verification;
6. `B` sends the calculated ciphertext to the smart contract using `publishVerifierDecryptionKey` which updates the Smart Contract state and the verifier is officially in the campaign, finally, `VerifierDecryptionKeyReady` event is emitted;
7. `V` is triggered by the event and get K_{dec}

After these steps the verifier can verify data and start the verification workflow, that will be discussed in Section 3.2.6.

With this implementation the verifier enrollment reflects the design choices made in Section 2.3.3.

3.2.4 Data submission

The data submission is the core phase where the crowdsensing campaign obtains data.

The SDK implementation for the senders is written in `SenderPrivacyFirst` (`SPF`) and is shown in Listing 3.14.

```
1 async publishData(plainCid: string, position: any) {
2 if (!this.kenc) throw new Error("Not␣enrolled␣-␣Kenc␣not␣
    available");
3
4 const encrypted = this.cipher.encryptStringHybrid(this.kenc,
    plainCid);
5 const cidBytes = toUtf8Bytes(JSON.stringify(encrypted));
6 return this.contract.connect(this.signer).publishIpfsCid(
    cidBytes, position);
```

Listing 3.14: Sender `publishData` method to send data to the campaign

Basically, the method checks if K_{enc} is available, if not, it throws an error, otherwise, uses K_{enc} to encrypt the CID returned by the IPFS node after uploading the

encrypted data, where they have submitted the data and publishes the encrypted CID using the smart contract function `publishIpfsCid`.

The function, which is shown in Listing 3.15, performs a series of checks such as bond availability, maximum data count per chunk and per sender, and geographical area validation. If any of these checks fails, then the entire transaction is reverted, but if the data is submitted a struct to store submission information is created and the data index is added to the pending queue, waiting for verification. In this phase of the protocol, no action from the campaign initiator is required; the `DataVerificationRequestedEvent` event is emitted to notify verifiers that new data is available for verification.

```
1
2  function publishIpfsCid(bytes calldata encryptedCid,
       GlobalPosition calldata position) external virtual {
3    require(_campaignOpen, "Campaign␣is␣closed");
4    require(_senders[msg.sender], "Only␣senders␣allowed...");
5    require(position.isValid(), "Invalid␣Coordinates");
6    require(_senderInformation[msg.sender].dataSubmitted
7      < _config.needs.maxDataPerSender
8      || _config.needs.maxDataPerSender == 0,
9    "Max␣data␣limit␣reached");
10
11   // if maxDataPerSender == 0 (unlimited), bond check is
       skipped
12   // since total bond cannot be pre-computed
13   require(_senderInformation[msg.sender].bondBalance
14     >= _config.rewardConfig.dataSenderBondPerData
15     || _config.needs.maxDataPerSender == 0,
16     "Insufficient␣DST␣balance␣in␣bond");
17
18   PackedPosition memory packed = position.packCoordinates();
19   (uint16 chunkIndex, bool inside) = _getChunkIndex(packed);
20   require(inside, "Outside␣campaign␣area");
21   require(_dataPerChunkReceived[chunkIndex]
22     < _config.needs.maximumDataPerChunk
23     || _config.needs.maximumDataPerChunk == 0,
24   "Too␣many␣data␣in␣this␣chunk!");
25
26   _dataEntries[_dataCount] = DataEntry({
27     sender: msg.sender,
28     encryptedCid: encryptedCid,
29     status: DataStatus.PENDING,
30     completedChecks: 0,
```

```
    positiveChecks: 0,
    assignedChecks: 0,
    chunkIndex: chunkIndex,
    earliestSlotDeadline: 0});
  _dataPerChunkReceived[chunkIndex]++;
  _senderInformation[msg.sender].dataSubmitted++;
  emit DataVerificationRequestedEvent(_dataCount);

  _addToPending(_dataCount);

  _dataCount++;

  if (_config.needs.maxDataPerSender > 0) {
    unchecked {
      _senderInformation[msg.sender].bondBalance =
         _senderInformation[msg.sender].bondBalance -
      _config.rewardConfig.dataSenderBondPerData;
    }}
}
```

Listing 3.15: `publishIpfsCid` function implementation

This code implements the Data submission workflow described in Section 2.3.4.

3.2.5 Spatial data validation

The spatial data validation is required for the protocol because the data need to be localized in a certain area of the geographical space. In the `publishIpfsCid` function there are some lines of code that handle these checks and locate the data in the appropriate chunks.

```
function publishIpfsCid(//...
  GlobalPosition calldata position)
  external virtual {
  //...
  require(position.isValid(), "Invalid␣Coordinates");
  //...
  PackedPosition memory packed = position.packCoordinates();
  (uint16 chunkIndex, bool inside) = _getChunkIndex(packed);
  require(inside, "Outside␣campaign␣area");
  //...
  _dataPerChunkReceived[chunkIndex]++;
  //...
}
```

Listing 3.16: `publishIpfsCid` function spatial validation code

The `publishIpfsCid` function requires, among others, the `position` parameter of type `GlobalPosition`, has four fields, two for latitude and two for longitude. For both coordinates a field is used for the integer part, the second one for the fractional part as up to seven decimal digits. The struct is shown in Listing 3.17.

```
struct GlobalPosition {
  int32 latitudeInteger; // [-90, +90]
  int32 latitudeDecimal; // (-10_000_000, +10_000_000) sign
     used iff integer is 0.
  int32 longitudeInteger;// [-180, +180]
  int32 longitudeDecimal;// (-10_000_000, +10_000_000) sign
     used iff integer is 0.
}
```

Listing 3.17: `GlobalPosition` implementation struct

The struct comes with a library, called `GlobalPositionLibrary`. The aim of this library is straightforward: make a clear interface to work with the introduced struct. The library has two functions: `isValid` and `packCoordinates`.

The `isValid` function returns true if the coordinate is valid, false otherwise. A GlobalPosition is valid when its integer parts stay within the standard geographic ranges, latitude between -90 and +90 inclusive, longitude between -180 and +180 inclusive. Each decimal part has a magnitude strictly below $10,000,000$, since that's the value of exactly one degree in E7 precision and anything at or above it would overflow into the next integer step.

The sign of the coordinate follows a simple rule: when the integer part is non-zero it carries the sign, and the decimal contributes only its magnitude; when the integer part is zero, the decimal carries the sign instead. This second branch is what makes the band (-1°, 0°) representable on both axes, without it, every point between the equator and one degree south, or between Greenwich and one degree west, would be unreachable.

The `packCoordinates` function collapses the split (integer, decimal) representation into a single signed 32-bit integer per axis, scaled by 10^7.

The resulting E7 format has about 1.11 cm of precision.

The code is shown in Listings 3.18.

```
library GlobalPositionLibrary {
```

```
2   int32 internal constant DEC_BOUND = 10_000_000;
3
4   function isValid(GlobalPosition memory p) internal pure
       returns (bool) {
5     if (p.latitudeInteger < -90
6       || p.latitudeInteger > 90) return false;
7     if (p.latitudeDecimal <= -DEC_BOUND
8       || p.latitudeDecimal >= DEC_BOUND) return false;
9     if ((p.latitudeInteger == 90 || p.latitudeInteger == -90)
10      && p.latitudeDecimal != 0) return false;
11    if (p.longitudeInteger < -180 ||
12      p.longitudeInteger > 180) return false;
13    if (p.longitudeDecimal <= -DEC_BOUND ||
14      p.longitudeDecimal >= DEC_BOUND) return false;
15    if ((p.longitudeInteger == 180
16      || p.longitudeInteger == -180)
17      && p.longitudeDecimal != 0) return false;
18    return true;
19  }
20
21  function packCoordinates(GlobalPosition memory p) internal
       pure returns (PackedPosition memory) {
22    //valid checks ...
23    return PackedPosition({
24    latE7: int32(_combine(p.latitudeInteger,
25      p.latitudeDecimal)),
26    lonE7: int32(_combine(p.longitudeInteger,
27      p.longitudeDecimal))
28    });
29  }
30
31  function _combine(int32 integerPart, int32 decimalPart)
       private pure returns (int64) {
32    int64 base = int64(integerPart) * 10_000_000;
33    int64 dec  = int64(decimalPart);
34    int64 absDec = dec < 0 ? -dec : dec;
35    if (integerPart > 0) return base + absDec;
36    if (integerPart < 0) return base - absDec;
37    return dec;
38  }
39 }
```

Listing 3.18: `GlobalPositionLibrary` implementation

The packed position is used by the smart contract to identify the chunk and if the sent data is in the campaign area or not. The function that consumes the `PackedPosition` struct is shown in Listing 3.19 and the `_getChunkIndex` function is shown in Listing 3.20.

```
struct PackedPosition {
  int32 latE7; // 45.1234567 DEG -> 451234567
  int32 lonE7; //  9.0000000 DEG ->  90000000
}
```

Listing 3.19: `PackedPosition` struct definition

```
function _getChunkIndex(PackedPosition memory packed)
   internal view returns (uint16, bool) {
  // Check bounds
  if (packed.latE7 < _smartContractOrigin.latE7 ||
    packed.latE7 >= _smartContractEnd.latE7) return (0, false
       );
  if (packed.lonE7 < _smartContractOrigin.lonE7 ||
    packed.lonE7 >= _smartContractEnd.lonE7) return (0, false
       );

  // Calculating row and col
  int32 chunkSize = int32(uint32(_config.chunkConfig.
     chunkSize));
  uint16 row = uint16(uint32((packed.latE7 -
     _smartContractOrigin.latE7) / chunkSize));
  uint16 col = uint16(uint32((packed.lonE7 -
     _smartContractOrigin.lonE7) / chunkSize));

  uint16 index = row * _config.chunkConfig.chunk4Side + col;
  return (index, true);
}
```

Listing 3.20: `_getChunkIndex` function implementation

3.2.6 Verification workflow

Verifiers have listeners on the `DataVerificationRequestedEvent` event and they are waiting for it. As shown in Section 3.2.4, at the end of the encrypted CID publishing the smart contract emits the `DataVerificationRequestedEvent`. This event is emitted in this phase only by the `PrivacyFirstContract`, not in the extended one. Although `DataVerificationRequestedEvent` has a parameter that indicates the

dataId, this parameter is ignored, because the smart contract selects which data to verify in order to enforce a cooldown mechanism.

The `VerifierPrivacyFirst` (`VPF`) exposes the method `requestAndVerify` that allows the verifier to join the verification process of data, the same name is used to describe a function in the smart contract that validates and checks the verifier's requests. At the end of the validation performed by the smart contract, the `VerifierElected` event is emitted and the verifier can proceed with validity checks. The SDK `requestAuthToVerify` method calls the smart contract function and returns the `decryptedCid`, after which the verifier can execute checks. The smart contract `requestAuthToVerify` function is shown in Listing 3.21.

```
function requestAuthToVerify() external {
  //Checks on enrollment and bond

  for (uint256 i = 0; i < _pendingDataIds.length; i++) {
    uint256 dataIndex = _pendingDataIds[i];

    //cooldown check
    if (_verifierInformation[msg.sender].hasVerified) {
      uint256 last = _verifierInformation[msg.sender].
         lastVerified;
      if (dataIndex >= last &&
        dataIndex < last + _config.needs.verifierCooldown)
           continue;
    }

    //checks on data slots
    if (_dataEntries[dataIndex].assignedChecks
      >= _config.needs.checksRequired
      && _dataEntries[dataIndex].earliestSlotDeadline
      > block.timestamp) {continue;}

    (uint8 slotIndex, bool found) = _findAvailableSlot(
       dataIndex);
    if (!found) continue;

    //Underflow safety. Bond decreasing
    unchecked {
        _verifierInformation[msg.sender].bondBalance =
          _verifierInformation[msg.sender].bondBalance
          - _config.rewardConfig.dataVerifierBondPerData;
    }

```

```
30      uint48 deadline = uint48(block.timestamp + _config.needs.
           verificationTimeout);
31
32      _verificationSlots[dataIndex][slotIndex] =
           VerificationSlot({
33        verifier: msg.sender,
34        result: false,
35        completed: false,
36        deadline: deadline});
37
38      // Update earliestSlotDeadline if unset or if the new
           deadline is sooner
39      if (_dataEntries[dataIndex].earliestSlotDeadline == 0 ||
40        deadline <
41        _dataEntries[dataIndex].earliestSlotDeadline) {
42          _dataEntries[dataIndex].earliestSlotDeadline
43            = deadline;}
44
45      //Assigning slots
46      if (slotIndex >=
47        _dataEntries[dataIndex].assignedChecks) {
48          _dataEntries[dataIndex].assignedChecks
49          = slotIndex + 1;}
50
51      _verifierInformation[msg.sender].verifyingData =
           dataIndex;
52      _verifierInformation[msg.sender].isVerifying = true;
53
54      _emitVerifierElected(msg.sender, dataIndex, slotIndex);
55      return;
56    }
57    revert("No data available for verification");
58  }
```

Listing 3.21: `requestAuthToVerify` function implementation

The function performs multiple checks to guarantee a fast data retrieval with the minimum amount of gas usage as discussed in Section 3.5, the `_findAvailableSlot` function checks if a data has a free slot, if not, the function analyzes the next pending entry. In `_emitVerifierElected`, a `VerifierElected` event is emitted; the verifiers are waiting for that event, when triggered by the event, they can invoke the `verifyCallback` that is given in the constructor, as explained in Section 3.2.3. In addition, the verifiers can use `submitVerifyResult` function of the smart con-

tract to deliver the verification result. A summary of verify workflow is shown in `requestAndVerify` method of `VerifyPrivacyFirst` class, whose implementation in shown in Listing 3.22.

```
async requestAndVerify(): Promise<{ dataId: number;
    decryptedCid: string;
    result: VerificationResult }> {
    try{
    const {dataId, decryptedCid}
      = await this.requestAuthToVerify();
    const result = this.verifyCallback(decryptedCid, dataId);
    const tx = await this.submitResult(result);
    await tx.wait();
    return {dataId, decryptedCid, result};
  } catch (ignored){}
  return {dataId: -1, decryptedCid: "", result:
     VerificationResult.CANNOT_COMPLETE};
}
```

Listing 3.22: `VPF`'s `requestAndVerify` method implementation

When the majority of required verifiers votes with the same significant result (data is valid or invalid), the validated data is removed from the pending queue and the smart contract rewards the correct verifiers and the sender, if the data was valid. The whole verifier workflow is implemented looking at Section 2.3.5; the final decisions of the smart contract implement the data lifecycle shown in Figure 2.7.

3.3 Economic model

The economic model described in Section 2.4 executes its logic on-chain through the smart contract and the use of the DST ERC-20 token. The same token is used to manage bond deposits. The following subsections describe how the reward logic and bond system are enforced.

3.3.1 DST token integration

The adoption of a custom token instead of using ETH is justified because of a certain numbers of wanted aspects, like reentrancy protection, because ERC-20 has no callback and hooks to exploit to perform the attack [6]. Table 3.2 shows a comparison between use ETH native and a custom ERC-20 token, like DST. In addition, the absence of payable functions blocks the receiving of ETH by error, because solidity on compile time adds a check to avoid this type of errors [45].

Aspect	ETH native	ERC-20 (DST)
Deposit mechanism	`msg.value` in `payable` function	`approve` + `transferFrom`
Reentrancy risk	High: `.call` triggers recipient's `receive`/`fallback`	Lower: not overridden `transfer` does not trigger external code
Multiple payouts	Every `.call` can fail independently	`transfer` reverts
`payable` functions	Necessary	Not necessary
Extra step	-	`approve` is required before deposit
Gas per transfer	21,000 gas [34]	$\sim$ 65,000 gas (measured in Table 4.4)

Table 3.2: Comparison between ETH native and ERC-20 (DST)

The DST token is implemented thought the `DataSentTokenContract`, which implementation is shown in Listing 3.23.

```
contract DataSentTokenContract is ERC20, Ownable {
    constructor(uint256 initialSupply) ERC20("DataSentToken",
        "DST") Ownable(msg.sender) {
        _mint(msg.sender, initialSupply);
    }

    function mint(address to, uint256 amount) public
       onlyOwner {
        _mint(to, amount);
    }

    function burn(uint256 amount) public {
        _burn(msg.sender, amount);
    }
}
```

Listing 3.23: `DataSentTokenContract` (DST) token implementation

3.3.2 Reward logic

Rewards incentivize senders to contribute data to the campaign. For verifiers, the reward is the incentive to act honestly. The reward is granted to senders that submit

valid data, and is also paid to verifiers that vote in agreement with the majority on whether a given datum is valid.

In `PrivacyFirstContract` the reward for the sender is constant and equal to `dataSenderPayout`, described in Table A.3. For `PrivacySecondContract` the behavior is described in Section 3.4.4.

For both protocols, the reward for verifiers is equal to `dataVerifierPayout`.

3.3.3 Bond logic

The bond, as described in Section 2.4.3, is useful to discourage dishonest behaviors, such as a sender who submits invalid data or a verifier that votes incorrectly. The bond implementation lives in `PrivacyFirstContract`, and it is kept in the second version of the contract.

In the SDK classes, the `senderPayBond` and `verifierPayBond` smart contract functions are called in the `payBond` methods, which are called in `setup` methods, for both actors. For senders and verifiers, the bond deducted per action is `dataSenderBondPerData` for senders and `dataVerifierBondPerData` for verifiers, both described in Table A.3. The total amount of bond to supply is calculated in the constructor as shown in Listing 3.24.

```
//...
_requiredSenderBond =
  _config.needs.maxDataPerSender *
  _config.rewardConfig.dataSenderBondPerData;
_requiredVerifierBond =
  _config.needs.maxDataPerVerifierPerBond *
  _config.rewardConfig.dataVerifierBondPerData;
//...
```

Listing 3.24: DST token to supply as bond for senders and verifiers

Senders cannot pay bond twice because the number of DST token to supply to the contract is enough only for the `maxDataPerSender` parameter, described in Table A.1. Listing 3.25 shows the code of the `senderPayBond` function, which executes checks whether the sender already paid a bond and registers the payment.

```
function senderPayBond() external {
  require(_senders[msg.sender], "Not␣a␣sender");
  require(!_senderInformation[msg.sender].bondPaid,
    "Bond␣already␣paid");
  _dstContract.transferFrom(msg.sender, address(this),
     _requiredSenderBond);
```

```
  _senderSupplyBond = _senderSupplyBond + _requiredSenderBond
  _senderInformation[msg.sender].bondBalance =
     _requiredSenderBond;
  _senderInformation[msg.sender].bondPaid = true;
}
```

Listing 3.25: Smart contract's `senderPayBond` function

Verifiers can refill their bond for this reason, there are no checks in the `verifierPayBond` function, whose code is shown in Listing 3.26.

```
function verifierPayBond() external {
  require(_verifiers[msg.sender], "Not a verifier");
  _dstContract.transferFrom(msg.sender, address(this),
     _requiredVerifierBond);
  _verifierSupplyBond = _verifierSupplyBond +
     _requiredVerifierBond;
  _verifierInformation[msg.sender].bondBalance =
    _verifierInformation[msg.sender].bondBalance +
    _requiredVerifierBond;
}
```

Listing 3.26: Smart contract's `verifierPayBond` function

The bond balance is deducted when Sender and Verifier perform an action that could lead to misbehavior. Senders do this when they publish the CID of the data they send, and verifiers do this when they request verification of a piece of data and it is found. Both pieces of code that handle this logic are shown in Listing 3.27.

```
//Senders
function publishIpfsCid(
    bytes calldata encryptedCid,
    GlobalPosition calldata position
) external virtual {
  // ...
  unchecked {
    _senderInformation[msg.sender].bondBalance =
      _senderInformation[msg.sender].bondBalance -
      _config.rewardConfig.dataSenderBondPerData;
  }
  // ...
}

//Verifiers
function requestAuthToVerify() external {
```

```
17    // ...
18    unchecked {
19    _verifierInformation[msg.sender].bondBalance =
20      _verifierInformation[msg.sender].bondBalance -
21      _config.rewardConfig.dataVerifierBondPerData;
22    }
23    // ...
24  }
```

Listing 3.27: Bond deducted from sender and verifier balance

If the actor acts honestly, the reward and the bond are transferred together to avoid multiple transfers. Otherwise, the bond is lost and no reward is granted. The total payout calculation is shown in Listing 3.28.

```
1  // For senders
2  uint256 total = _config.rewardConfig.dataSenderPayout +
       _config.rewardConfig.dataSenderBondPerData;
3
4  // For verifiers
5  uint256 total = _config.rewardConfig.dataVerifierPayout +
       _config.rewardConfig.dataVerifierBondPerData;
6
7  // For both
8  _dstContract.transfer(slot.verifier, total);
```

Listing 3.28: Total payout calculation

3.4 Extended protocol

In this section, all the different behaviors of `PrivacySecondContract` are described. All the changes have been made according to Section 2.6.

Sections 3.4.1 and 3.4.2 explain how the principle of least privilege [46] is enforced, as discussed in Section 2.6.1.

3.4.1 Extended verifier enrollment

In `PFC`, as described in Section 3.2.3, all verifiers receive K_{dec} on enrollment and this introduces the vulnerability exploited by the attack described in Section 2.5.2 exploits. Addressing this vulnerability is straightforward: create a key pair per data (KD_{enc}, KD_{dec}) and deliver KD_{dec} only when the verifier needs it to decrypt and

verify the assigned data. For this reason, in the PSC enrollment phase no key is delivered.

Listing 3.29 shows how the event handler for verifier enrollment phase is overridden and how an empty string is passed as placeholder for the key to preserve ABI compatibility with PFC, Listing also shows how the public keys are stored in a map to encrypt KD_{dec} when the verifier is elected in the verification phase. In addition, Listing 3.30 shows that at the end of the function the VerifierEnrolled event is emitted, instead of VerifierDecryptionKeyReady, which is unused in PSC.

```
override async handleVerifierEnrollment(
    verifierAddress: string,
    encryptedPubKeyJson: string,
): Promise<void> {
    const bundle = JSON.parse(encryptedPubKeyJson);
    const verifierPubKey = this.cipher.decryptStringHybrid(
       this.keys.privateKey, bundle);
    this.verifierPublicKeys.set(verifierAddress,
       verifierPubKey); //Store verifiers keys

    await this.contract
        .connect(this.signer)
        .publishVerifierDecryptionKey(verifierAddress, "");
}
```

Listing 3.29: B's handleVerifierEnrollment method in OPS

```
function publishVerifierDecryptionKey(address verifier,
   string calldata) public override {
    require(_campaignOpen, "Campaign is closed");
    require(msg.sender == _owner, "Not the owner");
    _verifiers[verifier] = true;
    _verifierInformation[verifier] = VerifierInformation({
        lastVerified: 0,
        hasVerified: false,
        verifyingData: 0,
        isVerifying: false,
        bondBalance: 0
    });
    delete _verifierEnrolledRequest[verifier];
    emit VerifierEnrolled(verifier);
}
```

Listing 3.30: PSC publishVerifierDecryptionKey overridden function

3.4.2 Extended verification phase

The verification phase in PSC is the phase that delivers KD_{dec} to them. Before the Verifier election and insertion in verify queue, in this protocol, the campaign initiator has to create a key pair per data (KD_{enc}, KD_{dec}). Since senders S have received K_{enc} in enrollment phase (Section 3.2.2), a trusted third party is required to generate and manage the key pair; this role is fulfilled by the campaign initiator.

In PSC, the publishIpfsCid function is overridden, the change that is relevant to this phase is the emission of DataRequiredVerificationKeyGeneration event, which is listened by B's SDK. The event handler is shown in Listing 3.31, the called smart contract function is shown in Listing 3.32. The called function emits the DataVerificationRequestedEvent which is the same event emitted by PFC.

After the verification authorization request, which follows the same flow as in PFC, _emitVerifierElected is invoked, that function is overridden in PSC, the code of both functions is shown in Listing 3.33.

```
async handleDataRequiresKey(dataId: number, encryptedCid:
   string): Promise<void> {
    this.encryptedCidCache.set(dataId, encryptedCid);

    await this.contract
        .connect(this.signer)
        .notifyVerificationKeyReady(dataId);
}
```

Listing 3.31: B's handleDataRequiresKey method in OPS

```
function notifyVerificationKeyReady(uint256 dataId) external{
  require(msg.sender == _owner, "Not the owner");
  require(_campaignOpen, "Campaign is closed");
  require(_awaitingKeyGeneration[dataId],
    "Data not awaiting key generation");
  delete _awaitingKeyGeneration[dataId];
  _addToPending(dataId);
  emit DataVerificationRequestedEvent(dataId);
}
```

Listing 3.32: PSC notifyVerificationKeyReady function

```
//PrivacyFirstContract
function _emitVerifierElected(address verifier, uint256
   dataId, uint8 slotIndex) internal virtual {
  emit VerifierElected(verifier, dataId, slotIndex,
     _dataEntries[dataId].encryptedCid);
```

```
}

//PrivacySecondContract
function _emitVerifierElected(address verifier, uint256
    dataId, uint8 slotIndex) internal override {
  delete _bundleDelivered[dataId][slotIndex];
  emit VerifierElectedForData(verifier, dataId);
}
```

Listing 3.33: PFC and PSC `_emitVerifierElected` implementations"

For PSC, the function shown in Listing 3.33 emits the `VerifierElectedForData` event. Listing 3.34 shows how the campaign initiator handles the event.

The smart contract's function `deliverVerificationBundle` performs check and then emits `VerifierVerificationBundleReady` event, which is listened by verifiers and contains the verification bundle. Once the verifier receives the bundle, they can submit the verification result as in PFC.

```
async handleVerifierElected(verifier: string,
dataId: number): Promise<void> {
  const pkV = this.verifierPublicKeys.get(verifier);
  if (!pkV) throw new Error(`...`);

  const encryptedCid = this.encryptedCidCache.get(dataId);
  if (!encryptedCid) throw new Error(`...`);

  const bundleContent = JSON.stringify({encryptedCid,
    kdec: this.encryptionKeys.decryptionKey});

  const bundleEnvelope = this.cipher.encryptStringHybrid(pkV,
      bundleContent);
  const bundleBytes = toUtf8Bytes(JSON.stringify(
     bundleEnvelope));

  await this.contract
    .connect(this.signer)
    .deliverVerificationBundle(dataId, verifier,bundleBytes);
```

Listing 3.34: B's `handleVerifierElected` method in OPS

3.4.3 Verifier additional penalties

The implementation of additional penalties for verifiers comes from the weakness of bond system if a verifier votes wrong, as described in Section 2.6.3. For this reason two new functions have been included: `_recordCorrectVote` and `_recordWrongVote`, both in Listing 3.35.

The functions are called according to the verification process result, the function `_recordCorrectVote` counts how many correct votes have been cast. The counting process starts if the verifier made a mistake in a previous voting process, in this case the smart contract counts how many good votes are registered towards `pardonWrongVoteAfterRightVotes` described in Table A.5. On `pardonWrongVoteAfterRightVotes` votes, a wrong vote is pardoned. If the parameter is 0 then no pardon system is implemented.

The `_recordWrongVote` function registers a wrong vote, once the wrong vote count reaches `kickVerifierAfterWrongVotes`(described in Table A.5), the verifier is automatically kicked. If `kickVerifierAfterWrongVotes` is 0, then no auto-kick system is enabled and this feature is disabled.

The `_kickVerifier` function handles the kick process, transferring the remaining bond to the kicked verifier.

```
1  function _recordCorrectVote(address verifier) internal {
2      uint256 pardon = _novelParams.
           pardonWrongVoteAfterRightVotes;
3      if (pardon == 0 || _wrongVotes[verifier] == 0) return;
4
5      _rightVotesAccumulator[verifier]++;
6      if (_rightVotesAccumulator[verifier] >= pardon) {
7          _wrongVotes[verifier]--;
8          _rightVotesAccumulator[verifier] = 0;
9      }
10 }
11
12 function _recordWrongVote(address verifier) internal {
13     _wrongVotes[verifier]++;
14     uint256 kick = _novelParams.kickVerifierAfterWrongVotes;
15     if (kick > 0 && _wrongVotes[verifier] >= kick) {
16         _kickVerifier(verifier);
17         emit VerifierKicked(verifier);
18     }
19 }
20
21 function _kickVerifier(address verifier) internal {
```

```
22      uint256 bond = _verifierInformation[verifier].bondBalance
            ;
23      if (bond > 0) {
24          unchecked { _verifierSupplyBond -= bond; }
25          _verifierInformation[verifier].bondBalance = 0;
26          _dstContract.transfer(verifier, bond);
27      }
28      if (_verifierInformation[verifier].isVerifying) {
29          uint256 dataId = _verifierInformation[verifier].
                verifyingData;
30          (uint8 slotIndex, bool found) = _getVerifierSlotIndex
                (dataId, verifier);
31          if (found) _removeVerifierFromData(dataId, slotIndex)
                ;
32          _makeVerifierAbleToOperate(verifier);
33      }
34      _verifiers[verifier] = false;
35  }
```

Listing 3.35: PSC `_recordCorrectVote`, `_recordWrongVote` and `_kickVerifier` functions

The campaign initiator can also provide the manual kicking of a certain verifier calling the `kickVerifier` function shown in Listing 3.36.

```
1  function kickVerifier(address verifier) external {
2      require(msg.sender == _owner, "Not␣the␣owner");
3      require(_verifiers[verifier], "Not␣a␣verifier");
4      _kickVerifier(verifier);
5      emit VerifierKicked(verifier);
6  }
```

Listing 3.36: PSC public `kickVerifier` function

3.4.4 Introducing geographical incentives

Geographical incentives, described in Section 2.5.3, are introduced to cover areas with a lower presence of people in the campaign.

For this reason, when a sender `S` sends a datum, they can expect a lower payout if the datum is in a densely covered area. The decay function, based on how many data are in a chunk (expressed by factor *alpha*, evaluated using Equation 2.3) is shown in Figure 2.13.

The smart contract uses the babylonian square root formula, as shown in Listing 3.37.

```
function _computeDecayedReward(uint256 countBeforeSubmission)
    internal view returns (uint256) {
  uint256 maxPerChunk = _config.needs.maximumDataPerChunk;

  if (!_novelParams.rewardDecayEnabled || maxPerChunk == 0)
    return _config.rewardConfig.dataSenderPayout;

  uint256 remaining = maxPerChunk - countBeforeSubmission;
  uint256 ratio = remaining * SCALE / maxPerChunk;

  uint256 k = _novelParams.rewardDecayFactor;
  for (uint256 i = 0; i < k;) {
    ratio = _sqrt(ratio * SCALE);
    unchecked { i++; }
  }

  return _config.rewardConfig.dataSenderPayout * ratio /
     SCALE;
}
```

Listing 3.37: `PSC _computeDecayedReward` function

3.5 Gas optimization

Gas is the unit measuring computational effort on the Ethereum blockchain; the aim is to minimize its consumption.

For this reason, as shown before, the bond and reward payout are combined in a single transfer, but another technique is used to avoid high gas usage: the pending data to verify queue is based on swap-and-pop mechanism.

The data structure is based on swap-and-pop using index mapping, the result is an unordered set, but it is gas-efficient and useful for this use case, because the verification order is not critical. Listing 3.38 shows how the data structure is implemented, how data are inserted into and removed from it. The usage is shown in Listing 3.21, all the operations have $O(1)$ computational cost, an example of `_addToPending` function usage is shown in Listing 3.15.

```
//Auxiliary data structures
uint256[] internal _pendingDataIds;
mapping(uint256 => uint256) internal _pendingIndex;

```

```
//Insert into data structure
function _addToPending(uint256 dataId) internal {
    _pendingIndex[dataId] = _pendingDataIds.length;
    _pendingDataIds.push(dataId);
}

//Remove from data structure
function _removeFromPending(uint256 dataId) internal {
    uint256 idx = _pendingIndex[dataId];
    uint256 lastId = _pendingDataIds[_pendingDataIds.length -
        1];
    _pendingDataIds[idx] = lastId;
    _pendingIndex[lastId] = idx;
    _pendingDataIds.pop();
    delete _pendingIndex[dataId];
}
```

Listing 3.38: Gas-optimized pendingData unordered set implementation

Chapter 4

Evaluation

The protocols designed in Chapter 2 and implemented in Chapter 3 are evaluated along two complementary axes. Section 4.1 analyses the gas consumption of the three contract variants, across a 4×4 scenario grid of sender and verifier counts, quantifying the cost of the swap-and-pop optimisation and the overhead introduced by the novel protocol extensions. Section 4.2 examines the behaviour of the network under concurrent campaign load, showing that the per-block gas limit introduces a critical section that forces partial serialisation when multiple campaigns execute simultaneously. All analyses are expressed in absolute units of measure, such as gas units for computational cost and DST tokens for economic rewards. Finally, Section 4.3 evaluates the behaviour of the protocol on a dataset of taxi traces from the city of Porto.

4.1 Gas cost analysis

There are three contracts under analysis: `PFC`, `PSC`, and `FIFO`. `FIFO` shares the same base implementation as `PFC` but replaces the swap-and-pop pending list with a linear-shift FIFO approach.

The methodology of these experiments is described in Section 4.1.1. Section 4.1.3 presents a comparison between the swap-and-pop data structure and FIFO queue. Section 4.1.4 contains a comparison to show the overhead introduced by the novel features, i.e., the cost of extending `PFC` to `PSC`.

Finally, Section 4.1.5 estimates the cost of organizing a crowdsensing campaign in USD.

	$V = 2$	$V = 10$	$V = 100$	$V = 1000$
$S = 1$	S: 1 V: 2	S: 1 V: 10	S: 1 V: 100	S: 1 V: 1000
$S = 10$	S: 10 V: 2	S: 10 V: 10	S: 10 V: 100	S: 10 V: 1000
$S = 100$	S: 100 V: 2	S: 100 V: 10	S: 100 V: 100	S: 100 V: 1000
$S = 1000$	S: 1000 V: 2	S: 1000 V: 10	S: 1000 V: 100	S: 1000 V: 1000

Table 4.1: Simulation scenarios: 16 combinations of sender and verifier cardinalities.

4.1.1 Methodology

The test environment is based on Hardhat, the experiments are executed on a simulated Ethereum environment on the `cancun` hardfork was selected to avoid the per-transaction gas cap of 2^{24} gas introduced by EIP-7825 [47] in the Fusaka hard fork. In addition, the gas limit per block is modified to 10^9 to avoid interruption on worst-case.

A grid of scenarios, a grid made with a dimension of 4×4, the scenarios are a combination between

$$\forall S \in \{1, 10, 100, 1000\}$$
$$\forall V \in \{2, 10, 100, 1000\}$$

A grid of scenarios in Table 4.1

Each scenario submits 10,000 CIDs. All participants are assumed to behave honestly, following the happy path of the protocol.

The results report gas units. First call operations are highlighted because first calls initialize storage slots, incurring higher costs due to cold SSTORE operations.

4.1.2 Baseline cost breakdown of PFC

Figure 4.1 shows a comparison between the first call and subsequent ones.

The `data_submission` function requires 194k gas units for the first call and an average of 165k for subsequent ones. It is the most expensive function in the smart contract; this cost is justified by SSTOREs, event emission, and input checks.

The owner function that executes K_{dec} delivery, `owner_send_kdec`, averages 133k gas units, with no significant difference from the first call.

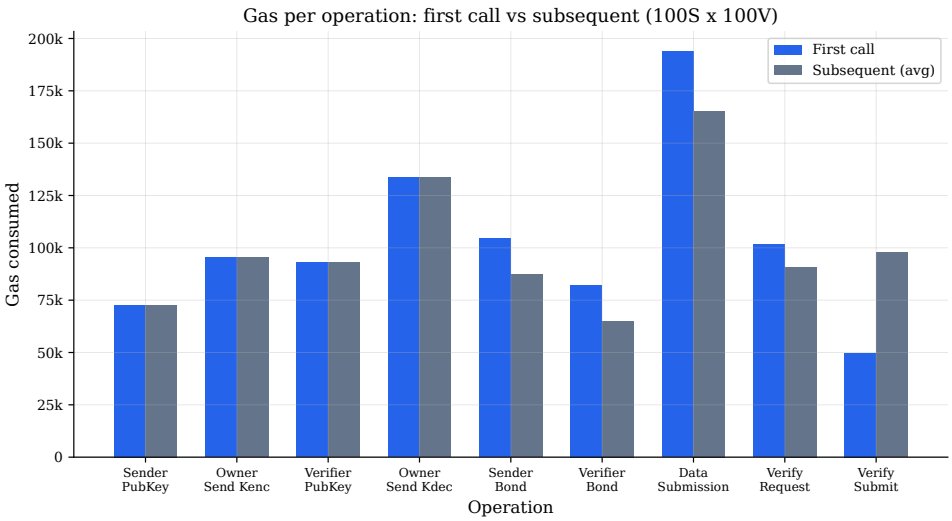

Figure 4.1: Per-operation gas cost of `PrivacyFirstContract` scenario $100\,\mathrm{S} \times 100\,\mathrm{V}$. Operations that do not allocate new storage slots show no premium.

The `verification_request` function has a cost dominated by iteration over the pending data structure.

The `verification_submit` function exhibits behaviour that will be explained in Section 4.1.3; all other operations include an SSTORE and a DST transfer that increase gas usage.

Per-operation gas costs of `PrivacyFirstContract`, summarised across the 16 simulated scenarios, are shown in Table 4.2. The *1st call* column reports the gas consumed by the first invocation of each operation. The *Avg subsequent* column reports the mean across scenarios of the per-scenario average; the *Range* column gives the [min, max] interval of that same metric. The narrow range confirms that per-call cost is a property of the contract, not of the workload.

4.1.3 PFC and FIFO pending list management

`PrivacyFirstContract` applies a swap-and-pop approach to the `_removeFromPending` function. Deletion runs in $O(1)$, but order is not preserved.

In this context, it is necessary to introduce a new smart contract implementing a FIFO queue for pending data awaiting verification, as shown in Listing C.1. This smart contract is a variant that inherits all functions from `PFC` and overrides `_addToPending` and `_removeFromPending` to implement a FIFO strategy.

For the `verification_submit` function, Figure 4.2 plots the gas consumption per

Operation	1st call	Avg subsequent	Range
`sender_upload_pubkey`	72,671	72,671	[72,671, 72,671]
`owner_send_kenc`	95,327	95,325	[95,324, 95,326]
`verifier_upload_pubkey`	92,871	92,871	[92,871, 92,871]
`owner_send_kdec`	133,543	133,542	[133,501, 134,000]
`sender_pay_bond`	104,354	87,254	[87,254, 87,254]
`verifier_pay_bond`	82,192	65,092	[65,092, 65,092]
`data_submission`	193,677	165,620	[165,280, 166,509]
`verification_request`	101,557	91,086	[90,711, 92,067]
`verification_submit`	49,694	98,634	[97,336, 102,023]

Table 4.2: Per-operation gas cost of PFC, averaged across the 16 simulated scenarios.

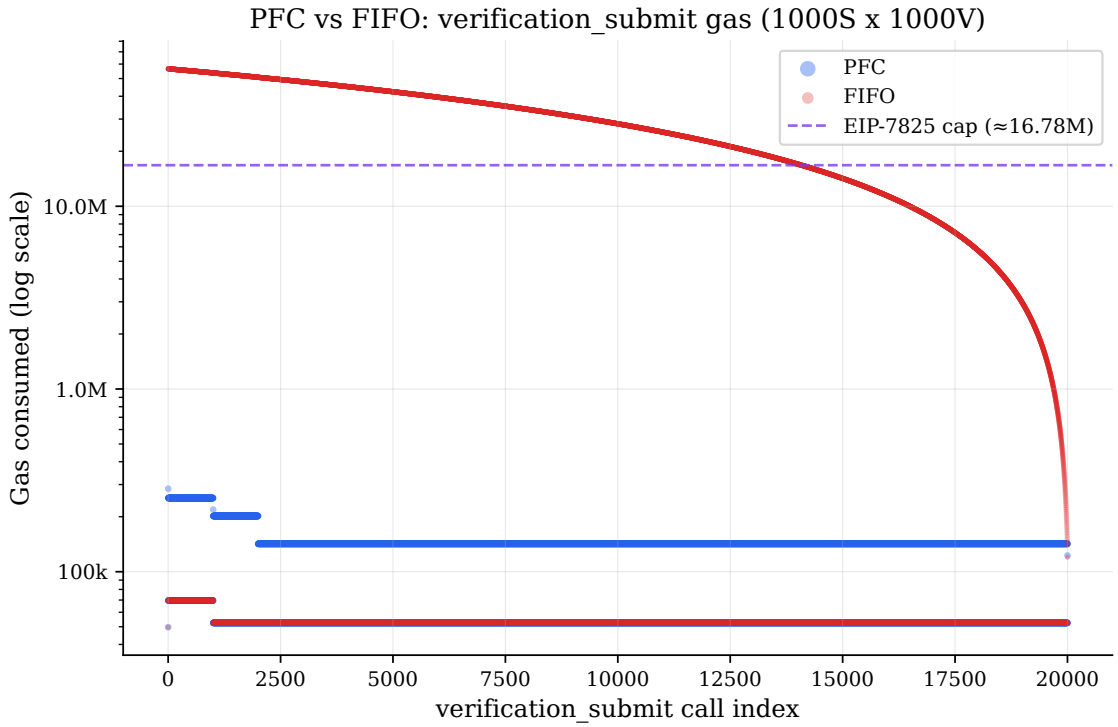

Figure 4.2: Per-call gas consumption of `verification_submit` in the 1000 S × 1000 V scenario. The dashed line marks the EIP-7825 per-transaction gas cap of 2^{24} gas, active from the Fusaka hard fork.

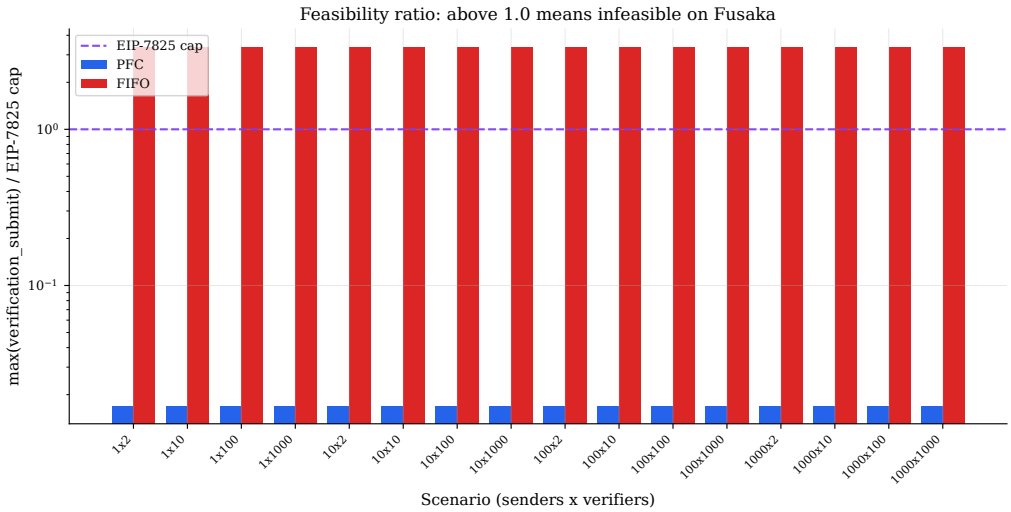

Figure 4.3: FIFO baseline infeasibility under EIP-7825 across all scenarios.

call for the `PFC` and `FIFO` contracts. The comparison is clear: `PFC` gas usage falls in the range $[120, 300]$ k gas units and becomes constant after 2 500 submissions.

For the `FIFO` contract the situation is different: there are two types of submissions: those that do not complete a verification process, and those that trigger one. If a verification submission completes a verification process, the verified data is removed from the pending data structure; `_removeFromPending` is therefore called, and in the `FIFO` contract this causes a left-shift of every subsequent entry.

Additionally, the two contracts differ in gas usage for the following functions:

- `verification_request`: the `FIFO` contract must iterate more times to satisfy the cooldown constraint;
- `data_submission`: `FIFO` is cheaper than `PFC` by 22 k gas units.

Despite the `data_submission` savings, PFC's advantage of 14 M gas units per `verification_submit` call far outweighs the $22\,\text{k} \times 10000 = 220\,\text{M}$ gas units saved by FIFO across all data submissions.

Figure 4.3 shows the ratio of worst-case `verification_submit` gas to the EIP-7825 cap, for each of the 16 scenarios.

Values above 1.0 indicate scenarios where the contract is infeasible on a Fusaka-era chain. `PrivacyFirstContract` stays consistently below 0.02, two orders of magnitude under the cap. `PrivacyFirstContractFifo` exceeds the cap in all scenarios by a factor of $\approx 3.4\times$.

Scenario	PFC max (gas)	FIFO max (gas)	Ratio FIFO/PFC	FIFO/EIP-7825
1 S × 2 V	284,522	56,541,167	198.7×	3.4×
1 S × 10 V	284,522	56,541,167	198.7×	3.4×
1 S × 100 V	284,522	56,541,167	198.7×	3.4×
1 S × 1000 V	284,522	56,541,167	198.7×	3.4×
10 S × 2 V	284,522	56,541,167	198.7×	3.4×
...	...	...	...	...
100 S × 1000 V	284,522	56,541,167	198.7×	3.4×
1000 S × 2 V	284,522	56,541,167	198.7×	3.4×
1000 S × 10 V	284,522	56,541,167	198.7×	3.4×
1000 S × 100 V	284,522	56,541,167	198.7×	3.4×
1000 S × 1000 V	284,522	56,541,167	198.7×	3.4×

Table 4.3: Worst-case `verification_submit` gas: PFC vs FIFO.

The ratio is independent of $(senders, verifiers)$ because the worst-case shift cost is driven by `TOTAL_CIDS`, not by the actor count.

A full comparison of gas usage between the two protocols is shown in Table 4.3. For all scenarios, the maximum gas for both protocols is participant-independent, as evidenced by identical values across all rows. For the `FIFO` contract the ratio with the EIP-7825 limit exceeds one, so with $10,000$ CIDs `FIFO` is not feasible.

4.1.4 PFC and PSC feature overhead

`PrivacySecondContract` inherits `PrivacyFirstContract` functions and implements the features described in Section 2.6.

Each feature has a measurable gas cost; this section therefore evaluates the overhead of each implemented feature and identifies which contributes most.

Figure 4.4 shows the variation in gas usage for functions common to both contracts.

- **Deployment**: bytecode size increases by 15.8% due to the additional bytecode;
- `data_submission`: gas usage increases by 20.7% due to the computation of the decayed reward for each submission;
- `verification_submit`: increases by 9.1% due to verifier vote tracking (correct and incorrect) and a newly introduced `require` check;
- `owner_send_kdec`: gas usage decreases by 55.5%, as the function now only emits an event, registers a mapping for the accepted verifier, and stores the

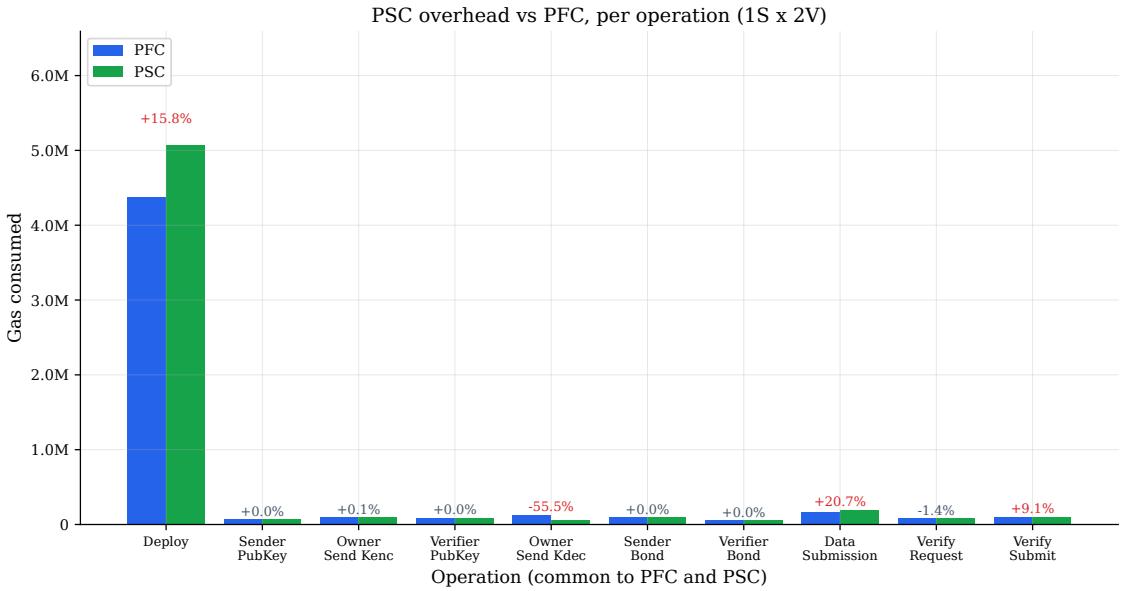

Figure 4.4: Operations common to PFC and PSC. Percentage deltas annotated on each pair.

Verifier struct.

`PrivacySecondContract` does not alter the behaviour of existing functions, but introduces new ones with their own costs, as shown in Figure 4.5. The three new functions are:

- `owner_notify_key_ready`: consumes 76 k gas units after the creation of the per-data key pair.
- `owner_deliver_bundle`: uses 63 k gas units to deliver the verification bundle: containing the per-data decryption key to the verifier.
- `owner_kick_verifier`: penalty path, invoked by the owner against malicious verifiers; costs 30 k gas units.

Table 4.4 reports the per-operation gas cost of `PrivacySecondContract` compared to `PrivacyFirstContract` in the 1000 S $\times$ 1000 V scenario. The upper section lists operations common to both contracts; the lower section lists operations introduced by the extended protocol of Section 2.6 that are absent from `PrivacyFirstContract`.

4.1.5 Real-world deployment costs across networks

Gas costs exist to prevent replay attacks and to incentivize verifiers to stake their funds and validate new blocks [34]. The formula to evaluate the USD cost of a

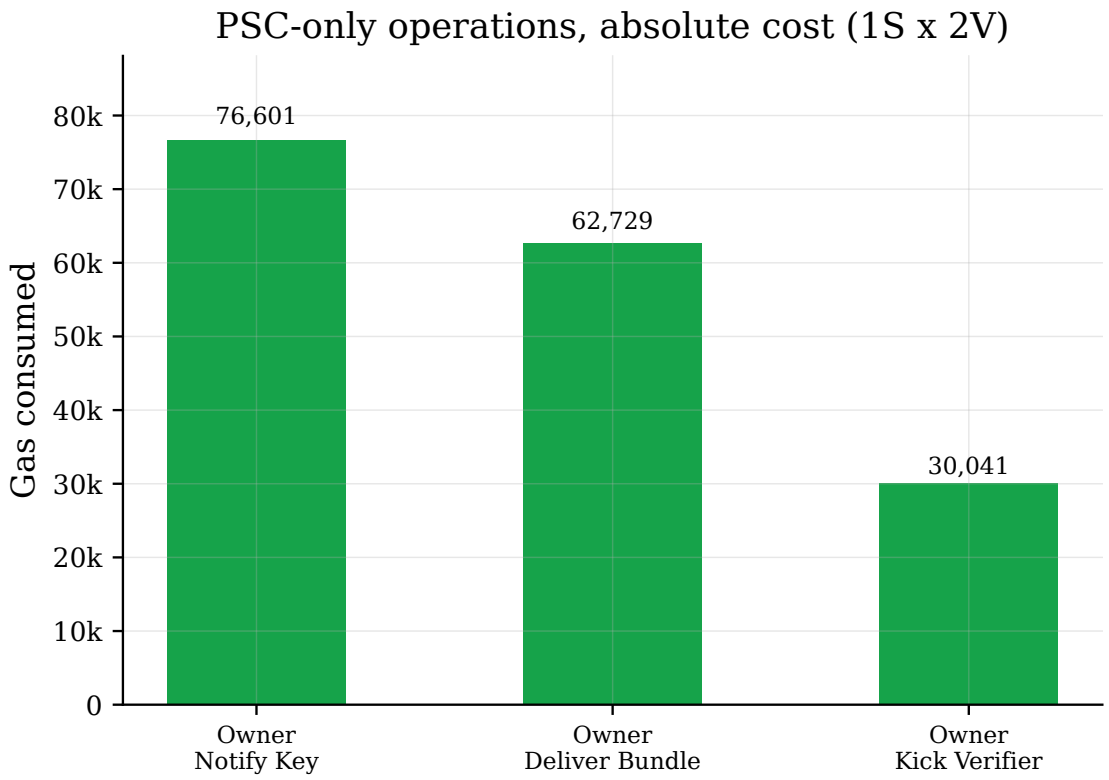

Figure 4.5: Operations introduced by PSC that have no PFC counterpart.

Operation	PFC (gas)	PSC (gas)	Δ (gas)	Δ (%)
Operations common to PFC and PSC				
`contract_deployment`	4,379,691	5,070,783	+691,092	+15.8%
`sender_upload_pubkey`	72,671	72,693	+22	+0.0%
`owner_send_kenc`	95,326	95,383	+57	+0.1%
`verifier_upload_pubkey`	92,871	92,893	+22	+0.0%
`owner_send_kdec`	133,512	59,378	-74,134	-55.5%
`sender_pay_bond`	87,254	87,276	+22	+0.0%
`verifier_pay_bond`	65,092	65,092	+0	+0.0%
`data_submission`	166,509	200,735	+34,226	+20.6%
`verification_request`	92,067	90,781	-1,286	-1.4%
`verification_submit`	102,023	110,920	+8,897	+8.7%
Operations exclusive to PSC				
`owner_notify_key_ready`	-	76,601	-	-
`owner_deliver_bundle`	-	62,734	-	-
`owner_kick_verifier`	-	30,053	-	-

Table 4.4: Per-operation PSC overhead over PFC at $1000\,\mathrm{S} \times 1000\,\mathrm{V}$.

certain gas usage is the following:

$$\mathrm{GAS}_{\mathrm{COST}} = \mathrm{GAS}_{\mathrm{USED}} \times \mathrm{GAS}_{\mathrm{PRICE}} \times \mathrm{TOKEN}_{\mathrm{COST}} \tag{4.1}$$

Equation 4.1 explains how the $\mathrm{GAS}_{\mathrm{COST}}$ is calculated as a function of the gas used, the gas price, and the native network token cost.

Tables 4.6, 4.7 and 4.8 show a comparison between the roles in the $1000\,\mathrm{S} \times 1000\,\mathrm{V}$ scenario, respectively for `FIFO`, `PFC` and `PSC` contracts. For verifiers and senders, the gas is calculated per actor; the mean, median, minimum, and maximum gas usage are reported. In addition, for each network all gas usage data are reported, while the corresponding gas cost reflects the selected network

Table 4.5 shows the gas cost and the token cost for each network [48; 49; 50; 51; 52; 53]. The data used are the median values for each category recorded in 2025. The median was chosen because it is robust to outliers.

The most expensive role is the owner, not only because they have to supply the `DST` tokens to distribute as rewards, but also because they hold the role with the highest gas usage for `PFC` and `PSC` contracts. For `FIFO`, the most expensive role is the verifier, due to the higher gas usage caused by the handling of the pending IDs data structure. The sender, in every scenario, has the lowest consumption, because in every protocol the submission is straightforward to register and requires only a

Token	GAS_{PRICE}	$TOKEN_{USD}$ [54]
Arbitrum (ARB)	0.0169	0.381549
Binance Coin (BNB)	1.0889	696.734456
Ethereum (ETH)	2.0980	2977.162263
Gnosis (GNO)	3.3204	131.286112
Optimism (OP)	0.0153	0.706806
Polygon (POL)	76.8760	0.225441

Table 4.5: Token prices and gas price used in the evaluation

Token	Role	Gas mean	Gas median	Gas min	Gas max	USD mean	USD median	USD min	USD max
ARB	owner	233,345,642	233,345,642	233,345,642	233,345,642	$0.001505	$0.001505	$0.001505	$0.001505
	sender	1,603,363	1,601,623	1,601,623	1,650,099	$0.000010	$0.000010	$0.000010	$0.000011
	verifier	285,325,886	271,285,028	3,083,235	595,692,881	$0.001840	$0.001749	$0.000020	$0.003841
BNB	owner	233,345,642	233,345,642	233,345,642	233,345,642	$177.03	$177.03	$177.03	$177.03
	sender	1,603,363	1,601,623	1,601,623	1,650,099	$1.22	$1.22	$1.22	$1.25
	verifier	285,325,886	271,285,028	3,083,235	595,692,881	$216.47	$205.82	$2.34	$451.94
ETH	owner	233,345,642	233,345,642	233,345,642	233,345,642	$1,457.50	$1,457.50	$1,457.50	$1,457.50
	sender	1,603,363	1,601,623	1,601,623	1,650,099	$10.01	$10.00	$10.00	$10.31
	verifier	285,325,886	271,285,028	3,083,235	595,692,881	$1,782.17	$1,694.47	$19.26	$3,720.75
GNO	owner	233,345,642	233,345,642	233,345,642	233,345,642	$101.72	$101.72	$101.72	$101.72
	sender	1,603,363	1,601,623	1,601,623	1,650,099	$0.6989	$0.6982	$0.6982	$0.7193
	verifier	285,325,886	271,285,028	3,083,235	595,692,881	$124.38	$118.26	$1.34	$259.68
OP	owner	233,345,642	233,345,642	233,345,642	233,345,642	$0.002523	$0.002523	$0.002523	$0.002523
	sender	1,603,363	1,601,623	1,601,623	1,650,099	$0.000017	$0.000017	$0.000017	$0.000018
	verifier	285,325,886	271,285,028	3,083,235	595,692,881	$0.003086	$0.002934	$0.000033	$0.006442
POLY	owner	233,345,642	233,345,642	233,345,642	233,345,642	$4.04	$4.04	$4.04	$4.04
	sender	1,603,363	1,601,623	1,601,623	1,650,099	$0.0278	$0.0278	$0.0278	$0.0286
	verifier	285,325,886	271,285,028	3,083,235	595,692,881	$4.94	$4.70	$0.0534	$10.32

Table 4.6: Per-role gas and USD cost: $1000\,\mathrm{S} \times 1000\,\mathrm{V}$ - FIFO.

Token	Role	Gas mean	Gas median	Gas min	Gas max	USD mean	USD median	USD min	USD max
ARB	owner	233,323,743	233,323,743	233,323,743	233,323,743	$0.001505	$0.001505	$0.001505	$0.001505
	sender	1,825,063	1,823,343	1,823,343	1,851,919	$0.000012	$0.000012	$0.000012	$0.000012
	verifier	4,039,766	4,036,867	3,041,701	5,085,633	$0.000026	$0.000026	$0.000020	$0.000033
BNB	owner	233,323,743	233,323,743	233,323,743	233,323,743	$177.02	$177.02	$177.02	$177.02
	sender	1,825,063	1,823,343	1,823,343	1,851,919	$1.38	$1.38	$1.38	$1.41
	verifier	4,039,766	4,036,867	3,041,701	5,085,633	$3.06	$3.06	$2.31	$3.86
ETH	owner	233,323,743	233,323,743	233,323,743	233,323,743	$1,457.36	$1,457.36	$1,457.36	$1,457.36
	sender	1,825,063	1,823,343	1,823,343	1,851,919	$11.40	$11.39	$11.39	$11.57
	verifier	4,039,766	4,036,867	3,041,701	5,085,633	$25.23	$25.21	$19.00	$31.77
GNO	owner	233,323,743	233,323,743	233,323,743	233,323,743	$101.71	$101.71	$101.71	$101.71
	sender	1,825,063	1,823,343	1,823,343	1,851,919	$0.7956	$0.7948	$0.7948	$0.8073
	verifier	4,039,766	4,036,867	3,041,701	5,085,633	$1.76	$1.76	$1.33	$2.22
OP	owner	233,323,743	233,323,743	233,323,743	233,323,743	$0.002523	$0.002523	$0.002523	$0.002523
	sender	1,825,063	1,823,343	1,823,343	1,851,919	$0.000020	$0.000020	$0.000020	$0.000020
	verifier	4,039,766	4,036,867	3,041,701	5,085,633	$0.000044	$0.000044	$0.000033	$0.000055
POLY	owner	233,323,743	233,323,743	233,323,743	233,323,743	$4.04	$4.04	$4.04	$4.04
	sender	1,825,063	1,823,343	1,823,343	1,851,919	$0.0316	$0.0316	$0.0316	$0.0321
	verifier	4,039,766	4,036,867	3,041,701	5,085,633	$0.0700	$0.0700	$0.0527	$0.0881

Table 4.7: Per-role gas and USD cost: $1000\,\mathrm{S} \times 1000\,\mathrm{V}$ - PFC.

Token	Role	Gas mean	Gas median	Gas min	Gas max	USD mean	USD median	USD min	USD max
ARB	owner	2,180,651,354	2,180,651,354	2,180,651,354	2,180,651,354	$0.0141	$0.0141	$0.0141	$0.0141
	sender	2,167,395	2,165,677	2,165,181	2,217,449	$0.000014	$0.000014	$0.000014	$0.000014
	verifier	4,192,008	4,189,109	3,100,083	5,331,735	$0.000027	$0.000027	$0.000020	$0.000034
BNB	owner	2,180,651,354	2,180,651,354	2,180,651,354	2,180,651,354	$1,654.40	$1,654.40	$1,654.40	$1,654.40
	sender	2,167,395	2,165,677	2,165,181	2,217,449	$1.64	$1.64	$1.64	$1.68
	verifier	4,192,008	4,189,109	3,100,083	5,331,735	$3.18	$3.18	$2.35	$4.05
ETH	owner	2,180,651,354	2,180,651,354	2,180,651,354	2,180,651,354	$13,620.54	$13,620.54	$13,620.54	$13,620.54
	sender	2,167,395	2,165,677	2,165,181	2,217,449	$13.54	$13.53	$13.52	$13.85
	verifier	4,192,008	4,189,109	3,100,083	5,331,735	$26.18	$26.17	$19.36	$33.30
GNO	owner	2,180,651,354	2,180,651,354	2,180,651,354	2,180,651,354	$950.59	$950.59	$950.59	$950.59
	sender	2,167,395	2,165,677	2,165,181	2,217,449	$0.9448	$0.9441	$0.9439	$0.9666
	verifier	4,192,008	4,189,109	3,100,083	5,331,735	$1.83	$1.83	$1.35	$2.32
OP	owner	2,180,651,354	2,180,651,354	2,180,651,354	2,180,651,354	$0.0236	$0.0236	$0.0236	$0.0236
	sender	2,167,395	2,165,677	2,165,181	2,217,449	$0.000023	$0.000023	$0.000023	$0.000024
	verifier	4,192,008	4,189,109	3,100,083	5,331,735	$0.000045	$0.000045	$0.000034	$0.000058
POLY	owner	2,180,651,354	2,180,651,354	2,180,651,354	2,180,651,354	$37.79	$37.79	$37.79	$37.79
	sender	2,167,395	2,165,677	2,165,181	2,217,449	$0.0376	$0.0375	$0.0375	$0.0384
	verifier	4,192,008	4,189,109	3,100,083	5,331,735	$0.0727	$0.0726	$0.0537	$0.0924

Table 4.8: Per-role gas and USD cost: $1000\,\mathrm{S} \times 1000\,\mathrm{V}$ - PSC.

few SSTORE operations. The difference between `FIFO` and `PFC` is evident in verifier gas costs, amounting to approximately $1,740$USD on the Ethereum network. The additional functionality of `PSC` compared to `PFC` has a cost of about $12,000$USD on Ethereum network.

4.2 Block Saturation Under Concurrent Campaigns

The Ethereum blockchain enforces a per-block gas limit, set to 60,000,000 gas units under the Fusaka hard fork [55], which bounds the total computational work that validators may include in a single block [34]. This experiment aims to verify whether this constraint introduces a critical section on block capacity when multiple campaigns execute concurrently. To this end, the block count consumed by each campaign is measured as a function of the number of concurrent campaigns N.

4.2.1 Gas Consumption Baseline

The gas consumption of every contract is reported in Tables 4.6, 4.7, and 4.8. `PFC` is the contract with the lowest gas consumption. This experiment was conducted using `PFC`, because if multiple concurrent executions of `PFC` exhibit a critical section over the per-block gas limit, then contracts with a higher gas consumption will necessarily exhibit the same contention. For this reason, `PFC` was chosen as the baseline.

4.2.2 Experimental Setup

The simulation was executed on a local Hardhat EVM node configured with a block gas limit of 60,000,000 gas [55] and interval mining at 200 ms, so that blocks are

N	Mean blocks	Ratio vs. N=1	Per-campaign blocks
1	21 564	1.00×	[21 564]
2	34 160	1.58×	[34 145, 34 176]
4	78 627	3.65×	[78 732, 78 538, 78 753, 78 485]

Table 4.9: Blocks consumed per campaign under concurrent execution (`PFC`, 1,000 senders, 100 verifiers, 5,000 CIDs).

produced at a fixed cadence regardless of mempool pressure. Between each batch of N campaigns, the chain state was reset via `hardhat_reset` to ensure independent measurements.

Each campaign runs the `PFC` contract with 1,000 senders, 100 verifiers, 5,000 CIDs, and `checksRequired`= 3. Every actor belongs to exactly one campaign; no wallets are shared across campaigns. Concurrency is achieved by launching all N campaign runners inside a single `Promise.all`, so that transactions from different campaigns interleave in the mempool and compete for block space under the shared gas limit. The values of N evaluated are $N \in \{1, 2, 4\}$.

The metric recorded for each campaign is the number of Ethereum blocks consumed, defined as $\Delta b = b_{\text{end}} - b_{\text{start}}$, where b_{start} is the block number snapshotted immediately before `Promise.all` fires and b_{end} is the block number of the last confirmed transaction of that campaign. Wall-clock time is intentionally excluded, as it reflects the speed of the local Hardhat node rather than mainnet behaviour.

4.2.3 Results and analysis

Table 4.9 reports the mean block count consumed per campaign as a function of N for `PFC`, together with the individual per-campaign values.

As N increases, each campaign consumes measurably more blocks than it would in isolation. At $N = 4$, the mean block count reaches 3.65×, this directly evidences a critical section on block capacity: concurrent campaigns compete for the 60,000,000-gas per-block limit [55].

4.3 Spatial simulation

Section 2.6.2 describes a mechanism to address the geographical incentive imbalance. The following simulation aims to empirically validate the implemented solution, using the Taxi Trajectory dataset from Porto [56].

Senders	Total CIDs	Covered chunks	Coverage
500	21 408	47 / 100	47%
1000	45 554	50 / 100	50%

Table 4.10: Spatial coverage results for the Porto taxi dataset across different numbers of senders. The grid consists of 100 chunks (`chunk4Side` = 10, `maximumDataPerChunk` = 10 000). Coverage saturates at 50% between $N = 500$ and $N = 1{,}000$, reflecting the geographic concentration of the Porto taxi fleet rather than a protocol limitation.

4.3.1 Experimental setup

The experiment is conducted over the city of Porto using the taxi trajectory dataset [56]. The grid comprises 10 chunks per side, for a total of 100 chunks. The campaign area is a square of approximately 28 km per side, where each chunk covers approximately 2.8 km × 2.1 km. The origin of the area, located at the south-west corner, is at 41.14°N, 8.61°W. The experiment is conducted with 500 and 1,000 senders; since coverage increases by only 3 chunks between the two configurations, only the results for the 1,000-sender scenario are presented. Each taxi trajectory is associated with a distinct sender, and each chunk accepts a maximum of 10,000 data entries.

4.3.2 Spatial coverage

Table 4.10 shows how chunk coverage varies across the two simulation configurations. The coverage gain is negligible, whereas the number of submitted CIDs increases by a factor greater than 2. This stagnation is attributable to the geographic concentration of the Porto taxi fleet: trajectories are predominantly located in the city centre, where population density is higher, as described in Section 2.5.3. An overview of the geographical distribution of the submitted data is shown in Figure 4.6.

4.3.3 Reward decay evaluation

Figures 4.7 and 4.8 show how rewards vary as k ranges over $\{1, 2, 3, 4\}$. For cold chunks, those with no submissions, the reward remains constant at its maximum value of 1,000 DST. For heavily saturated chunks the difference is substantial: on chunk 53, the most saturated one ($\alpha = 0.931$), the expected reward for the next CID is 263 DST with $k = 1$ and 846 DST with $k = 4$. Consequently, a higher value of k yields a higher reward for saturated chunks, resulting in a more gradual decay curve.

Figure 4.6: Geographical distribution of received CIDs across chunks, overlaid on a map of Porto ($N = 1{,}000$ senders).

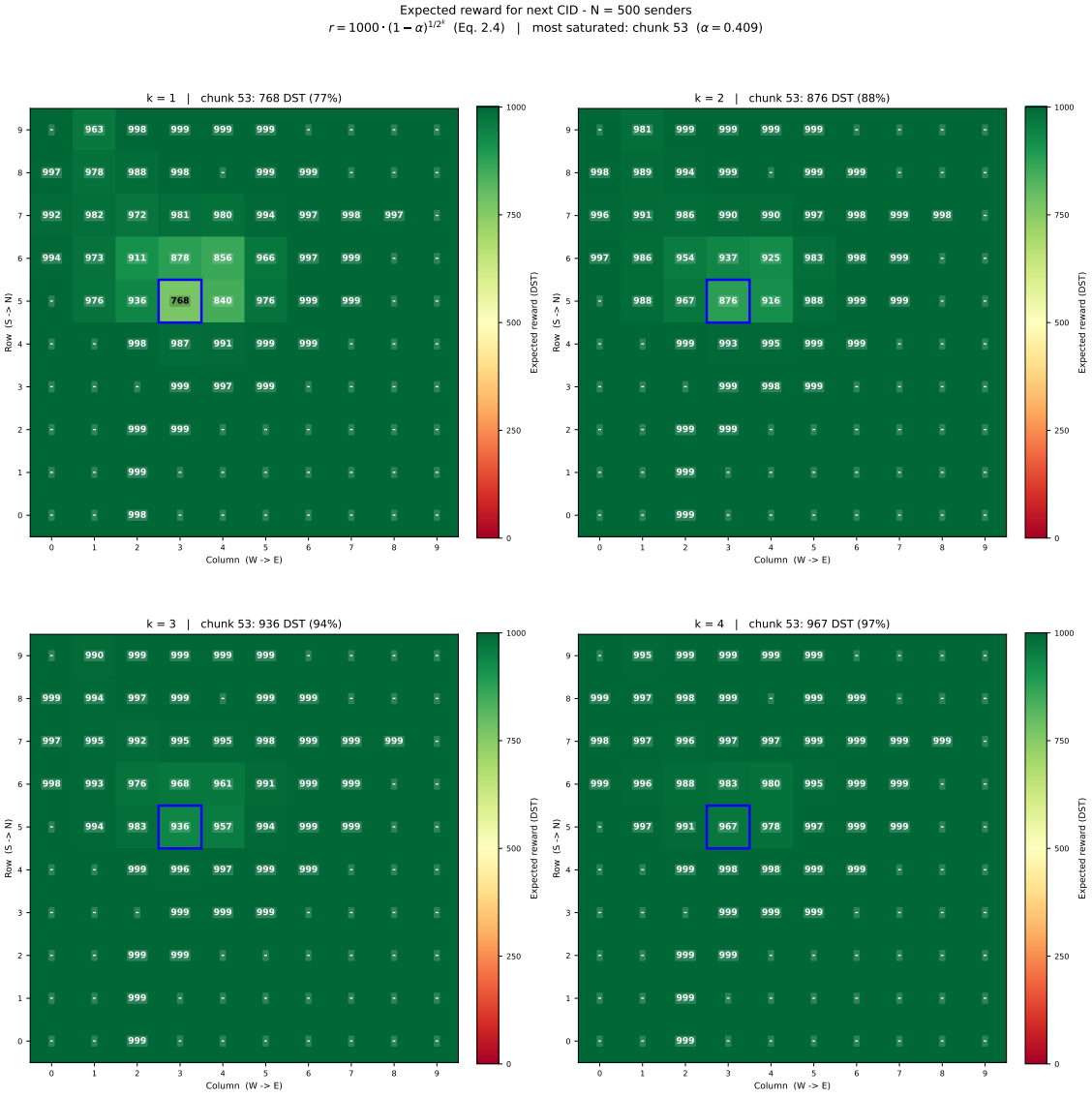

Figure 4.7: Expected reward per chunk for the next CID submission, with $N = 500$ senders and $k \in \{1, 2, 3, 4\}$.

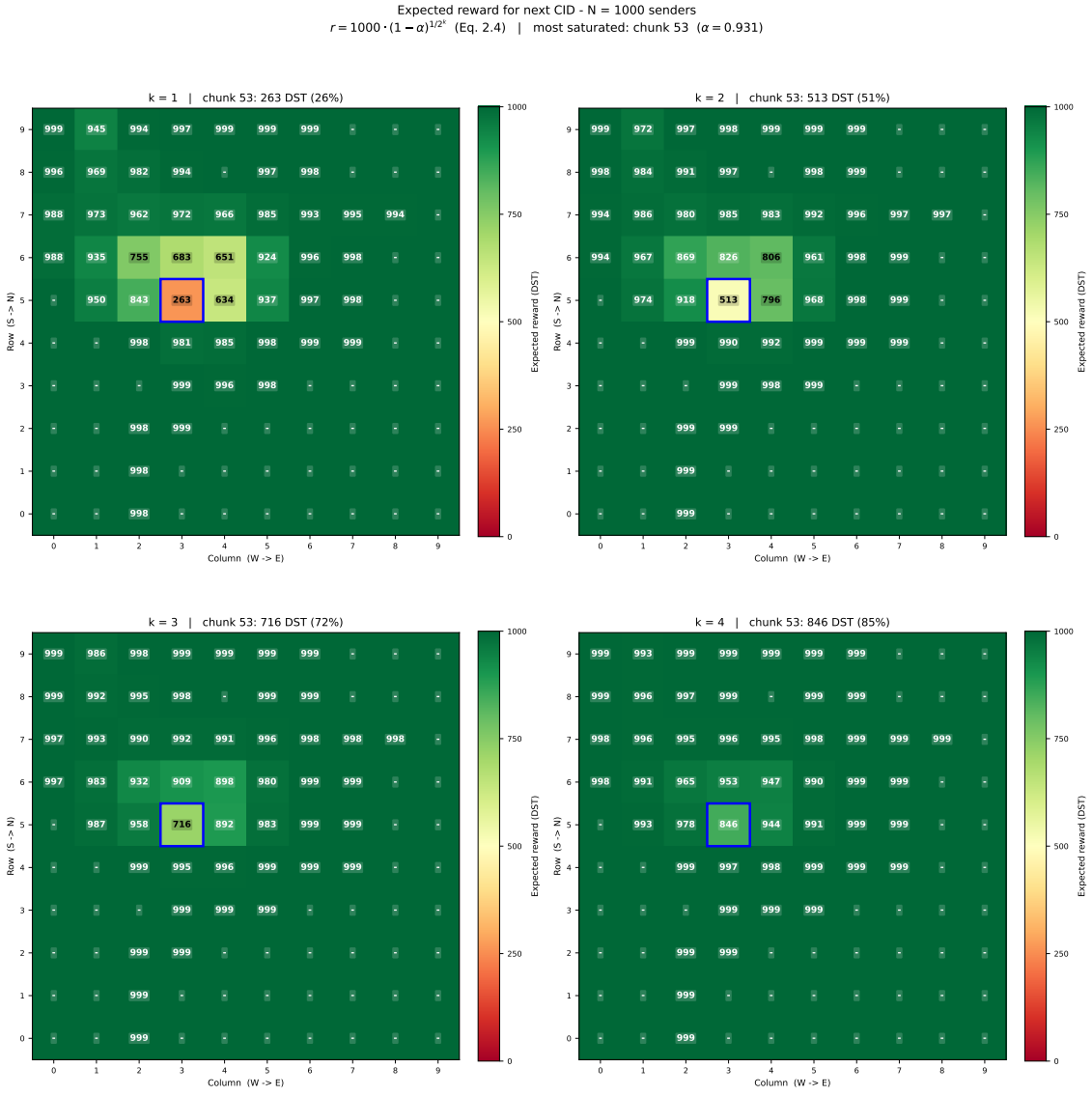

Figure 4.8: Expected reward per chunk for the next CID submission, with $N = 1{,}000$ senders and $k \in \{1, 2, 3, 4\}$.

4.3.4 Discussion

The simulation is conducted with at most 1,000 senders due to RAM constraints. The results confirm that the decay mechanism operates as designed: Equation 2.4 in Section 2.6.2 produces an economic signal that incentivises rational senders to contribute to underrepresented chunks in order to maximise their reward. Furthermore, this experiment confirms that the implementation described in Section 3.4.4 is correct.

4.4 Result and discussion

The experiments presented in this chapter lead to several conclusions.

Section 4.1 compares the three contract variants. `FIFO` is infeasible under EIP-7825 [47]: its $O(n)$ pending list removal causes the worst-case gas of `submitVerifyResult` to exceed the per-transaction cap already at moderate CID counts. `PFC` is the most efficient contract, but it does not address the issues analysed in Section 2.5. `PSC` addresses those issues at a measurable but controlled overhead: the additional gas introduced by its three extensions remains within the same order of magnitude as `PFC`. Across all three contracts, deployment and operation on Ethereum mainnet is expensive for the campaign initiator, while sender and verifier costs are acceptable. On L2 networks the costs become negligible for all roles.

Section 4.2 shows that N concurrent campaigns produce a critical section on block capacity. In production, concurrent campaigns should be staggered or deployed on L2 networks with a higher block gas limit or a shorter block time to reduce contention.

Section 4.3 confirms that the reward decay mechanism operates as designed. The 50% spatial coverage observed on the Porto dataset reflects the intrinsic geographic concentration of the taxi fleet rather than a protocol limitation. The decay parameter k controls the steepness of the incentive curve: higher values produce a gentler decay, preserving a meaningful reward even for highly saturated chunks, while lower values more aggressively redirect senders towards underrepresented areas.

Chapter 5

Conclusions

This thesis presented the design, implementation, and evaluation of a privacy-preserving decentralised crowdsensing protocol based on smart contracts. This chapter summarizes the contributions of the work, discusses its limitations, and outlines directions for future research.

5.1 Summary of contributions

The work presented in this thesis covers three main areas: protocol design, implementation, and evaluation.

In the design phase, `PrivacyFirstContract` was developed, building on the crowdsensing protocol proposed by Bedogni and Ferretti [7]. A subsequent security and incentive analysis identified several open issues, which were addressed in `PrivacySecondContract`, an extension that introduces a mechanism restricting each verifier to the data it is elected to verify, a geographical reward decay, and a verifier penalty system.

The implementation phase produced two Solidity smart contracts reflecting the above design, together with a TypeScript SDK layer that exposes a per-actor API and handles off-chain cryptographic operations, providing a complete reference implementation of the protocol.

The evaluation phase assessed the contracts along three axes: gas consumption and its real-world cost across multiple networks, block capacity contention under concurrent campaign execution, and spatial coverage and reward distribution on a real-world taxi trajectory dataset from the city of Porto.

5.2 Limitations

The work presented in this thesis has several limitations.

The geographic simulation was conducted with at most 1 000 senders due to RAM constraints on the test machine; a larger population could not be evaluated without dedicated infrastructure. Furthermore, the contracts were tested exclusively on a local Hardhat node and were not deployed on Ethereum mainnet or any L2 network, so the gas cost figures reported in Section 4.1 are based on local measurements rather than live network conditions.

At the protocol level, the centrality of the campaign initiator `B` remains an open issue: `PrivacySecondContract` does not eliminate `B` as a trust point, as discussed in Section 2.5.

Finally, the ProVerif analysis described in Section 2.5.1 assumes honest enrollment: the sender and verifier enrollment channels are modelled as private, so the attacker cannot enrol as a malicious participant. As a consequence, the formal verification only guarantees secrecy properties under this assumption; scenarios involving a compromised sender or verifier are outside the scope of the model, as noted in Appendix B.

5.3 Future work

Several directions are open for future investigation. The centrality of the campaign initiator `B` could be eliminated by replacing the current key distribution mechanism with Proxy Re-Encryption(PRE) [57]: senders would encrypt their data under their own public key, and a decentralised proxy network, such as the Threshold Network[58], would re-encrypt each ciphertext for the elected verifier without ever learning the plaintext, removing `B` as an online trust point entirely.

The formal security analysis could be strengthened by extending the ProVerif model to cover dishonest enrollment scenarios, or by porting the verification to CryptoVerif [59], which operates in the computational model and provides guarantees over concrete cryptographic primitives rather than perfect symbolic abstractions.

On the evaluation side, the geographic simulation should be replicated on datasets with a more uniform spatial distribution, such as rural or multi-city traces, to assess the effectiveness of the reward decay mechanism beyond the concentrated urban setting of the Porto taxi fleet. Finally, the contracts should be deployed on a public testnet and subsequently on an L2 network to validate the gas cost figures reported in Section 4.1 under real network conditions and to measure end-to-end latency across all protocol phases.

Bibliography

[1] Raghu K Ganti, Fan Ye, and Hui Lei. Mobile crowdsensing: current state and future challenges. *IEEE communications Magazine*, 49(11):32–39, 2011.

[2] Andrea Capponi, Claudio Fiandrino, Burak Kantarci, Luca Foschini, Dzmitry Kliazovich, and Pascal Bouvry. A survey on mobile crowdsensing systems: Challenges, solutions, and opportunities. *IEEE Communications Surveys & Tutorials*, 21(3):2419–2465, 2019.

[3] Zhiyan Chen, Claudio Fiandrino, and Burak Kantarci. On blockchain integration into mobile crowdsensing via smart embedded devices: A comprehensive survey. *Journal of Systems Architecture*, 115:102011, 2021.

[4] Ruiyun Yu, Ann Move Oguti, Mohammad S. Obaidat, Shuchen Li, Pengfei Wang, and Kuei-Fang Hsiao. Blockchain-based solutions for mobile crowdsensing: A comprehensive survey. *Computer Science Review*, 50:100589, 2023.

[5] Juan Benet. IPFS - content addressed, versioned, p2p file system. Technical report, Protocol Labs, 2014.

[6] Fabian Vogelsteller and Vitalik Buterin. EIP-20: Token standard. Ethereum Improvement Proposals, 2015.

[7] Luca Bedogni and Stefano Ferretti. Smart contract coordinated privacy preserving crowdsensing campaigns. In *2025 IEEE 22nd Consumer Communications & Networking Conference (CCNC)*, pages 1–4, 2025.

[8] Lorenzo Gigli, Federico Montori, Mirko Zichichi, Luca Bedogni, Stefano Ferretti, and Marco Di Felice. On the decentralization of mobile crowdsensing in distributed ledgers: An architectural vision. In *Proceedings of the 2024 IEEE 21st Annual Consumer Communications and Networking Conference (CCNC)*, pages 1–7. IEEE, 2024.

[9] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 82–96. IEEE, 2001.

[10] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[11] Andrew S. Tanenbaum and Maarten Van Steen. Distributed systems principles and paradigms. *Network*, 2(28):1, 2002.

[12] R. Fielding, M. Nottingham, and J. Reschke. HTTP semantics. RFC 9110, IETF, 2022.

[13] P. Mockapetris. Domain names - concepts and facilities. RFC 1034, IETF, 1987.

[14] R. Droms. Dynamic host configuration protocol. RFC 2131, IETF, 1997.

[15] 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.

[16] Bram Cohen et al. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.

[17] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Self-published*, 2008.

[18] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. *Self-published*, 2014.

[19] Eldar Kapengut and Bruce Mizrach. An event study of the ethereum transition to proof-of-stake. *Commodities*, 2(1):1–15, 2023.

[20] Ethereum Foundation. 51% attack, 2023.

[21] Xu Kang, Liang Liu, and Huadong Ma. Enhance the quality of crowdsensing for fine-grained urban environment monitoring via data correlation. *Sensors*, 17(1):88, 2017.

[22] Qipei Mei, Mustafa Gül, and Nima Shirzad-Ghaleroudkhani. Towards smart cities: crowdsensing-based monitoring of transportation infrastructure using in-traffic vehicles. *Journal of Civil Structural Health Monitoring*, 10:653–665, 2020.

[23] Nicholas D Lane, Emiliano Miluzzo, Hong Lu, Daniel Peebles, Tanzeem Choudhury, and Andrew T Campbell. A survey of mobile phone sensing. *IEEE Communications magazine*, 48(9):140–150, 2010.

[24] Shihong Zou, Jinwen Xi, Honggang Wang, and Guoai Xu. Crowdblps:

A blockchain-based location-privacy-preserving mobile crowdsensing system. *IEEE transactions on industrial informatics*, 16(6):4206–4218, 2019.

[25] Maha Kadadha, Rabeb Mizouni, Shakti Singh, Hadi Otrok, and Anis Ouali. Abcrowd an auction mechanism on blockchain for spatial crowdsourcing. *IEEE Access*, 8:12745–12757, 2020.

[26] Junqin Huang, Linghe Kong, Long Cheng, Hong-Ning Dai, Meikang Qiu, Guihai Chen, Xue Liu, and Gang Huang. Blocksense: Towards trustworthy mobile crowdsensing via proof-of-data blockchain. *IEEE Transactions on Mobile Computing*, 23(2):1016–1033, 2022.

[27] Dimitris Chatzopoulos, Sujit Gujar, Boi Faltings, and Pan Hui. Privacy preserving and cost optimal mobile crowdsensing using smart contracts on blockchain. In *2018 IEEE 15th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, pages 442–450. IEEE, 2018.

[28] Yuan Lu, Qiang Tang, and Guiling Wang. Zebralancer: Private and anonymous crowdsourcing system atop open blockchain. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 853–865. IEEE, 2018.

[29] Chenhao Ying, Haiming Jin, Jie Li, Xueming Si, and Yuan Luo. Incentive mechanism design via smart contract in blockchain-based edge-assisted crowdsensing. *Frontiers of Computer Science*, 19(1):193802, 2025.

[30] Jong Wook Kim, Kennedy Edemacu, and Beakcheol Jang. Privacy-preserving mechanisms for location privacy in mobile crowdsensing: A survey. *Journal of Network and Computer Applications*, 200:103315, 2022.

[31] Robert Muth and Florian Tschorsch. Smartdhx: Diffie-hellman key exchange with smart contracts. In *2020 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, pages 164–168, 2020.

[32] Armando Ruggeri, Antonio Celesti, Maria Fazio, Antonino Galletta, and Massimo Villari. BCB-X3DH: A blockchain based improved version of the extended triple Diffie-Hellman protocol. In *2020 Second IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, pages 73–78. IEEE, 2020.

[33] Nsikak P. Owoh and Manjit Singh Singh. Applying diffie-hellman algorithm to solve the key agreement problem in mobile blockchain environment. In *Proceedings of the International Conference on Advances in Cyber Security*. Springer, 2019.

[34] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014.

[35] National Geospatial-Intelligence Agency. World Geodetic System 1984: Its Definition and Relationships with Local Geodetic Systems. Technical Report NGA.STND.0036_1.0.0, National Geospatial-Intelligence Agency, 2014.

[36] Federico Montori and Luca Bedogni. Privacy preservation for spatio-temporal data in mobile crowdsensing scenarios. *Pervasive and Mobile Computing*, 90:101755, 2023.

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

[38] Gilles Brassard, David Chaum, and Claude Crépeau. Minimum disclosure proofs of knowledge. *Journal of Computer and System Sciences*, 37(2):156–189, 1988.

[39] Ethereum Foundation. Commit-reveal voting, 2018.

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

[41] Olga Kosheleva. Babylonian method of computing the square root: Justifications based on fuzzy techniques and on computational complexity. In *Proceedings of the 28th Annual Conference of the North American Fuzzy Information Processing Society (NAFIPS 2009)*, Cincinnati, Ohio, USA, June 2009.

[42] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. CRC Press, 3 edition, 2020.

[43] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. PKCS #1: RSA cryptography specifications version 2.2. RFC 8017, IETF, 2016.

[44] Morris Dworkin. Recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC. Special Publication 800-38D, National Institute of Standards and Technology, 2007.

[45] Solidity Contributors. Contract ABI specification – Solidity documentation. `https://docs.soliditylang.org/en/latest/abi-spec.html`, 2024. Accessed: 2026.

[46] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[47] Giulio Rebuffo and Toni Wahrstätter. EIP-7825: Transaction gas limit cap. Ethereum Improvement Proposal, November 2024. Status: Final.

[48] Etherscan. Ethereum average gas price chart. https://etherscan.io/chart/gasprice, 2025. Median 2025 value. Accessed: 2026.

[49] PolygonScan. Polygon average gas price chart. https://polygonscan.com/chart/gasprice, 2025. Median 2025 value. Accessed: 2026.

[50] Arbiscan. Arbitrum average gas price chart. https://arbiscan.io/chart/gasprice, 2025. Median 2025 value. Accessed: 2026.

[51] Optimistic Etherscan. Optimism average gas price chart. https://optimistic.etherscan.io/chart/gasprice, 2025. Median 2025 value. Accessed: 2026.

[52] GnosisScan. Gnosis average gas price chart. https://gnosisscan.io/chart/gasprice, 2025. Median 2025 value. Accessed: 2026.

[53] BscScan. Bnb chain average gas price chart. https://bscscan.com/chart/gasprice, 2025. Median 2025 value. Accessed: 2026.

[54] CoinGecko. Cryptocurrency historical market data. https://www.coingecko.com, 2025. Median 2025 USD/token, per-network CSV exports. Accessed: 2026.

[55] Sophia Gold, Parithosh Jayanthi, Toni Wahrstätter, Carl Beekhuizen, Ansgar Dietrichs, Dankrad Feist, Alex Stokes, Josh Rudolph, Giulio Rebuffo, Storm Slivkoff, and Kamil Chodoła. EIP-7935: Set default gas limit to 60M. Ethereum Improvement Proposals, 2025. https://eips.ethereum.org/EIPS/eip-7935.

[56] Luis Moreira-Matias, Michel Ferreira, and João Mendes-Moreira. Taxi service trajectory – prediction challenge, ecml pkdd 2015, 2015.

[57] Giuseppe Ateniese, Kevin Fu, Matthew Green, and Susan Hohenberger. Improved proxy re-encryption schemes with applications to secure distributed storage. *ACM Transactions on Information and System Security (TISSEC)*, 9(1):1–30, 2006.

[58] Threshold Network. Threshold Network documentation. https://docs.threshold.network, 2024. Accessed: 2026.

[59] Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, 2008.

Appendices

Appendix A

Smart Contract Configuration Parameters

This appendix provides a comprehensive reference for all configurable parameters accepted by the smart contract at deployment time. The constructor receives a `Config` struct, which is composed of four sub-structures: `SmartContractNeeds`, `ChunkConfig`, `RewardConfig` and `GlobalPosition`. Each sub-structure is documented in a dedicated table below. In addition, the constructor receives the `B` public key and the address of `DataSentTokenContract` which is the token introduced for senders and verifiers rewarding. The details of rewarding model are described in the 2.4.2 subsection.

A convention adopted throughout the configuration is the **zero-means-unlimited** semantics: when a parameter that governs a limit is set to zero, the corresponding constraint is not enforced. This convention is explicitly noted for each applicable parameter.

A.1 Campaign Requirements (`SmartContractNeeds`)

The `SmartContractNeeds` structure defines the operational rules of the crowdsensing campaign, including participation limits, and data acceptance criteria. A.1 lists each parameter with its type, constraints, and description.

A.2 Spatial Chunking (`ChunkConfig`)

The `ChunkConfig` structure controls the spatial partitioning of the campaign area into a square grid of chunks. Table A.2 lists the two parameters involved.

Parameter	Type	Constraint	Description
`dataNeeded`	`uint256`	≥ 0 (0 = unlimited)	Number of positively verified data entries required to automatically close the campaign. When set to zero, the campaign does not autoclose based on data count alone.
`maximumDataPerChunk`	`uint256`	≥ 0 (0 = unlimited)	Maximum number of data submissions accepted within a single spatial chunk. Prevents over-representation of a geographic area.
`checksRequired`	`uint8`	odd, ≥ 1	Number of independent verification slots allocated per data entry. The odd constraint ensures that a strict majority can always be determined.
`verifierCooldown`	`uint256`	> 0	Minimum distance (in data entry indices) between consecutive assignments of the same verifier. Prevents a single verifier from being assigned to sequentially adjacent data entries, promoting verification independence.
`verificationTimeout`	`uint256`	> 0	Time window (in seconds) granted to a verifier to complete the assigned verification task. If the deadline expires, the verification slot becomes available for reassignment.
`minimumParticipants`	`uint256`	≥ 0 (0 = unset)	Minimum number of unique senders that must have at least one positively verified data entry for the campaign to auto-close. Combined with `dataNeeded` in a dual-threshold check.
`maxDataPerSender`	`uint256`	≥ 0 (0 = unlimited)	Maximum number of data entries a single sender is allowed to submit. When greater than zero, the bond mechanism is activated: senders must deposit a bond proportional to this limit before submitting data.
`maxDataPerVerifierPerBond`	`uint256`	≥ 0	Number of verification tasks covered by a single bond deposit. Used to compute the required verifier bond as `maxDataPerVerifierPerBond` $\times$ `dataVerifierBondPerData`.

Table A.1: Parameters of the `SmartContractNeeds` structure.

Parameter	Type	Constraint	Description
`chunkSize`	`uint256`	> 0	Side length of each square chunk, expressed in E7 coordinate units (i.e., 10^{-7} degrees). For reference, 1 000 E7 units correspond to approximately 11 meters at the equator.
`chunk4Side`	`uint16`	> 0	Number of chunks along each side of the square campaign area. The total number of chunks is `chunk4Side`^{2}. When set to 1, the spatial chunking system is effectively disabled and the entire campaign area is treated as a single region.

Table A.2: Parameters of the `ChunkConfig` structure.

The total geographic extent of the campaign area is derived as `chunkSize`$\times$`chunk4Side` E7 units per side, starting from the origin coordinates specified in the `GlobalPosition` parameter (see Section A.4.

A.3 Reward and Bond Configuration (`RewardConfig`)

The `RewardConfig` structure governs the economic model of the campaign, defining the total token supply, per-operation payouts, and bond amounts denominated in DST (DataSent Token). A.3 lists all parameters.

The bond mechanism serves a dual purpose: it discourages low-quality submissions and frivolous verifications, while ensuring that participants have a stake in the correctness of their contributions. Bonds are deducted at action time (submission or assignment) rather than at reward time, so that the balance check is enforced before the operation is recorded on-chain.

Parameter	Type	Constraint	Description
`totalDst`	`uint256`	> 0	Total amount of DST tokens that the campaign initiator must deposit into the contract to fund all sender and verifier rewards.
`dataSenderPayout`	`uint16`	$>$ `dataSenderBondPerData`	DST tokens paid to a sender when one of their data entries is positively verified. The constraint ensures that participation is always economically incentivized.
`dataVerifierPayout`	`uint16`	$>$ `dataVerifierBondPerData`	DST tokens paid to a verifier who voted with the majority on a resolved data entry.
`dataSenderBondPerData`	`uint16`	> 0	DST tokens deducted from the sender's bond balance at data submission time. Returned together with the payout upon positive verification.
`dataVerifierBondPerData`	`uint16`	> 0	DST tokens deducted from the verifier's bond balance when a verification task is assigned. Returned together with the payout upon a majority-aligned vote.

Table A.3: Parameters of the `RewardConfig` structure.

Field	Type	Valid Range	Description
`latitudeInteger`	`int32`	$[-90,\ +90]$	Integer part of the latitude in degrees.
`latitudeDecimal`	`uint32`	$[0,\ 9\,999\,999]$	Fractional part of the latitude, representing up to seven decimal digits.
`longitudeInteger`	`int32`	$[-180,\ +180]$	Integer part of the longitude in degrees.
`longitudeDecimal`	`uint32`	$[0,\ 9\,999\,999]$	Fractional part of the longitude, representing up to seven decimal digits.

Table A.4: Fields of the `GlobalPosition` structure.

A.4 Campaign Origin Coordinates (`GlobalPosition`)

The `GlobalPosition` structure defines the geographic origin (south-west corner) of the campaign area. The coordinate representation and the E7 packing mechanism are described in detail in the implementation chapter. Table A.4 lists the fields and their valid ranges.

A.5 Extended Protocol Parameters (`NovelParameters`)

The `NovelParameters` structure contains the parameters introduced by the protocol extensions described in Section 2.6. These parameters are accepted at deployment time alongside the base configuration. A.5 lists each parameter with its type, constraints, and description.

Parameter	Type	Constraint	Description
`rewardDecayFactor`	`uint256`	> 0	Controls the steepness of the reward decay curve as a chunk approaches saturation. Higher values produce a gentler decay, while lower values penalize contributions to already-populated chunks more aggressively. Used in Equation 2.4.
`kickVerifierAfterWrongVotes`	`uint256`	≥ 0 (0 = disabled)	Number of accumulated incorrect votes after which a verifier is automatically removed from the campaign. When set to zero, the automatic removal mechanism is disabled and only manual removal by `B` is available.
`pardonWrongVoteAfterRightVotes`	`uint256`	≥ 0 (0 = disabled)	Number of correct votes required to decrease the incorrect vote counter by one. Provides a forgiveness mechanism so that honest verifiers who occasionally make mistakes are not unfairly penalized over the course of a long campaign. When set to zero, incorrect votes are never forgiven.

Table A.5: Parameters of the `NovelParameters` structure.

Appendix B

Formal Verification with ProVerif

This appendix presents the complete ProVerif model used for the formal security analysis discussed in Section 2.5. The model captures the cryptographic key exchange and data submission protocol described in Chapter 2, abstracting away non-cryptographic smart contract logic which falls outside the scope of symbolic protocol verification.

B.1 Modeling Assumptions

The verification is performed under the Dolev-Yao attacker model [10], in which the adversary has full control over all public communication channels. In particular, the attacker is able to read, inject, modify, replay, and delete any message transmitted on a public channel. This provides a conservative upper bound on the adversary's capabilities: any property that holds under Dolev-Yao also holds against weaker, purely passive attackers.

The protocol involves four actors: the data sender S, the data verifier V, the campaign initiator B (owner), and the smart contract SC. Each of them modeled as a separate ProVerif process. The smart contract is modeled as an active intermediary rather than a passive public channel. This distinction is essential: in the real system, SC enforces access control through Solidity `require` statements and the EVM guarantees the authenticity of emitted events [18]. Table B.1 summarizes how each channel in the model corresponds to a specific contract function or blockchain-level guarantee.

The initiator's public key PK_B is passed as a parameter to processes S and V rather than being read from a public channel. This models the fact that PK_B is embedded in the smart contract at deployment time and retrieved via `getInitiatorPublicKey()`,

Channel	Contract function / mechanism	Security basis
ch_b_sender_key	publishSenderEncryptionKey()	require(msg.sender == _owner)
ch_b_verifier_key	publishVerifierDecryptionKey()	require(msg.sender == _owner)
ch_s_data	publishIpfsCid()	require(_senders[msg.sender])
ch_sc_to_s	SenderEncryptionKeyReady event	Event authenticity (EVM)
ch_sc_to_v	VerifierElected event	Event authenticity (EVM)
ch_sc_notify_b_*	SenderEnrolledEvent and similar	Event authenticity (EVM)
ch_s_enroll	publishEncryptedSenderPK()	Honest scenario assumption
ch_v_enroll	publishEncryptedVerifierPK()	Honest scenario assumption
pub	All emitted events	Public (attacker observes)
ipfs	IPFS data storage	Public (attacker observes)

Table B.1: Mapping between ProVerif channels, contract functions, and security mechanisms.

a read from immutable contract storage whose integrity is guaranteed by blockchain consensus [17; 18]. In addition, enrollment channels are modeled as private,restricting participation to honest senders and verifiers (see Section B.3 for further discussion).

B.2 Security Queries

Two secrecy queries are verified:

Q1: CID Confidentiality. The attacker cannot derive the plaintext Content Identifier (CID) from the encrypted value $Enc(CID, K_{enc})$ published on the blockchain.

Q2: Data Confidentiality. The attacker cannot derive the plaintext data from the encrypted content $Enc(data, K_{enc})$ stored on IPFS.

Both queries hold under the Dolev-Yao model, confirming that even an active adversary with full control over public channels cannot recover the plaintext CID or data, provided that: (i) the smart contract's access control mechanisms (`require` checks) function correctly, (ii) blockchain event authenticity is preserved, and (iii) all enrolled participants behave honestly.

B.3 Limitations

ProVerif operates in the symbolic (Dolev-Yao) model, which assumes perfect cryptographic primitives. Consequently, the verification does not account for implementation-level vulnerabilities such as weak randomness, side-channel attacks, or flaws in specific cryptographic libraries.

The enrollment channels (`ch_s_enroll`, `ch_v_enroll`) are modeled as private, reflecting the honest scenario in which only legitimate actors participate. In the real contract, these functions lack role-based access control and can be called by any address.

B.4 ProVerif Model Source Code

The complete ProVerif model is listed below.

```
(* ProVerif model: Privacy-Enhanced Decentralized Crowdsensing       *)
(*                                                                   *)
(* Four actors: S (sender), V (verifier), B (owner), SC (contract)  *)
(*                                                                   *)
(* SC mediates ALL interactions and enforces access control via      *)
(* require(msg.sender == ...) checks. Each require is modeled as     *)
(* a private channel that only the authorized actor can write to.    *)
(*                                                                   *)
(* The attacker observes all SC events on a public channel but       *)
(* cannot bypass SC's access control to inject messages.             *)



(* Cryptographic primitives *)
type skey.

fun pk(skey): bitstring.
fun aenc(bitstring, bitstring): bitstring.
reduc forall m: bitstring, k: skey;
    adec(aenc(m, pk(k)), k) = m.

(* Type converter: allows Kdec (a secret key) to be transmitted
   as a message payload from B to V via SC. *)
fun skey_to_bitstring(skey): bitstring [data, typeConverter].


(* Channels                                                          *)
(*                                                                   *)
(* Each private channel models a require()-protected SC function.    *)
(* The channel name reflects the Solidity function it represents.    *)
(*                                                                   *)
(* Public channel: models emitted events and public view functions.  *)
(* The Dolev-Yao attacker has full read/write access to pub and ipfs *)
(* but CANNOT read or write private channels.                        *)

(* Public: attacker observes all events *)
free pub:  channel.
free ipfs: channel.

(* S -> SC: publishEncryptedSenderPublicKey()
   In the contract, anyone can call this function. We model it as
   private because the attacker enrolling as a sender is addressed
   separately in compromise scenarios. In the honest scenario, only
   legitimate senders enroll. *)
free ch_s_enroll: channel [private].

(* V -> SC: publishEncryptedVerifierPublicKey()
   Same reasoning as sender enrollment. *)
```

```
free ch_v_enroll: channel.

(* B -> SC: publishSenderEncryptionKey()
   require(msg.sender == _owner) *)
free ch_b_sender_key: channel [private].

(* B -> SC: publishVerifierDecryptionKey()
   require(msg.sender == _owner) *)
free ch_b_verifier_key: channel [private].

(* SC -> B: event notifications for enrollment
   Models B monitoring SenderEnrolledEvent / VerifierEnrolledEvent.
   Private because SC events are authentic: the attacker cannot
   forge events from a contract they do not control. *)
free ch_sc_notify_b_sender:   channel [private].
free ch_sc_notify_b_verifier: channel [private].

(* SC -> S: SenderEncryptionKeyReady event
   Private because contract events are authentic and tamper-proof.
   The attacker observes the ciphertext on pub but cannot replace
   what S receives from SC. *)
free ch_sc_to_s: channel [private].

(* SC -> V: VerifierDecryptionKeyReady / VerifierElected events
   Same reasoning: authentic contract event delivery. *)
free ch_sc_to_v: channel [private].

(* S -> SC: publishIpfsCid()
   require(_senders[msg.sender]) so only enrolled senders *)
free ch_s_data: channel [private].



(* Secrets under test   *)
free secret_cid:  bitstring [private].
free secret_data: bitstring [private].



(* Events *)
event SenderSubmittedCID(bitstring).
event VerifierObtainedCID(bitstring).


(* Security queries  *)

(* Q1: Can the attacker learn the plaintext CID? *)
query attacker(secret_cid).

(* Q2: Can the attacker learn the plaintext data stored on IPFS? *)
query attacker(secret_data).



(* Process SC: Smart Contract                                          *)
(*                                                                     *)
(* SC is a passive mediator with three replicated handlers:            *)
(*   1. Sender enrollment: receives from S, notifies B, receives       *)
(*      B's response, delivers to S, emits public events.              *)
(*   2. Verifier enrollment: same flow for verifiers.                  *)
(*   3. Data submission: receives encrypted CID from enrolled S,       *)
(*      delivers to V, emits public event.                             *)
```

```
(*                                                                    *)
(* Every input comes from a private channel that enforces the        *)
(* corresponding require() check. Every event is also published      *)
(* on pub so the attacker can observe ciphertexts.                   *)

let processSC() =
    (
        (* --- Sender enrollment handler --- *)
        (* Solidity: publishEncryptedSenderPublicKey() then
                     publishSenderEncryptionKey() *)
        !(
            in(ch_s_enroll, enc_pkS: bitstring);
            out(pub, enc_pkS);                          (* emit SenderEnrolledEvent
                *)
            out(ch_sc_notify_b_sender, enc_pkS);        (* notify B *)

            in(ch_b_sender_key, enc_kenc: bitstring);
            out(pub, enc_kenc);                          (* emit
                SenderEncryptionKeyReady *)
            out(ch_sc_to_s, enc_kenc)                    (* deliver to S *)
        )
    |
        (* --- Verifier enrollment handler --- *)
        (* Solidity: publishEncryptedVerifierPublicKey() then
                     publishVerifierDecryptionKey() *)
        !(
            in(ch_v_enroll, enc_pkV: bitstring);
            out(pub, enc_pkV);                           (* emit
                VerifierEnrolledEvent *)
            out(ch_sc_notify_b_verifier, enc_pkV);       (* notify B *)

            in(ch_b_verifier_key, enc_kdec: bitstring);
            out(pub, enc_kdec);                          (* emit
                VerifierDecryptionKeyReady *)
            out(ch_sc_to_v, enc_kdec)                    (* deliver to V *)
        )
    |
        (* --- Data submission handler --- *)
        (* Solidity: publishIpfsCid() with require(_senders[msg.sender]) *)
        !(
            in(ch_s_data, enc_cid: bitstring);
            out(pub, enc_cid);                           (* emit
                DataVerificationRequestedEvent *)
            out(ch_sc_to_v, enc_cid)                     (* deliver to V via
                VerifierElected *)
        )
    ).



(* Process B: Campaign Initiator (Owner)                              *)
(*                                                                    *)
(* B generates key pairs, deploys SC (publishes pkB), and handles     *)
(* enrollment notifications by distributing Kenc to senders and       *)
(* Kdec to verifiers through SC's owner-only functions.               *)

let processB(skB: skey, skData: skey) =
    let kenc = pk(skData) in

    (* Deploy: pkB becomes public via getInitiatorPublicKey() *)
    out(pub, pk(skB));

```

```
    (
        (* --- Handle sender enrollment notifications --- *)
        !(
            in(ch_sc_notify_b_sender, enc_pkS: bitstring);
            let pkS = adec(enc_pkS, skB) in
            out(ch_b_sender_key, aenc(kenc, pkS))
        )
    |
        (* --- Handle verifier enrollment notifications --- *)
        !(
            in(ch_sc_notify_b_verifier, enc_pkV: bitstring);
            let pkV = adec(enc_pkV, skB) in
            out(ch_b_verifier_key, aenc(skey_to_bitstring(skData), pkV))
        )
    ).



(* Process S: Data Sender                                              *)
(*                                                                     *)
(* S generates (skS, pkS). Reads pkB from contract (parameter).      *)
(* Enrolls via SC, receives Kenc, encrypts data + CID.               *)
(*                                                                     *)
(* pkB is passed as parameter: in the real contract, it is read via  *)
(* getInitiatorPublicKey() from immutable contract state. This is    *)
(* an authenticated public value that the attacker cannot replace.   *)

let processS(skS: skey, pkB: bitstring, cid: bitstring, data: bitstring) =
    (* publishEncryptedSenderPublicKey(Enc(PKS, PKB)) *)
    out(ch_s_enroll, aenc(pk(skS), pkB));

    (* Read SenderEncryptionKeyReady from SC *)
    in(ch_sc_to_s, enc_kenc: bitstring);
    let kenc = adec(enc_kenc, skS) in

    (* Encrypt data with Kenc, upload to IPFS *)
    out(ipfs, aenc(data, kenc));

    (* publishIpfsCid(Enc(CID, Kenc)) *)
    event SenderSubmittedCID(cid);
    out(ch_s_data, aenc(cid, kenc)).



(* Process V: Data Verifier                                            *)
(*                                                                     *)
(* V generates (skV, pkV). Reads pkB from contract (parameter).      *)
(* Enrolls via SC, receives Kdec, retrieves and decrypts CID + data. *)
let processV(skV: skey, pkB: bitstring) =
    (* publishEncryptedVerifierPublicKey(Enc(PKV, PKB)) *)
    out(ch_v_enroll, aenc(pk(skV), pkB));

    (* Read VerifierDecryptionKeyReady from SC *)
    in(ch_sc_to_v, enc_kdec: bitstring);
    let kdec_bits = adec(enc_kdec, skV) in
    let skey_to_bitstring(kdec) = kdec_bits in

    (* Read VerifierElected: receive encrypted CID from SC *)
    in(ch_sc_to_v, enc_cid: bitstring);
    let cid = adec(enc_cid, kdec) in

    (* Retrieve encrypted data from IPFS *)
```

```
    in(ipfs, enc_data: bitstring);
    let plaintext_data = adec(enc_data, kdec) in

    event VerifierObtainedCID(cid).


(* Main process (Honest scenario)                                    *)
process
    new skB: skey;
    new skData: skey;
    let pkB = pk(skB) in
    (
        processSC()
      | processB(skB, skData)
      | new skS: skey;
        processS(skS, pkB, secret_cid, secret_data)
      | !new skS_i: skey; new cid_i: bitstring; new data_i: bitstring;
        processS(skS_i, pkB, cid_i, data_i)
      | !new skV_i: skey;
        processV(skV_i, pkB)
    )
```

Listing B.1: ProVerif model of the crowdsensing protocol.

The result of the execution is

```
...
--------------------------------------------------------------
Verification summary:

Query not attacker(secret_cid[]) is true.

Query not attacker(secret_data[]) is true.

--------------------------------------------------------------
```

Listing B.2: ProVerif verification output for the crowdsensing protocol model.

Appendix C

Source code of supporting contracts

This appendix collects the source code of supporting Solidity contracts referenced in the body of the thesis but not central to the protocol design itself. They are reproduced here in full to make the dissertation self-contained: a reader can verify experimental claims (in particular those of Section 4.1.3) without reaching for the companion repository.

C.1 The FIFO baseline contract

`PrivacyFirstContractFifo` is the FIFO-preserving variant of `PrivacyFirstContract` used as the baseline for the gas-cost comparison of Section 4.1.3. It inherits from `PrivacyFirstContract` and overrides only the two functions that manage the pending data list, `_addToPending` and `_removeFromPending`. The inheritance pattern guarantees that every other code path is bytecode-identical to the optimised contract, isolating the measured gas difference to the single algorithmic decision under study.

The chosen FIFO strategy is the most natural alternative to swap-and-pop: `_addToPending` appends to the tail, and `_removeFromPending` performs a linear scan to locate the entry followed by a left-shift of all subsequent elements. The result is an $O(n)$ removal that preserves the insertion order, in contrast to swap-and-pop's $O(1)$ removal that does not.

```
1 // SPDX-License-Identifier: UNLICENSED
2 pragma solidity ^0.8.28;
3
4 import "./PrivacyFirstContract.sol";
```

```
import "./inc/config/Config.sol";

contract PrivacyFirstContractFifo is PrivacyFirstContract {

constructor(
  string memory ownerPubKey,
  Config memory config,
  address dstContractAddress
  ) PrivacyFirstContract(ownerPubKey, config, dstContractAddress) {}

  function _addToPending(uint256 dataId) internal override {
    _pendingDataIds.push(dataId);
  }

  function _removeFromPending(uint256 dataId) internal override {
    uint256 len = _pendingDataIds.length;
    uint256 idx = len; // sentinel: not found
    for (uint256 i = 0; i < len; i++) {
      if (_pendingDataIds[i] == dataId) {
        idx = i;break;
      }}
    require(idx < len, "Data not in pending");
    for (uint256 i = idx; i + 1 < len; i++) {
      _pendingDataIds[i] = _pendingDataIds[i + 1];
    }
    _pendingDataIds.pop();
  }
}
```

Listing C.1: Full source of `PrivacyFirstContractFifo`.