4-Party Asynchronous Communication with Perfect Secrecy based on Split-Half d-Gap Protocol

---

CS 6903 Applied Cryptography

Spring 2026

Giovanni Di Crescenzo

Yang Zheng

Xuan Zhang

Introduction

Task Distribution

Task	Team Members
Protocol idea & design	Yang Zheng, Xuan Zhang
Protocol Code	Yang Zheng, Xuan Zhang
Testing Code	Yang Zheng
Project Report	Xuan Zhang, Yang Zheng

Problem Statement

Design a multi-party one-time pad communication protocol for asynchronous networks that maintains perfect secrecy (no pad is used twice), minimizes wasted pads, and achieves efficient runtime per message. In this project, we use m = 4 parties.

Main Result

a. Number of waste pads

To test our solution, we measured the average number of wasted pads in different scenarios (S.x), which means only x, randomly chosen, parties repeatedly send L-length messages, where the decision of who sends the next message is also randomly chosen. In our case, x = 1, 2, 4.

Parameters: n=2000, d=20, num_executions=200

Scenario	Avg wasted pads
S.1	1020
S.2	403.75
S.4	72.31

Maximum number of wasted pads across all Scenarios: 1020

b. Runtime to send a single message (exclude time for encryption)

Our send operation performs a constant number of integer operations and state updates, so runtime per message is O(1).

Informal Explanation

We split the pad sequence of length n into two equal halves. Parties (0,1) share the first half, and parties (2,3) share the second half. Inside each half, we run a two-party bidirectional d-gap protocol: one party consumes pads from the left, the other from the right, and they must maintain a gap of at least d pads between their positions. If using the next pad would violate the gap, the party stops.

General idea is showed below:

Pad Sequence (length n)

First Half [0, n/2)	Second Half [n/2, n)
P0 → ← P1	P2 → ← P3
← *gap ≥ d* →	← *gap ≥ d* →

0 n/2-1 n/2 n-1

Rigorous Description

Overview

- 4 parties share a pad sequence of length n

- Sequence is split into two halves: [0, n/2) and [n/2, n)

- Each half has 2 parties consumes pads from two ends

- Parties must maintain a gap of at least d pads between them

Pseudo Codes

```
ProtocolState:
      n ← total number of pad sequences
      d ← required gap size
      last_used ← a list stores each party's last used pad index,
                  initialize as [-1, n//2, n//2-1, n]

FourPartyProtocol(n, d):
      n ← n
      d ← d
```

```
    mid ← n // 2  #mid-index of pads
    state ← ProtocolState(n, d)
```

Get-Next-Pad(state, party_id): (Core algorithm)

```
    CASE party_id of:
      0: next ← state.last_used[0] + 1
         IF next ≥ state.last_used[1] - d THEN RETURN NULL
         RETURN next
      1: next ← state.last_used[1] - 1
         IF next ≤ state.last_used[0] + d THEN RETURN NULL
         RETURN next
      2: next ← last[2] + 1
         IF next ≥ state.last_used[3] - d THEN RETURN NULL
         RETURN next
      3: next ← last[3] - 1
         IF next ≤ state.last_used[2] + d THEN RETURN NULL
         RETURN next
    END CASE
```

SEND(party_id):

```
    pad ← Get-Next-Pad(state, party_id)
    state ← pad  #update state
    RETURN pad
```

GET_TOTAL_USED():

```
    total ← 0
    Check each value in state.last_used, if is not init value, update total
    RETURN total
```

GET_WASTED_PADS():

```
    RETURN n - total used
```

Proof

We split the pad sequence into two halves. In the first half, only parties (0,1) ever touch those pads, and they move inward from the two ends while keeping a safety gap of size d. Same thing independently in the second half for parties (2,3). So the two halves never interfere with each other. Inside each half it's basically the classic 2-party d-gap idea: one party increases, the other decreases, and we only allow a send if the next pad would still leave at least d pads between them. That means the "left pointer" and "right pointer" never cross, so they can't ever pick the same pad. Since halves are disjoint, no one can collide across halves either. So no pad is reused and perfect secrecy holds.

Look at one half (say pads 0..mid-1). When the protocol stops in that half, it's because the next pad someone wants would violate the d-gap. That can only happen when the remaining unused region between the two pointers is ≤ d. So the number of pads "left on the table" inside an active half is at most d. So if a half is active (someone in it sends at least once), it wastes ≤ d pads. If a half is inactive (nobody in it ever sends), it wastes basically the whole half: about n/2 pads.

S.1 (only 1 party sends): only one half is active, the other half is untouched. So wasted pads ≈ n/2 + d.

S.2 (2 parties send): if they're in different halves, both halves are active → wasted ≤ 2d. if they're in the same half, the other half is unused → wasted ≈ n/2 + d.

S.4 (all 4 parties send): both halves are definitely active → wasted ≤ 2d.

So compared to the split into 4 parts baseline (which can waste ~3n/4), our design is usually much better, especially when d << n and both halves get used. (For the cases that 3 parties send, it can be treated as one half shared by 2 parties and another half shared by one. So the same result as S.2.)

Generalization

When m = 3

For m=3, still split the pad into two halves. Parties (0,1) share the first half, and parties 2 uses the second half alone. When sending a new message, Party 0 and Party 1 check their gap distance, and Party 2 check the gap distance between state.last_used[2] and mid = n//2 (start index of party 1). For the worst case scenario that only one party is using the pad, total waste pad would be O(n/2 + d). When d << n, (n/2 + d) < (2/3)n.

When m < n/d

Let k = floor($\log_2$(m-1)), and divide the pad into 2^k equal parts. Now we have Party[0, 1, 2, ..., m-1] and pad_parts[0, 1, 2, ..., 2^k-1]. Assign each Party[i] to pad_parts[i mod 2^k]. Note that this method guarantees that each part of the pad is shared by at most two parties. Please check illustration.pdf to have a visualized interpretation of the general idea.

It's not hard to observe that the worst case waste pad is ((2^k-1)/2^k)*n+d, and this is strictly less than ((m-1)/m)*n when d << n, because 2^k is strictly less than m.

ReadME (included in README.md)

4-Party Asynchronous Communication with Perfect Secrecy

This is a Multi-party one-time pad protocol for asynchronous broadcast, using a split-half d-gap design.

Requirements

- Python 3.10+

No external dependencies. Standard library only.

Quick Demo for Running the Protocol

```
python protocol.py
```

Running the Testing program

```
python testing.py
```

With custom parameters you can choose:

```
python testing.py -n 2000 -d 20 -e 200 --seed 188
```

Options

-n : Pad sequence length (default: 1000)

-d : Gap size (default: 10)

-e , --executions : Number of protocol runs per scenario (default: 100)

--seed : Random seed (default: 42)

Scenarios

S.1: 1 random party sends

S.2: 2 random parties send

S.4: All 4 parties send

Each protocol run until at least one party cannot send. Output includes average wasted pads, max and min wasted pads etc.