

Deutsche Telekom Chair of Communication Networks
Technische Universität Dresden

Practical Implementations of Network Coding

Frank Fitzek // Summer Semester 2018

Random Linear Network Coding

Coding History

Evolution of Coding

Block Codes

1950s

- Free
- Proprietary close to patent expiry
- Proprietary with long patent life

Convolutional Codes

1960s

Modern Codes

- LDPCs – patent expired
- Turbo Codes – patents expired or expiring

1990s

1998

2003

2014

Rateless Codes

Raptor and related codes

- Rate-less (refinement to free E2E)
- Still E2E, still static



Fulcrum Codes codeon

- RLNC-enabled **steinwurf**
 - Fluid complexity (flexible field size)
 - Breaks performance-overhead trade-off

The Technology: RLNC

At the heart of many communications problems is a collectors' problem.

Traditional Approach

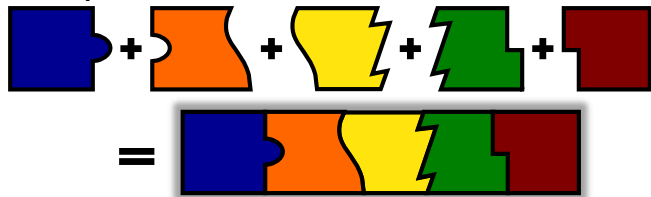
- Data broken into pieces



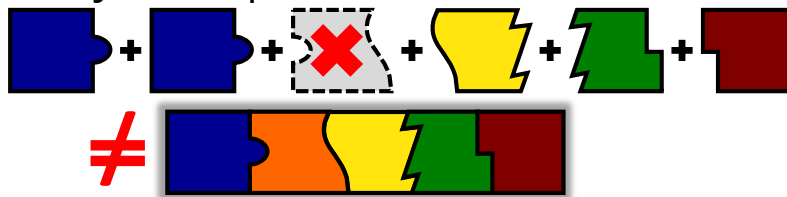
- k-piece data set \rightarrow k pieces



- All pieces needed

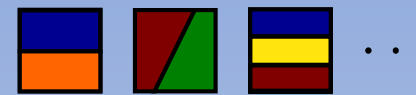


- Only these pieces will do

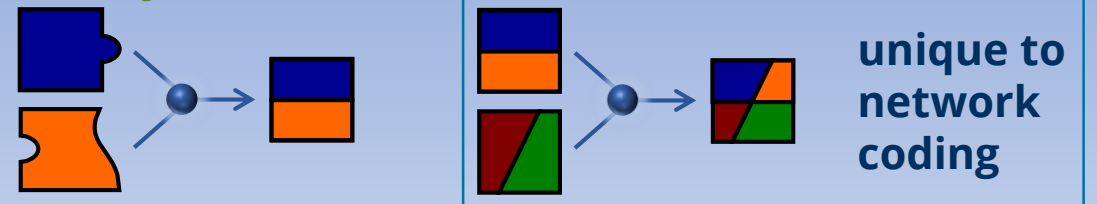


RLNC

- Mixtures created from pieces



- Any node can create mixtures



- Many mixtures possible

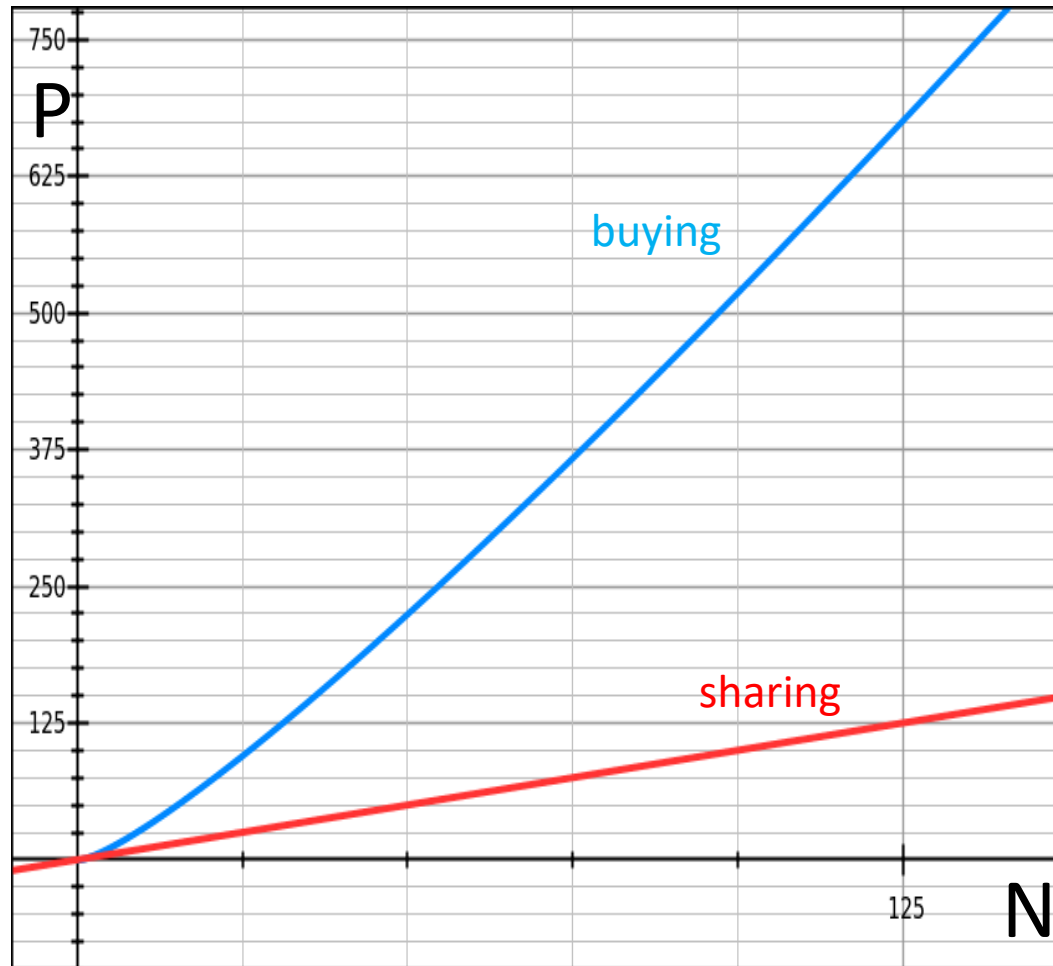


- Any k mixtures will do



Coupon Collector's Problem

$$P = N * (\ln(N) + 0.577)$$



Random Linear Network Coding

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} & \alpha_{1,5} & \alpha_{1,6} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} & \alpha_{2,5} & \alpha_{2,6} \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} & \alpha_{3,5} & \alpha_{3,6} \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ \alpha_{5,1} & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Original packets

Gaussian elimination $n \times n$ matrix requires
 $An^3 + Bn^2 + Cn$ operations.

Random Linear Network Coding

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{matrix} \text{coding} \\ \text{coefficients} \end{matrix} \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} & \alpha_{1,5} & \alpha_{1,6} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} & \alpha_{2,5} & \alpha_{2,6} \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} & \alpha_{3,5} & \alpha_{3,6} \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ \alpha_{5,1} & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Gaussian elimination $n \times n$ matrix requires
 $An^3 + Bn^2 + Cn$ operations.

Random Linear Network Coding

coded packets

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} & \alpha_{1,5} & \alpha_{1,6} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} & \alpha_{2,5} & \alpha_{2,6} \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} & \alpha_{3,5} & \alpha_{3,6} \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ \alpha_{5,1} & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

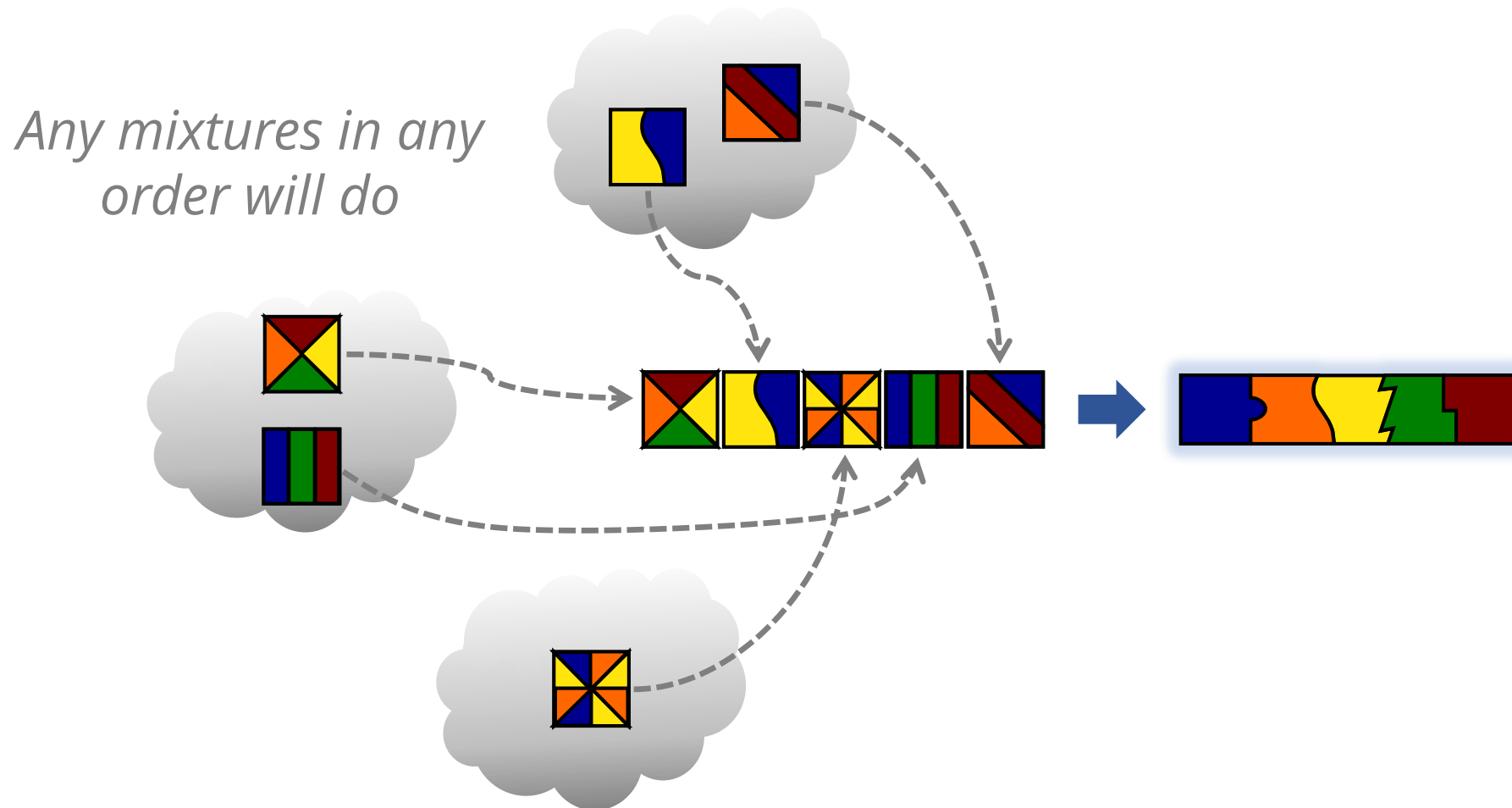
Gaussian elimination $n \times n$ matrix requires
 $An^3 + Bn^2 + Cn$ operations.

Random Linear Network Coding

$$\begin{pmatrix} C_1 \\ \vdots \\ C_G \\ C_{G+1} \\ \vdots \\ C_K \end{pmatrix} = \begin{pmatrix} \alpha_{1,1} & \cdots & \alpha_{1,G} \\ \vdots & \ddots & \vdots \\ \alpha_{G,1} & \cdots & \alpha_{G,G} \\ \alpha_{G+1,1} & \cdots & \alpha_{G+1,G} \\ \vdots & \ddots & \vdots \\ \alpha_{K,1} & \cdots & \alpha_{K,G} \end{pmatrix} \begin{pmatrix} P_1 \\ \vdots \\ P_G \end{pmatrix}$$

Rateless code: can output any number of coded packets.
(such as Fountain codes, but better than RS)

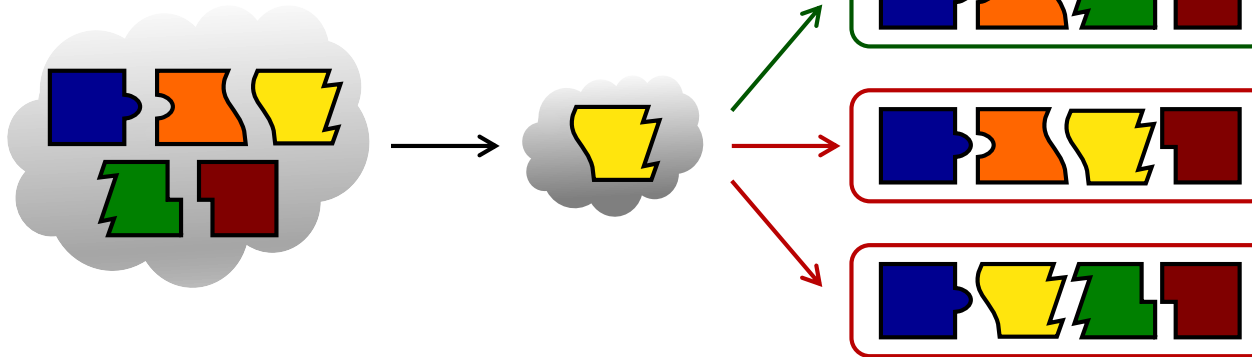
Multipath - Multicloud



- RLNC Enables “Stateless Communications”

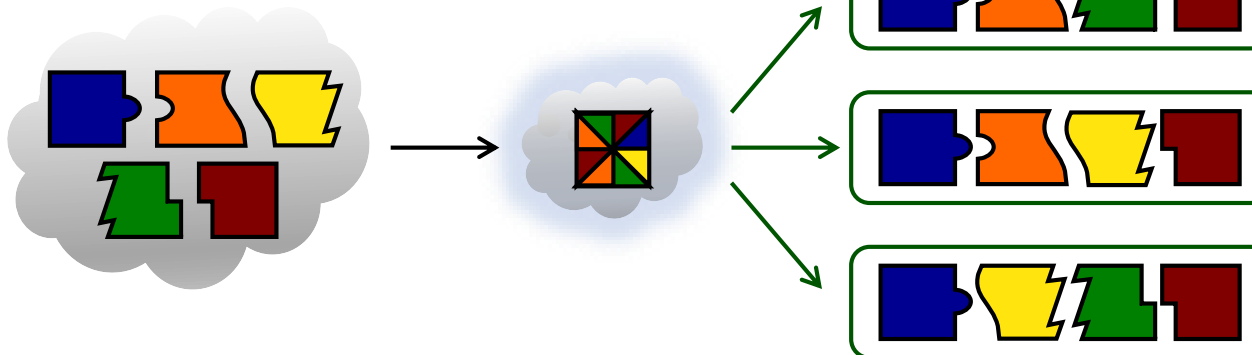
Coded Edge Caching

Traditional



- One small cache can't satisfy everyone
- Only the device missing the piece in the edge cache can reconstruct

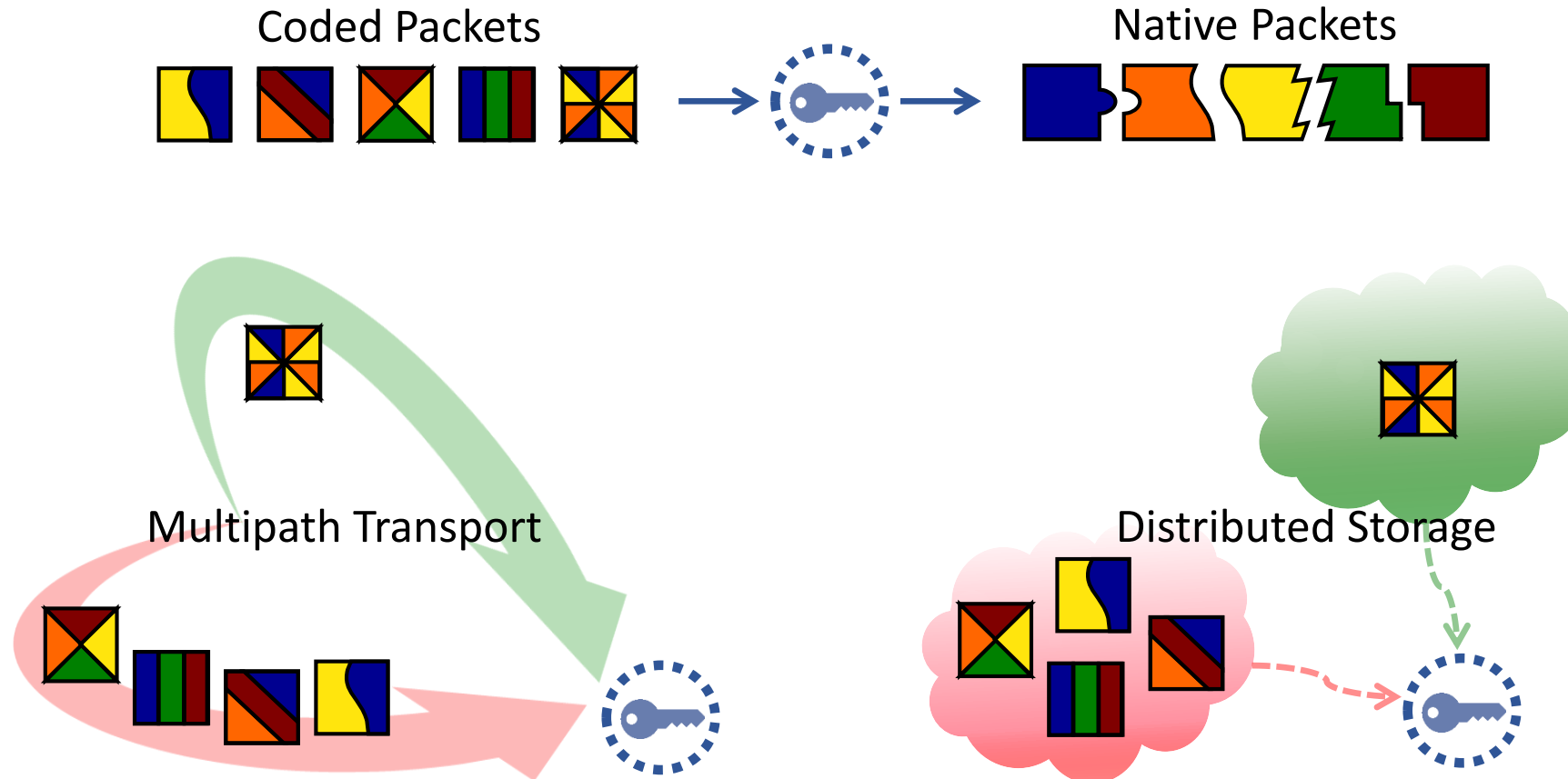
RLNC



- Mixtures enable all nodes to reconstruct
- An example of RLNC's enhancement of non-coded systems

- A Little RLNC Can Go a Long Way

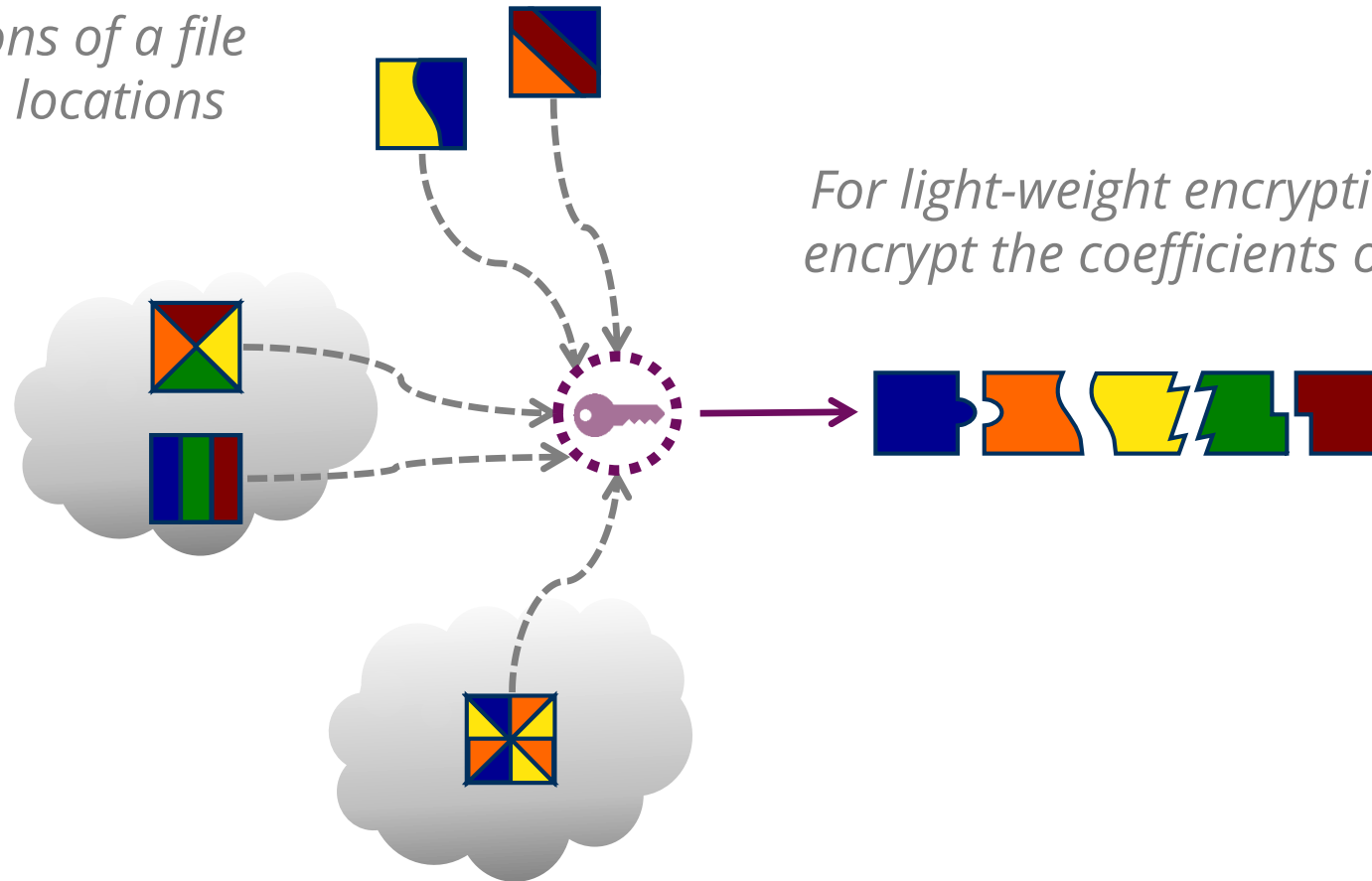
Coding as an Additional Security Measure



- Data on a given path/cloud acts as a cypher

RLNC Improves Cloud Security

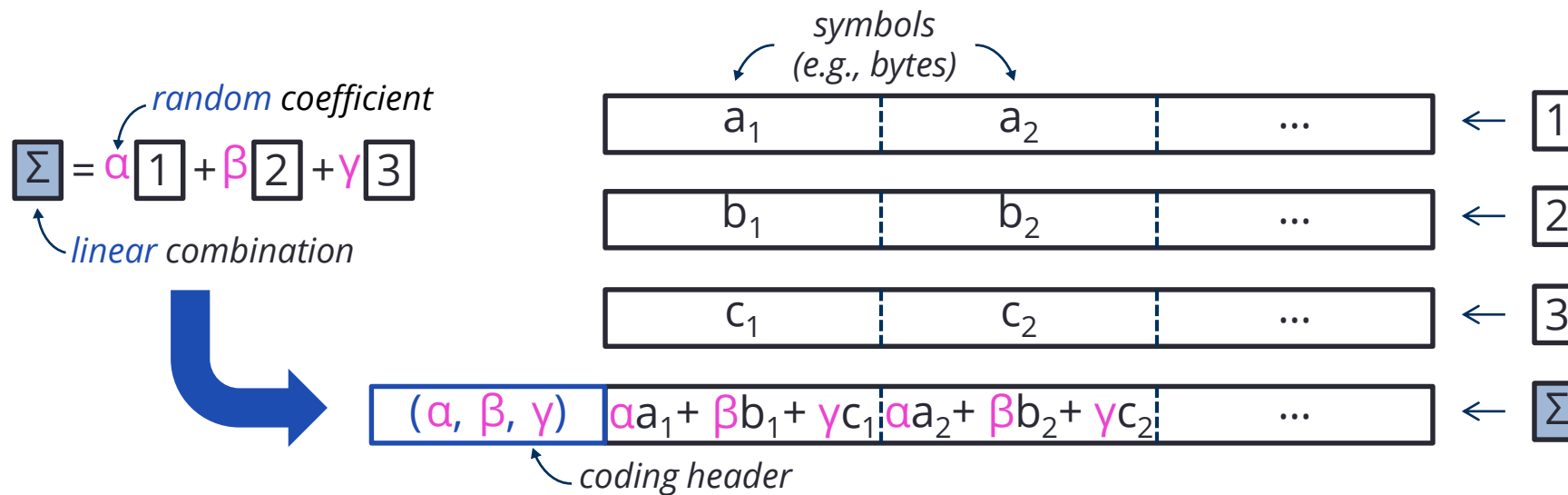
*Store portions of a file
in multiple locations*



*For light-weight encryption:
encrypt the coefficients only*

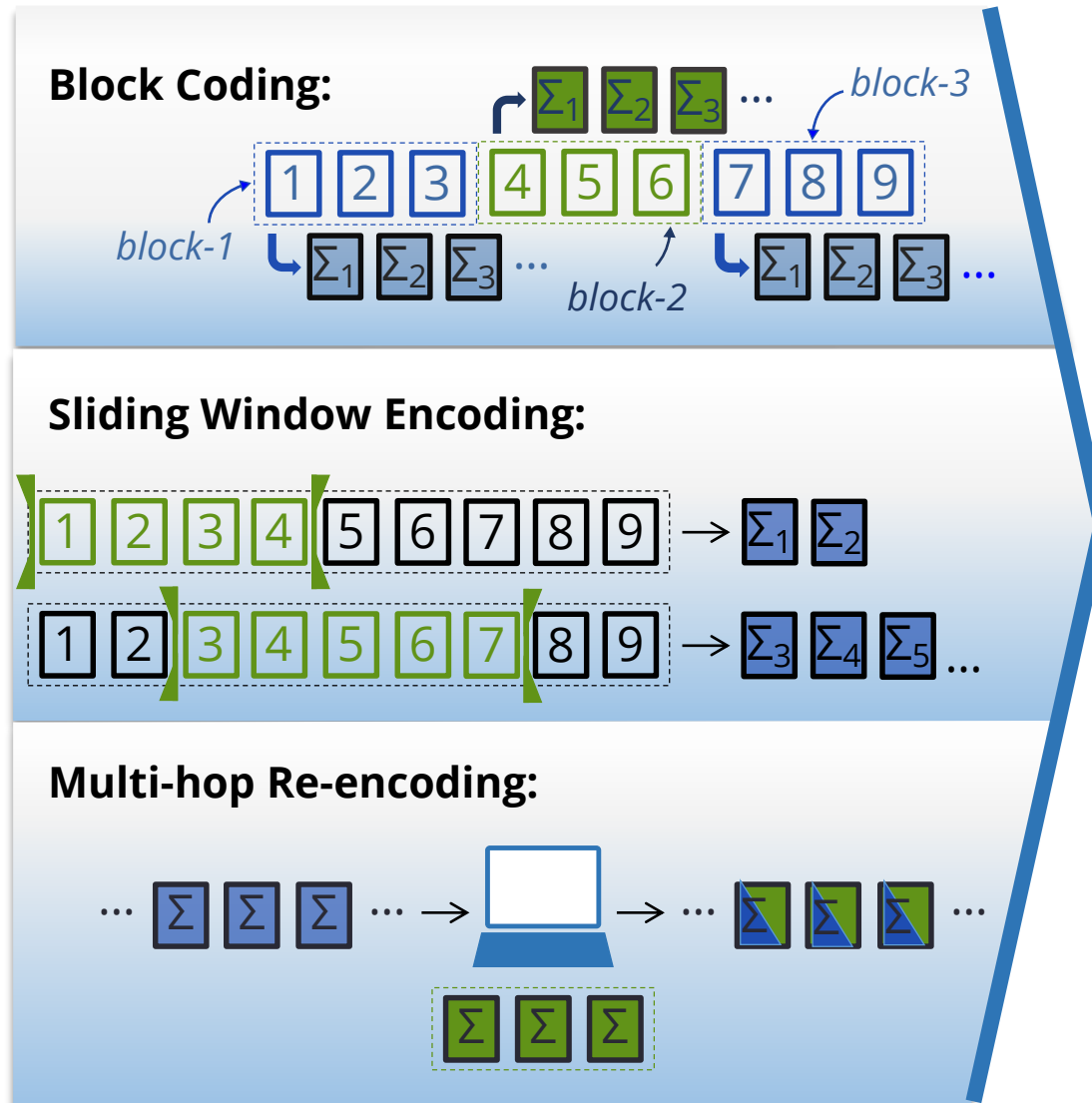
Mixture equations "unlock" data from mixtures

How does RLNC work?



- Random Generation of Coefficients
- Code Embedded within Data
- No State Tracking
- **Versatile** Code

RLNC: The Technology



decoding scheme
(simple equation-solving)

$$1 = \alpha'_1 \Sigma_1 + \beta'_1 \Sigma_2 + \gamma'_1 \Sigma_3$$

$$2 = \alpha'_2 \Sigma_1 + \beta'_2 \Sigma_2 + \gamma'_2 \Sigma_3$$

$$3 = 1 + \beta'_3 \Sigma_2 + \gamma'_3 \Sigma_3$$

obtained through
Gaussian Elimination

Can decode using both
encoded and un-encoded packets

RLNC: The Technology

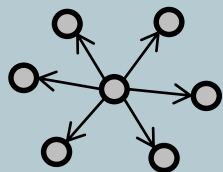
Coding Today

(all End-to-End)

Classical



Multicast



Coding Tomorrow with RLNC

Classical + Sliding Window Encoding



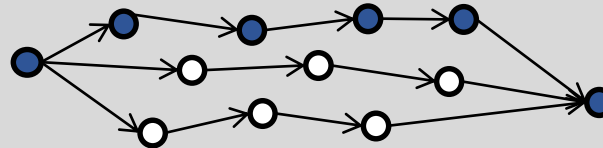
Real time video streaming,
TCP, SDN...

Multihop



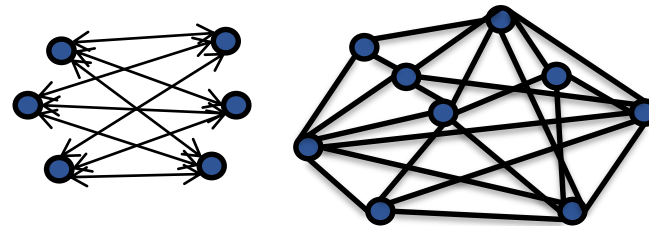
Edge caches, wireless mesh,
reliable multicast, satellites,
small relay topologies, SDN...

Multipath



Multi-source streaming
Multipath TCP, channel
bundling, heterogeneous
network combining, SDN...

Multisource – Multi-destination / Mesh



Distributed cloud, SDN,
advanced mesh (IoT, car2car,
M2M, smart grid) ...

Key Parameters of RLNC

- **Generation size:** number of packets that are (currently) coded together.
- **Field size:** number of elements in the finite field

- Both have an impact on:
 - Performance
 - Complexity

Field Size

Paket 1

1	0	1	1	0	0	1	0	1	0	1	0	1	1	0	0	1	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Paket 2

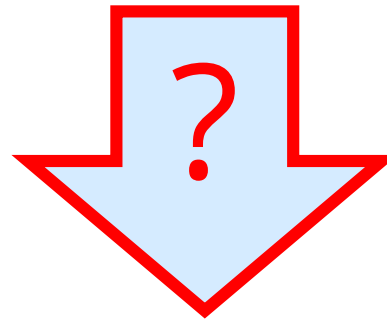
0	0	0	1	1	1	1	0	1	0	1	1	0	0	1	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Field Size

Paket 1

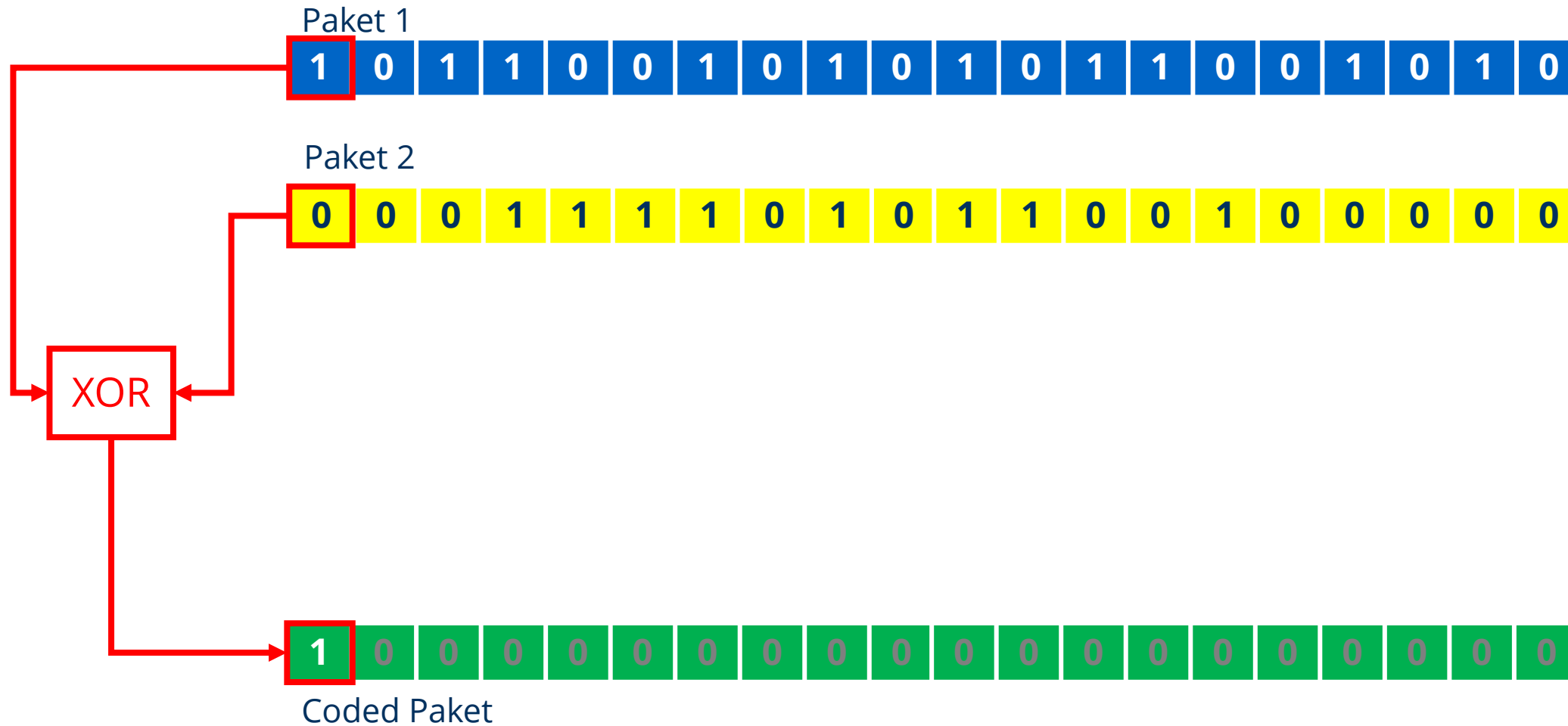


Paket 2

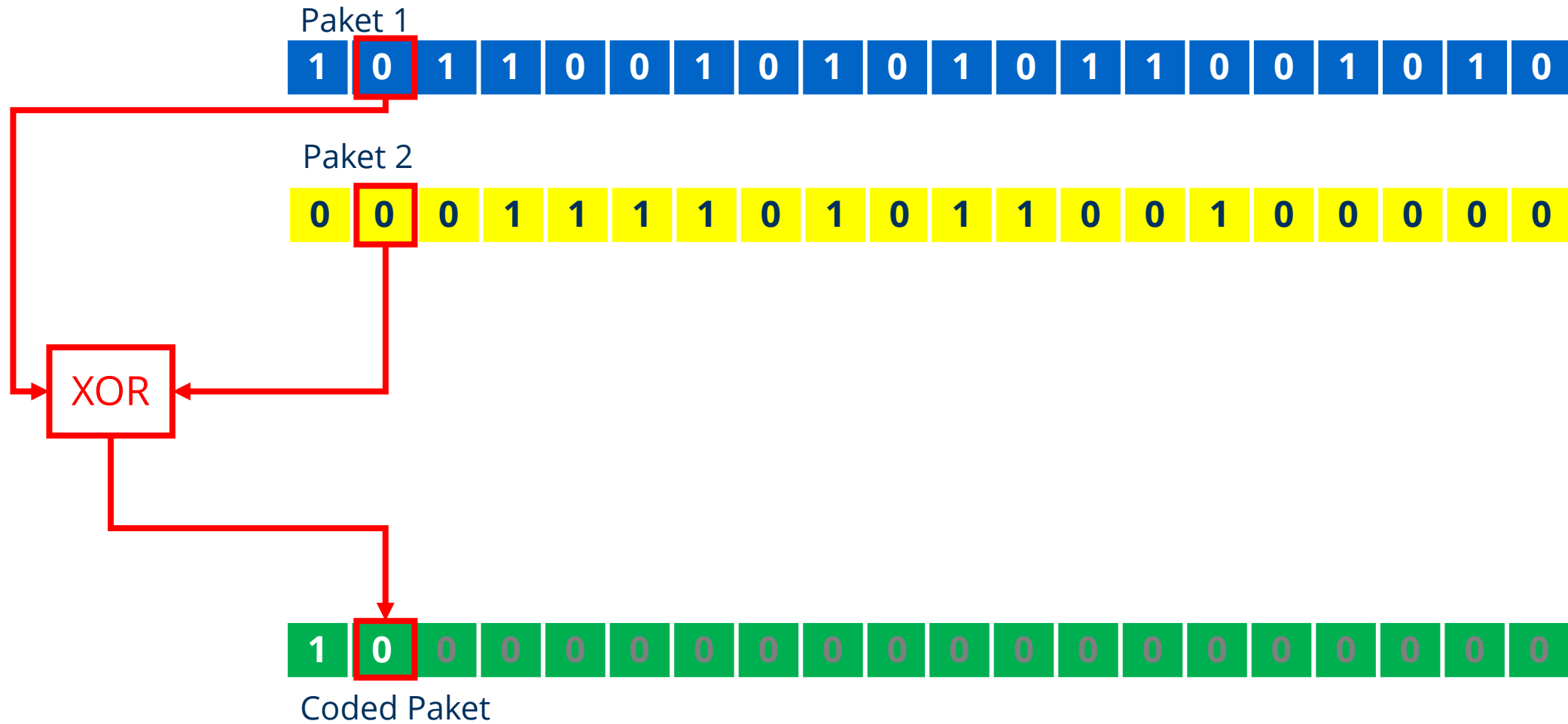


Coded Paket (initially empty)

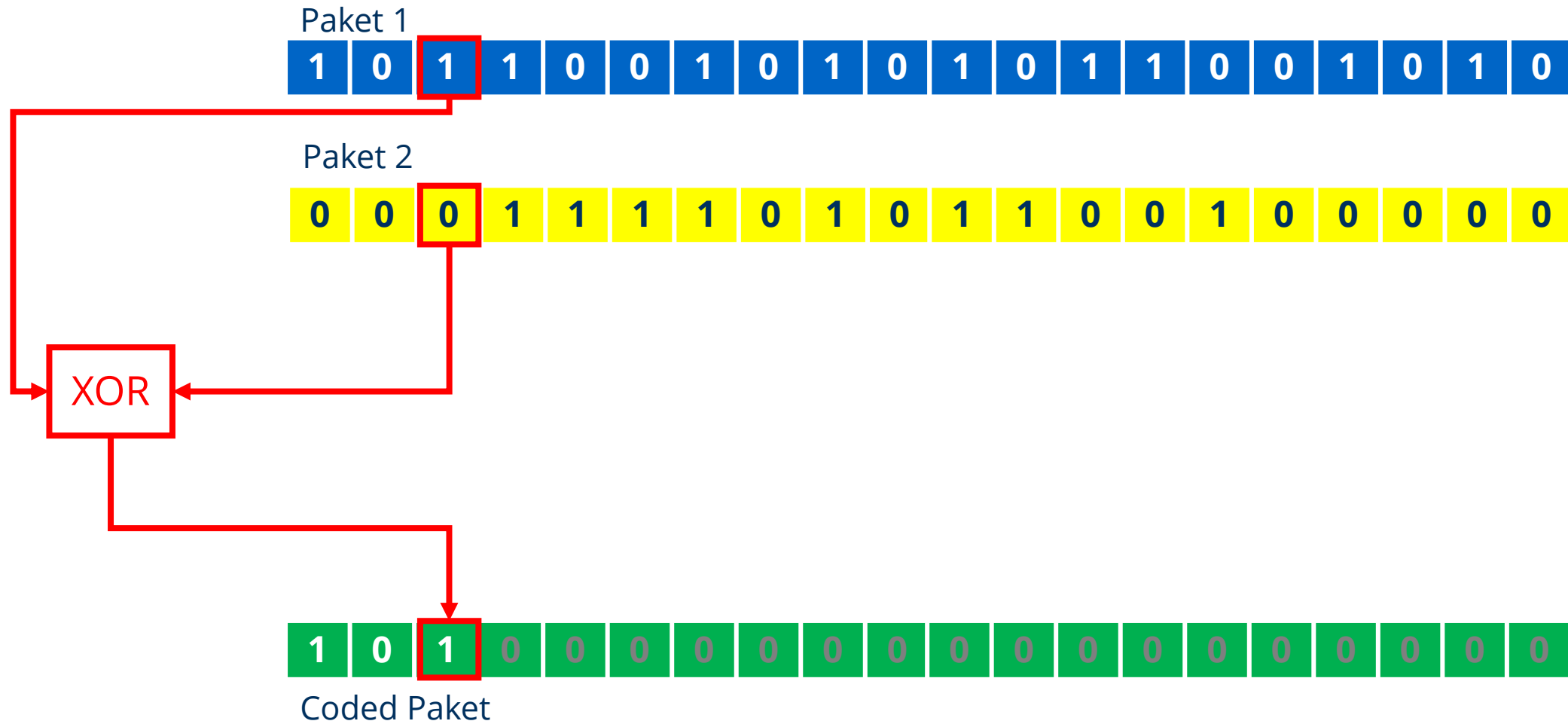
Field Size GF(2)



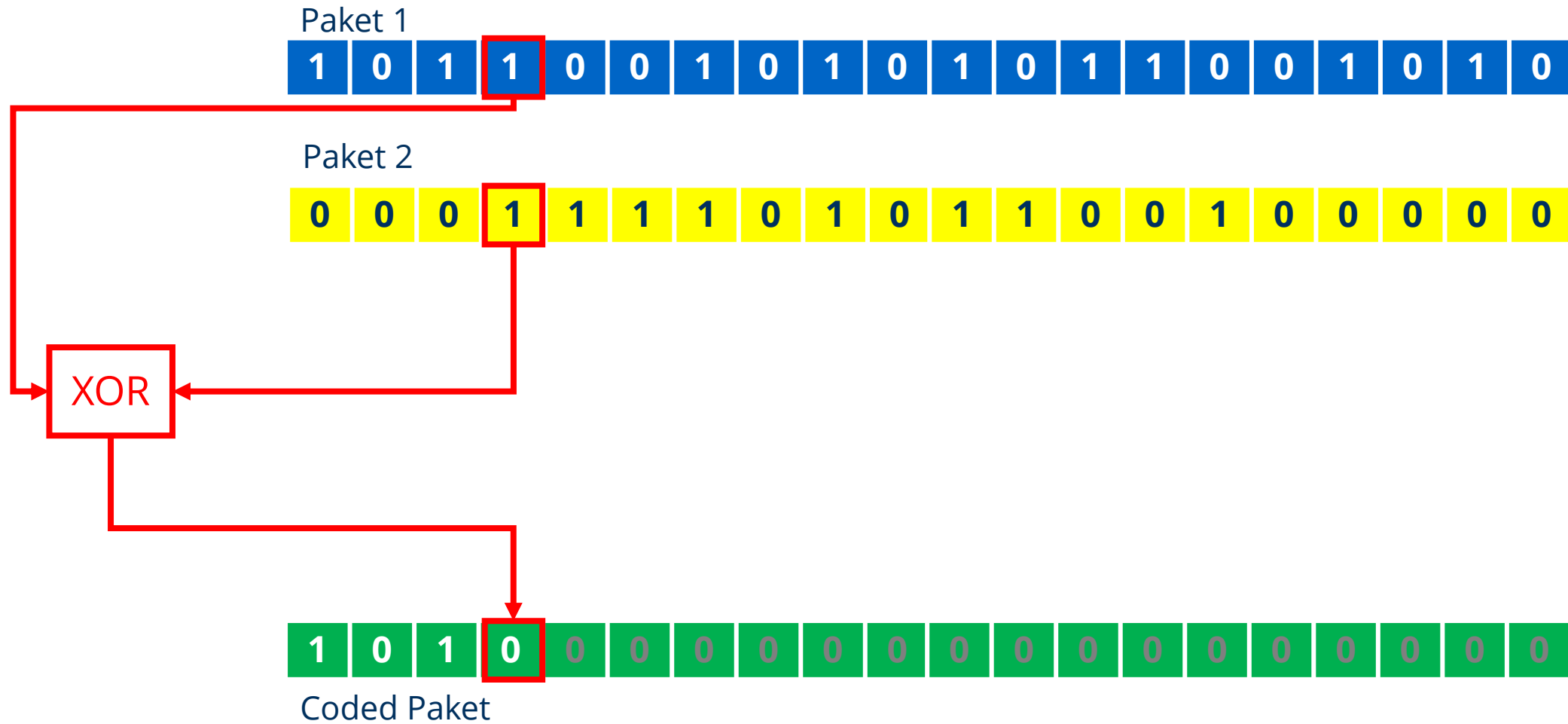
Field Size GF(2)



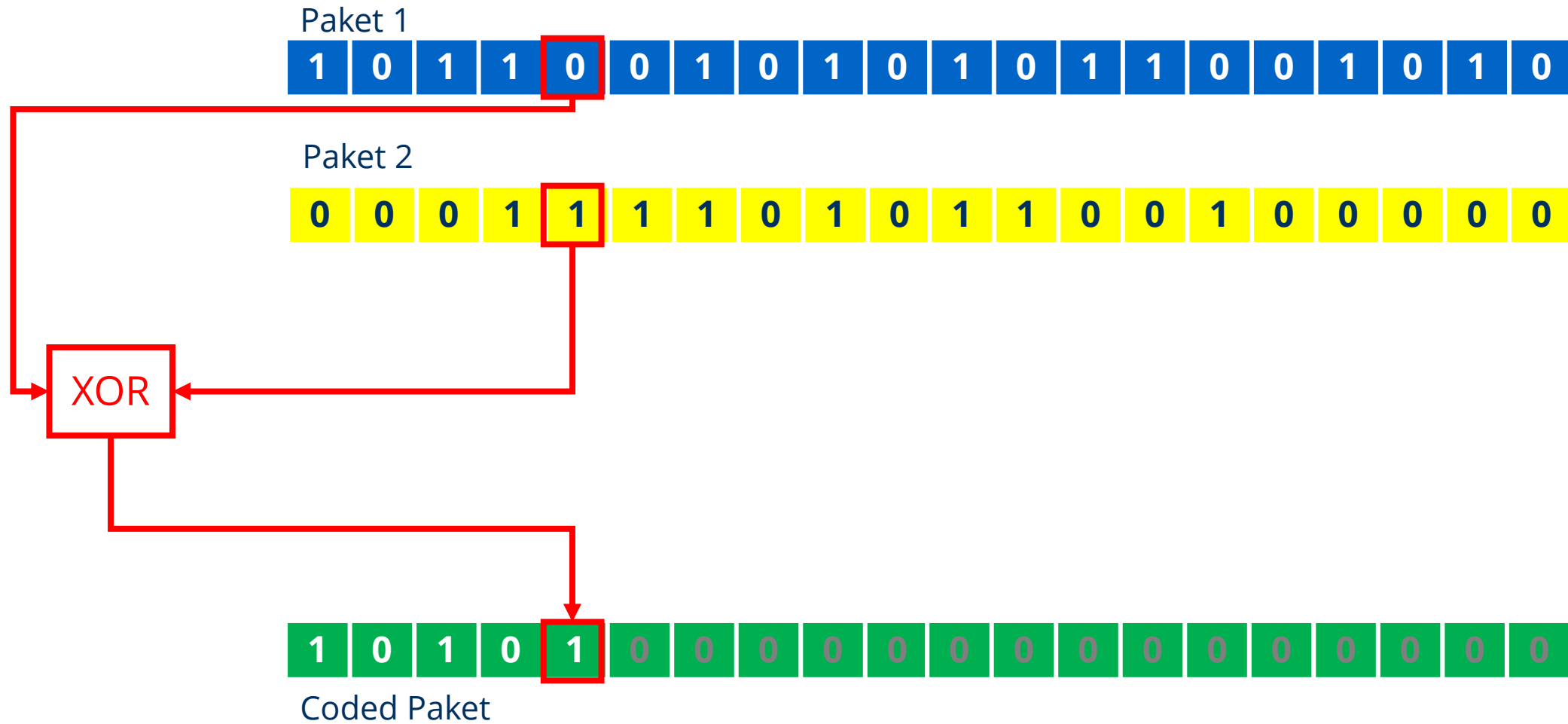
Field Size GF(2)



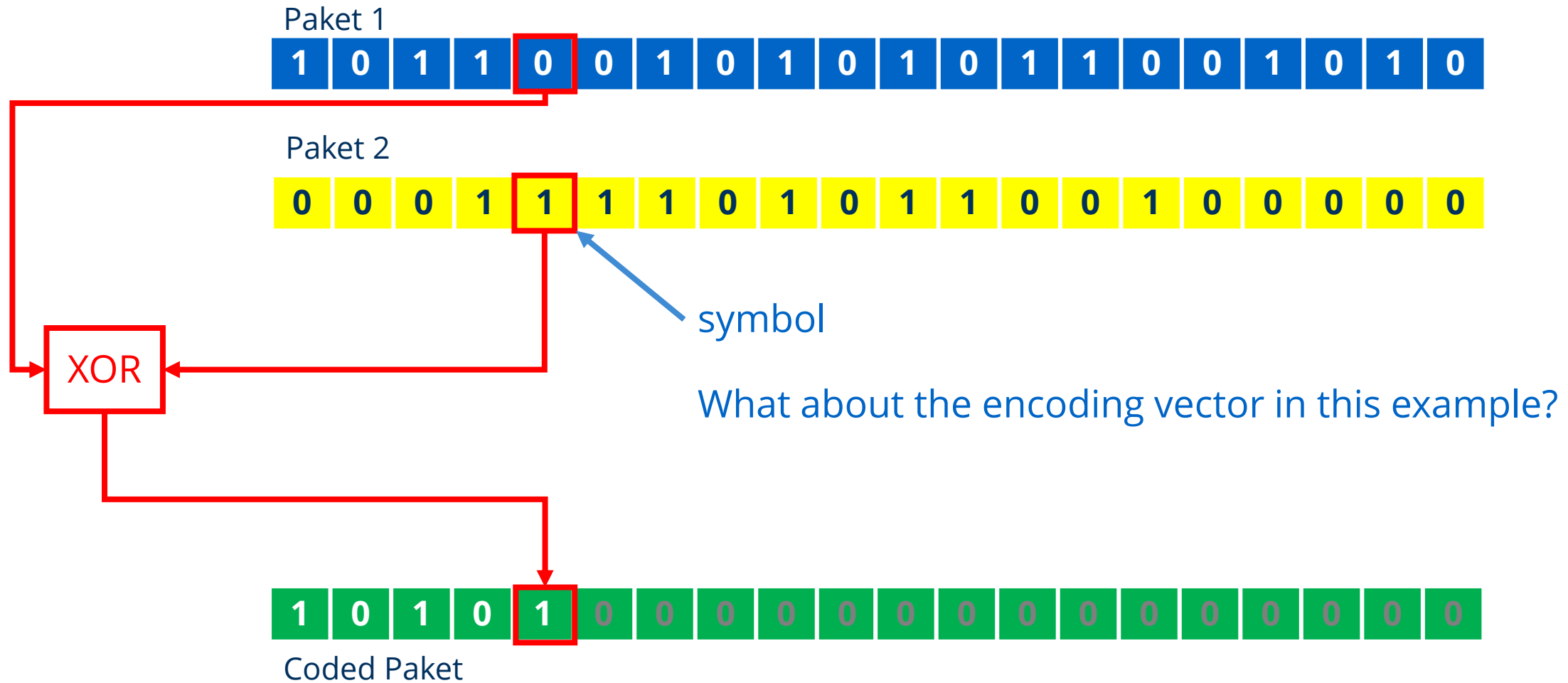
Field Size GF(2)



Field Size GF(2)

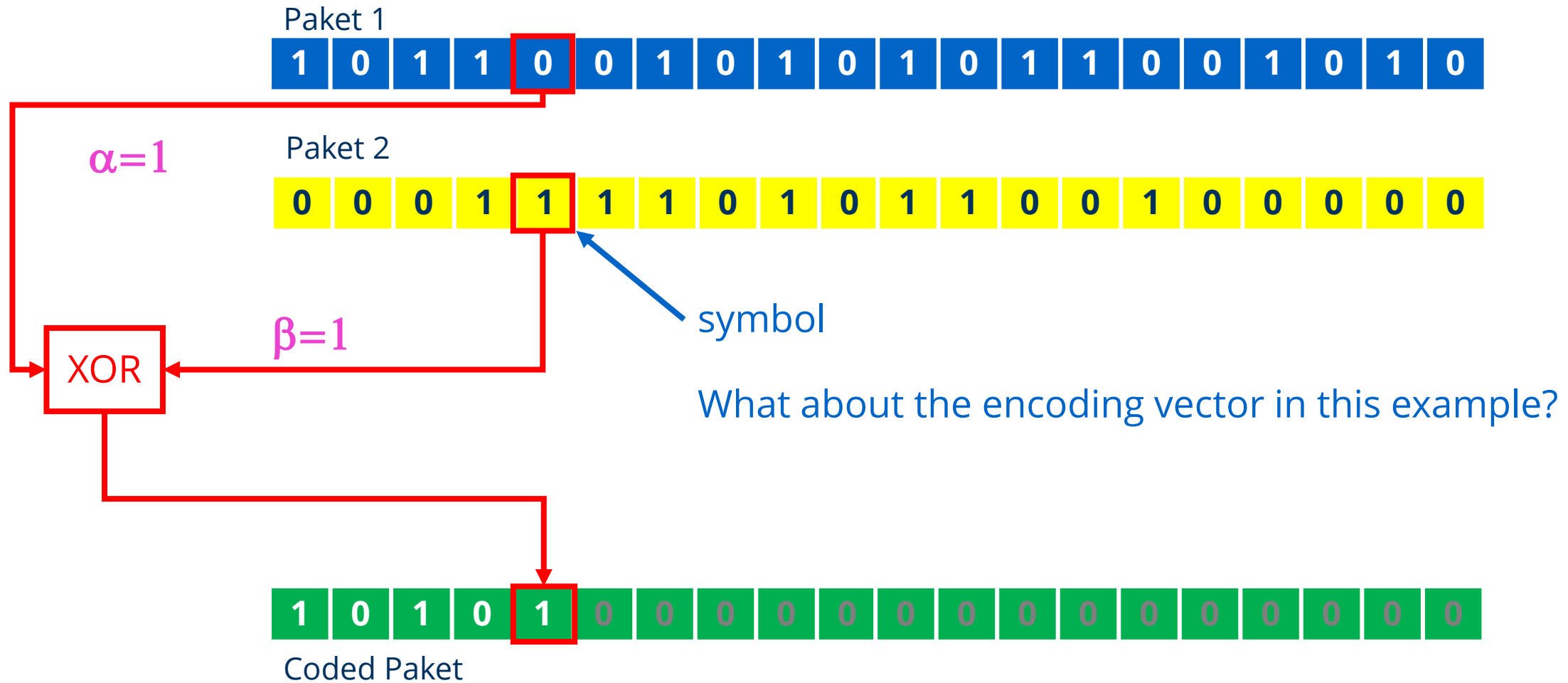


Field Size GF(2)



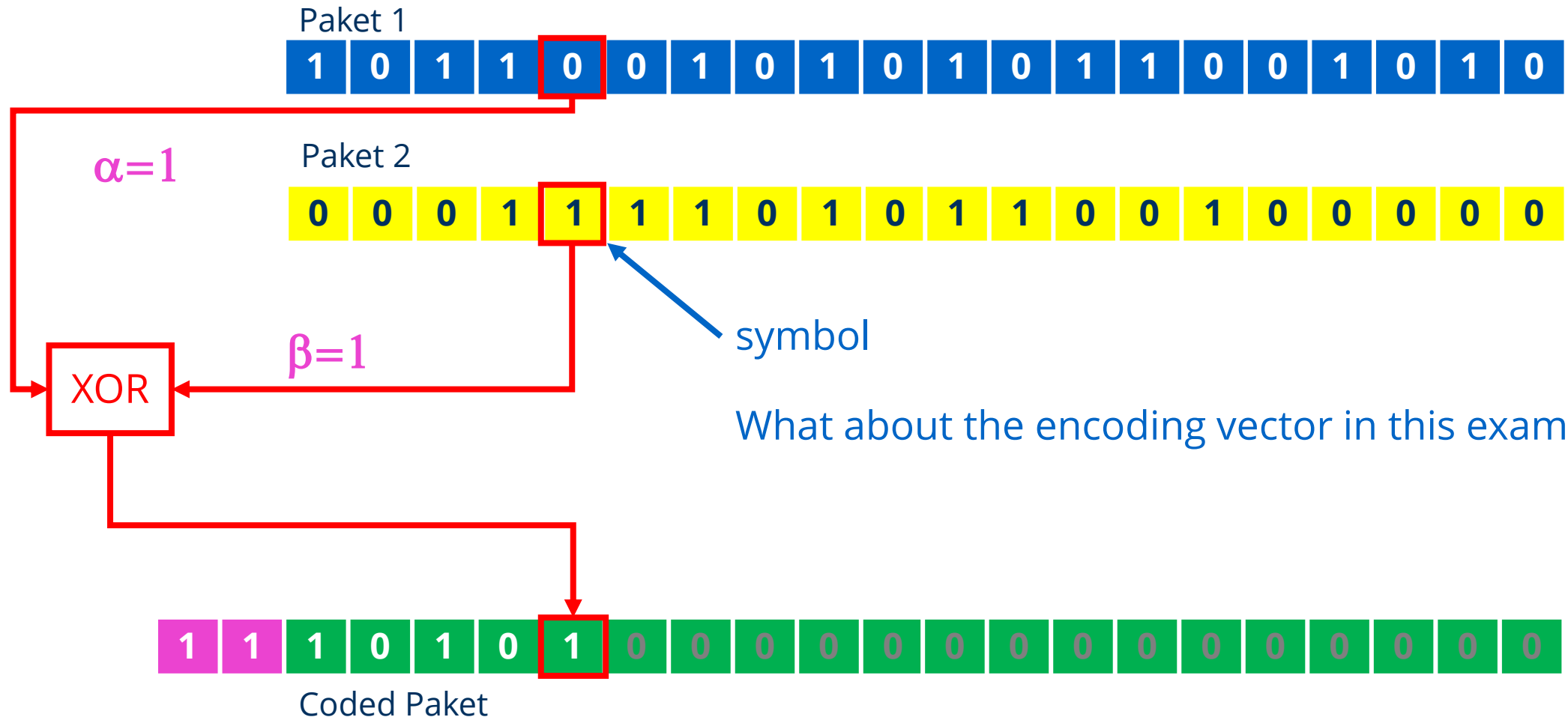


Field Size GF(2)

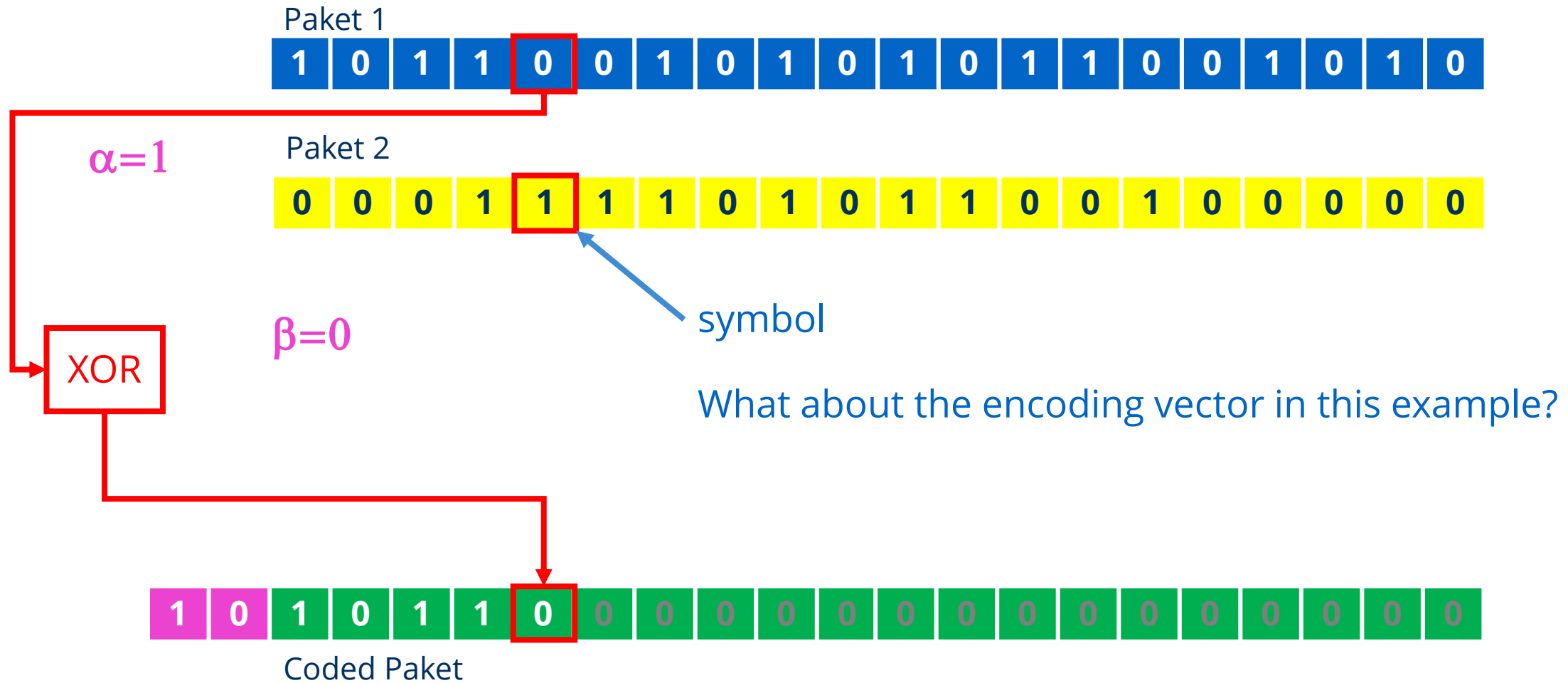




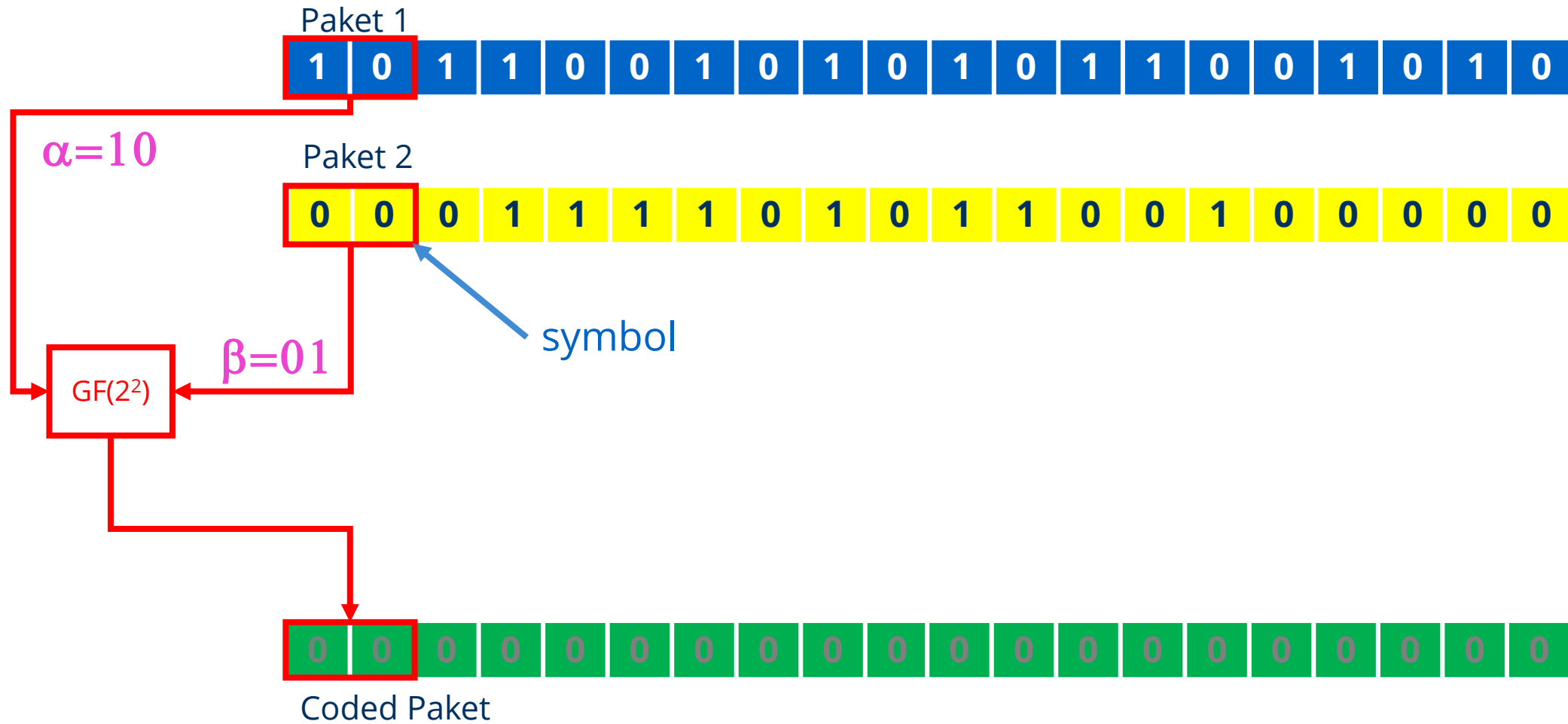
Field Size GF(2)



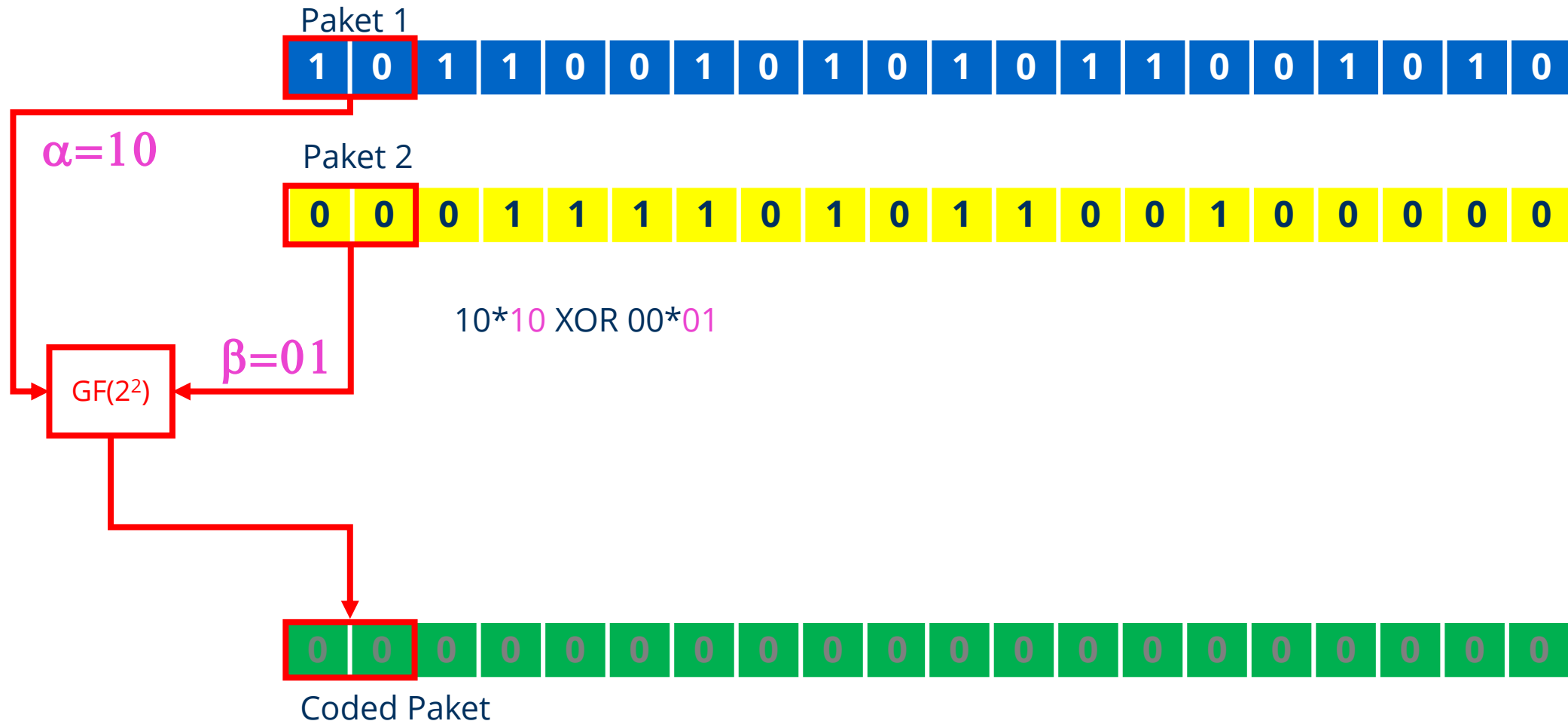
Field Size GF(2)



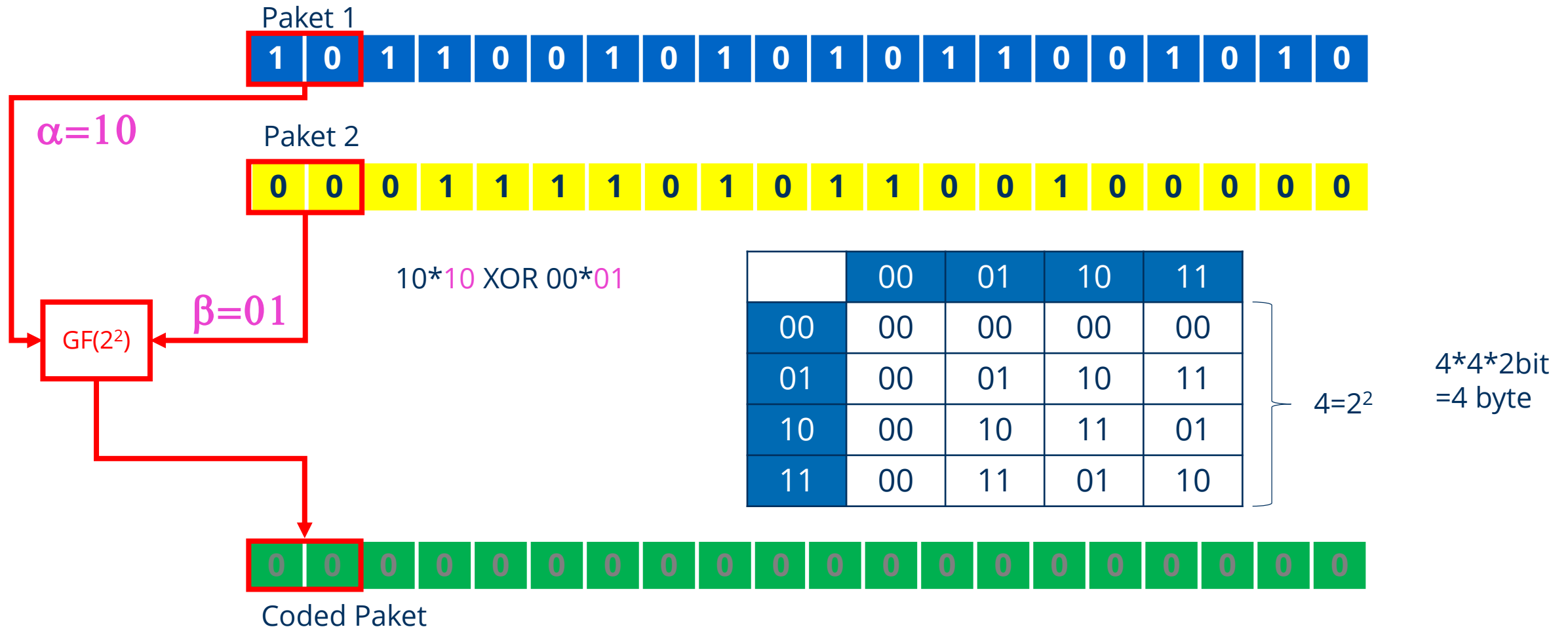
Field Size $GF(2^2)$



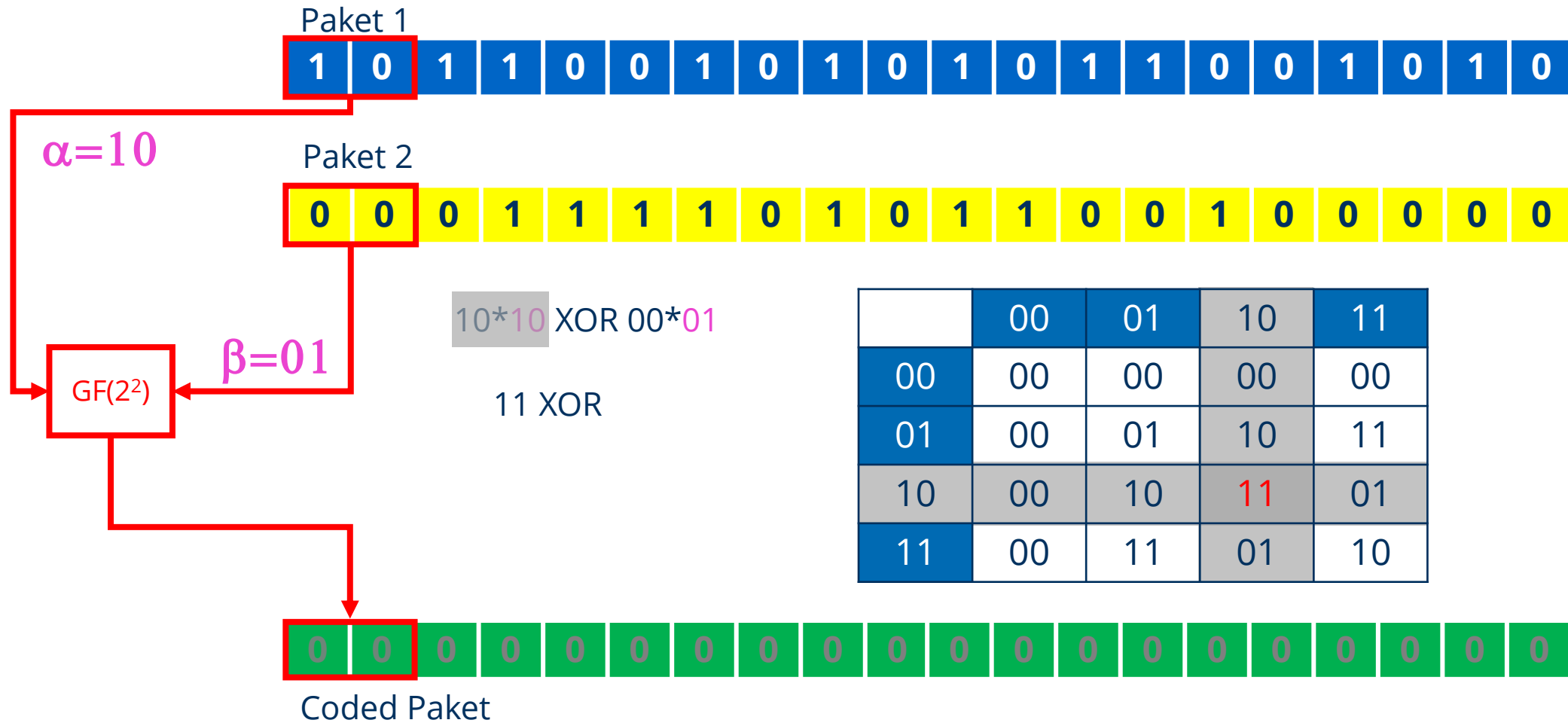
Field Size $GF(2^2)$



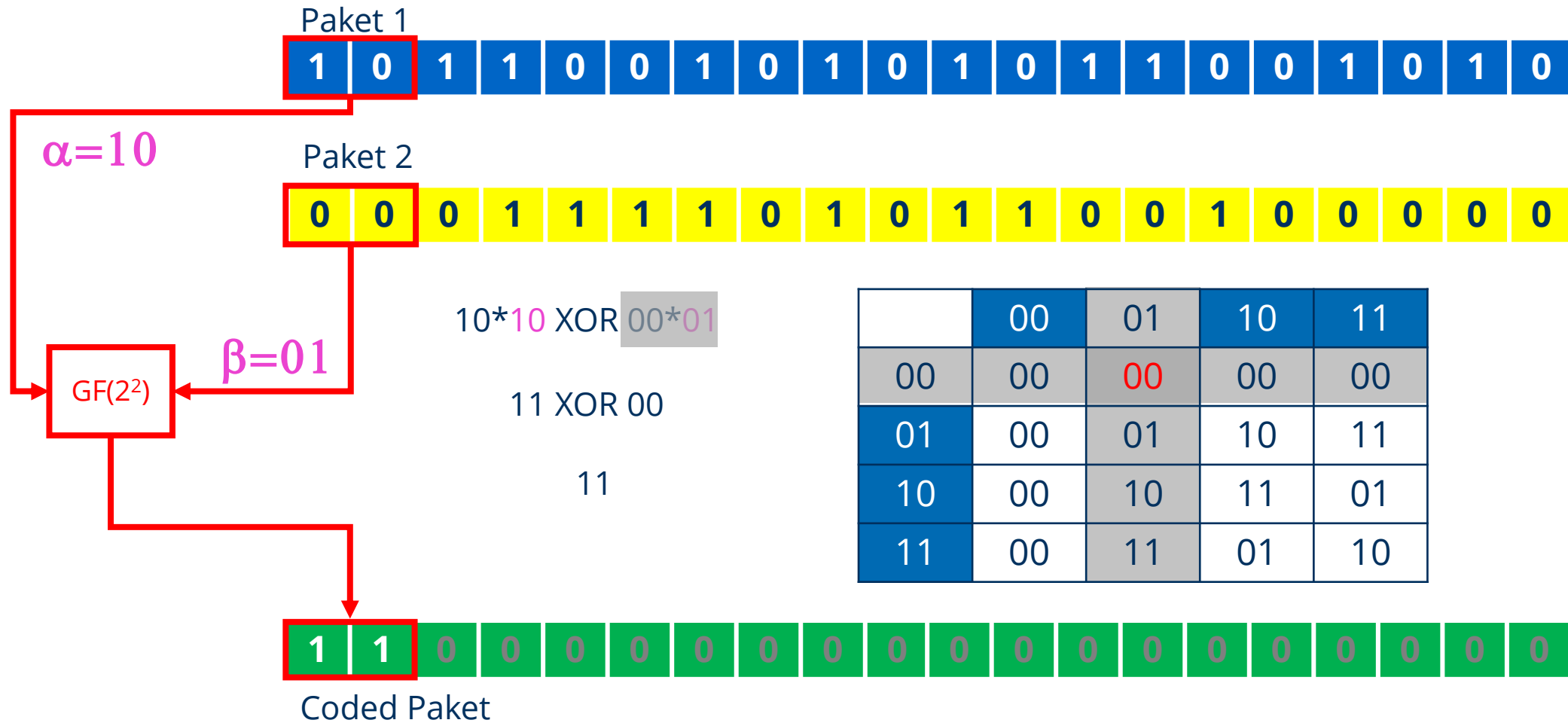
Field Size GF(2²)



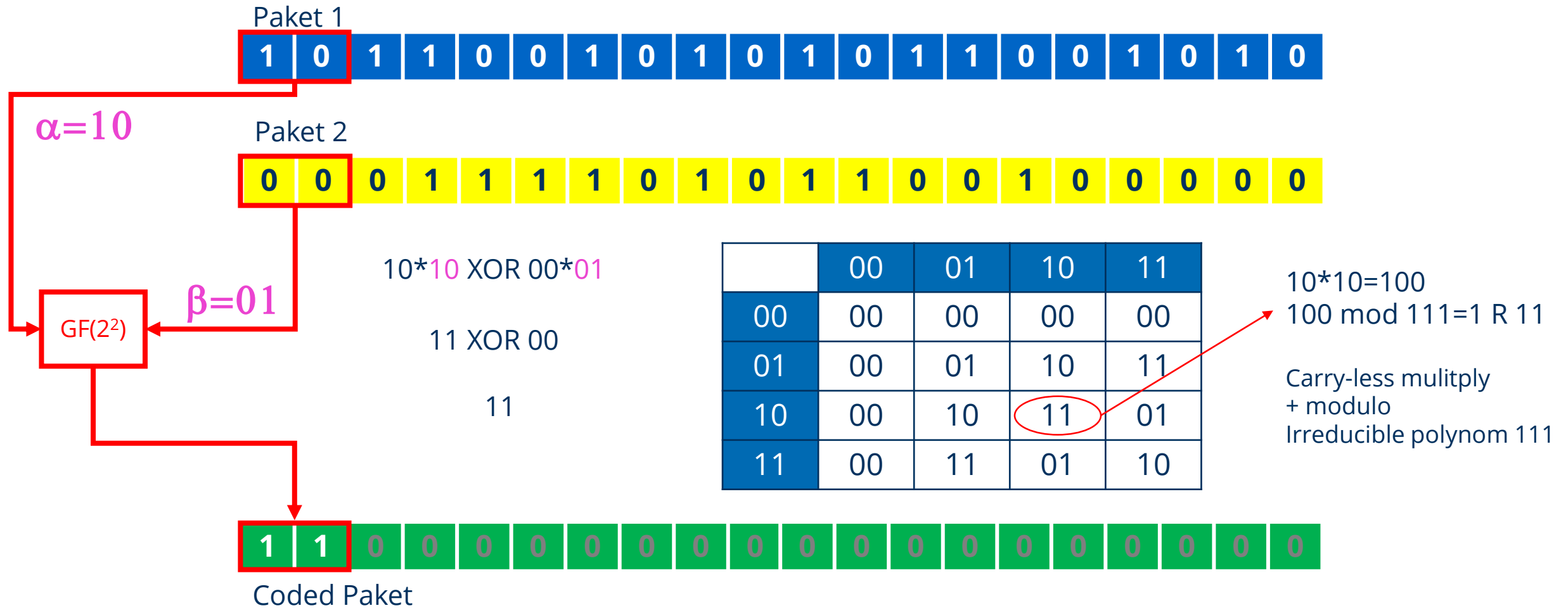
Field Size GF(2²)



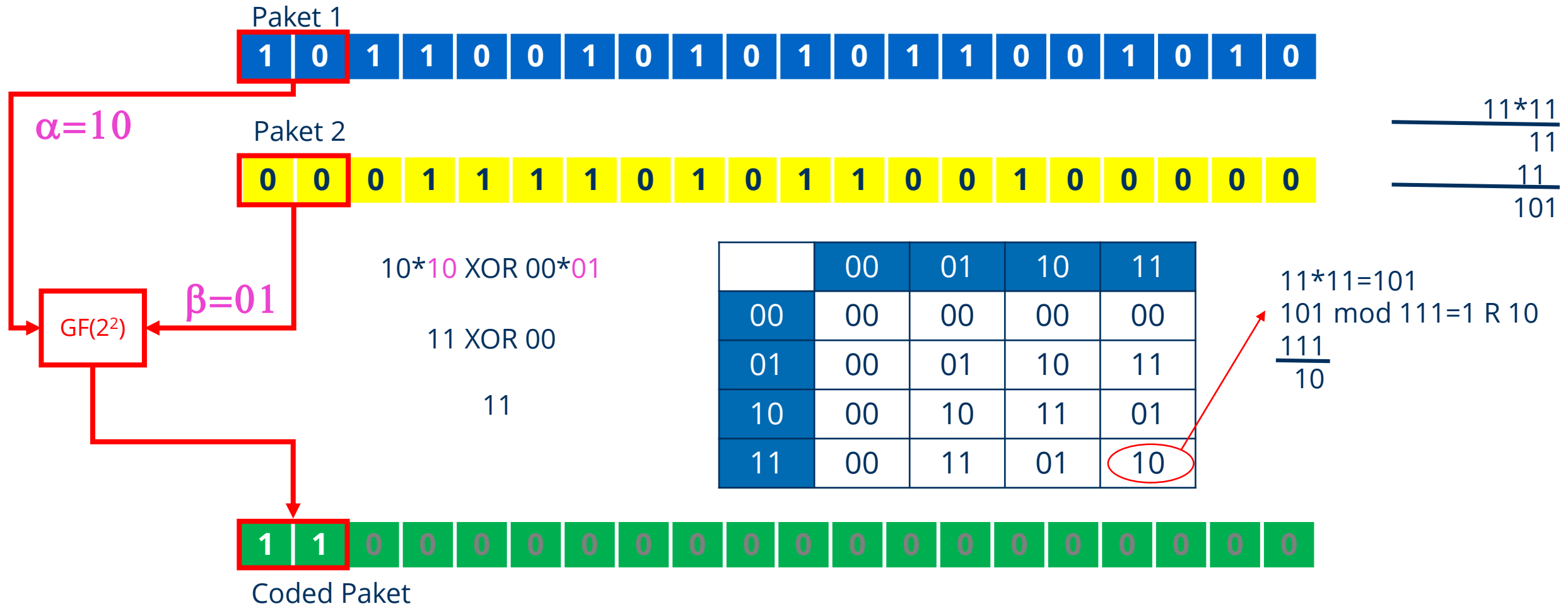
Field Size GF(2²)



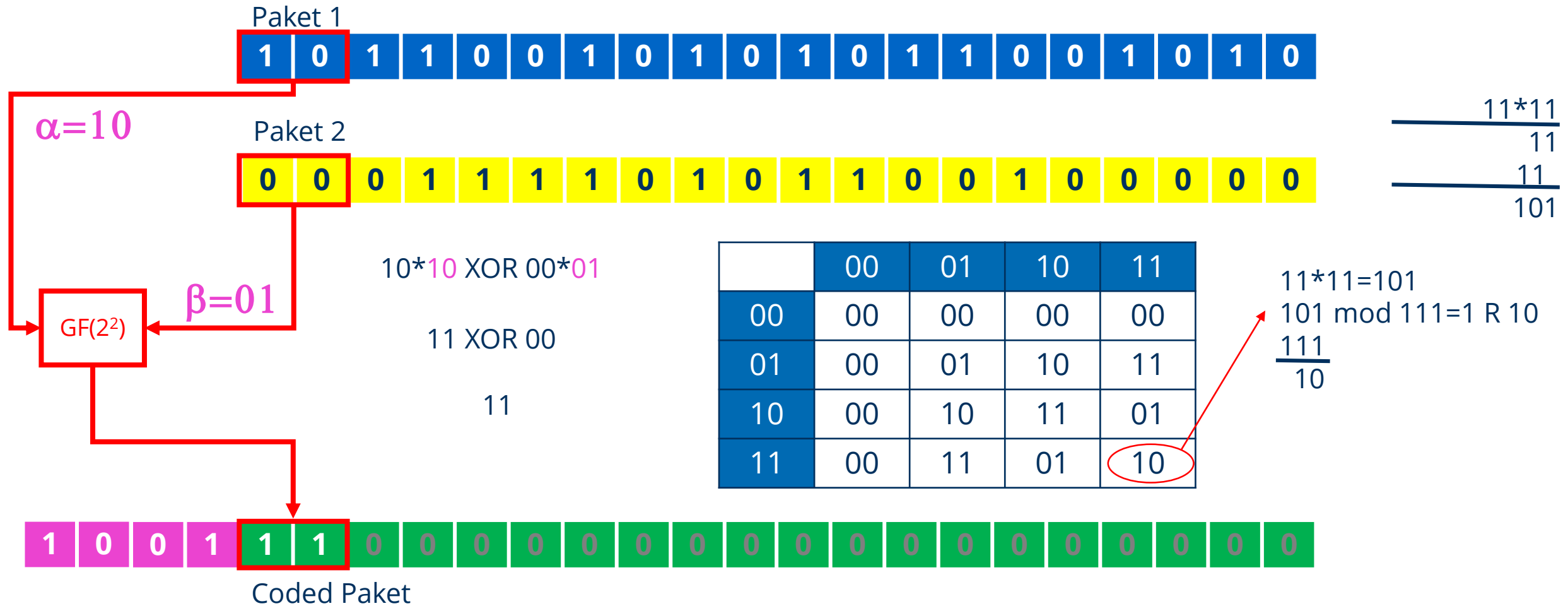
Field Size GF(2²)



Field Size GF(2²)

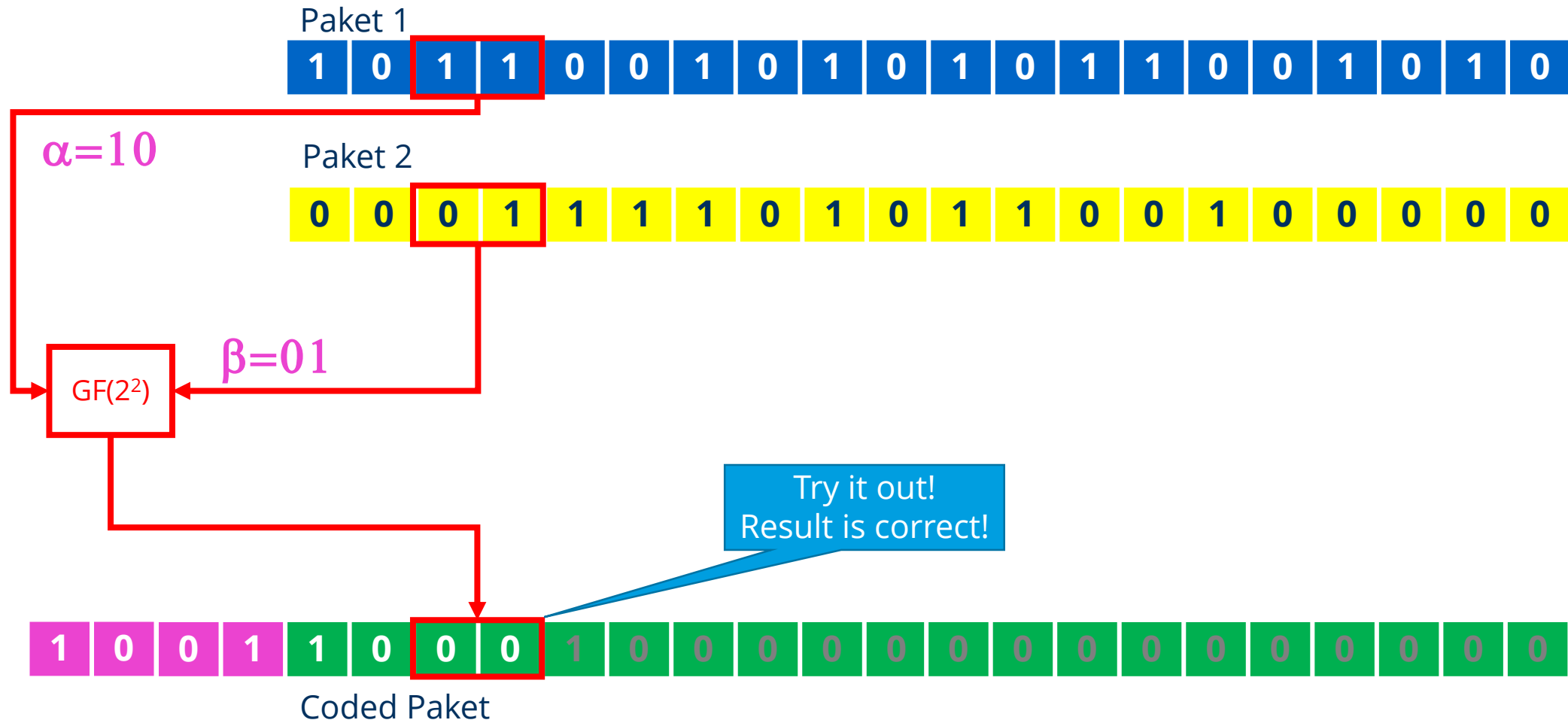


Field Size GF(2²)

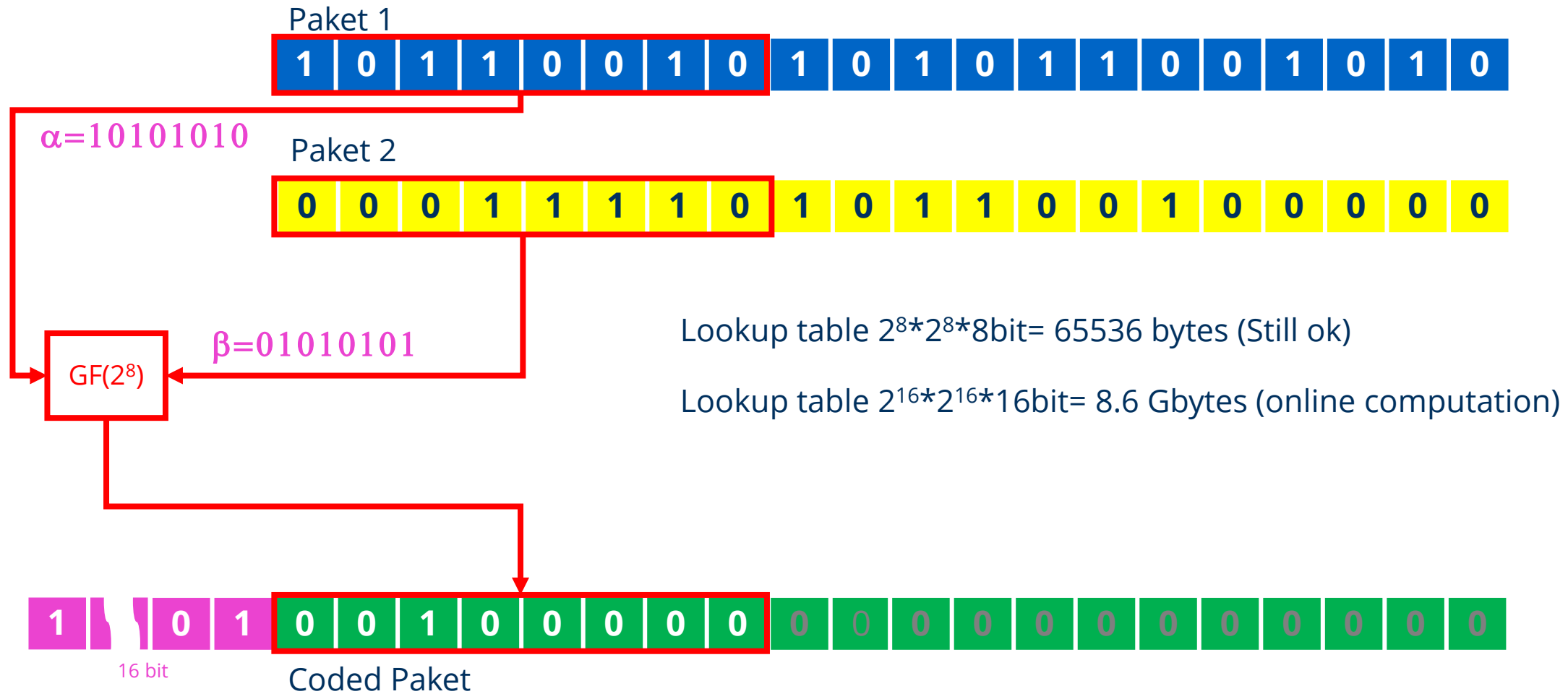




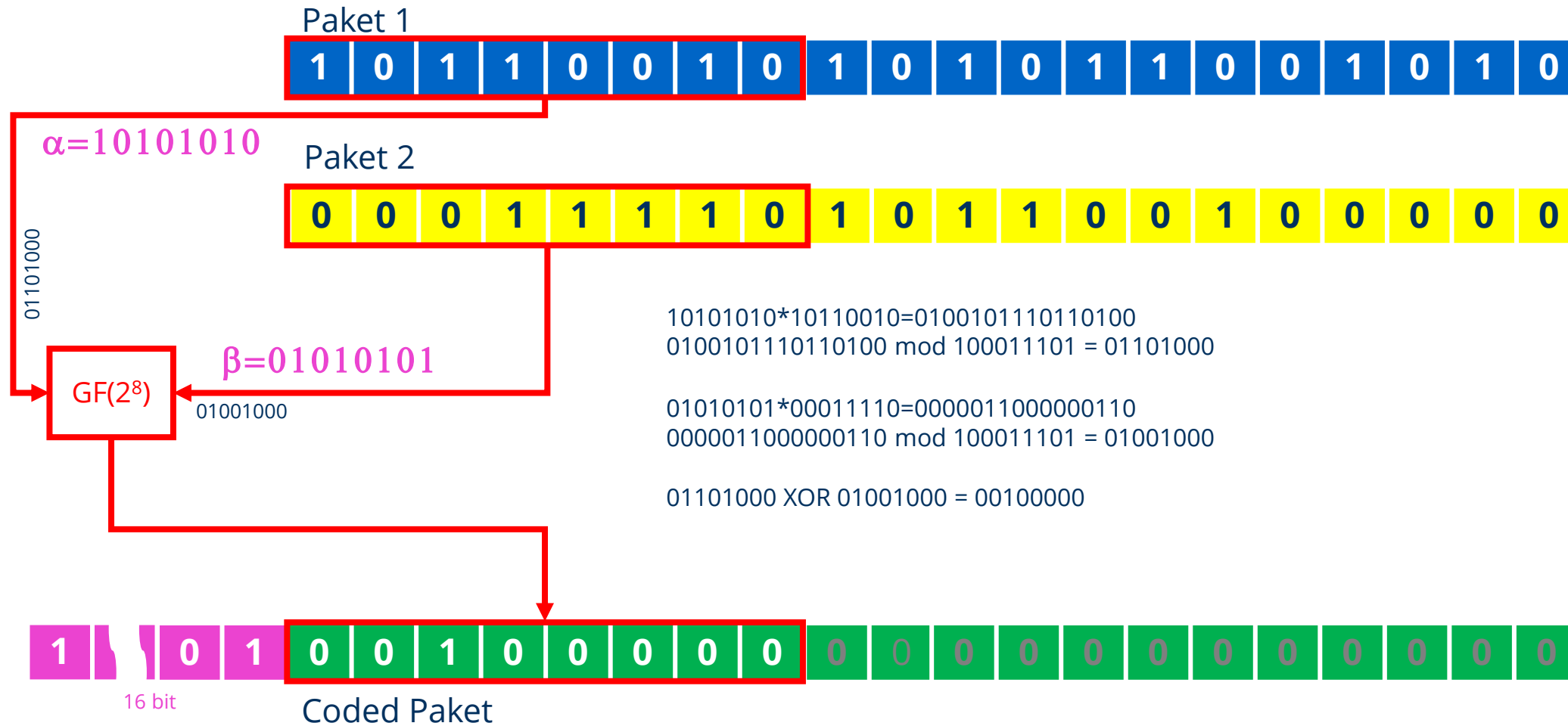
Field Size $GF(2^2)$



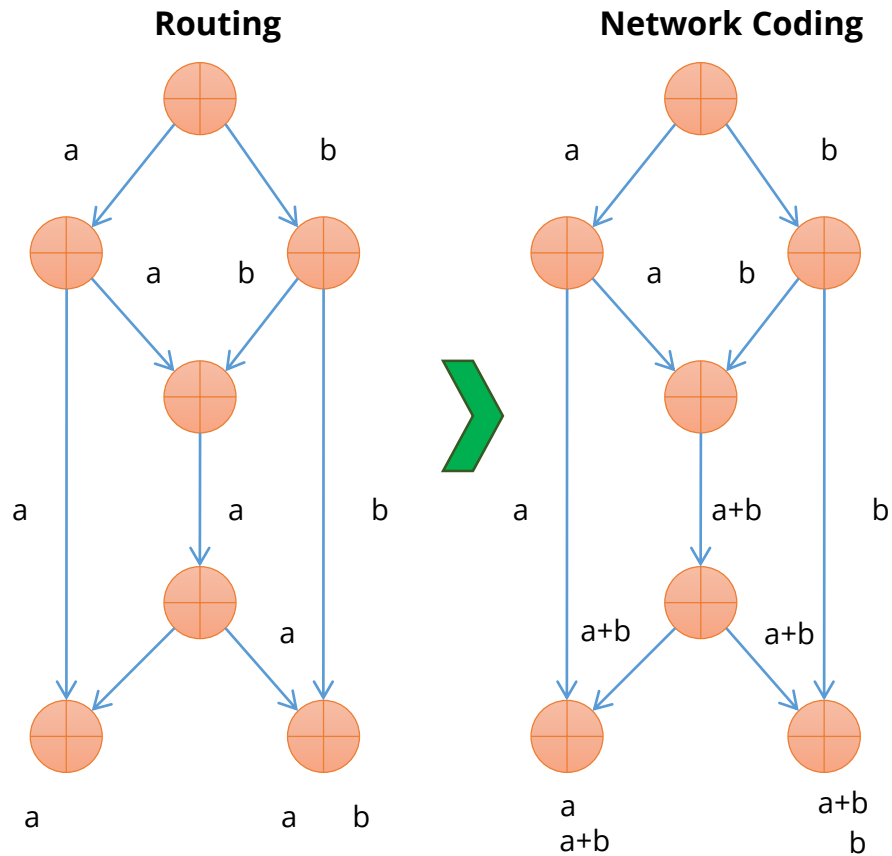
Field Size GF(2⁸)



Field Size GF(2⁸)



Network Coding



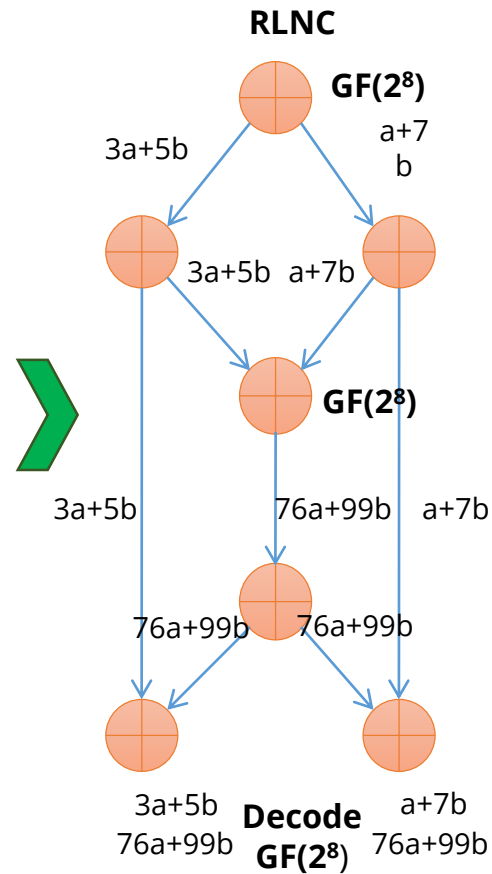
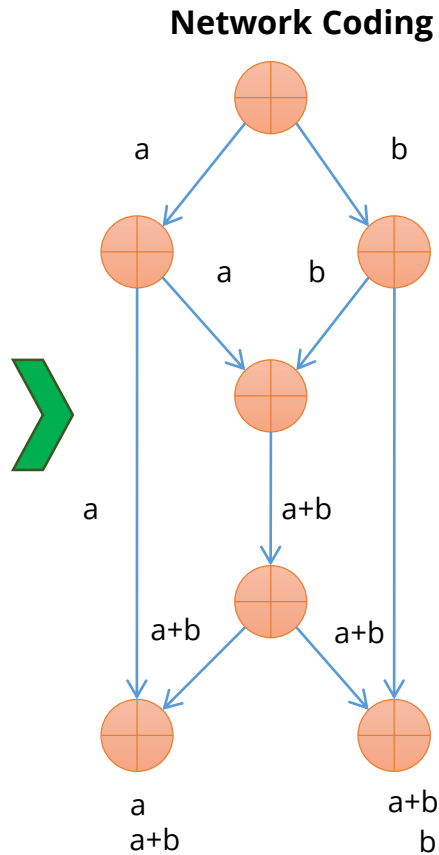
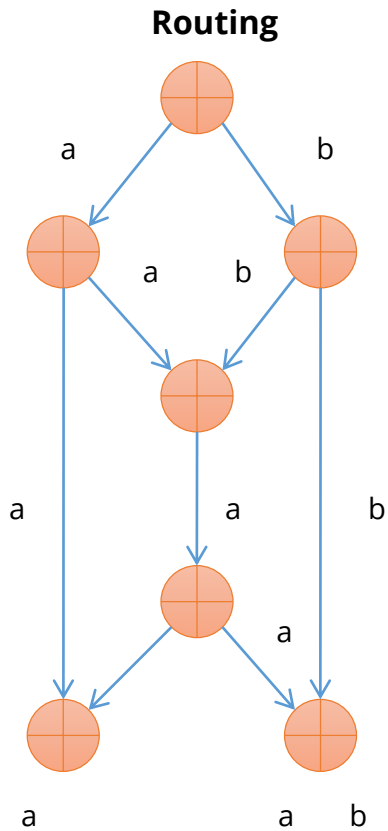
Rate: 1.5 symbols/time
 Distributed (but planned)
 Sub-optimal
 Low processing cost



Rate: 2 symbols/time
 Centralized, Planned
Optimal
Low-Medium processing cost
 One Finite Field in use
 Does not consider device

capabilities
 • One encoder, one decoder
 • One encoder
 • One decoder
 Prof. Dr. h.c. Frank H.P. Fitzek
 Technische Universität Dresden, Deutsche Telekom Chair of Communication Networks

Network Coding



a **a b**

Rate: 1.5 symbols/time
 Distributed (but planned)
 Sub-optimal
 Low processing cost

a b **a b**

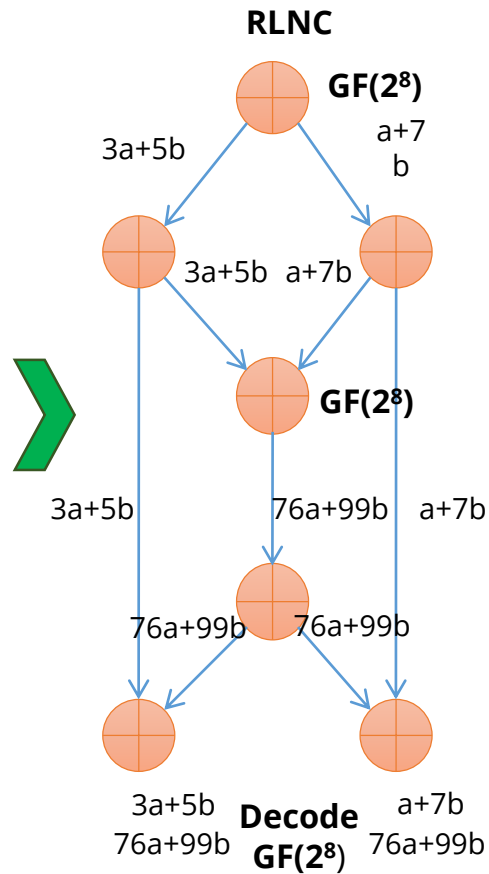
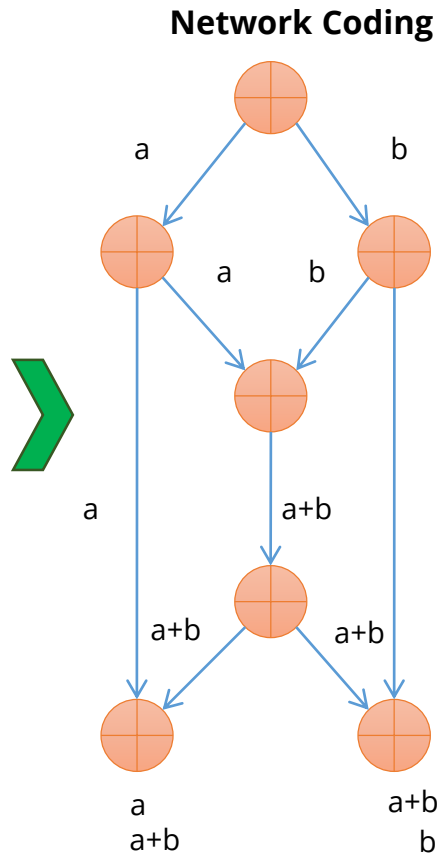
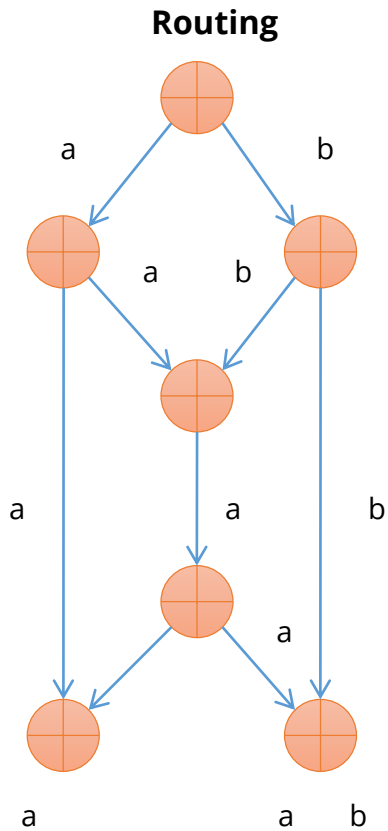
Rate: 2 symbols/time
 Centralized, Planned
Optimal
Low-Medium processing cost
 One Finite Field in use
 Does not consider device capabilities

a b **a b**

Rate: 2 symbols/time
Distributed (not planned)
Optimal (high probability)
 High processing cost
 One Finite Field in use
 Does not consider device capabilities

- One encoder, one decoder
- One recoder

Network Coding



What is the difference with respect to throughput and signalling?

a **a b**

Rate: 1.5 symbols/time
 Distributed (but planned)
 Sub-optimal
 Low processing cost

a b **a b**

Rate: 2 symbols/time
 Centralized, Planned
Optimal
Low-Medium processing cost
 One Finite Field in use
 Does not consider device capabilities

a b **a b**

Rate: 2 symbols/time
Distributed (not planned)
Optimal (high probability)
 High processing cost
 One Finite Field in use
 Does not consider device capabilities

Prepared by Dr. h.c. Frank H.P. Fitzek
 Network Coding
 Technische Universität Dresden, Deutsche Telekom Chair of Communication Networks

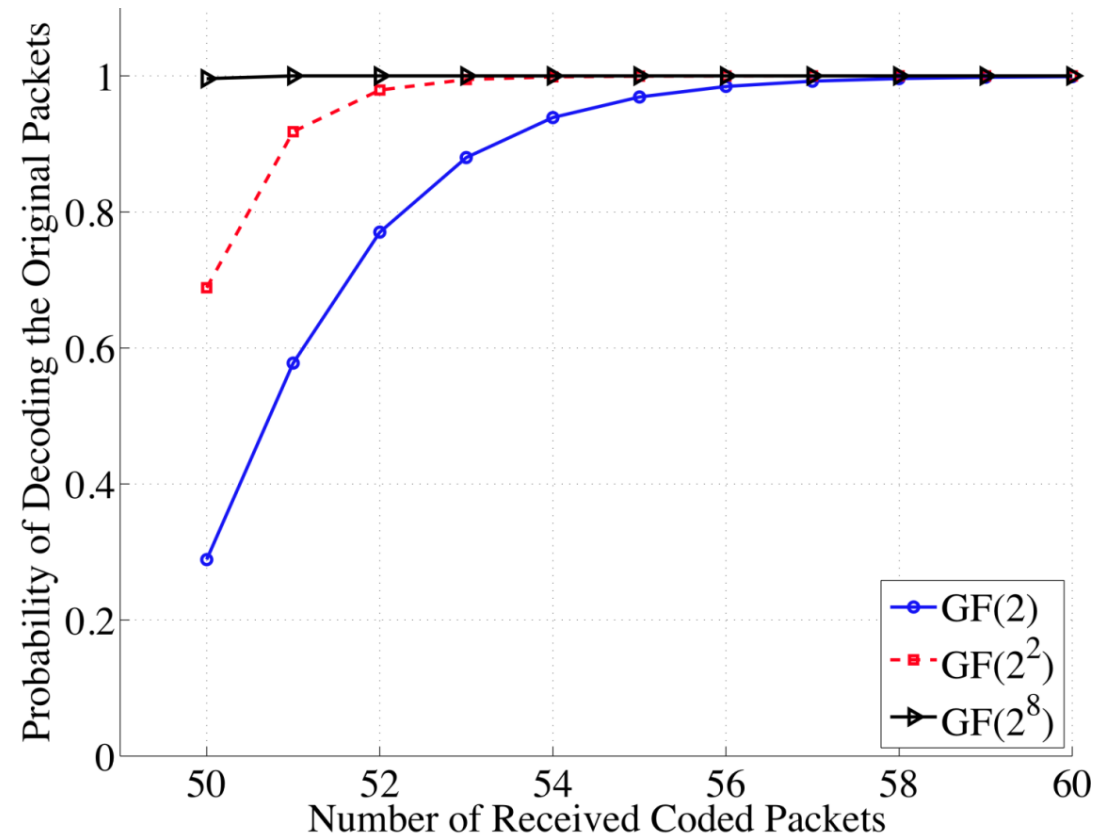
Field Size Analysis

Ralf Kötter: „How bad is binary?“

Field and Generation Size

Following scenario: we have an error-free communication and want to convey 50 packets from A and B

- Without coding, we would send exactly 50 packets
- With Reed-Solomon coding, we would send exactly 50 packets
- If we used RLNC (without systematic mode), we would need to send more packets, because of linear dependency of the packets (doubles so to speak)



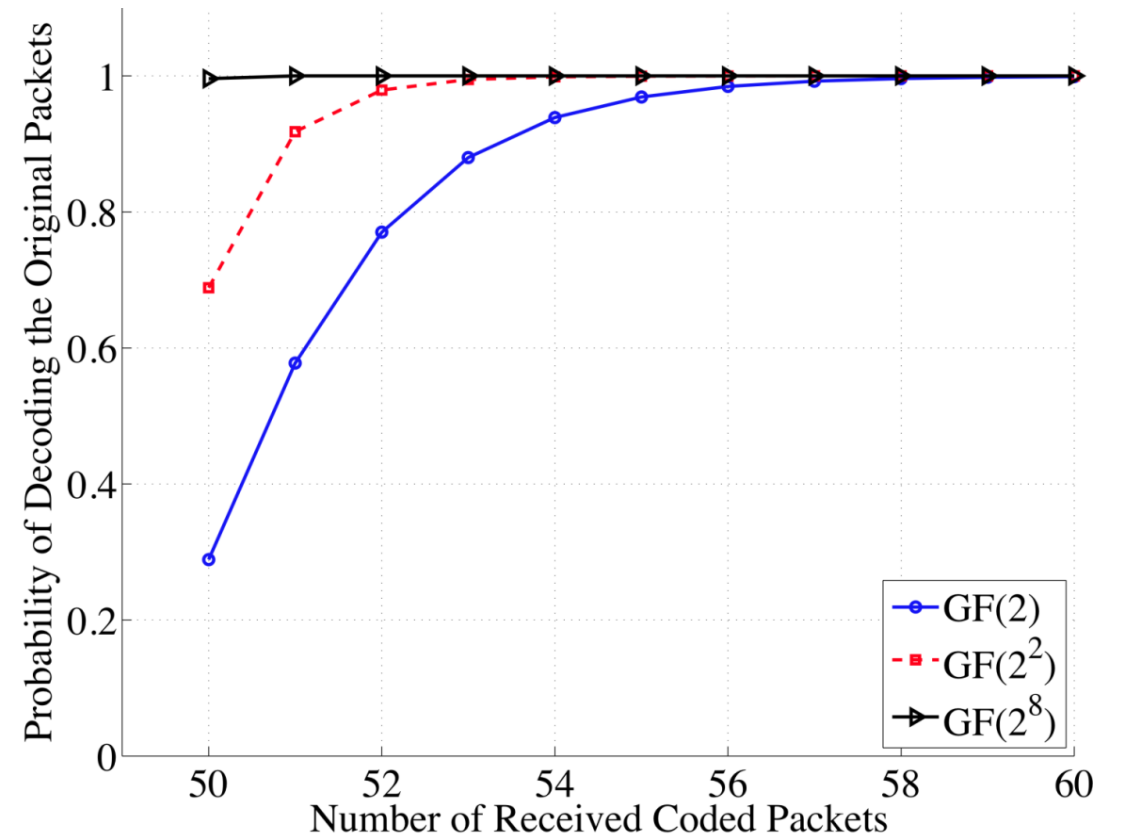
Field and Generation Size

Small field sizes are resulting in linear dependent coded packets.

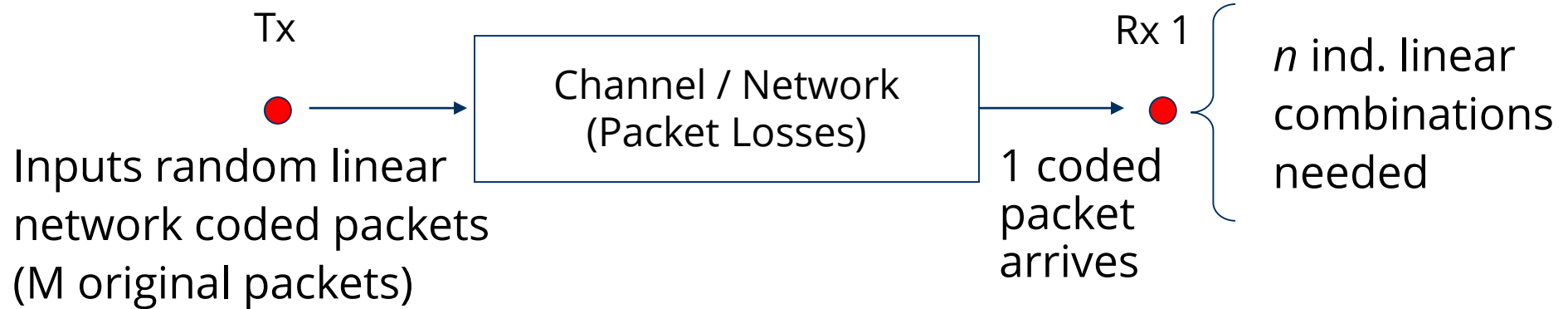
1.6 packets extra per generation in case of binary field sizes.

Large fields sizes (2^8 or higher) have nearly no linear dependency.

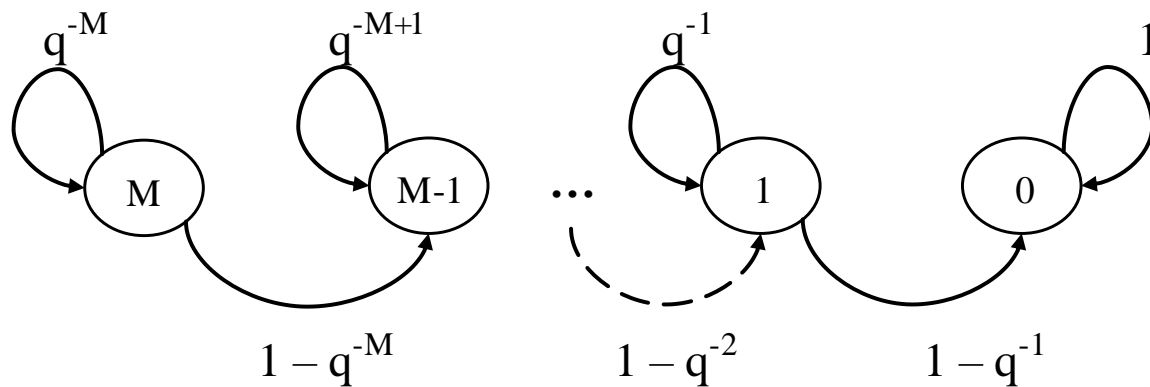
Theory is aiming for large field sizes and large generation sizes!



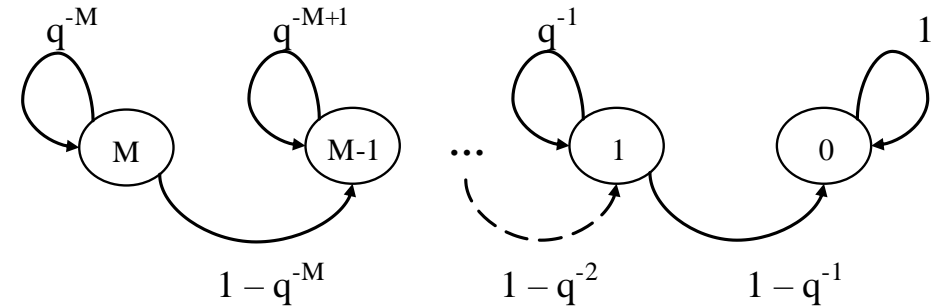
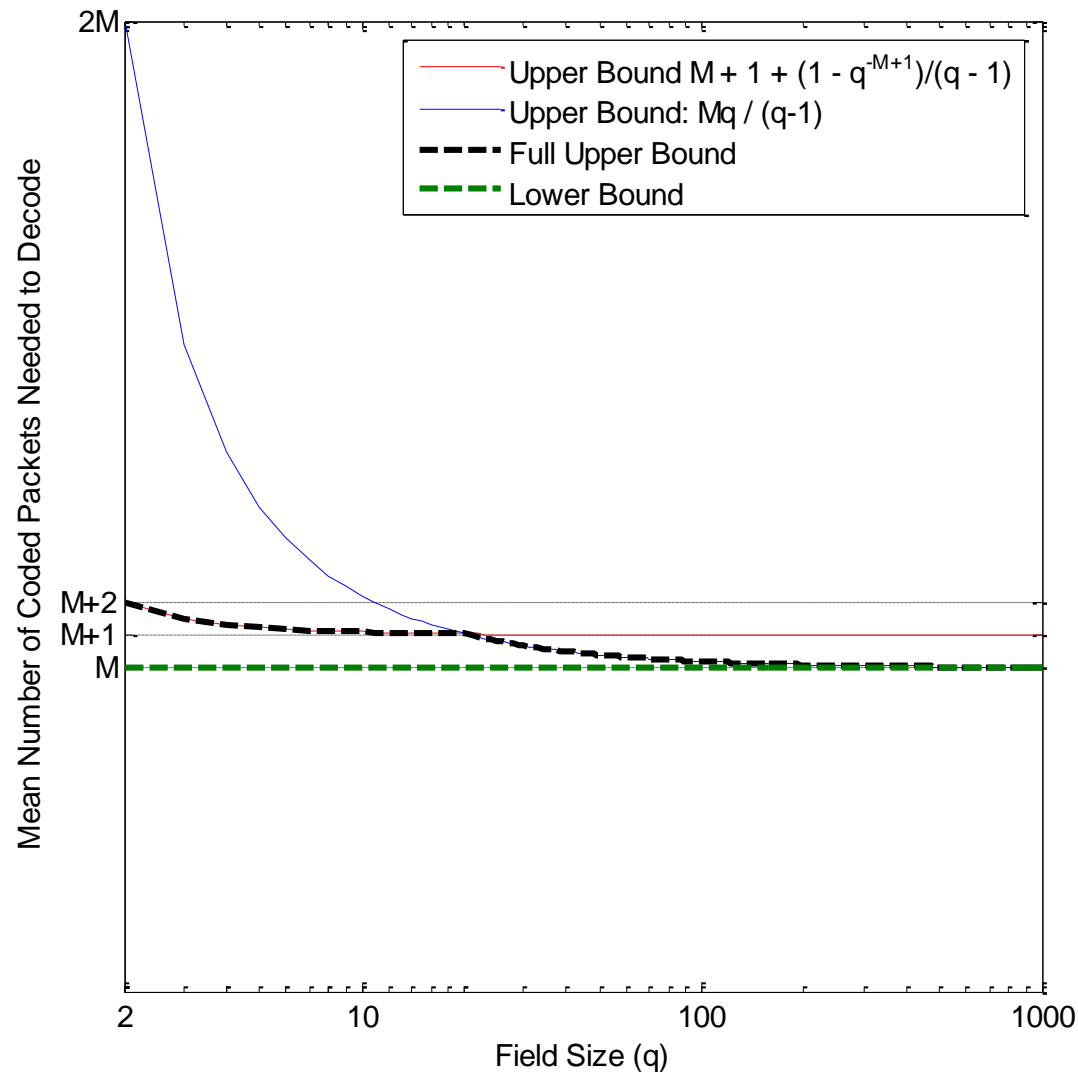
Field Size Analysis for RLNC



–Modeled as a Markov chain



Field Size Analysis



$$E[N_C] = \sum_{k=1}^M \frac{1}{1 - q^{-k}}$$

$$\leq \min \left(M \frac{q}{q-1}, M + 1 + \frac{1 + q^{-M+1}}{q-1} \right)$$

$$\leq M + 2$$

If M or q is large:

- Little overhead
- Small performance degradation



Field Size Analysis

$GF(2) G=4$

0000 → null vector → probability $1 / 2^4 = 1/16$

E.g. assuming three linear independent combinations

0110

1000

1101

KODO

1000

0110

0011 ← 0101 (XOR 0110) ← 1101 (XOR 1000)

One more step

1000

0101 (0110 XOR 0011)

0011

What is the probability to achieve a linear independent linear combination?

xxx1

(any combination with a ONE the last digit)

There are eight combinations out of 16 that fulfill that requirement, therefore 50% is the likelihood that we will receive a valuable combination for the last missing packet.

And for the second last combination? Here it is 25%!
More in the exercise!

Therefore, the mean value for transmissions over an error-free channel with binary coding is

$$E_n = G + 1.6$$

with G being the generation size.



Field Size Analysis

$GF(2) G=4$

0000 \rightarrow null vector \rightarrow probability $1 / 2^4 = 1/16$

E.g. assuming three linear independent combinations

0110 \leftarrow SEEN $X_2 X_3$

1000 \leftarrow DECODED X_1

1101 \leftarrow SEEN $X_1 X_2 X_4$

KODO

1000

0110

0011 \leftarrow 0101 (XOR 0110) \leftarrow 1101 (XOR 1000)

One more step

1000

0101 (0110 XOR 0011)

0011

What is the probability to achieve a linear independent linear combination?

$xxx1$

(any combination with a ONE the last digit)

There are eight combinations out of 16 that fulfill that requirement, therefore 50% is the likelihood that we will receive a valuable combination for the last missing packet.

And for the second last combination? Here it is 25%!
More in the exercise!

Therefore, the mean value for transmissions over an error-free channel with binary coding is

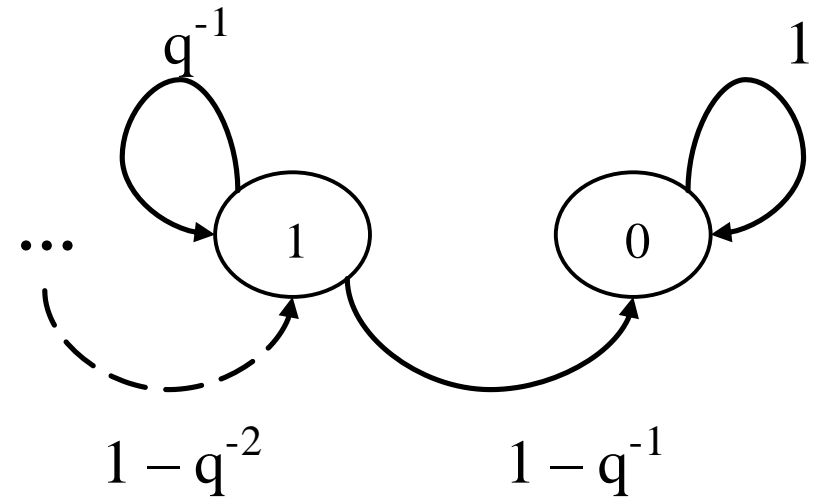
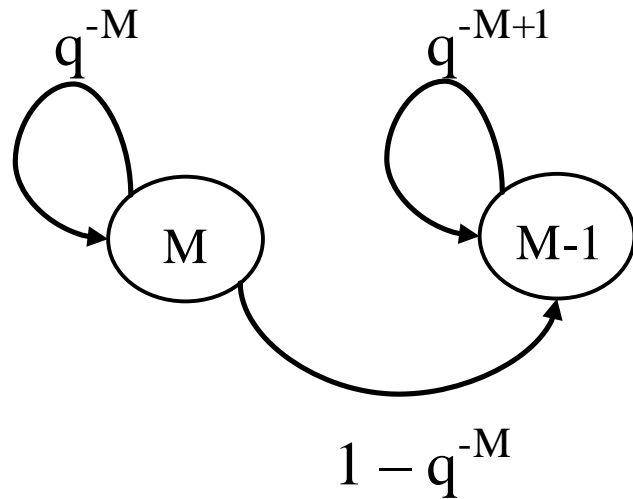
$$E_n = G + 1.6$$

with G being the generation size.

Field Size Analysis

$GF(2) G=4$

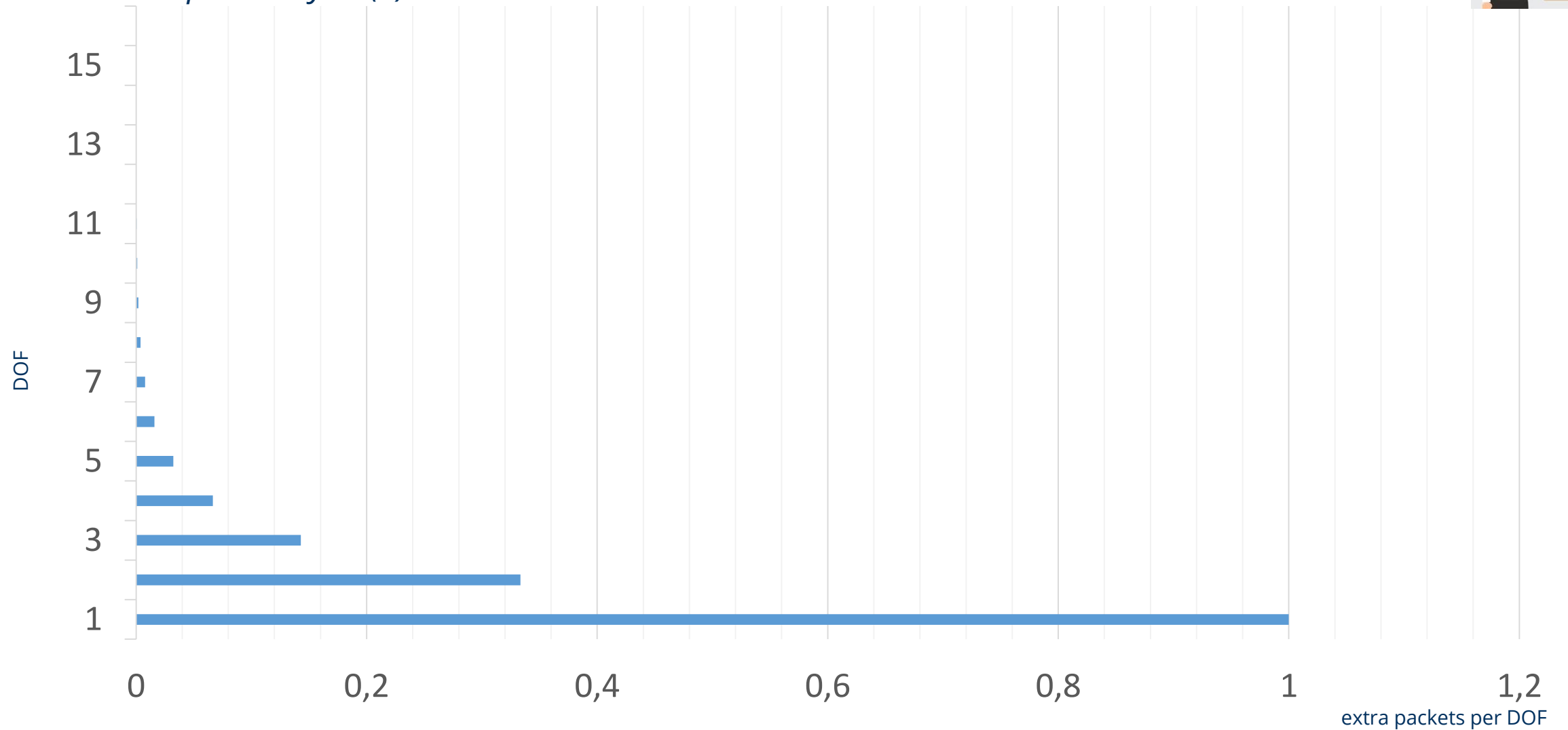
$GF(2^8)$	$(1/2^8)^4$	$(1/2^8)^3$	$(1/2^8)^2$	$1/2^8$	DONE
$GF(2)$	$1/16$	$1/8$	$1/4$	$1/2$	DONE
	0000	x000	xx00	xxx0	





Field Size Analysis

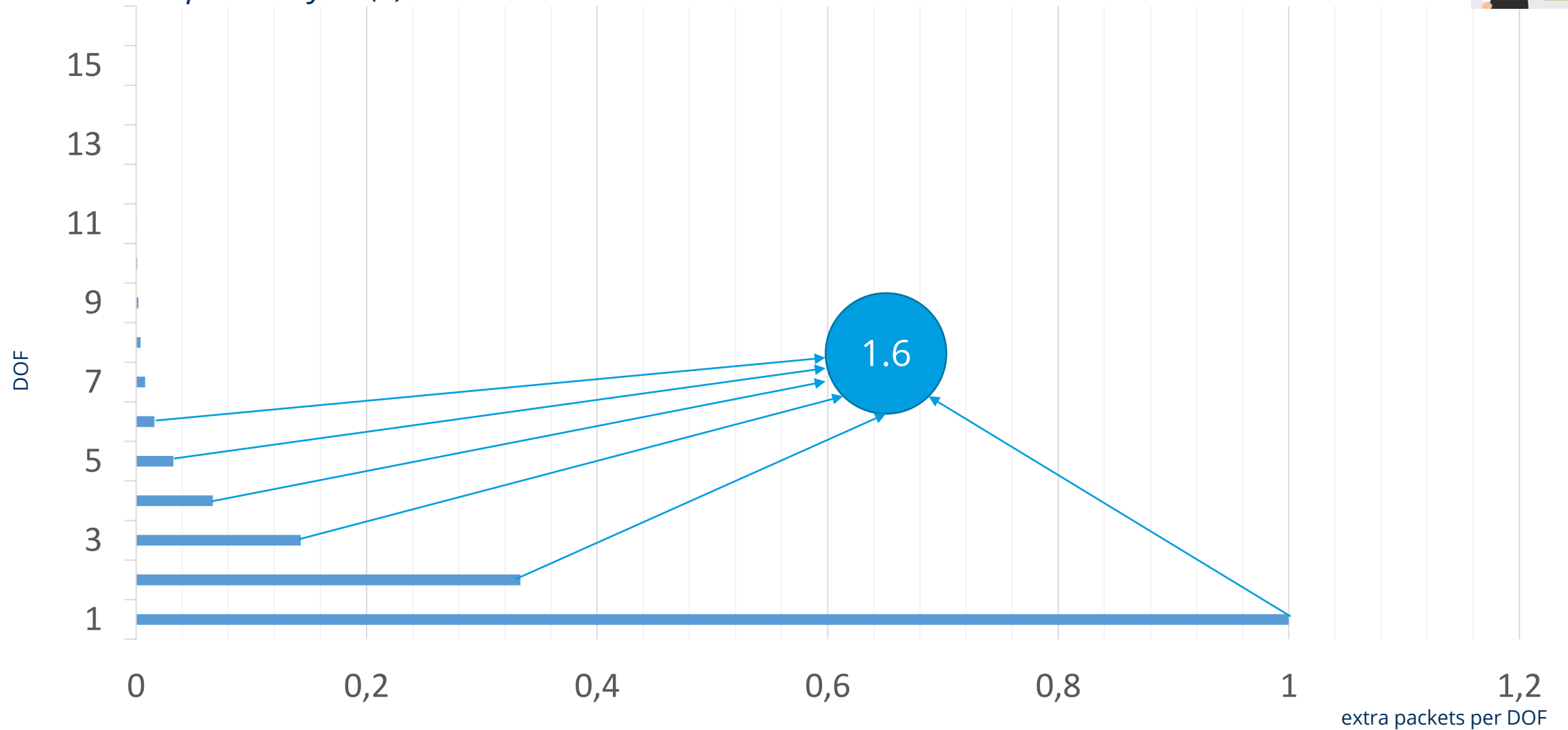
Linear dependency $GF(2)$ $G=16$





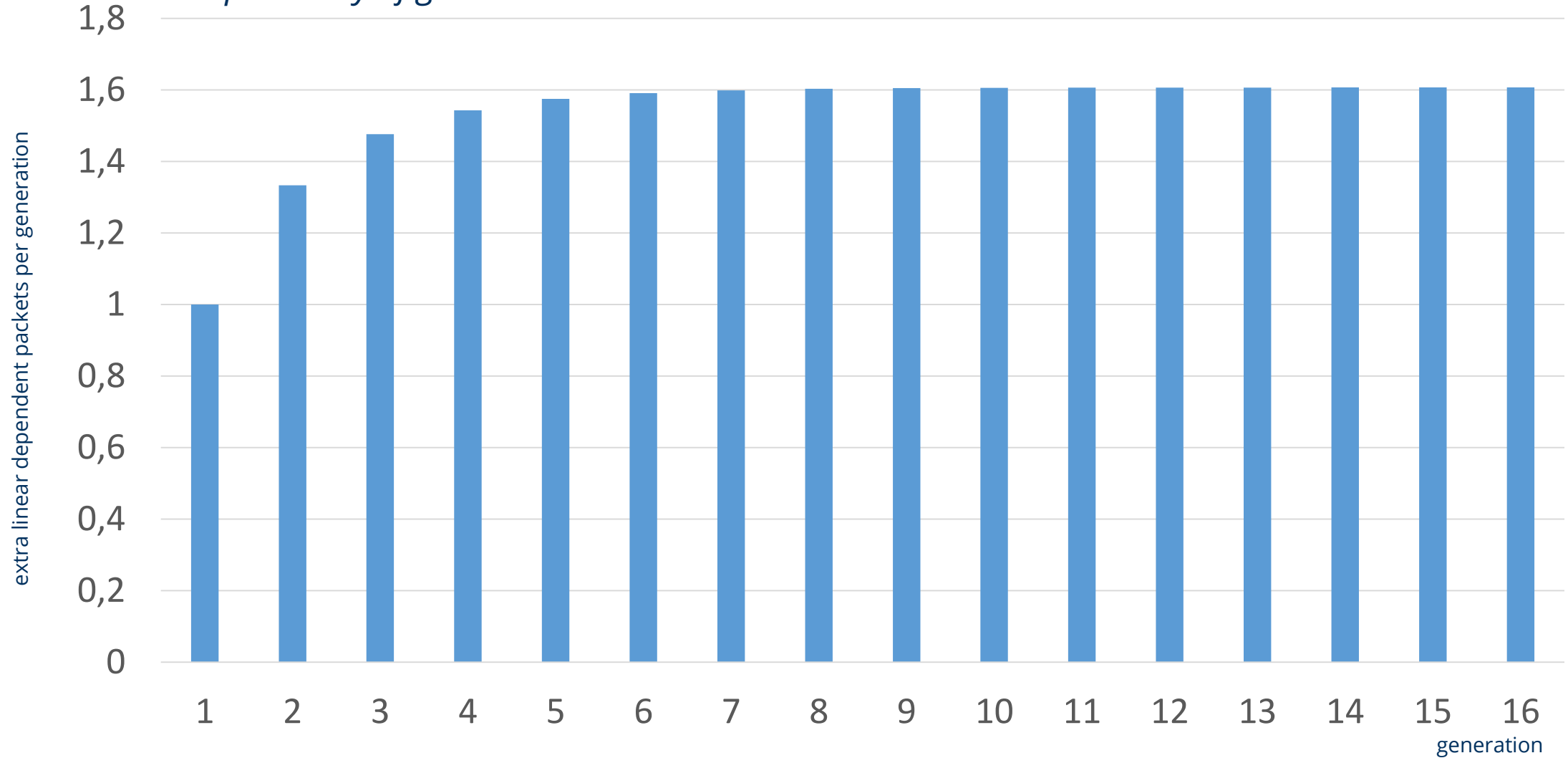
Field Size Analysis

Linear dependency $GF(2)$ $G=16$



Field Size Analysis

Overhead in dependency of generation size



http://notebook.deutsche-telekom.rocks



```
In [1]: # Copyright Steinwurf ApS 2015.  
# Distributed under the "STEINWURF RESEARCH LICENSE 1.0".  
# See accompanying file LICENSE.rst or  
# http://www.steinwurf.com/licensing
```

Basic example: Encoding and Decoding

We need to import kodo. It is simple

```
In [2]: import os  
import sys  
  
import kodo
```

We set some constants

```
In [3]: # Set the number of symbols (i.e. the generation size in RLNC  
# terminology) and the size of a symbol in bytes  
symbols = 8  
symbol_size = 160
```

To create an encoder and a decoder, we need to create a factory. After, we can build as many encoders and decoders as we want

```
In [4]: # In the following we will make an encoder/decoder factory.  
# The factories are used to build actual encoders/decoders  
encoder_factory = kodo.FullVectorEncoderFactoryBinary(symbols, symbol_size)  
encoder = encoder_factory.build()  
  
decoder_factory = kodo.FullVectorDecoderFactoryBinary(symbols, symbol_size)  
decoder = decoder_factory.build()
```



Extra packets per generation

How many extra packet do we need to send due to linear dependencies?

```
In [9]: packets_sent = []
        for i in xrange(1000):
            packet_number = 0
            decoder = decoder_factory.build()
            while not decoder.is_complete():
                # Generate an encoded packet
                packet = encoder.write_payload()

                # Pass that packet to the decoder
                decoder.read_payload(packet)
                packet_number += 1

            packets_sent.append(packet_number)

        # We print the average of extra sent packets
        mean_extra_packets = sum(packets_sent)/float(len(packets_sent)) - symbols
        mean_extra_packets
```

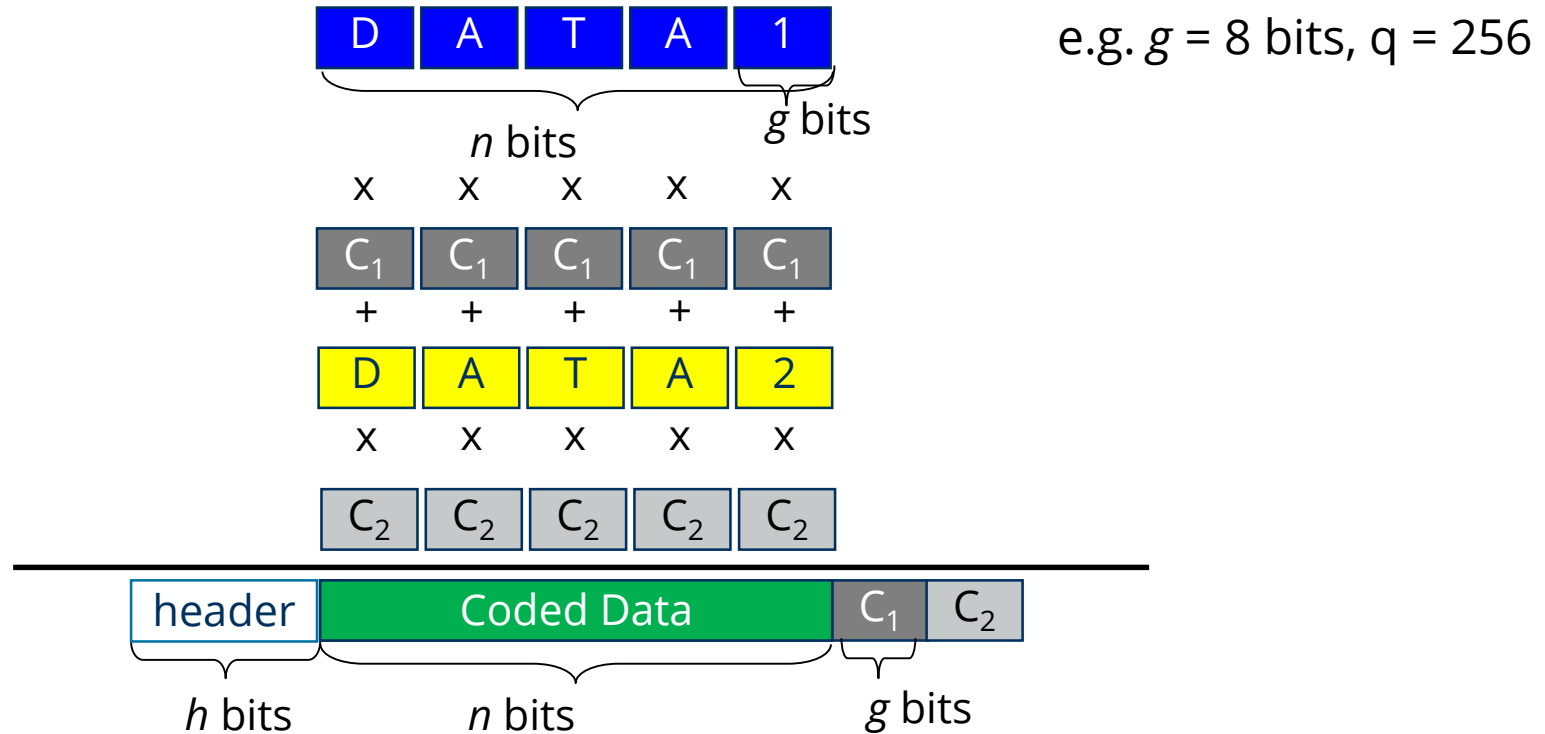
Out[9]: 1.5899999999999999

It is only 1.6 extra packets!

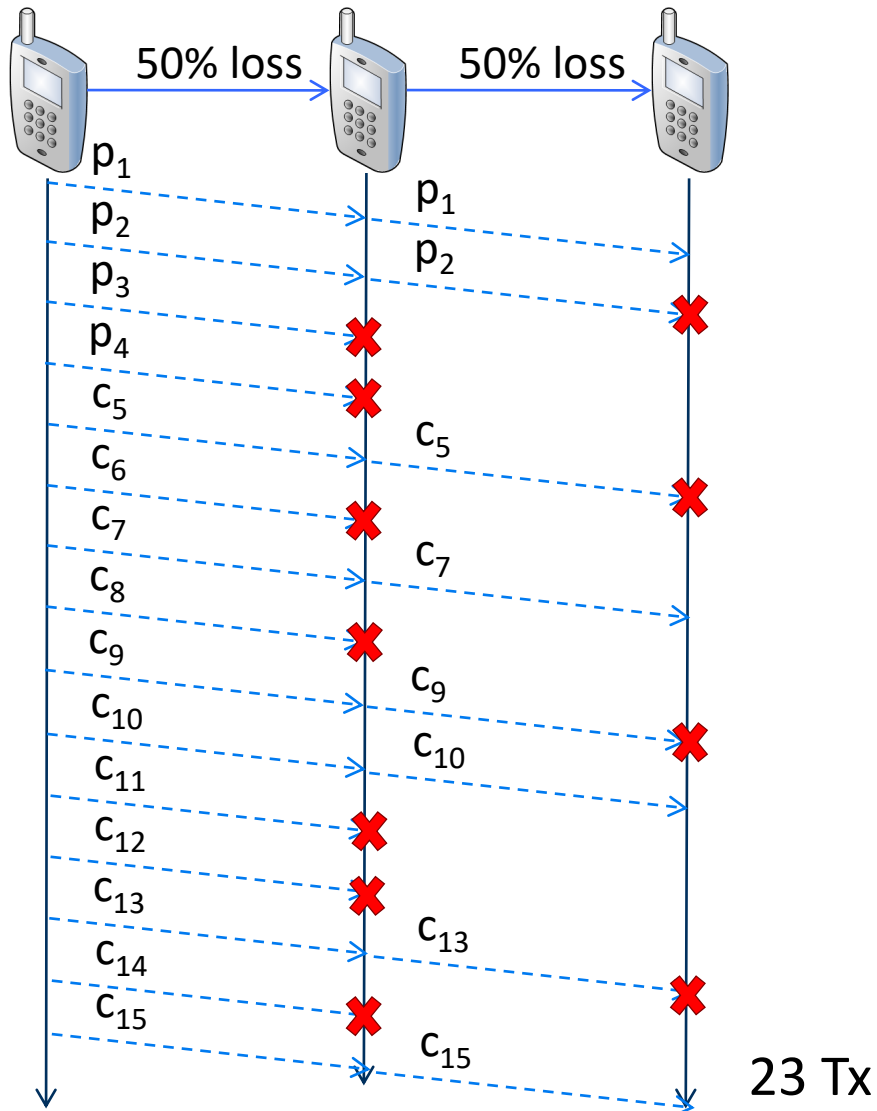
Recoding Potential

Generating a Coded Packet

- Generating a linear network coded packet (CP) $CP_j = \sum_i C_i P_i$
- Operations over finite field of size.



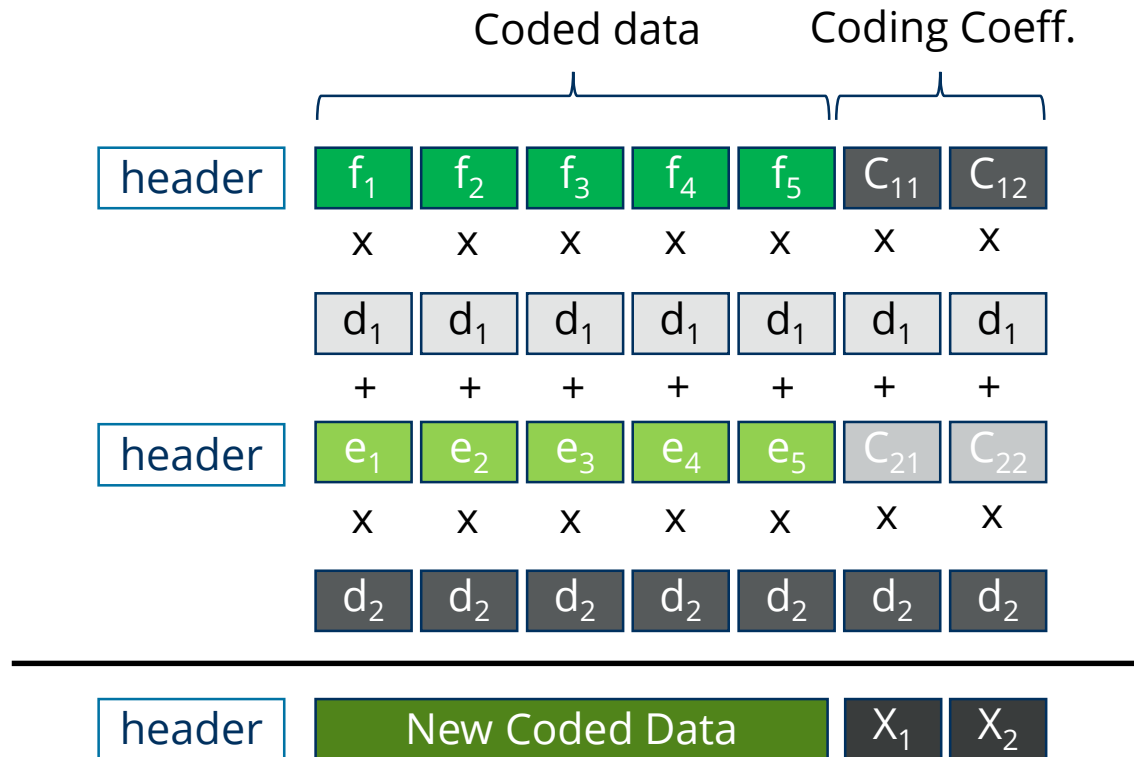
No Recoding



- Simple operation: forward
- Structure of code is preserved
- Issues
 - Delay per batch of packets
 - Missing transmission opportunities
 - Equivalent loss probability: compounding each channel's loss
- Loss of channel i : e_i
- Equivalent success prob of a packet:
 $(1-e_1)(1-e_2)$

Recoding Packets

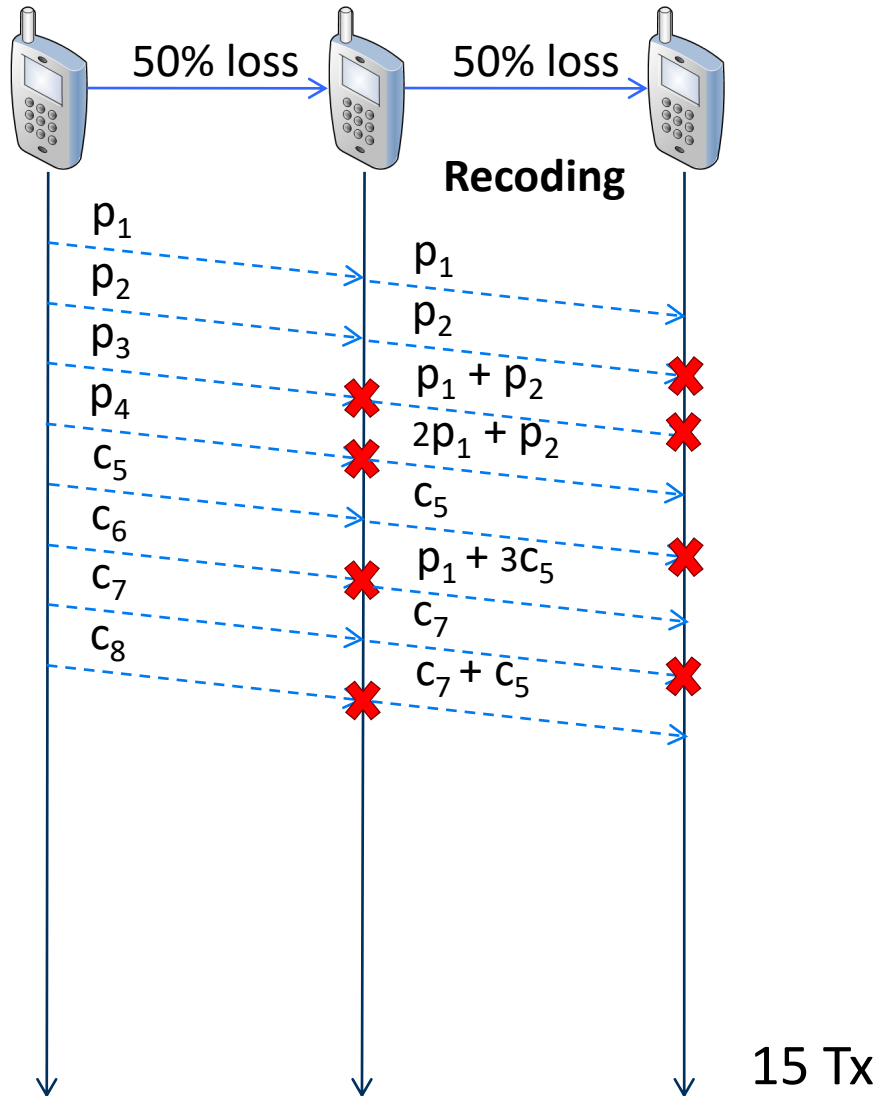
Generating a new linear network coded packet (CP)



$$X_1 = d_1 C_{11} + d_2 C_{21} \quad \text{and} \quad X_2 = d_1 C_{12} + d_2 C_{22}$$

Recall that: $f = C_{11}P_1 + C_{12}P_2$ and $e = C_{21}P_1 + C_{22}P_2$
 Thus, $d_1f + d_2e = d_1C_{11}P_1 + d_1C_{12}P_2 + d_2C_{21}P_1 + d_2C_{22}P_2 = X_1P_1 + X_2P_2$

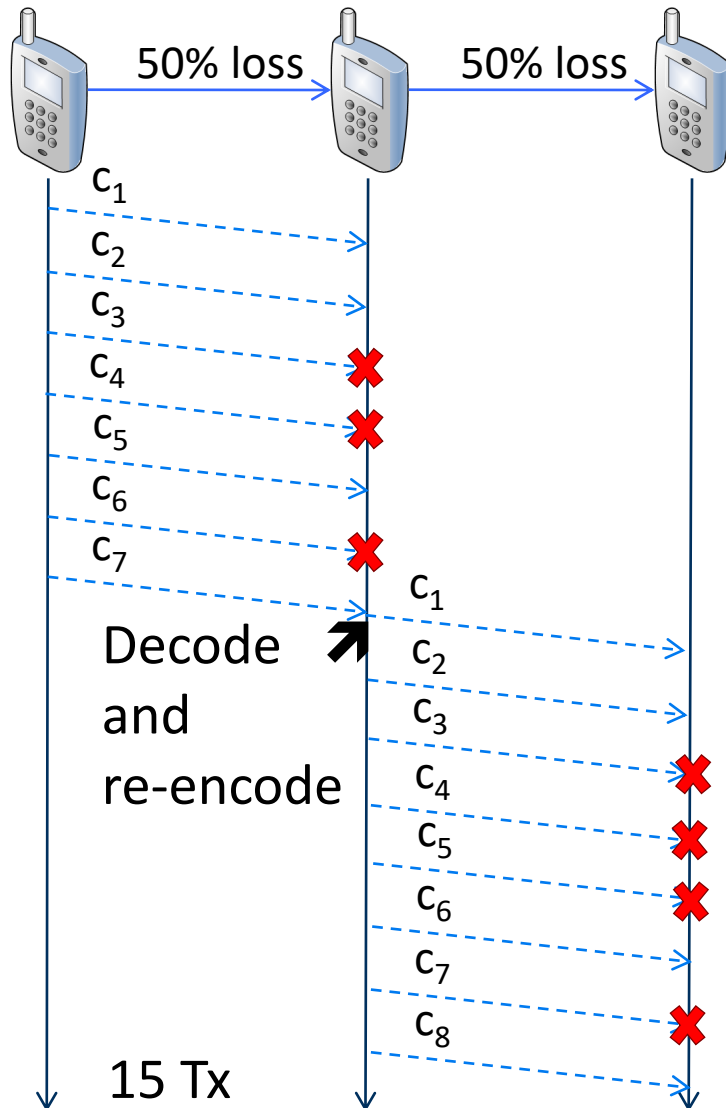
Recoding Packets



- A bit more complex: recode
 - Equivalent to encoding in worst case
 - Not so bad
- Structure of code may be changed
 - Some exceptions
- Issues
 - If left unchecked, can have unnecessary transmissions, e.g., c_8
 - Additional processing needed
- Advantage: equivalent success probability of a *linear combination*:

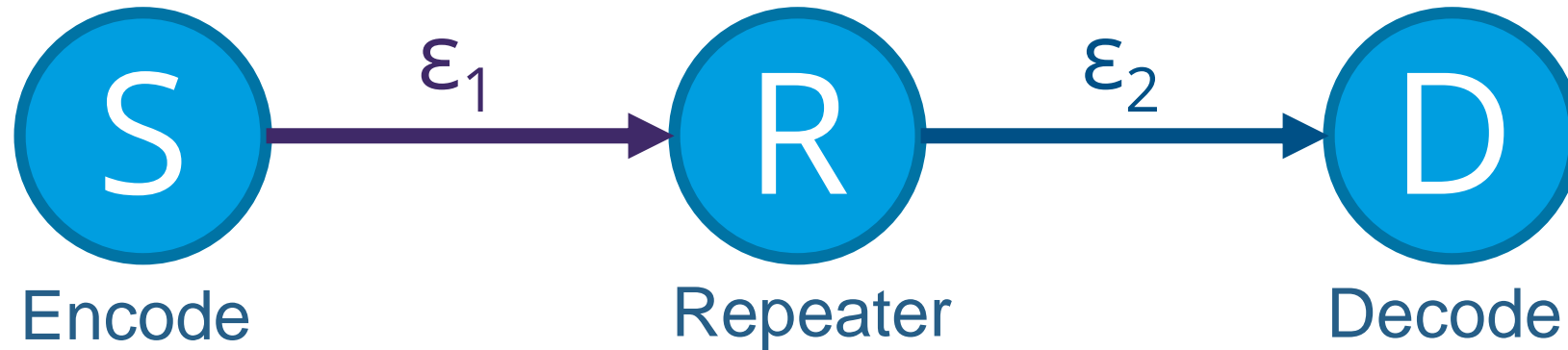
$$\min\{1-e_1, 1-e_2\}$$

Is “recoding” possible with other linear codes?



- Consider Reed-Solomon, LT, etc
- Structure is not composable
 - Mixing coded packets does not
 - produce a “valid” coded packet
 - Different structure, properties are lost
- Recoding means receiving enough
- coded packets, decode,
- and *then* re-encode
- Issues
 - Delay per batch
 - Computational effort
 - Can also have unnecessary Tx
- Success probability of a *linear combination*:
 $\min\{1-e_1, 1-e_2\}$

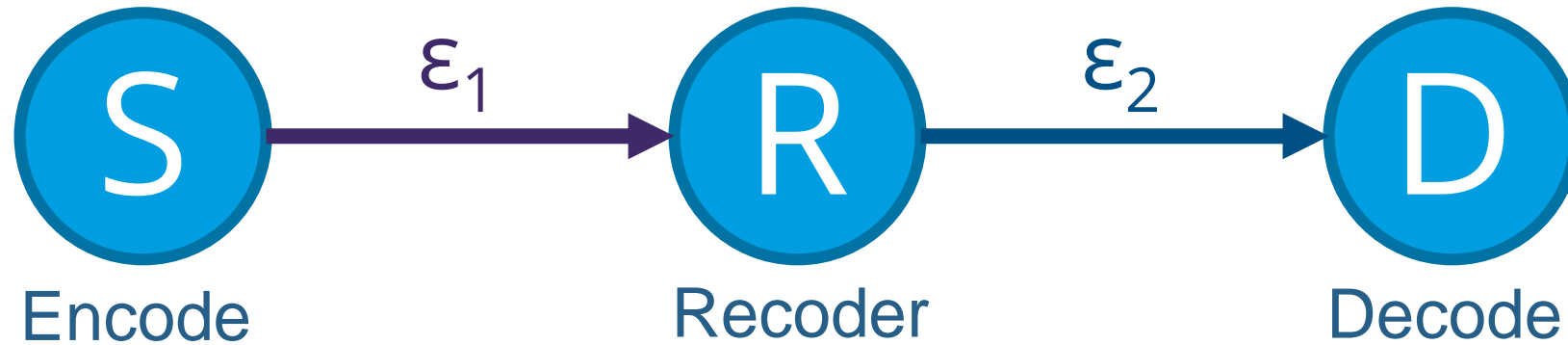
Network Coding



Reed Solomon
LDPC
LT
Raptor



Network Coding



Reed Solomon
LDPC
LT
Raptor

$$(1-\epsilon_1)(1-\epsilon_2)$$



$$\min\{(1-\epsilon_1);(1-\epsilon_2)\}$$

Network Coding



<http://notebook.deutsche-telekom.rocks>



- [encoding-decoding.ipynb](#) example
- [encoding-recoding-decoding.ipynb](#) example
- Proof the aforementioned gain of recoder over repeater
 - Losses of link one and two are 60% and 20%, respectively
- The „shark fin“ problem
- <https://arxiv.org/abs/1601.03201>

http://notebook.deutsche-telekom.rocks



```
In [1]: # Copyright Steinwurf ApS 2015.
# Distributed under the "STEINWURF RESEARCH LICENSE 1.0".
# See accompanying file LICENSE.rst or
# http://www.steinwurf.com/licensing

import os
import sys
import random

import kodo

"""
Encode recode decode example.
In Network Coding applications one of the key features is the
ability of intermediate nodes in the network to recode packets
as they traverse them. In Kodo it is possible to recode packets
in decoders which provide the recode() function.
This example shows how to use one encoder and two decoders to
simulate a simple relay network as shown below (for simplicity
we have error free links, i.e. no data packets are lost when being
sent from encoder to decoder1 and decoder1 to decoder2):

    +-----+   +-----+   +-----+
    | encoder |+---| decoder1 |+---| decoder2 |
    +-----+   | (recoder) |   +-----+
                  +-----+

In a practical application recoding can be using in several different
ways and one must consider several different factors e.g. such as
reducing linear dependency by coordinating several recoding nodes
in the network.
Suggestions for dealing with such issues can be found in current
research literature (e.g. MORE: A Network Coding Approach to
Opportunistic Routing).
"""
```



Simple forwarding

```
In [2]: e1 = ['loss'] * 2 + ['success'] * 8
e2 = ['loss'] * 6 + ['success'] * 4

packets_sent = []
for i in xrange(1000):
    decoder1 = decoder_factory.build()
    decoder2 = decoder_factory.build()
    packet_number = 0
    while not decoder2.is_complete():

        # Encode a packet into the payload buffer
        packet = encoder.write_payload()
        packet_number += 1

        # Pass that packet to decoder1 with (1-e1) probability
        if random.choice(e1) == 'success':
            decoder1.read_payload(packet)
        else:
            continue

        # Now produce a new recoded packet from the current
        # decoding buffer, and place it into the payload buffer
        packet = decoder1.write_payload()

        # Pass the recoded packet to decoder2 with (1-e2) probability
        if random.choice(e2) == 'success':
            decoder2.read_payload(packet)

    packets_sent.append(packet_number)

# We print the average packets sent
sum(packets_sent)/float(len(packets_sent))
```

Out[2]: 100.324



With recoding

```
In [4]: e1 = ['loss'] * 2 + ['success'] * 8
e2 = ['loss'] * 6 + ['success'] * 4

packets_sent = []
for i in xrange(1000):
    decoder1 = decoder_factory.build()
    decoder2 = decoder_factory.build()
    packet_number = 0
    while not decoder2.is_complete():

        # Encode a packet into the payload buffer
        packet = encoder.write_payload()
        packet_number += 1

        # Pass that packet to decoder1 with (1-e1) probability
        if random.choice(e1) == 'success':
            decoder1.read_payload(packet)

        # Now produce a new recoded packet from the current
        # decoding buffer, and place it into the payload buffer
        packet = decoder1.write_payload()

        # Pass the recoded packet to decoder2 with (1-e2) probability
        if random.choice(e2) == 'success':
            decoder2.read_payload(packet)

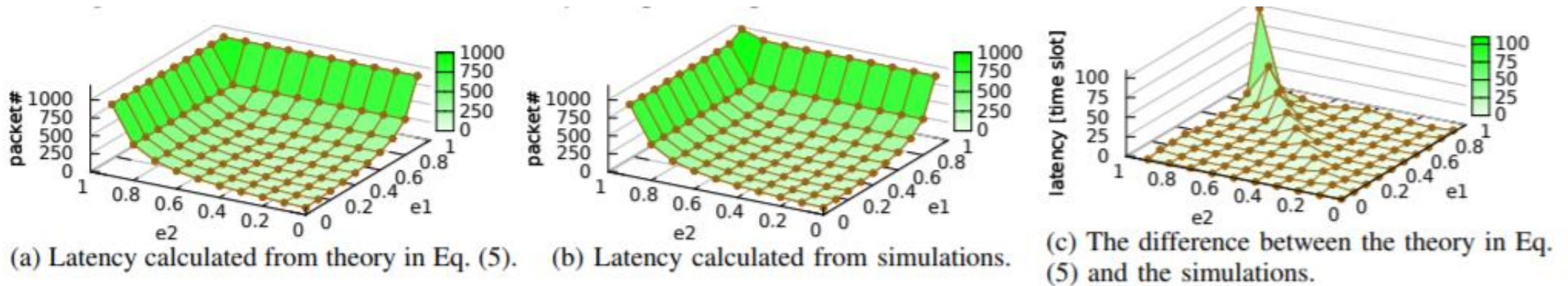
    packets_sent.append(packet_number)

# We print the average packets sent
sum(packets_sent)/float(len(packets_sent))
```

Out[4]: 80 323

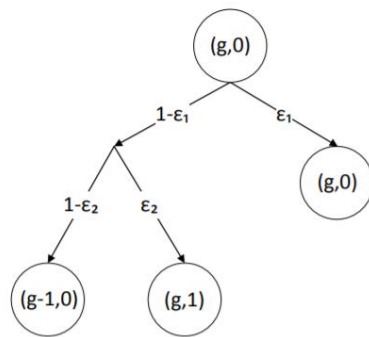
Sharkfin Problem

- If error rate is high and the same (or close) for both links, the theory and the practical implementation do not match.
- Underlines the importance for implementation and theory → Theory that matters!

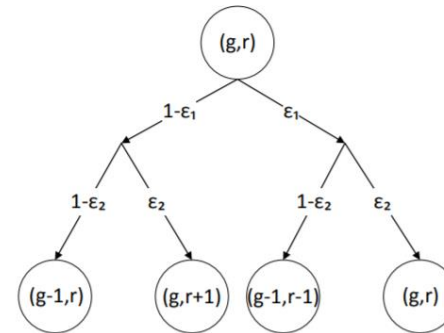


Sharkfin Problem

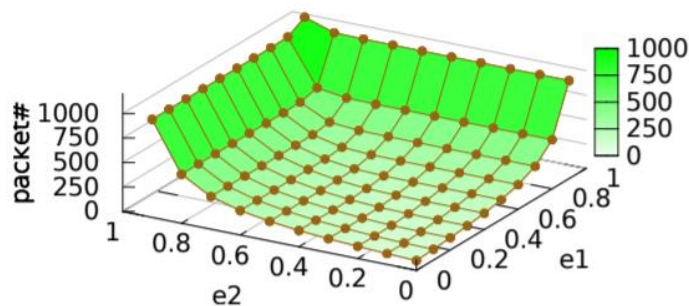
- Adjustment of the theory presented in <https://arxiv.org/abs/1601.03201>



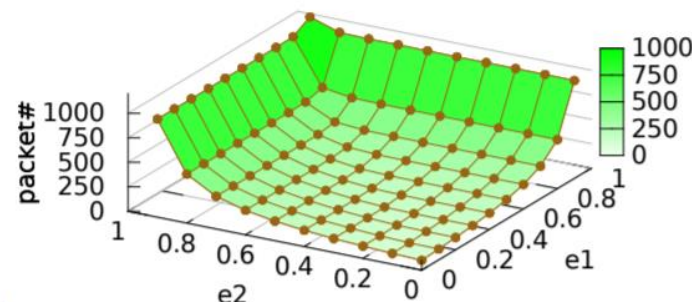
(a) State transition graph when the recoder is empty.



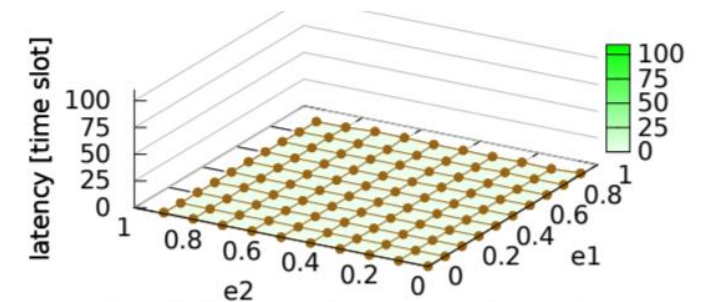
(b) State transition graph when the recoder has at least one linearly independent packet.



(a) Latency calculated from recursive formula in Eq. (7).



(b) Latency calculated from simulations.



(c) The difference between the values calculated from the recursive formula in Eq. (7) and the simulations.

The Recoding Advantage

Optimal and Dynamic Loss Compensation

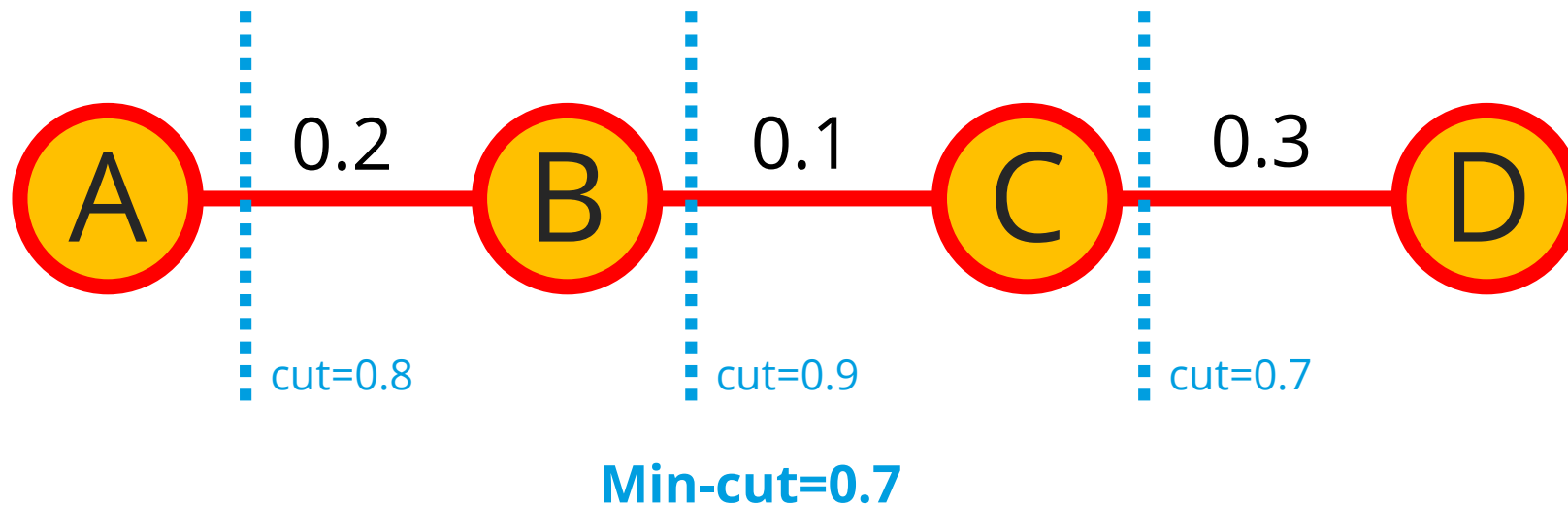
- Uncoded ~50%
- Coded
 - Hop by hop: Efficient but creates delay
 - E2E (A → D) plan for the worst link (30%)
 - RLNC: Recoding (as efficient as hop by hop without delay)



The Recoding Advantage

Optimal and Dynamic Loss Compensation

- Uncoded ~50%
- Coded
 - Hop by hop: Efficient but creates delay
 - E2E (A → D) plan for the worst link (30%)
 - RLNC: Recoding (as efficient as hop by hop without delay)



The Recoding Advantage

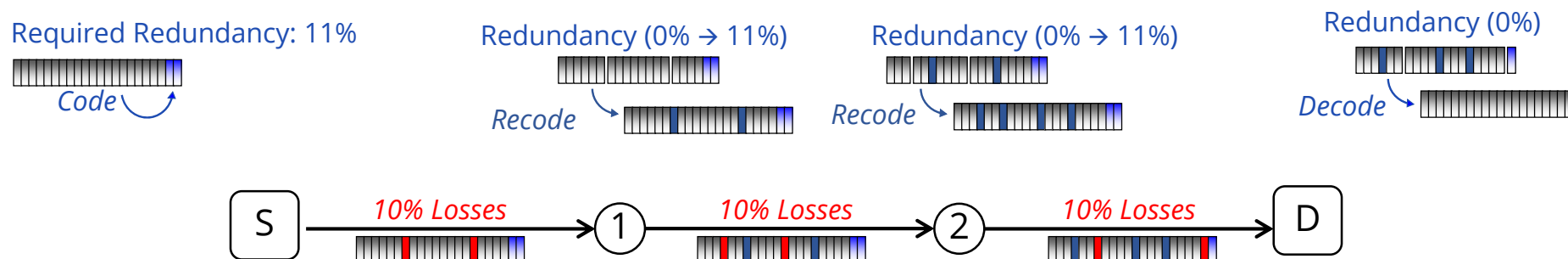
Optimal and Dynamic Loss Compensation



Coding End-to-End Overhead = Cumulative Losses (37%)



Re-Coding Overhead = Single Worst-Case Loss (11%)





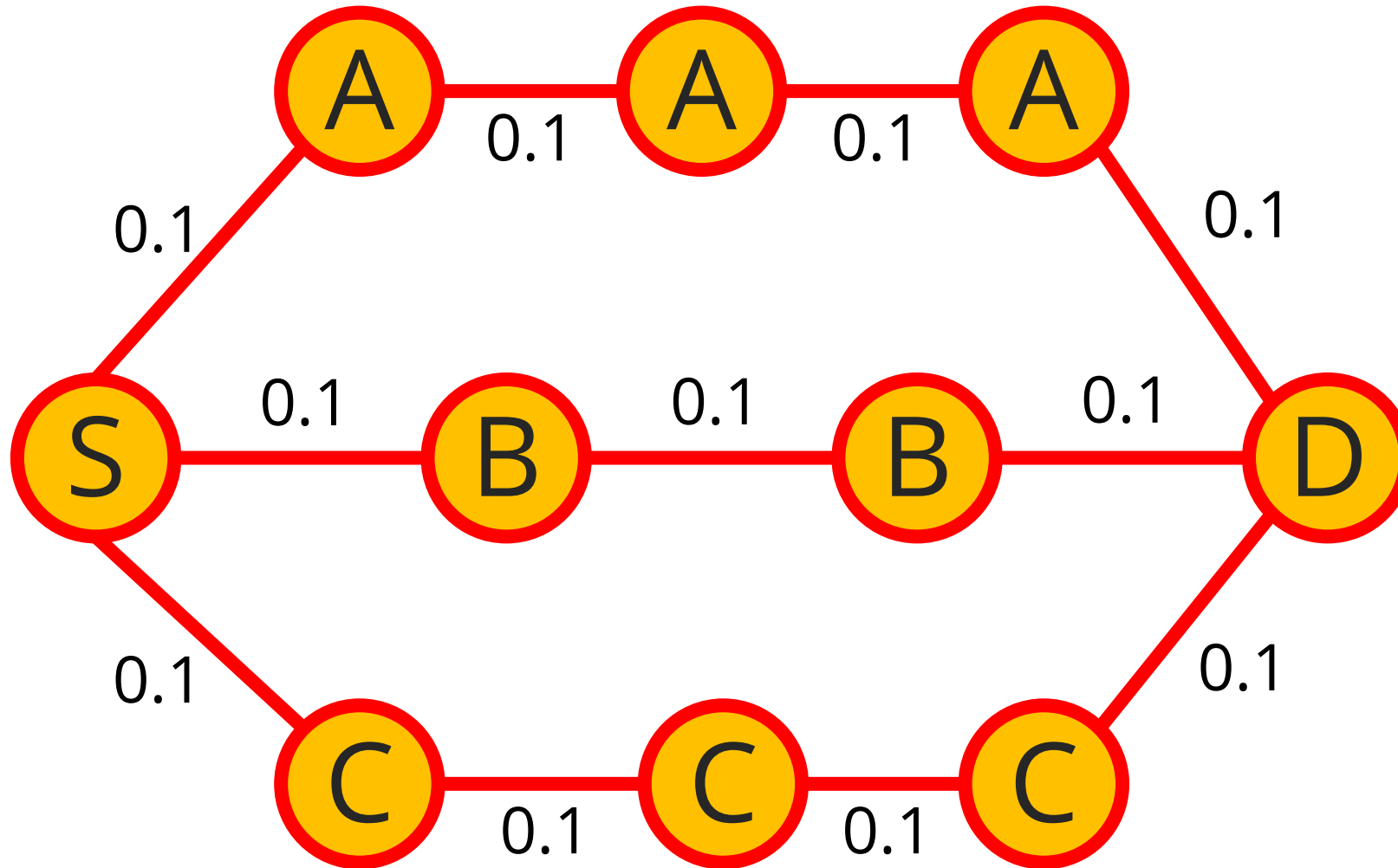
- Kodo_Multihop_Example.ipynb example
- $\text{Gain} = \# \text{timeslots_of_repeater} / \# \text{timeslots_of_recoder}$



#Relays	1	2	3	4	5	6	7	8	9
Repeater									
Recoder									
Gain									

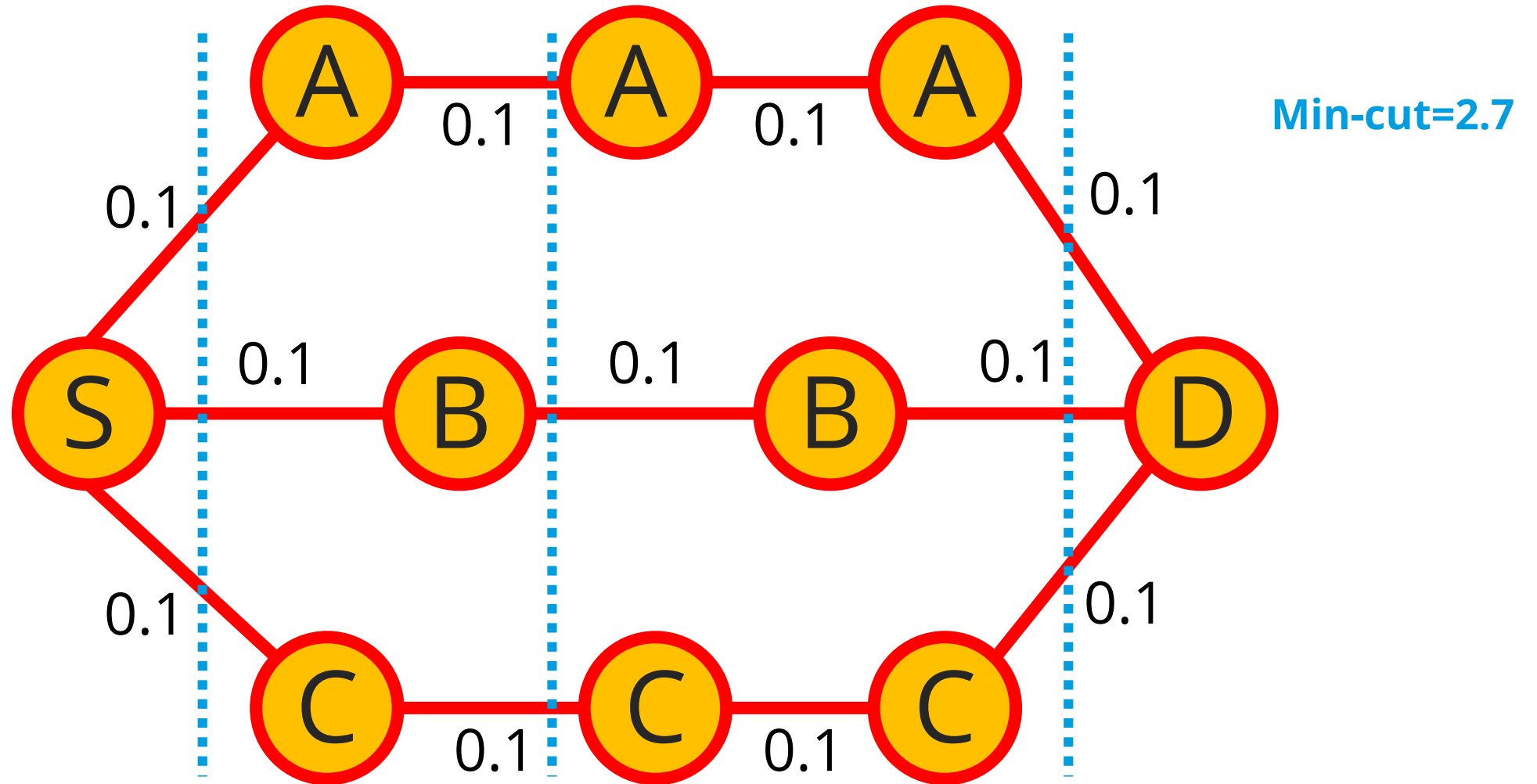
The Recoding Advantage

Optimal and Dynamic Loss Compensation



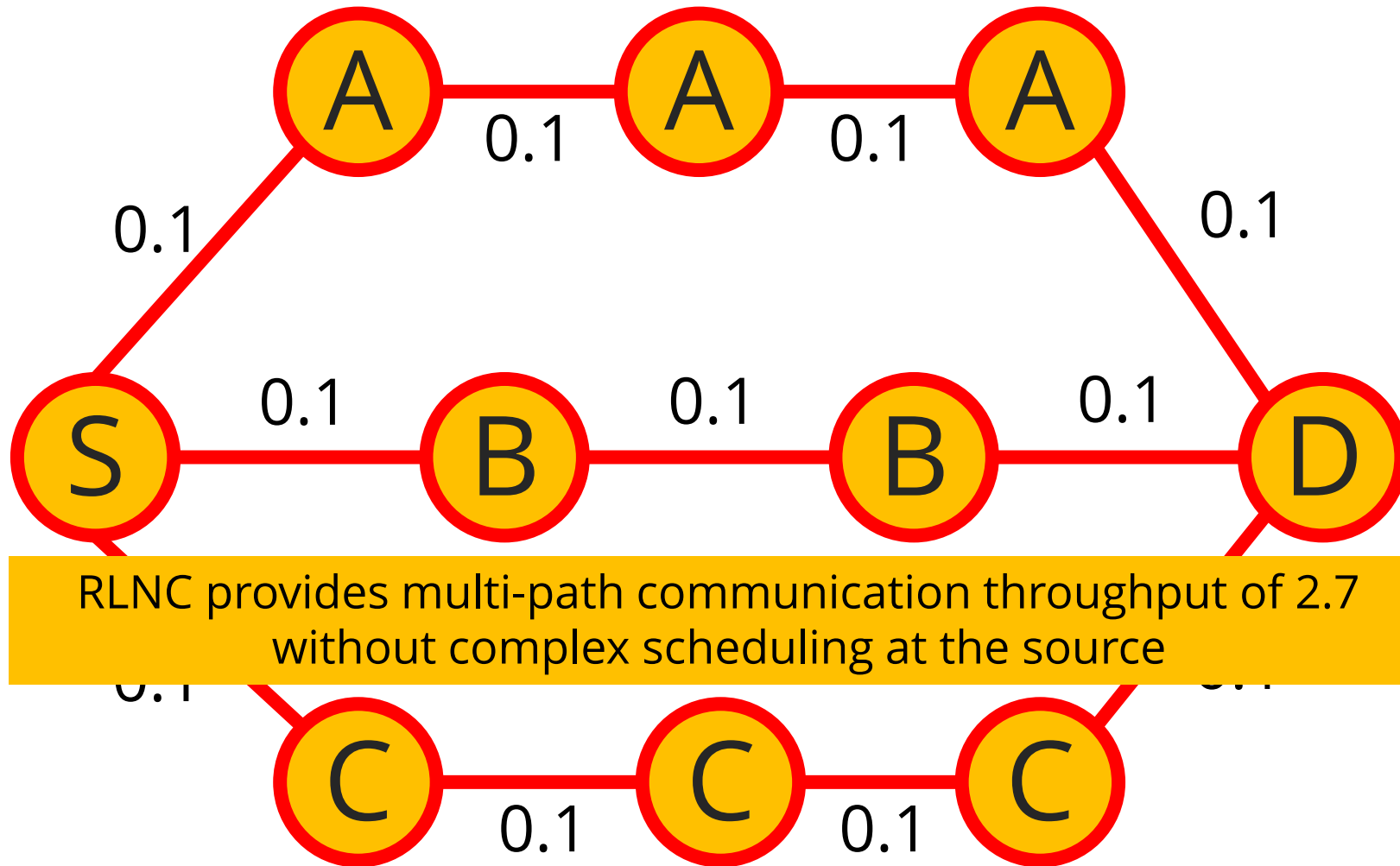
The Recoding Advantage

Optimal and Dynamic Loss Compensation



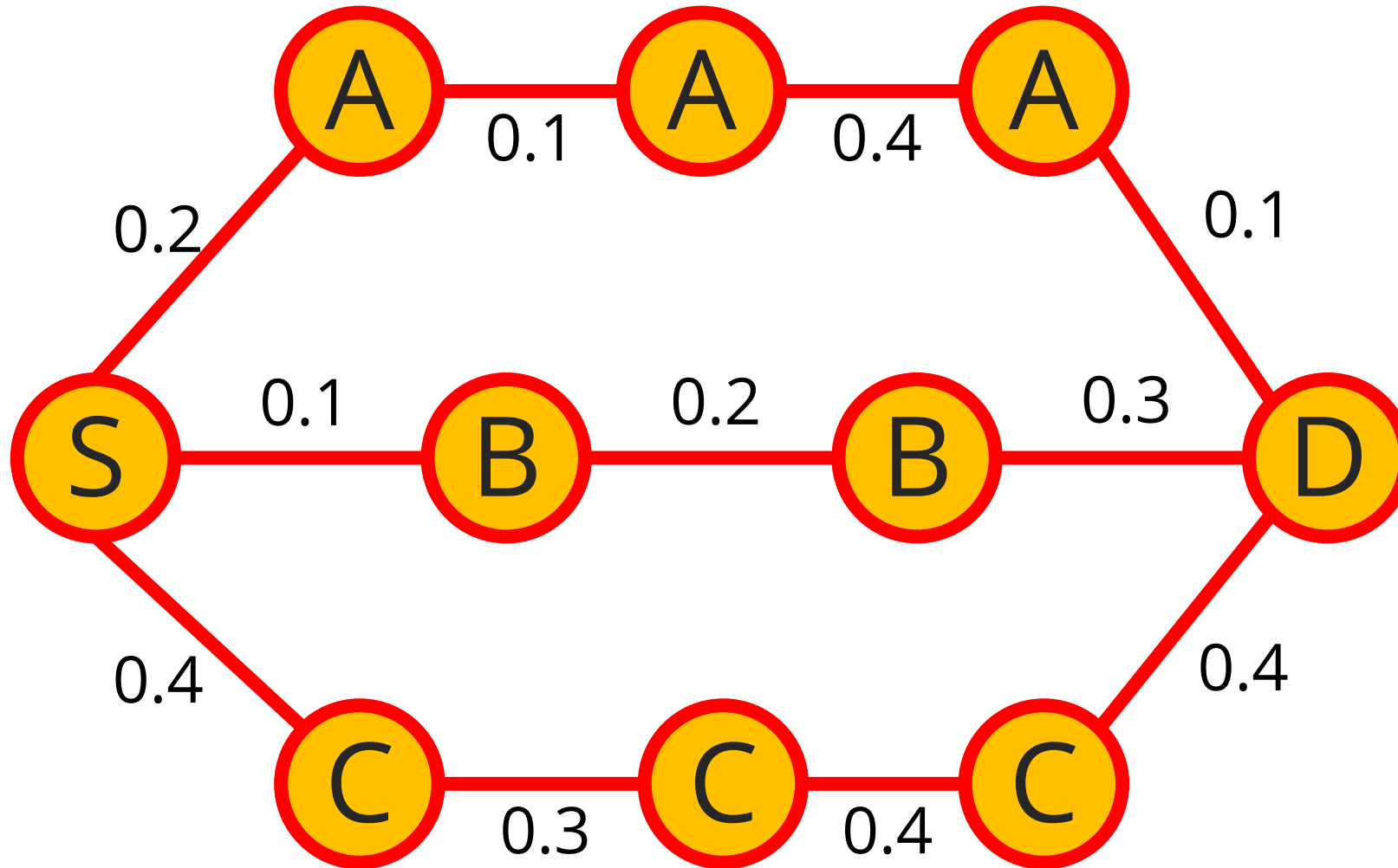
The Recoding Advantage

Optimal and Dynamic Loss Compensation



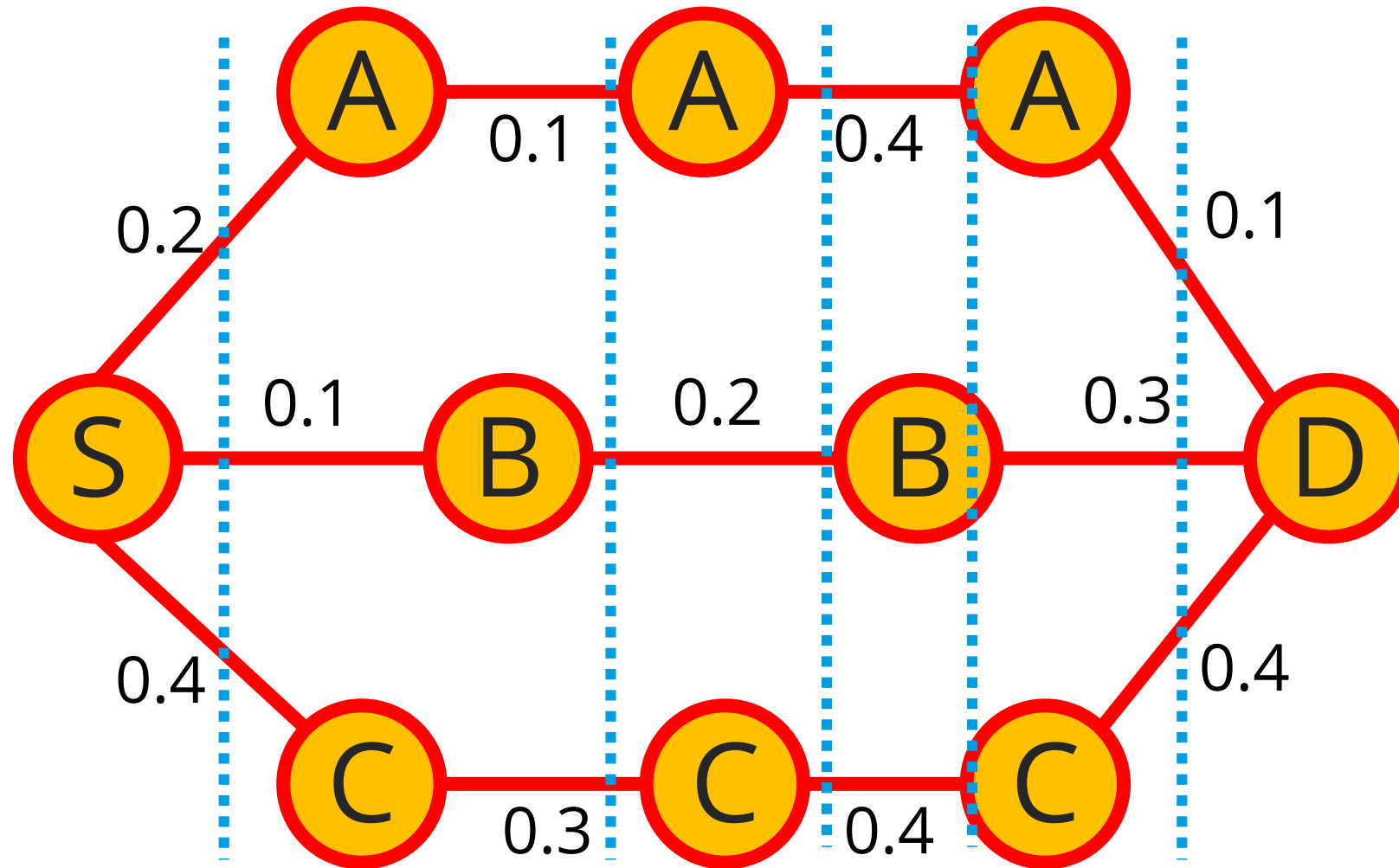
The Recoding Advantage

Optimal and Dynamic Loss Compensation



The Recoding Advantage

Optimal and Dynamic Loss Compensation



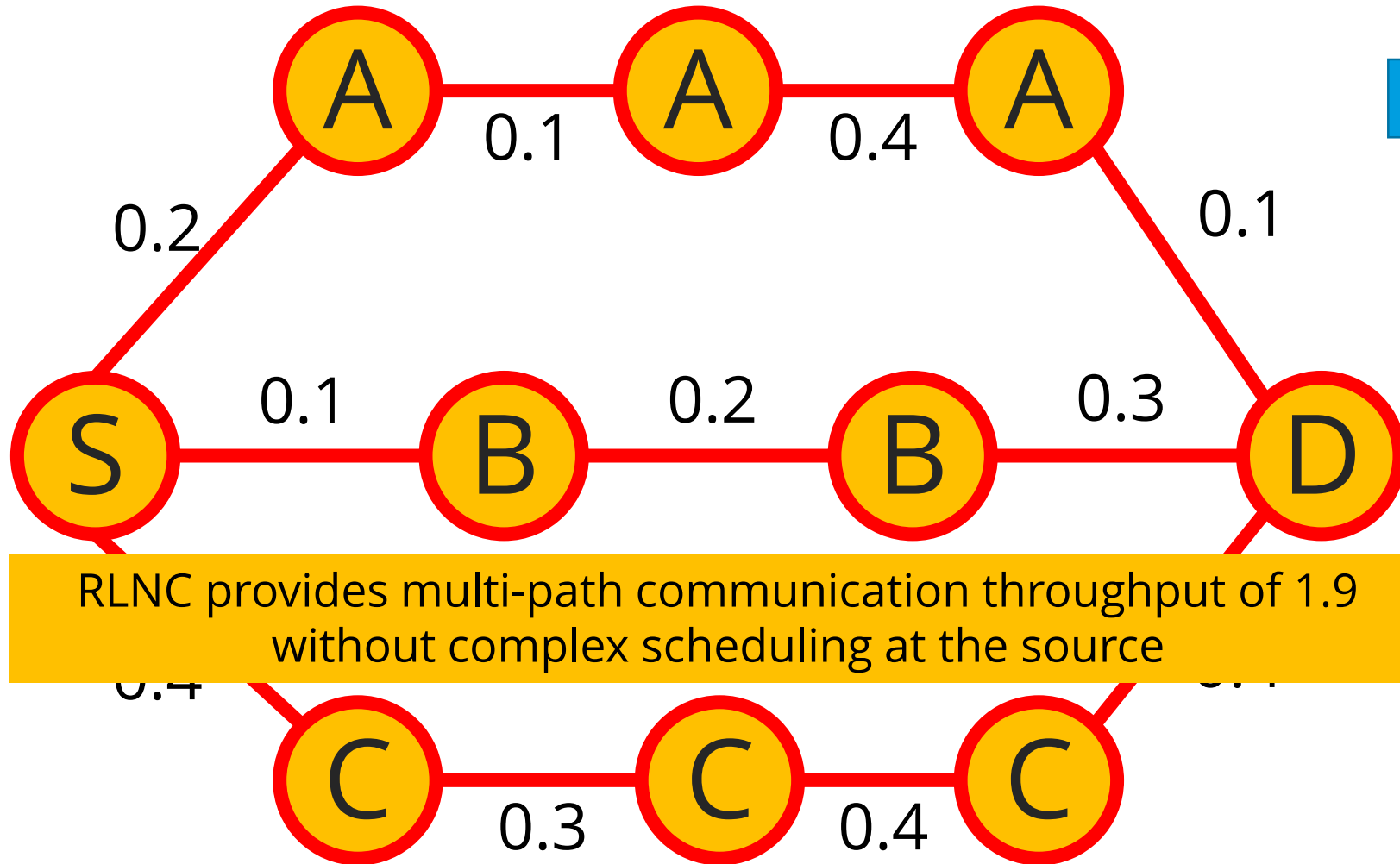
Min-cut=1.9

The Recoding Advantage

Optimal and Dynamic Loss Compensation



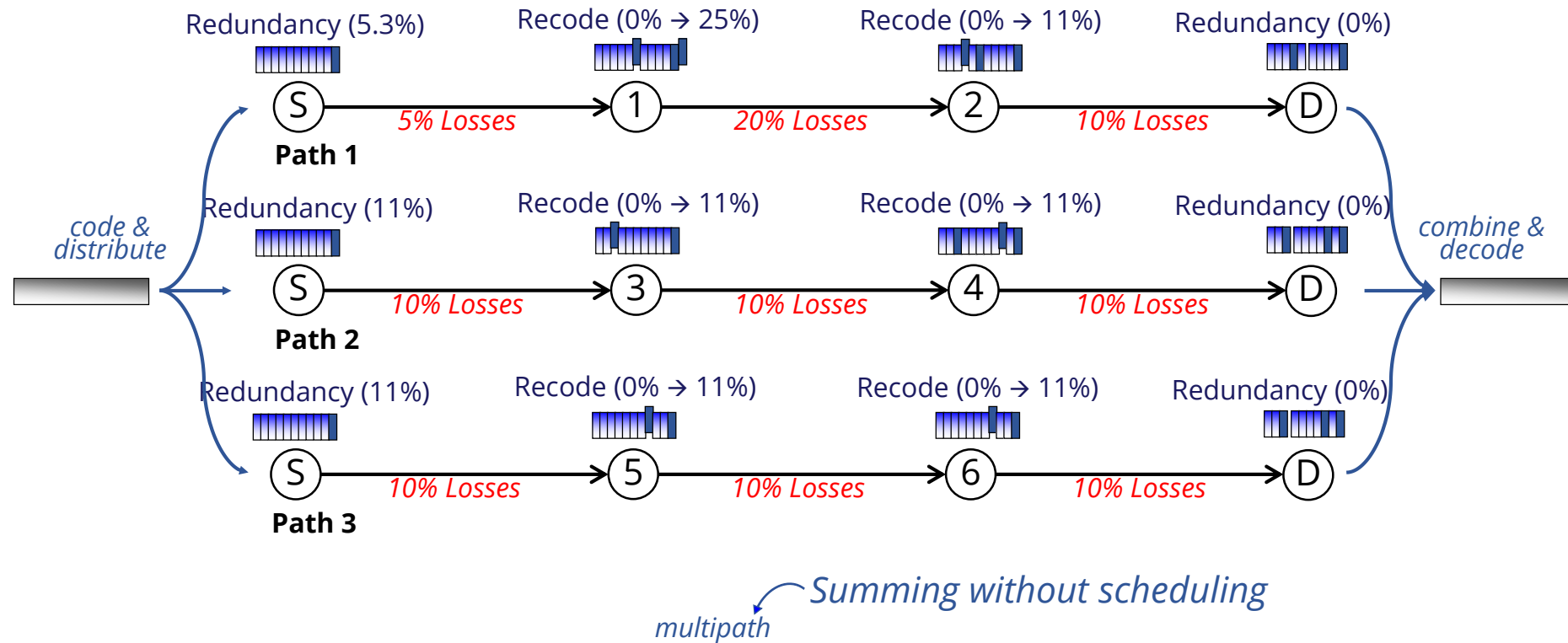
Tip: Sharkfin





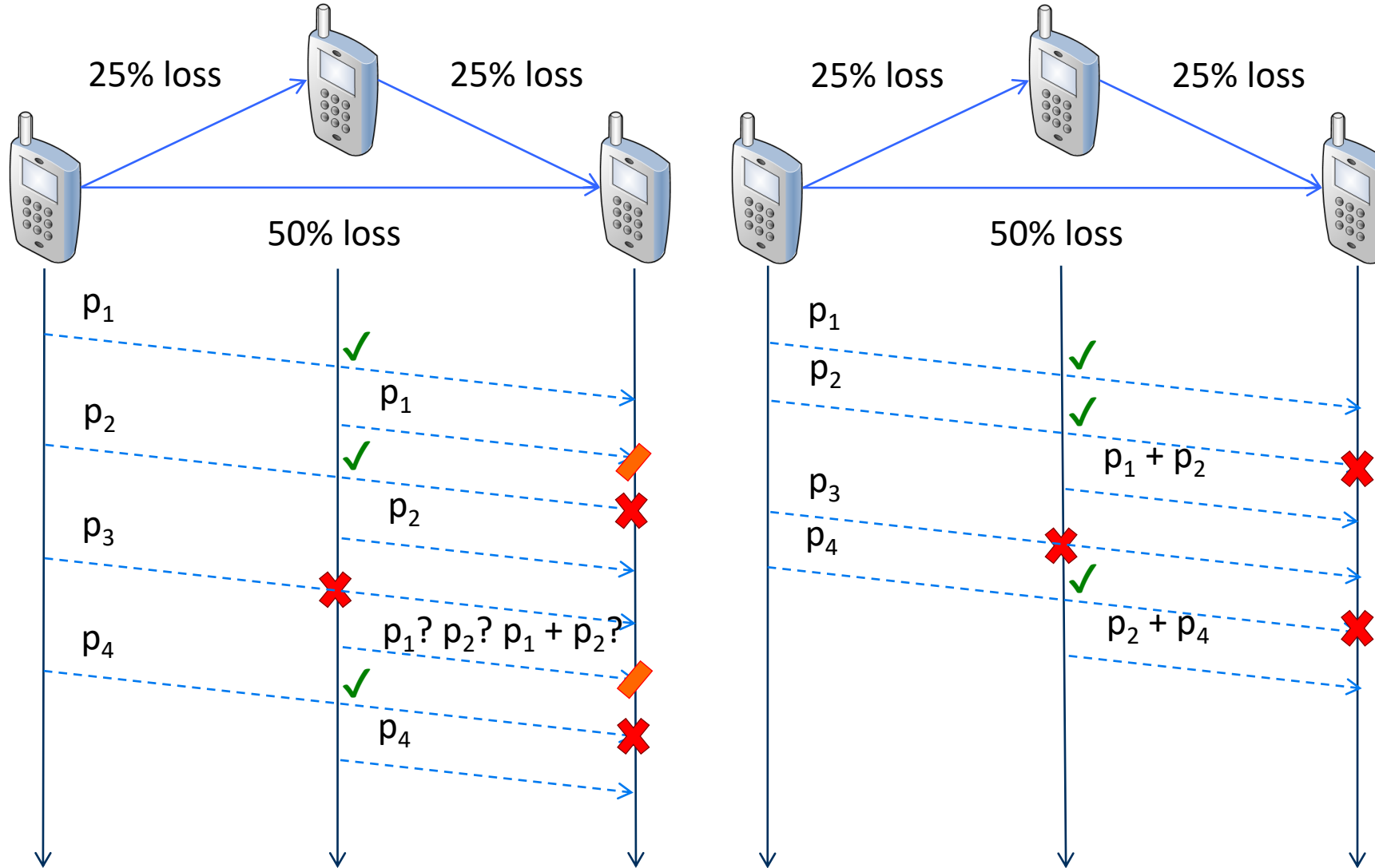
The Recoding Advantage plus Multipath Advantage

Native Bandwidth Aggregation



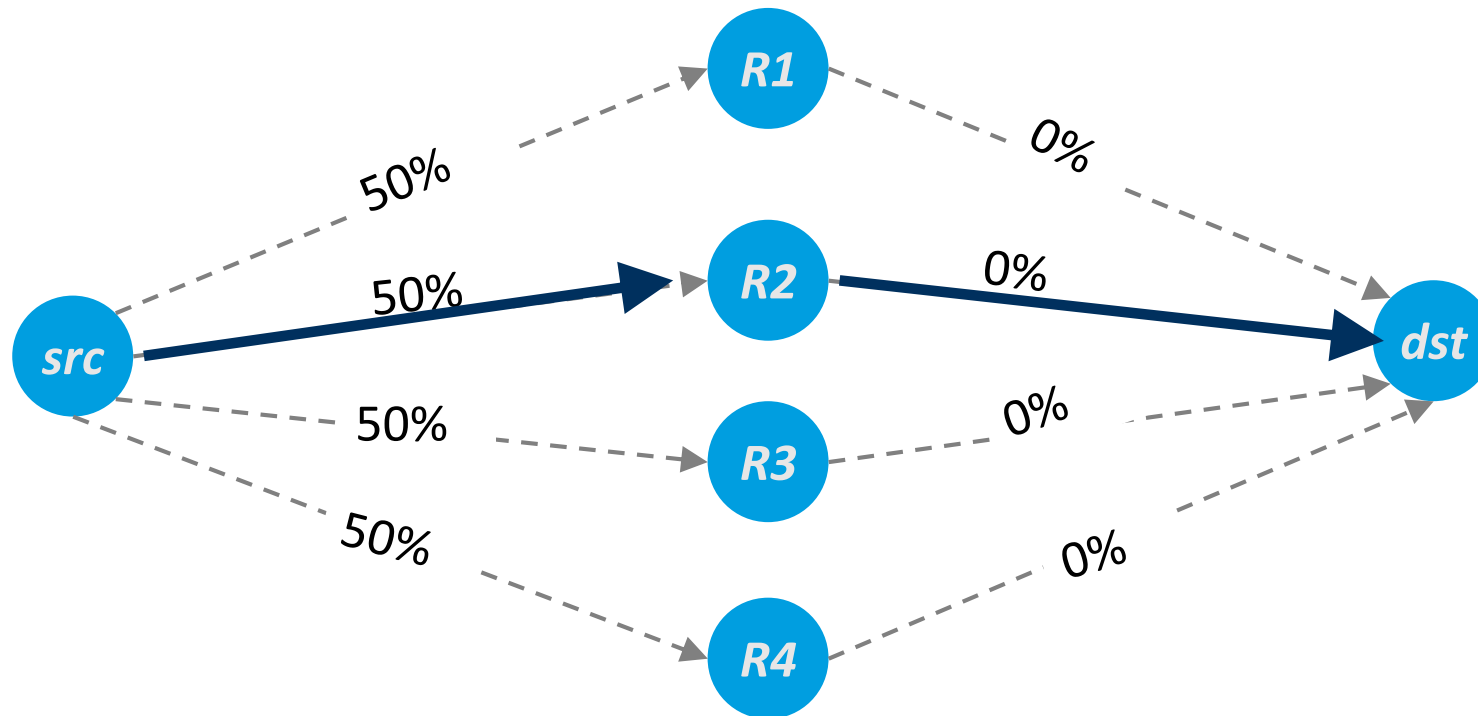
- Total Throughput = \sum (path rate - worst link)
- Inject 10Mbps at each path → Obtain 26Mbps seamlessly

Other Recoding Issues



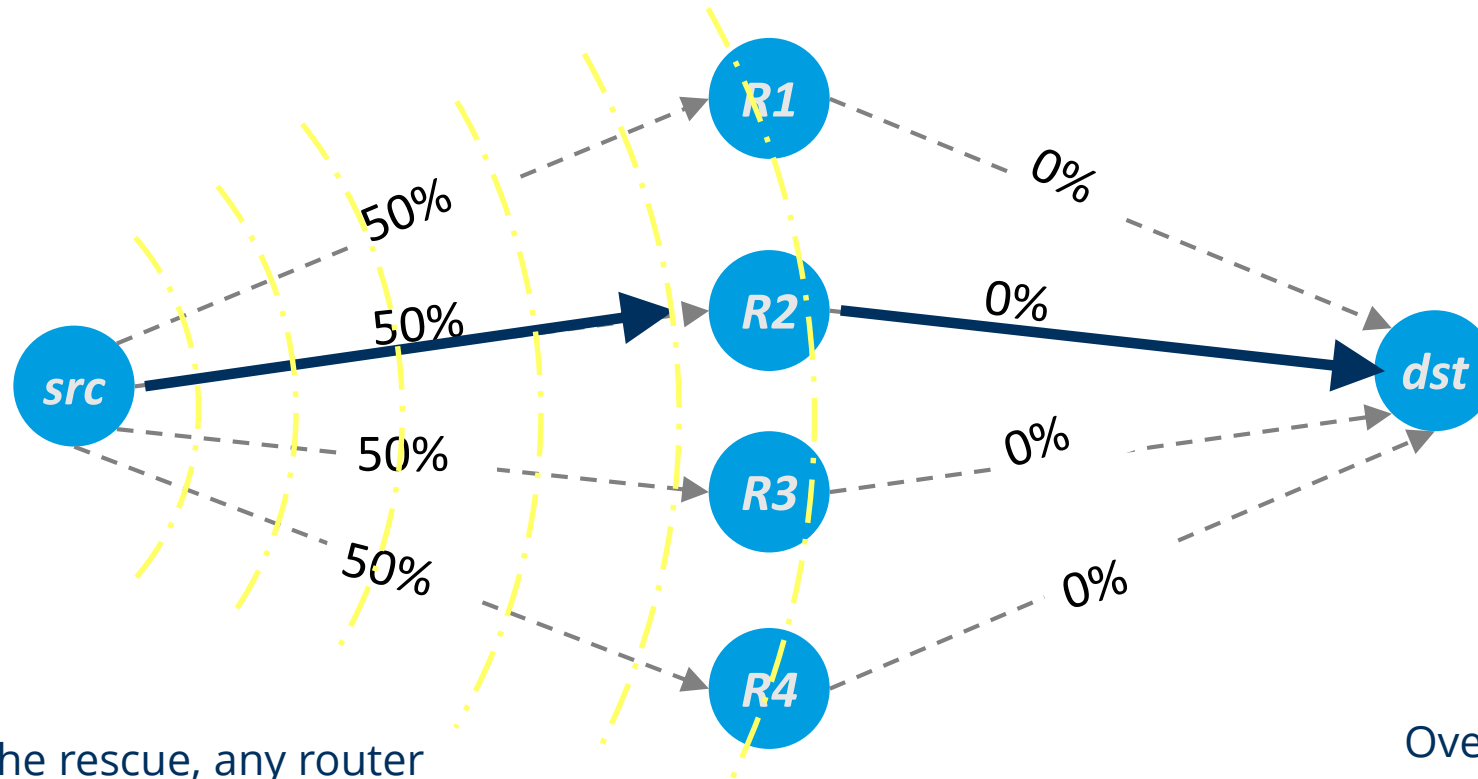
Other Recoding Issues

Best single path \rightarrow Prob. of loss 50%



Other Recoding Issues

Best single path \rightarrow Prob. of loss 50%

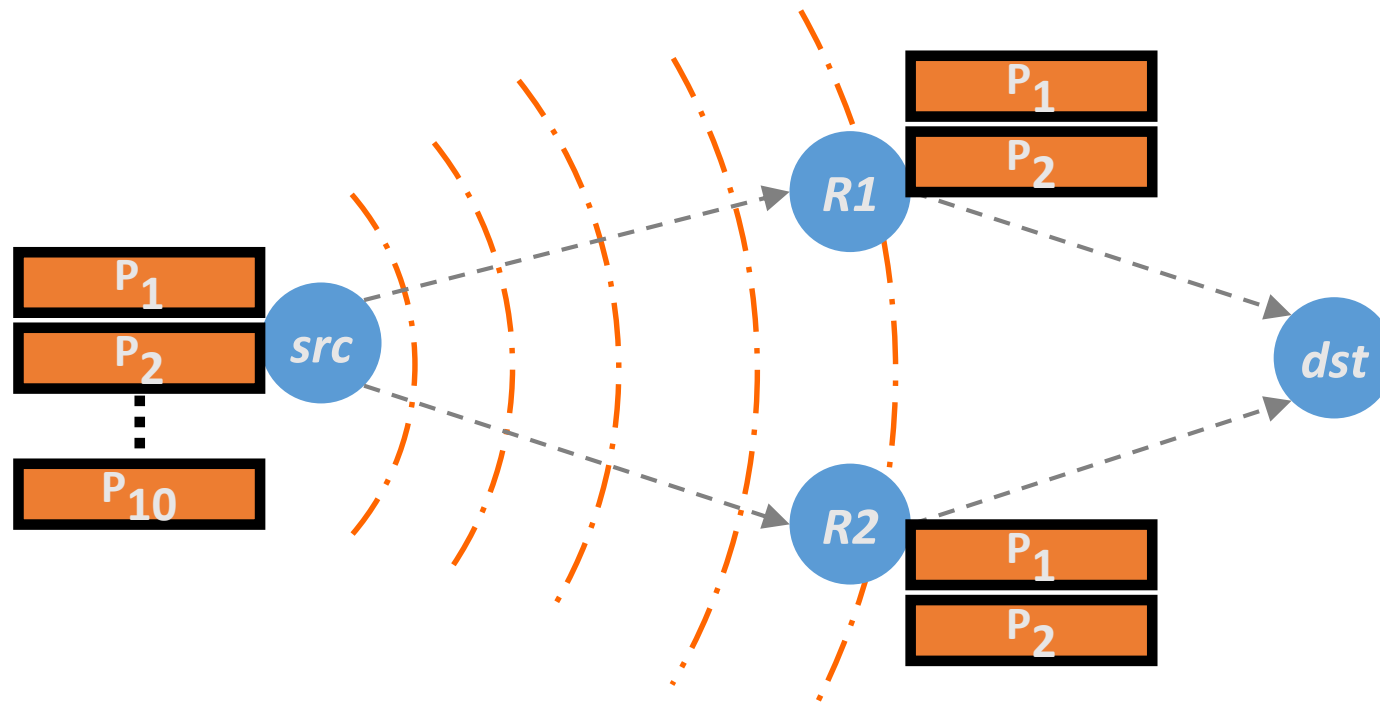


Spatial diversity to the rescue, any router forward packet \rightarrow Prob. of loss $0.54 = 6\%$

Overlap in received packets \rightarrow Routers forward duplicates

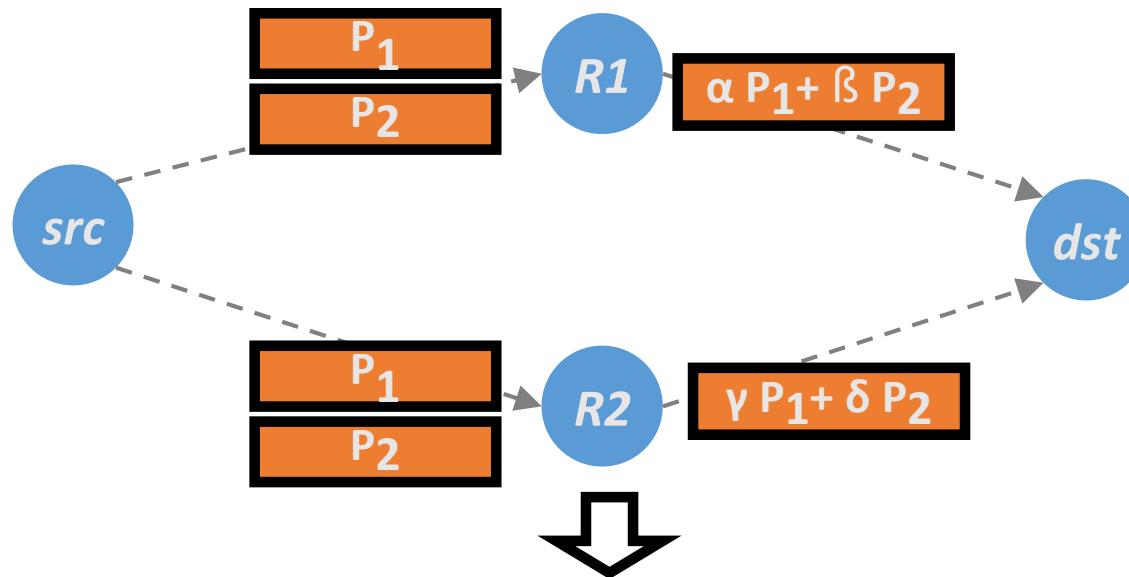
Challenge with Using Spatial Diversity

Overlap in received packets → Routers forward duplicates



Challenge with Using Spatial Diversity

Each router forwards random combinations of packets

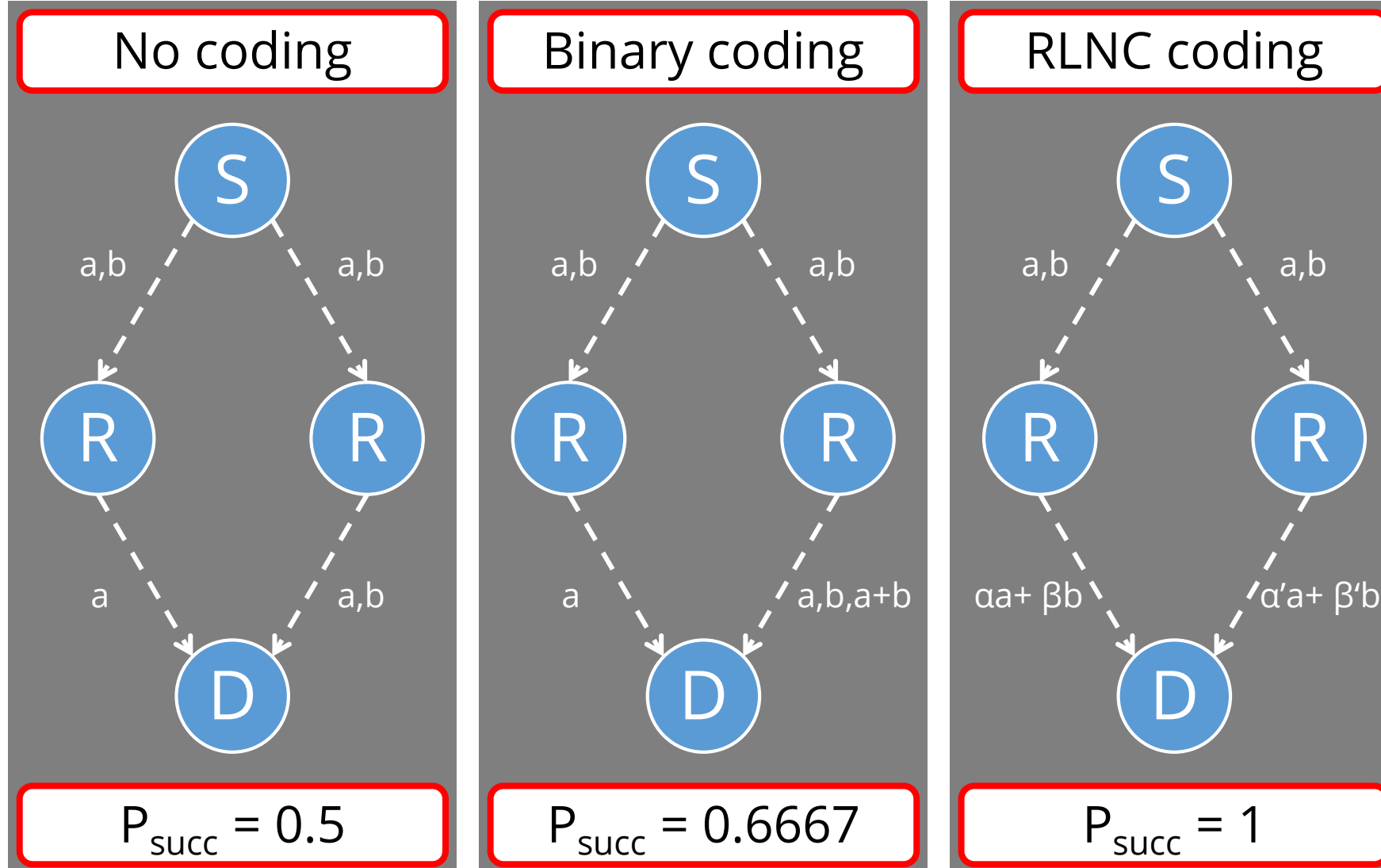


Randomness prevents duplicates

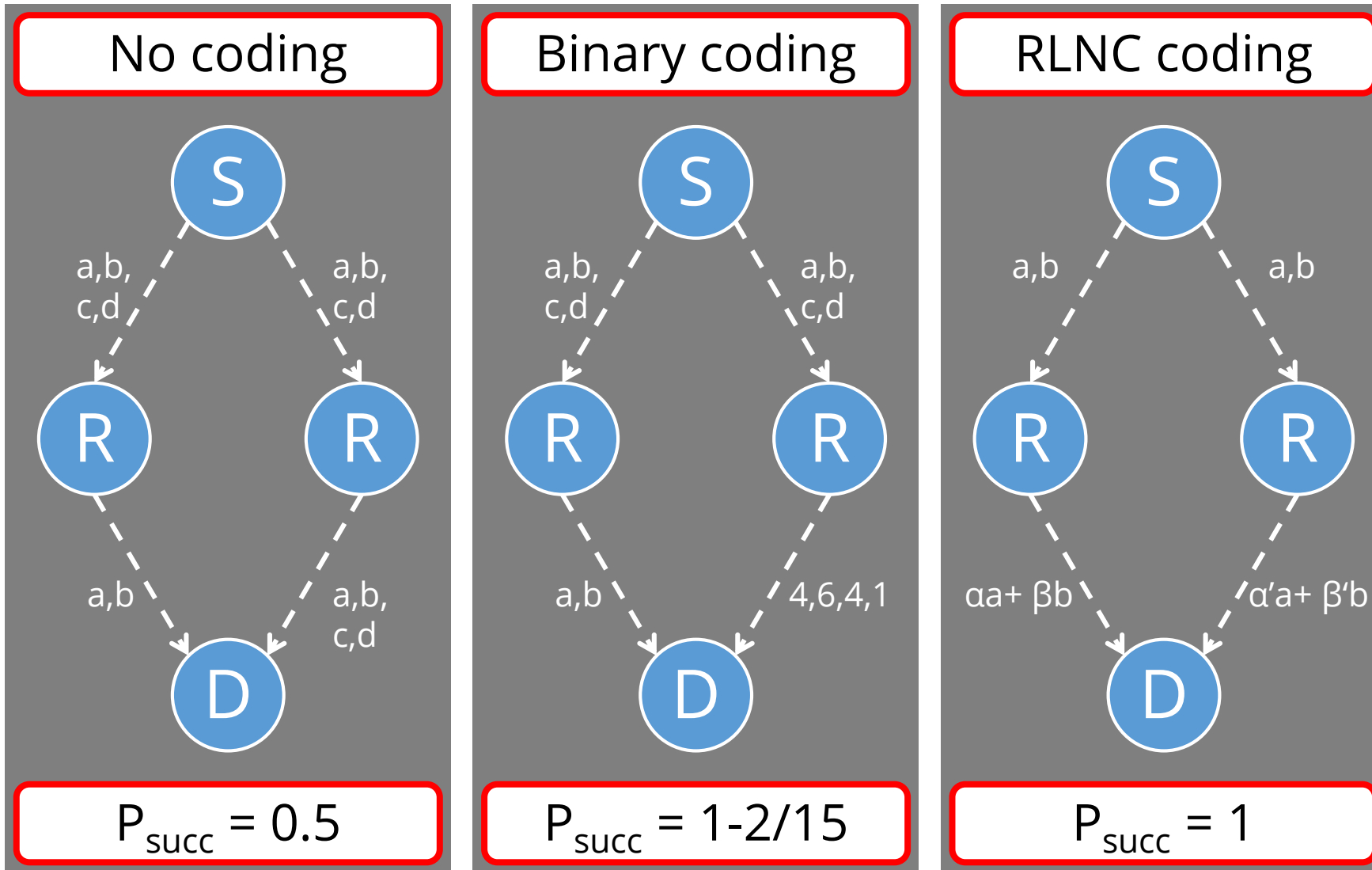


Network coding exploits spatial diversity to improve dead spots

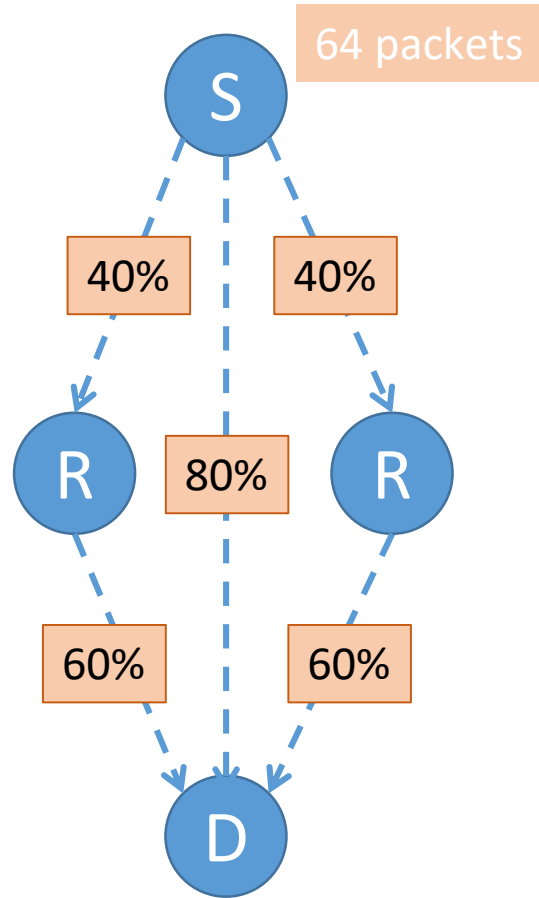
Impact of Recoding



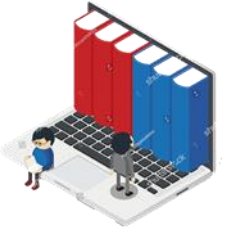
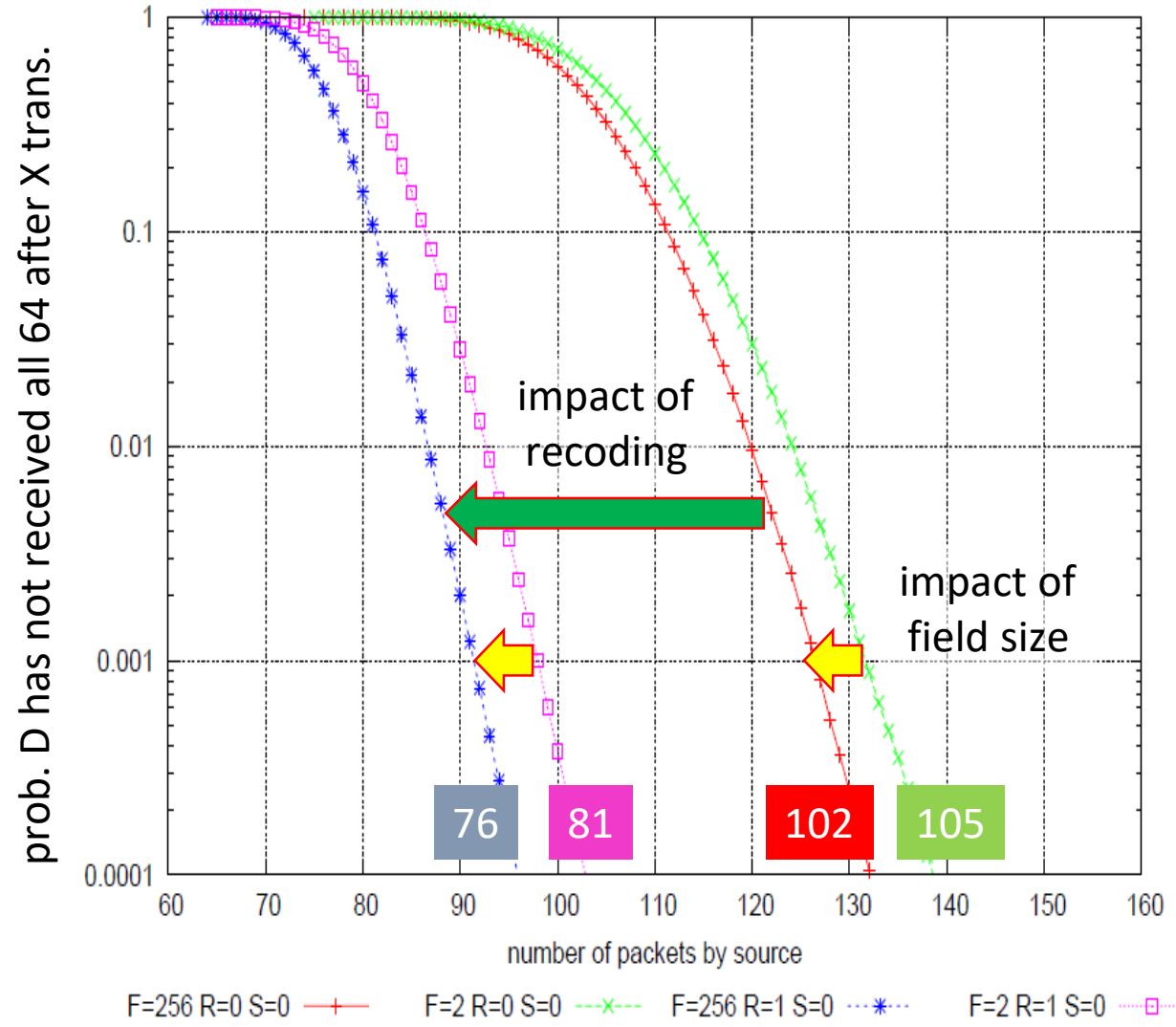
Impact of Recoding



Impact of Recoding

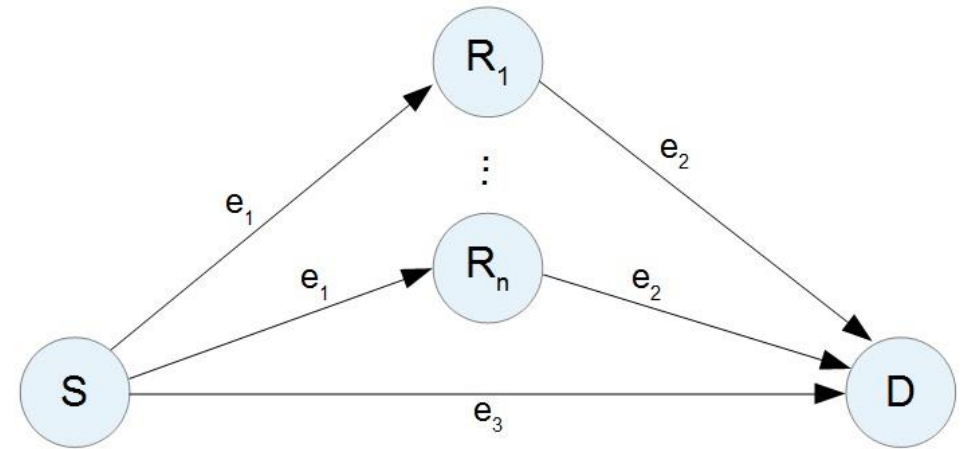


No need for signalling!





- Kodo_Multipath_Example.ipynb example

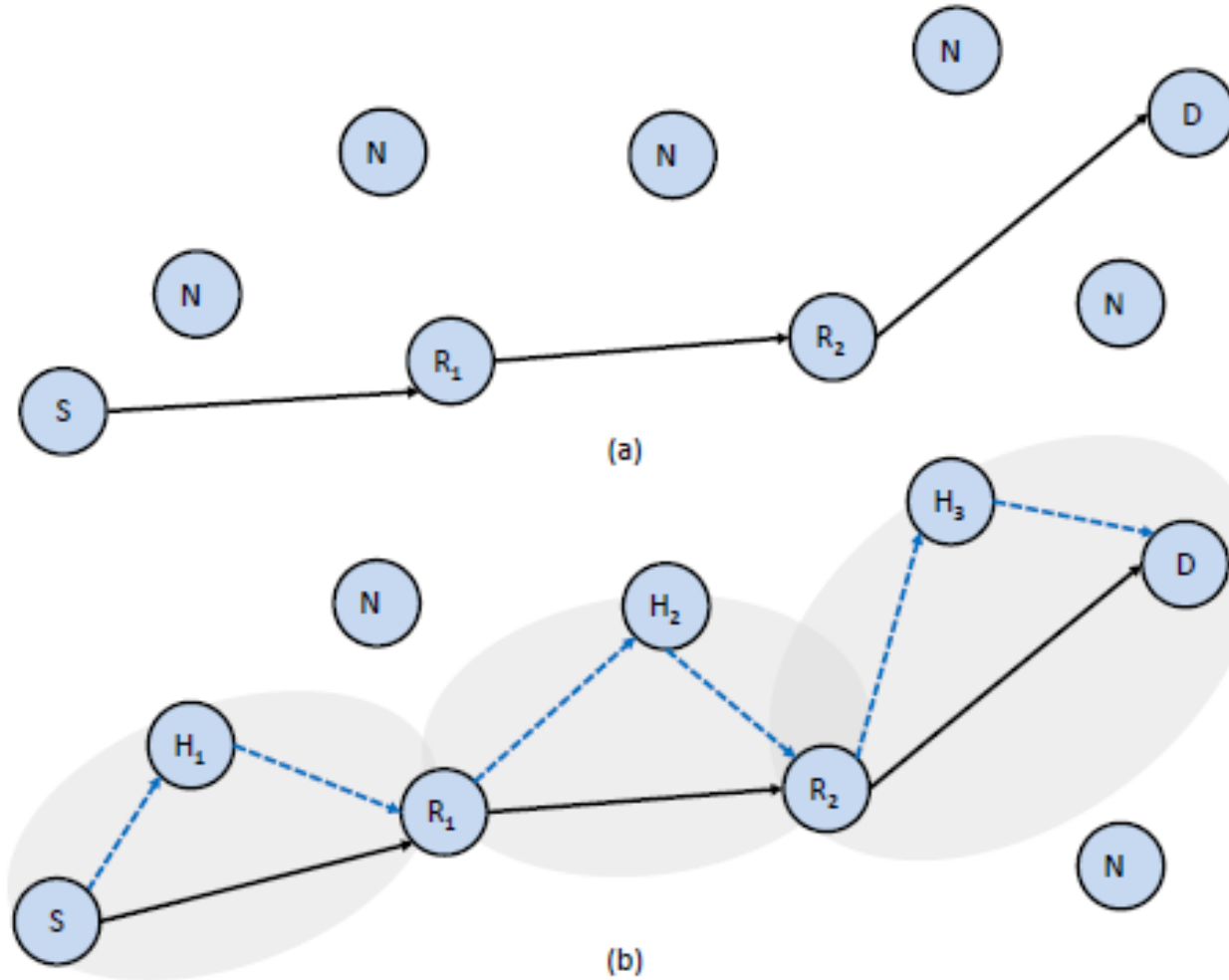


#Relays	1	2	3	4	5	6	7	8	9
Repeater									
Recoder									
Gain									

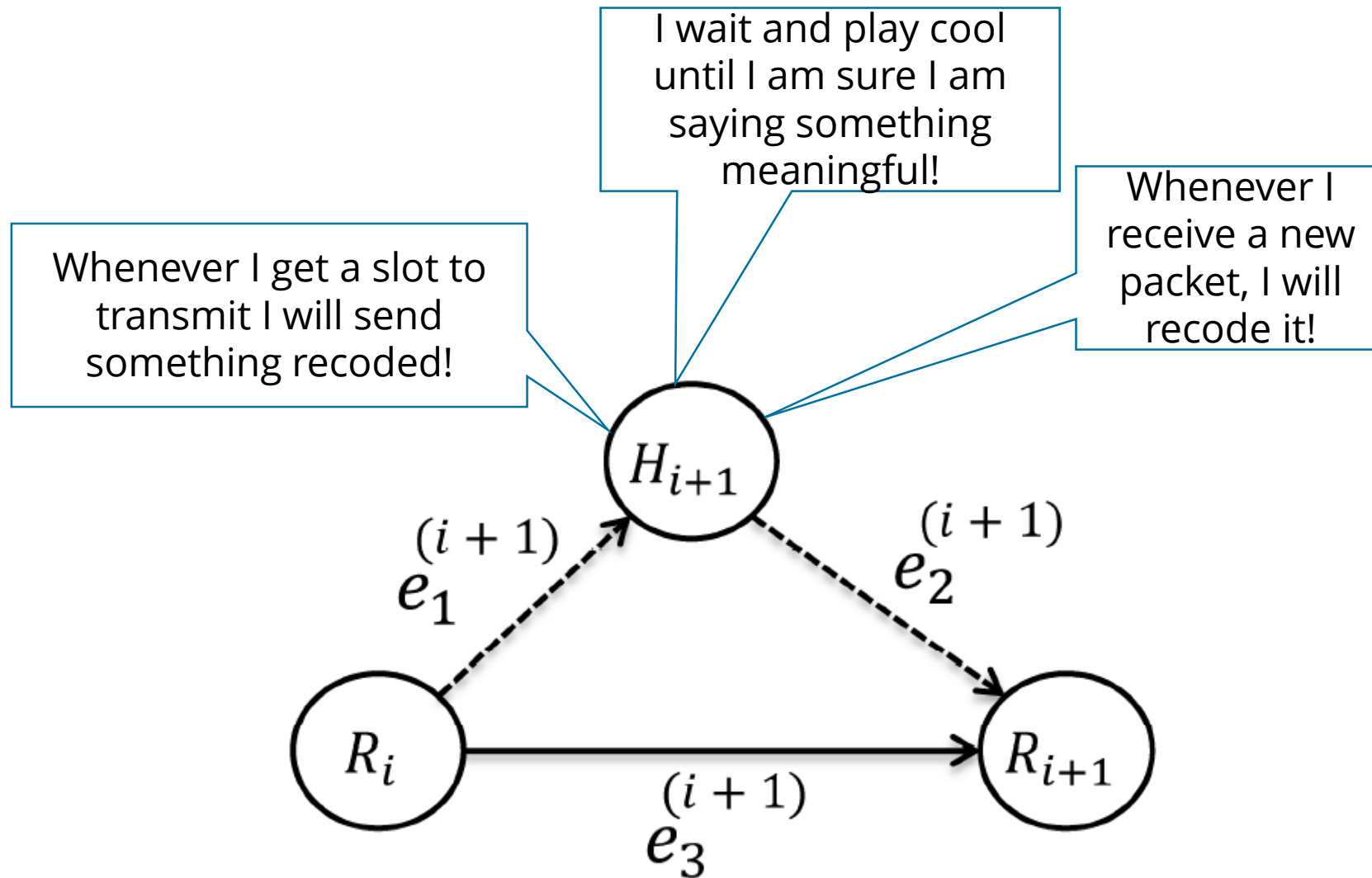
Impact of the protocol design

P. Pahlavani, D.E. Lucani, M.V. Pedersen, and F.H.P. Fitzek, "PlayNCool: Opportunistic Network Coding for Local Optimization of Routing in Wireless Mesh Networks," in Globecom 2013 Workshop - First International Workshop on Cloud-Processing in Heterogeneous Mobile Communication Networks - GLOBECOM 2013, Dec. 2013.

Wireless Mesh

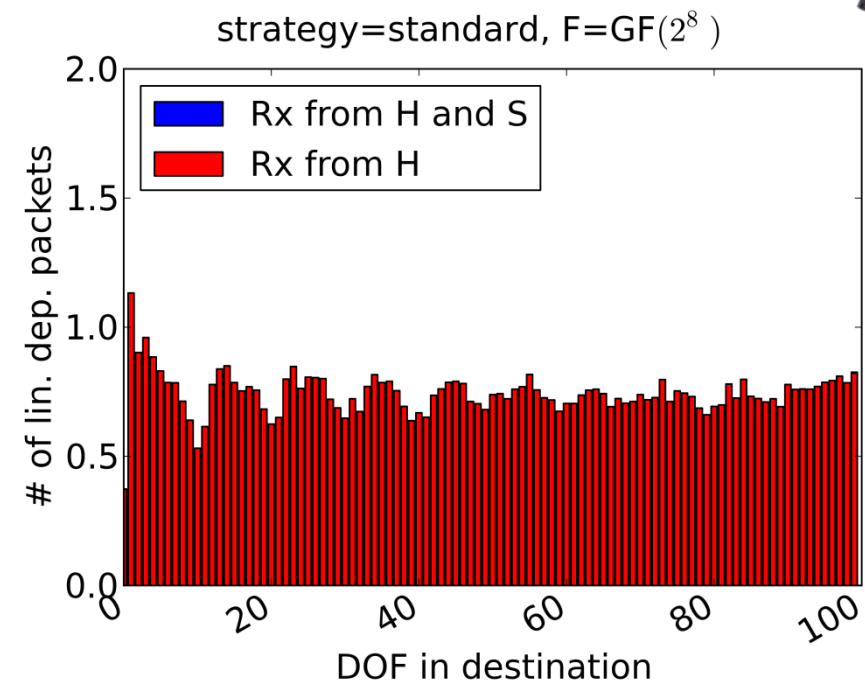
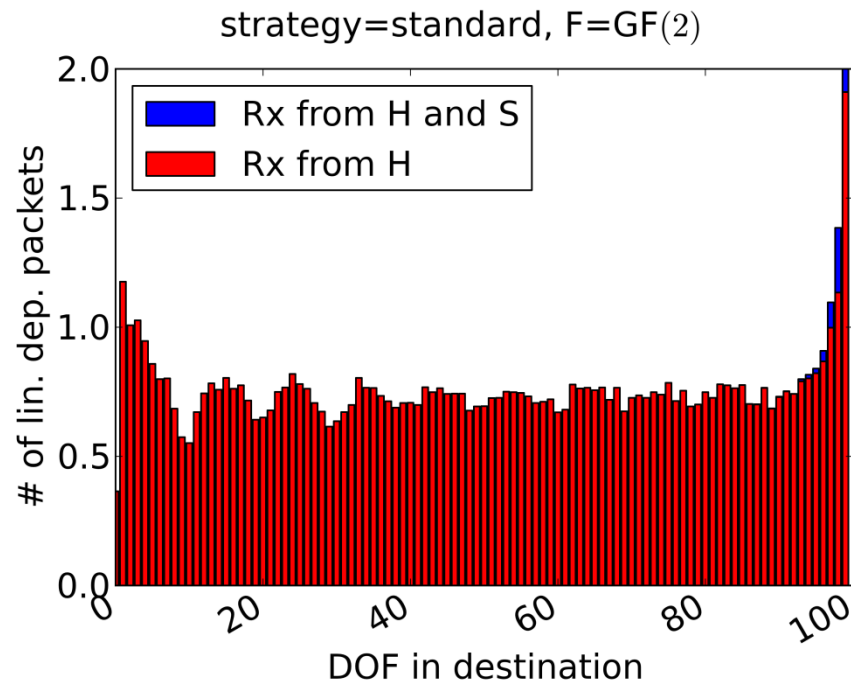
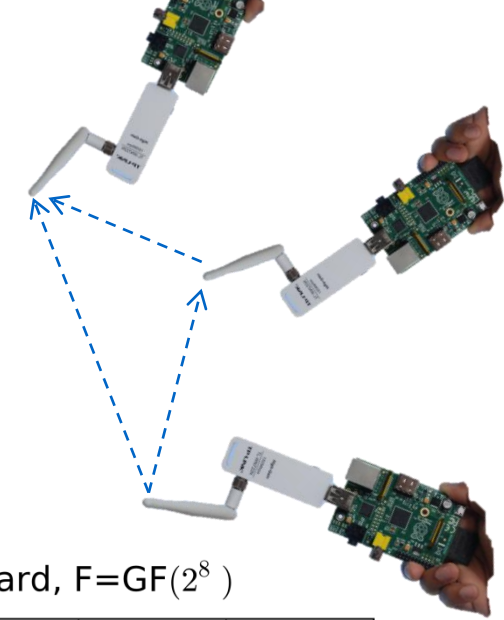
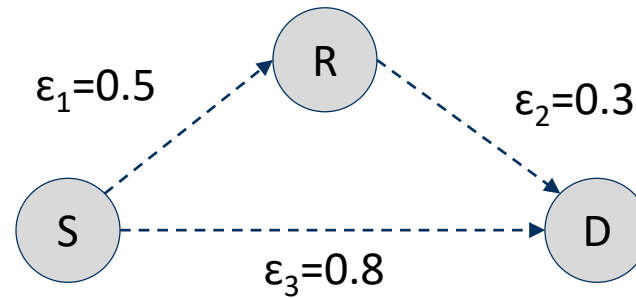


Strategies under Investigation



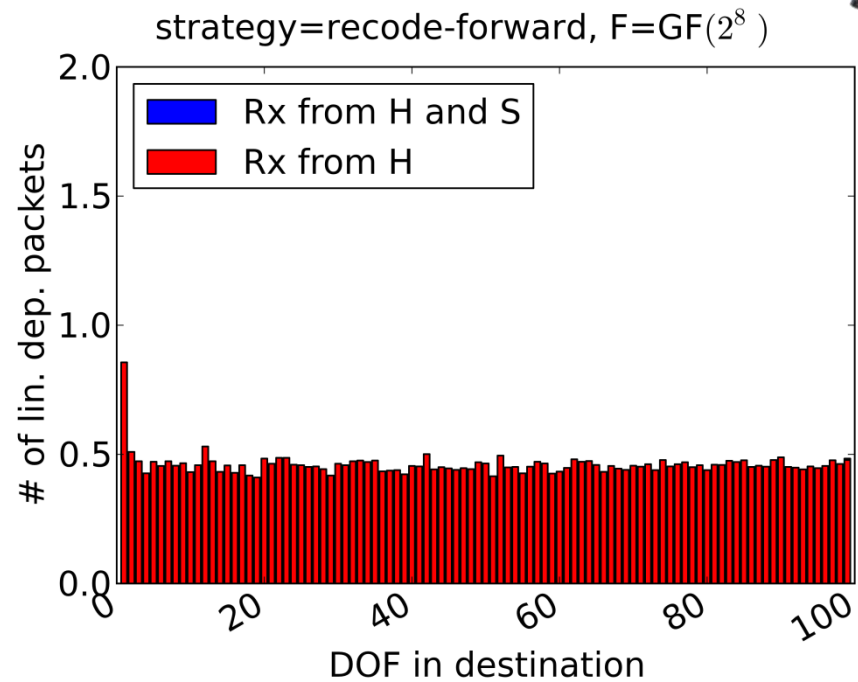
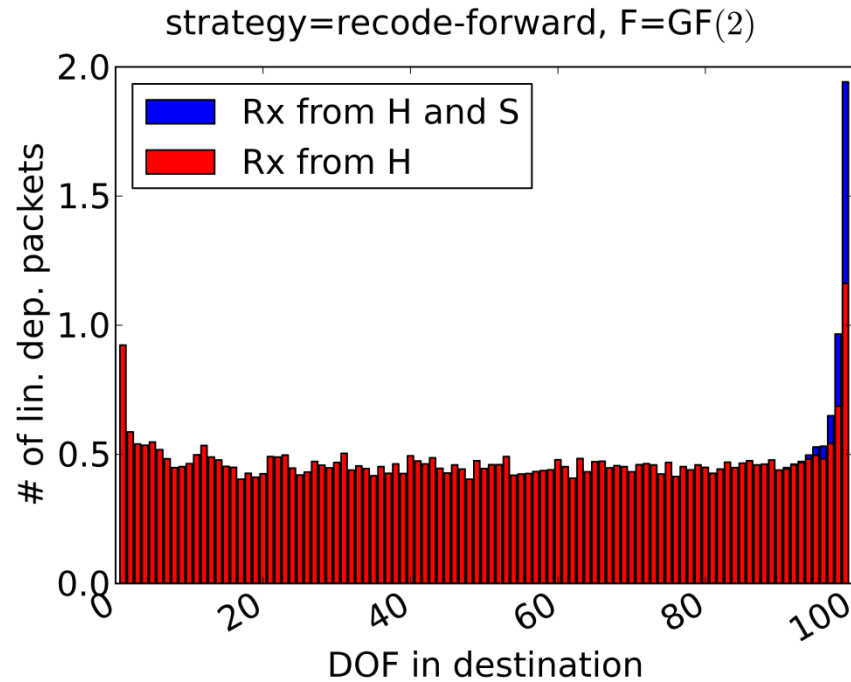
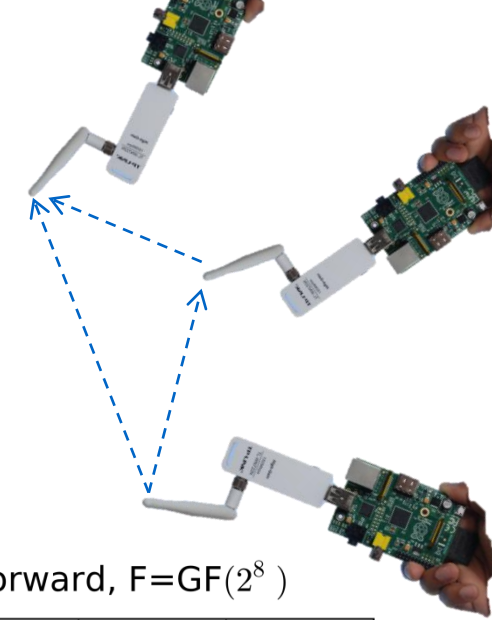
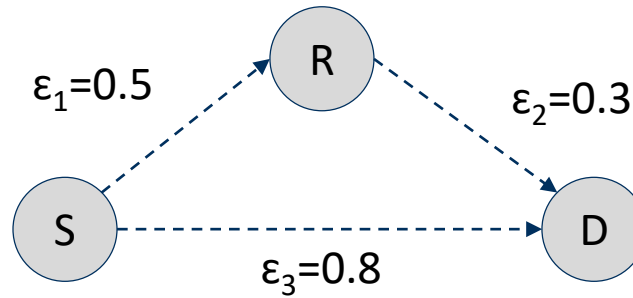
RLNC in real meshed

Agnostic recoding



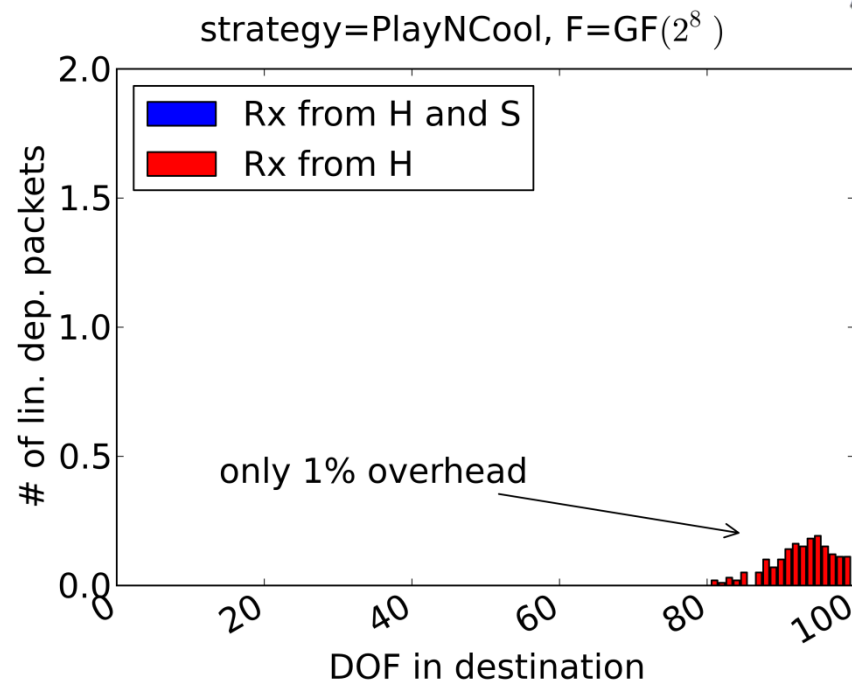
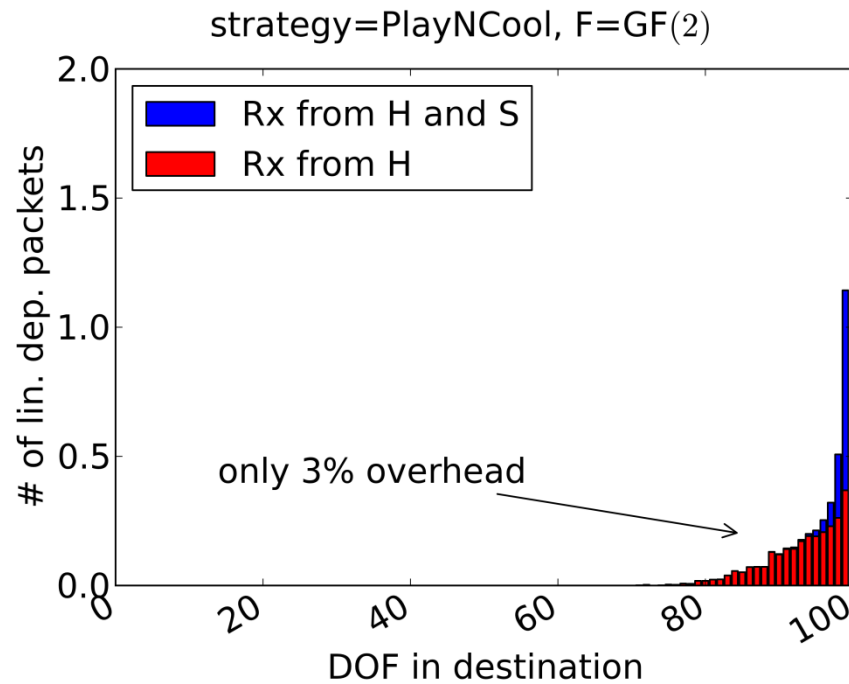
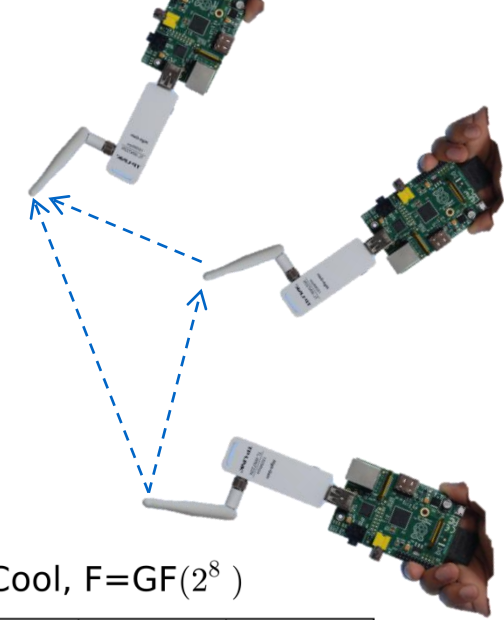
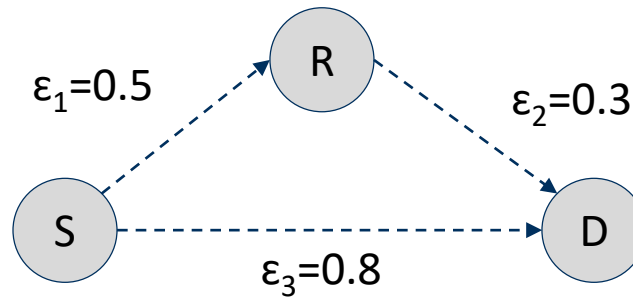
RLNC in real meshed

Rate-sensitive recoding



RLNC in real meshed

PlayN Cool



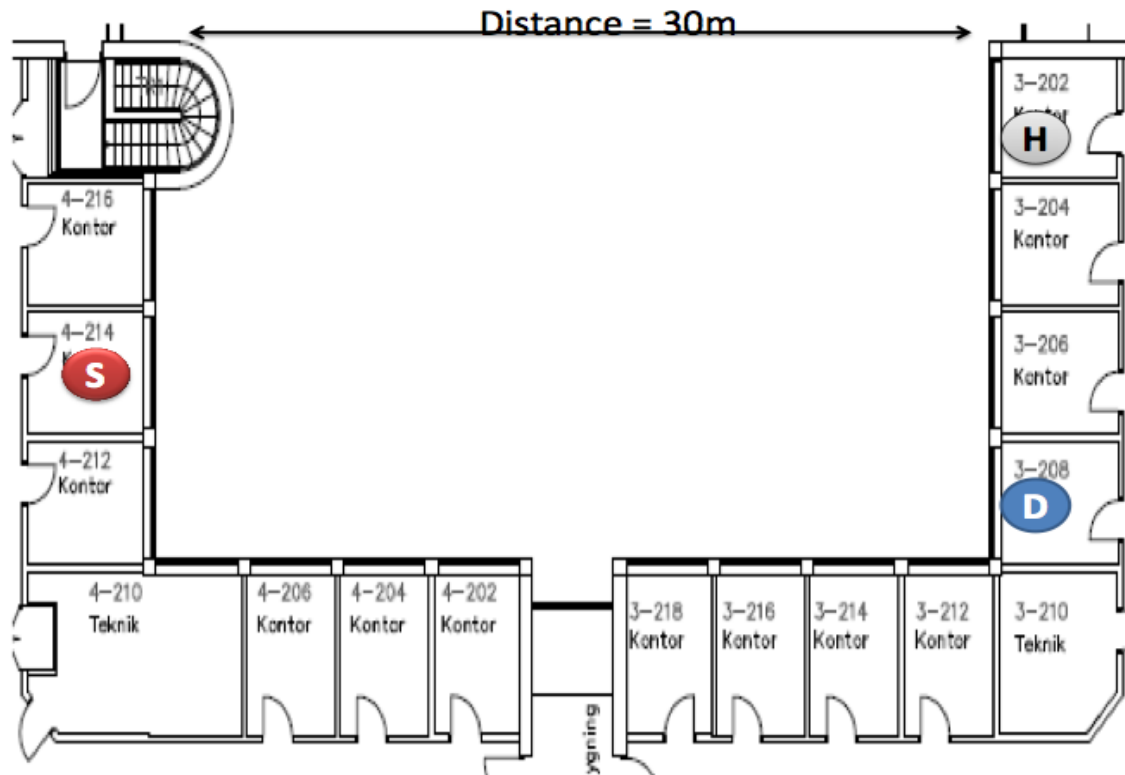
RLNC in real meshed

PlayN Cool

- Link quality information is inaccurate and not stable over run time
- Link quality information is not always available
- Retrieving the link quality information increases the overhead
- The waiting time (W), time the recoder waits to increase knowledge, is adaptive based on a feedback packet from the destination node.
- Feedback shows the usefulness of the packet
 - W decreases by UPDATE_VALUE if packet is useful
 - W increases by UPDATE_VALUE if RX packet is useless

RLNC in real meshed

PlayN Cool



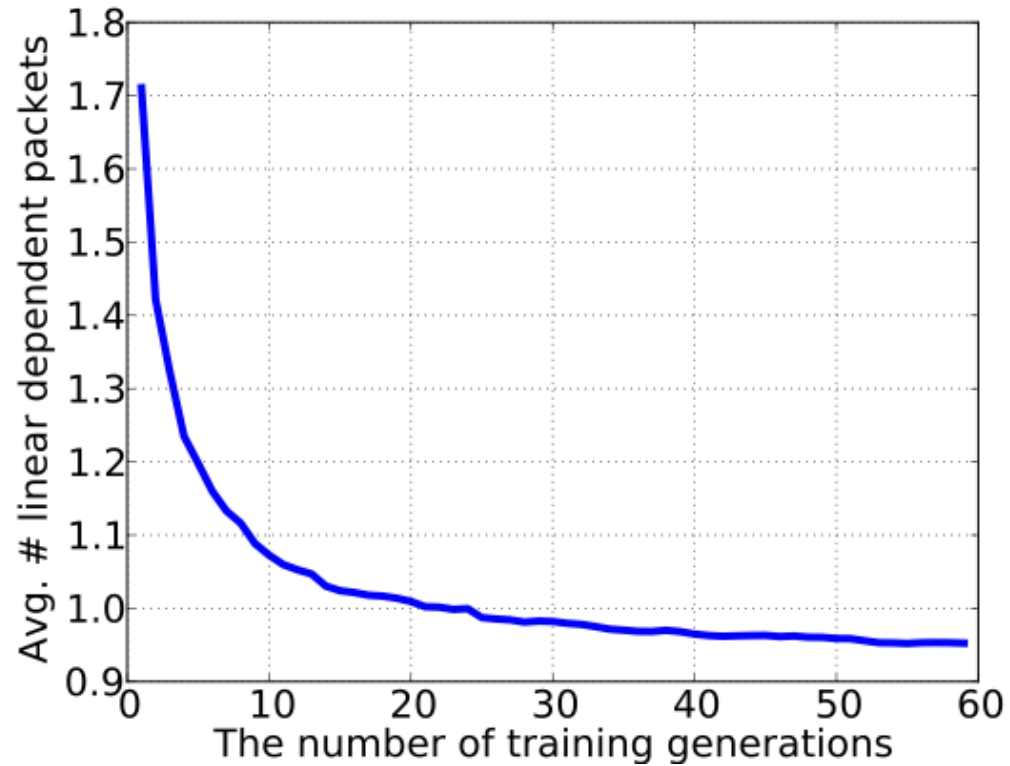
The testbed deployed at AAU



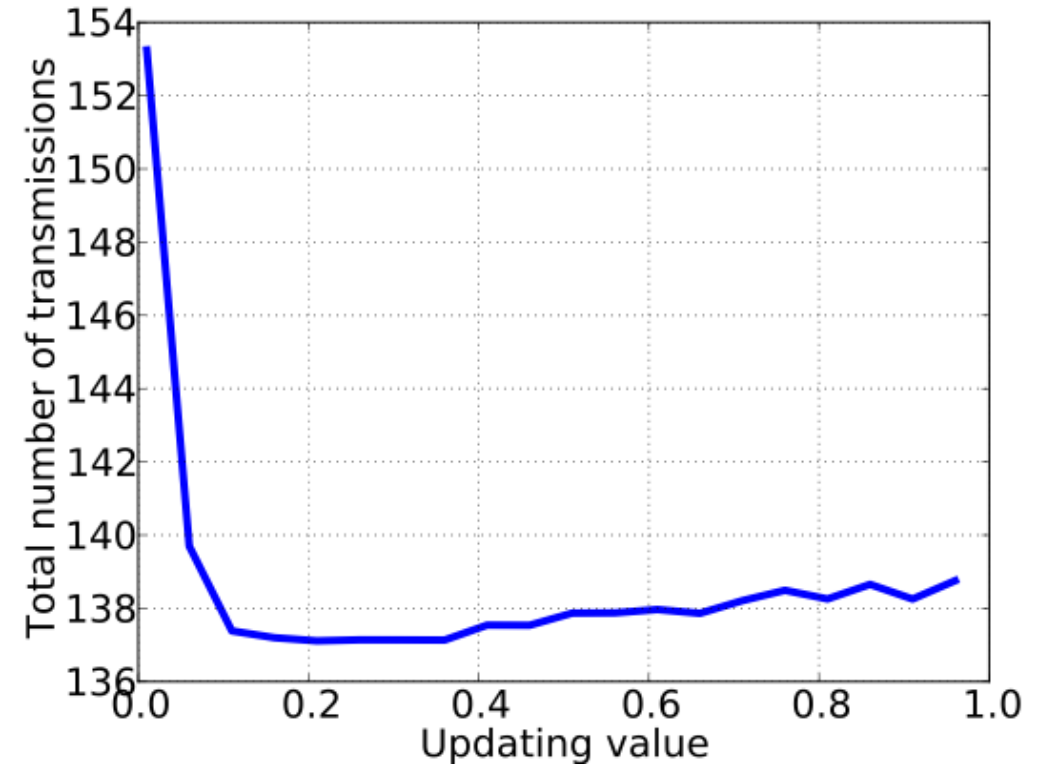
Raspberry Devices

RLNC in real meshed

PlayN Cool



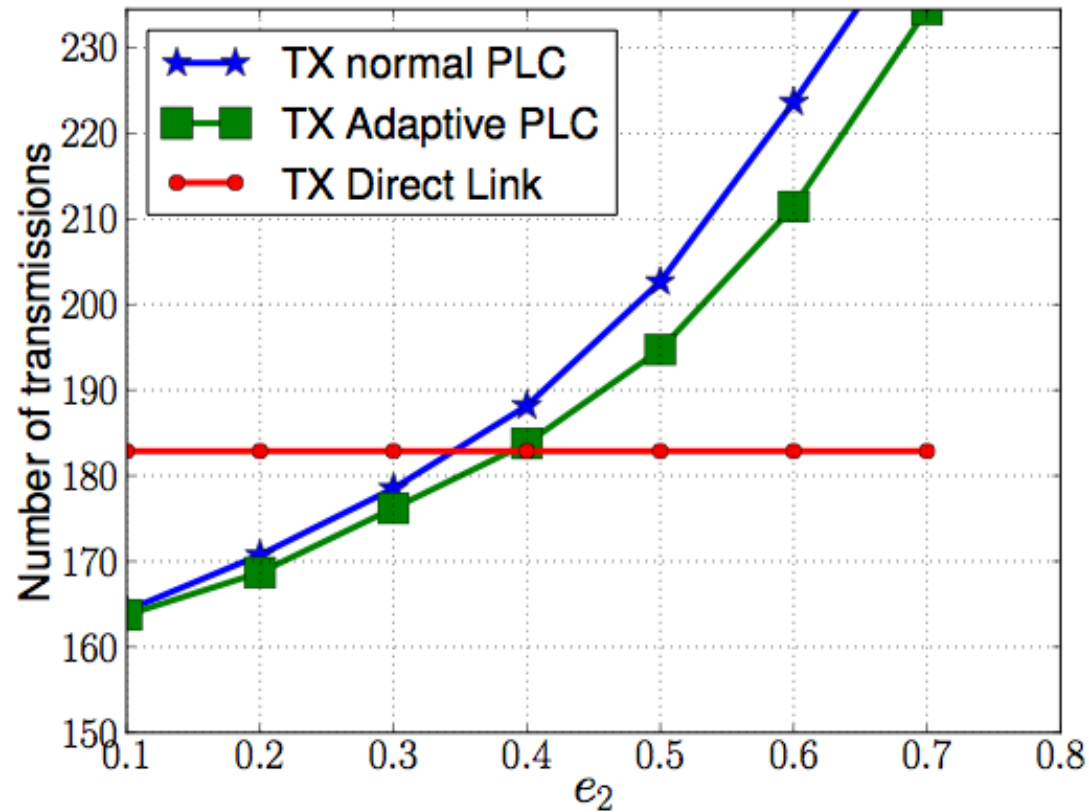
The effect of number of training generation



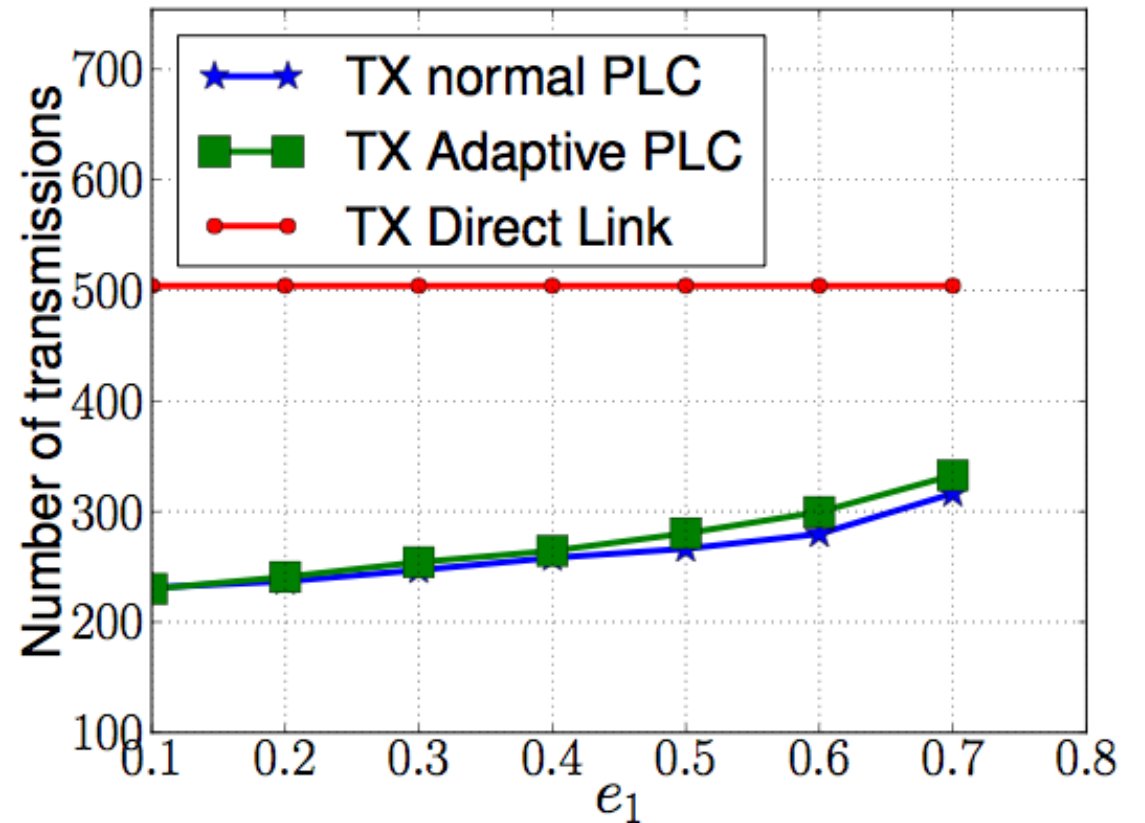
The effect the updating value

RLNC in real meshed

PlayN Cool



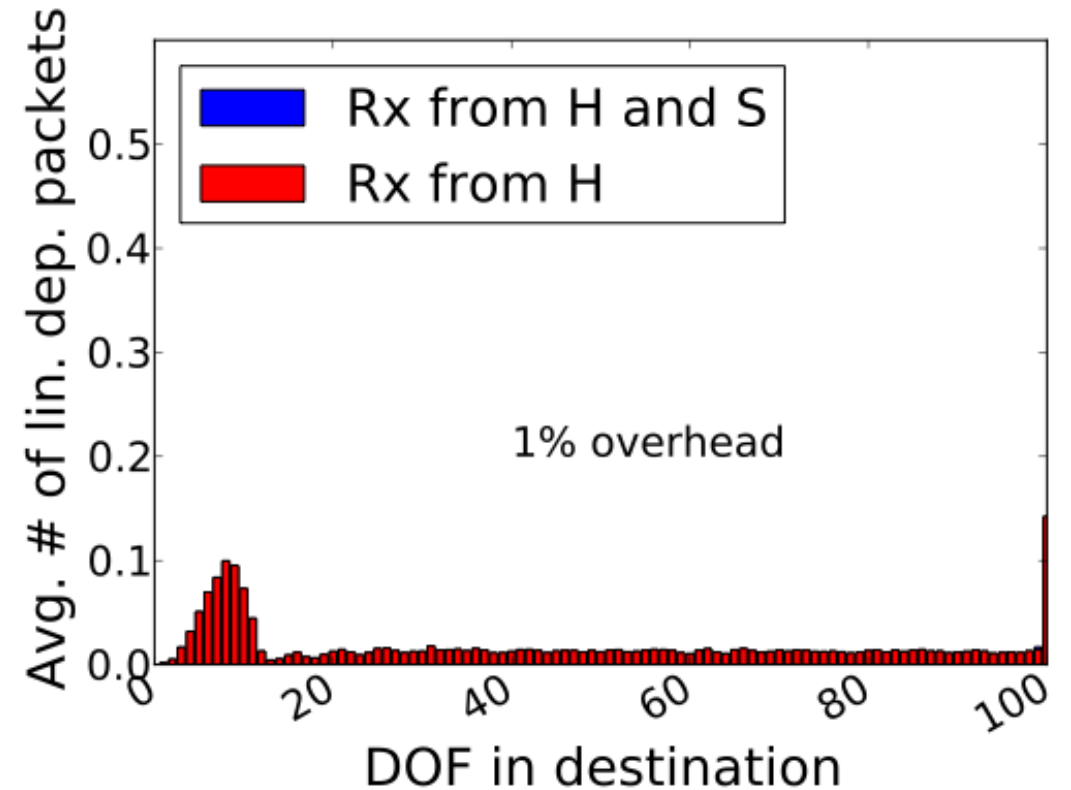
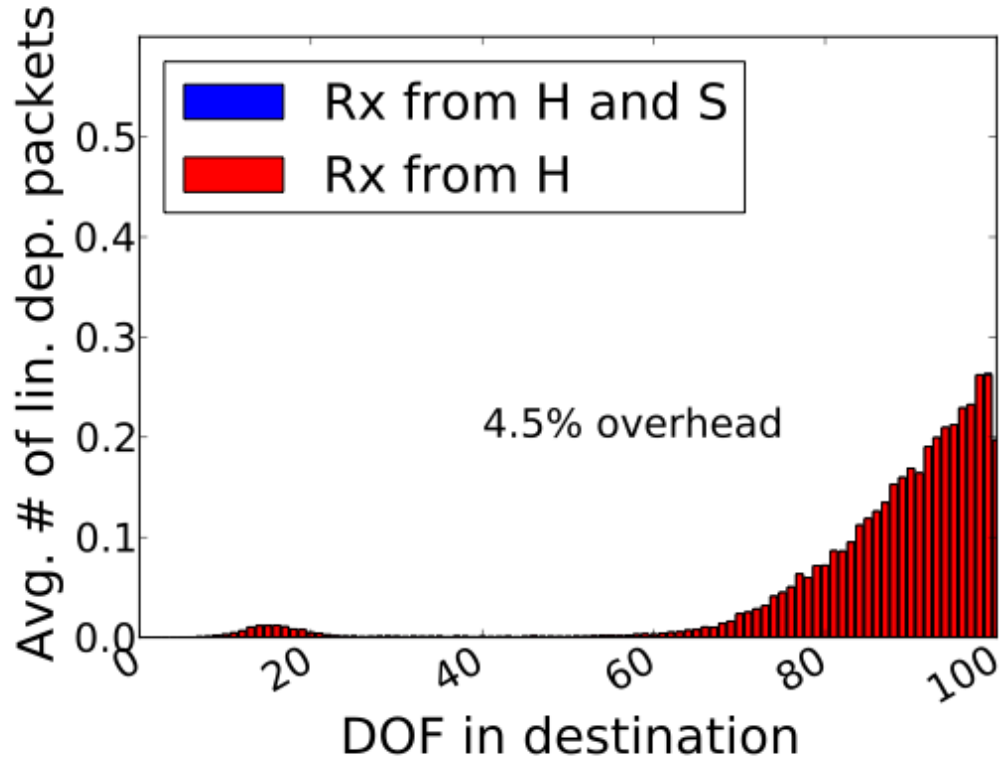
$e_2 = 0.3, e_3 = 0.8, g = 100$



$e_2 = 0.3, e_3 = 0.8, g = 100$

RLNC in real meshed

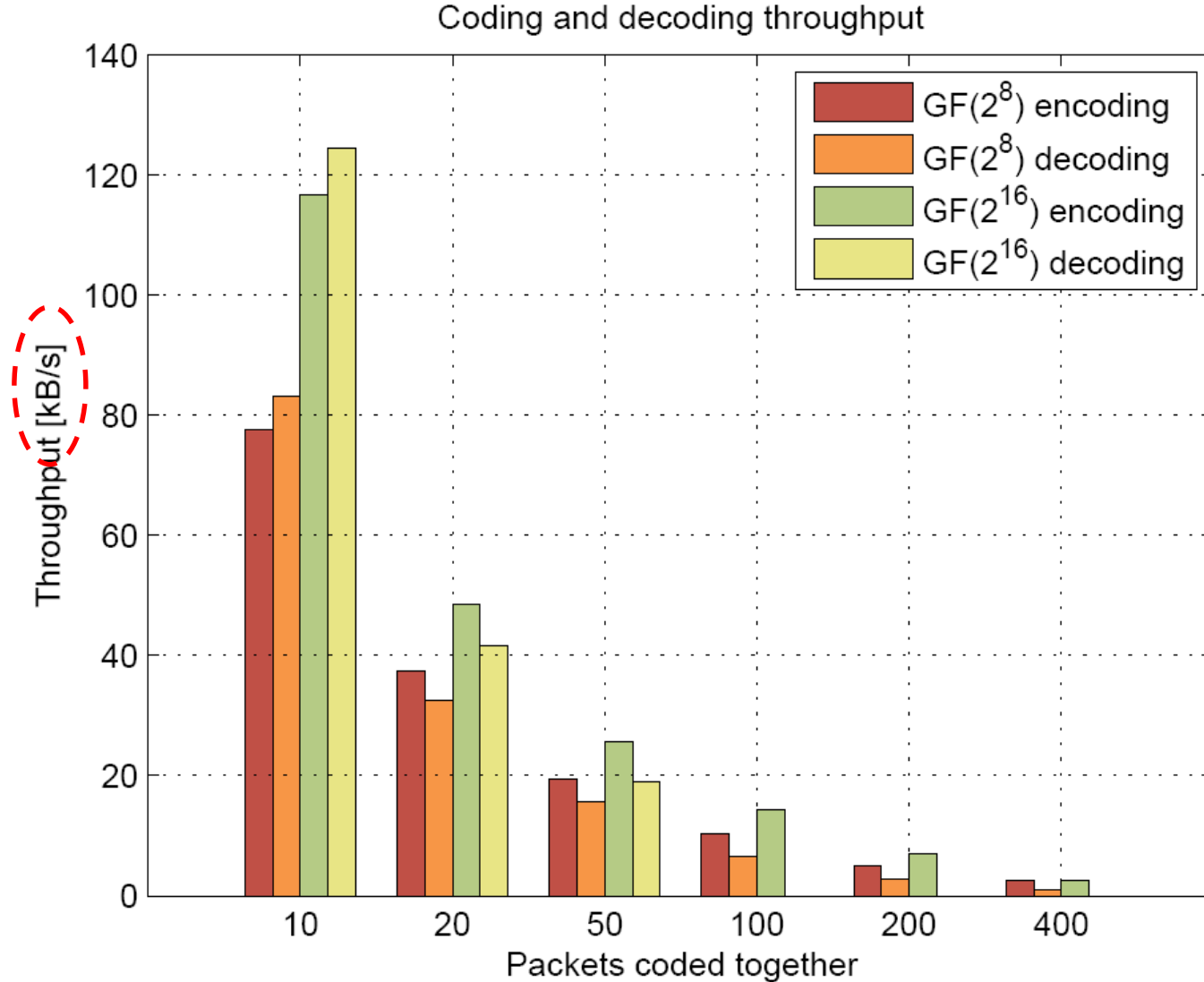
PlayN Cool



Peyman Pahlevani, Daniel E. Lucani, Frank H. P. Fitzek, "Adaptive Relay Activation in the Network Coding Protocols" *European Wireless 2015*

A Practical Guide to RLNC Libraries

S60 Implementation RLNC (2007)



Pre-allocated memory, generated the encoding vectors, so that we only had the raw encoding



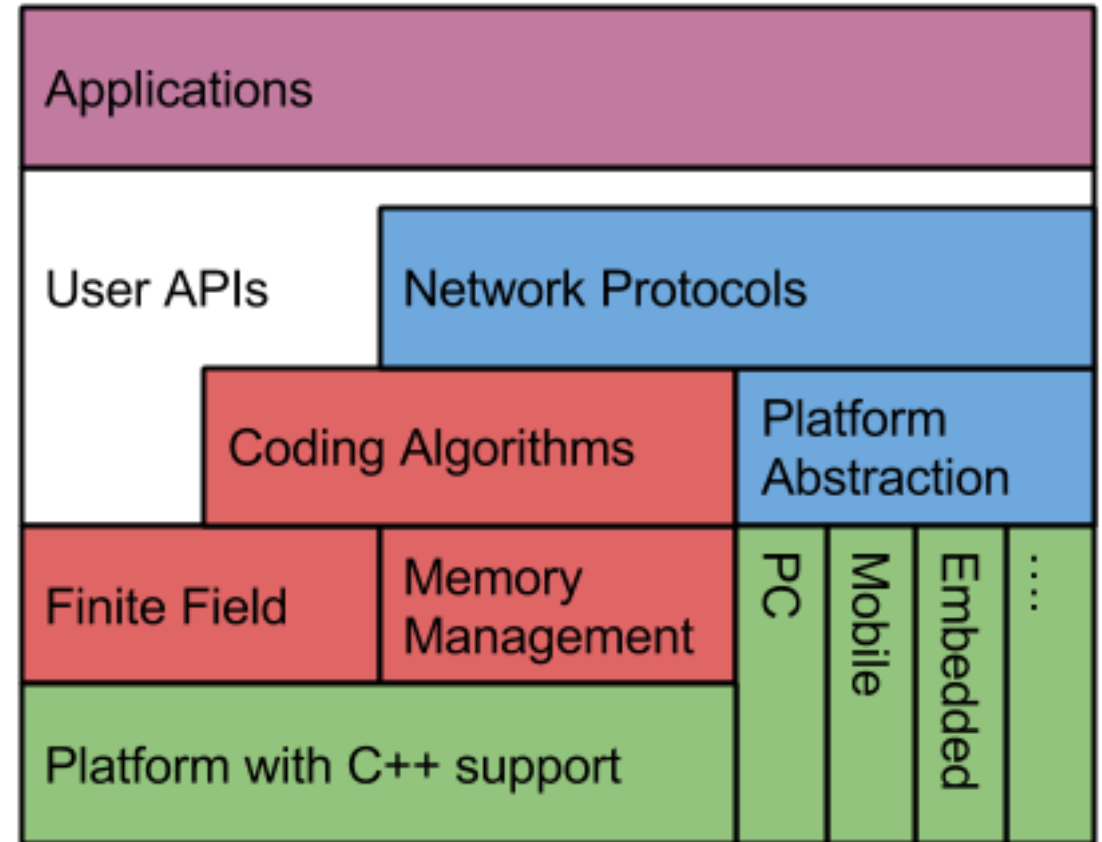
Key Technologies to Speed Up

- New software design
- Right choice of G and F
 - Binary case results in low complexity
- Hardware implementation
 - Dedicated hardware (OPENGL, SIMD)
 - Multi core / Many core (HAEC)
 - Kernel
- Sparse coding & Systematic coding
- Optimal Prime Fields (OPF), e.g., $2^{32}-5$

Software Library

Kodo: Software Implementation

- Software library for Network Coding
- Software library for Network Protocols
- Fully-tested Software
- Build System for several platforms
- In use by Customers



KODO: Shortens TTM

Implemented Features

- Recoding
- Systematic coding
- On-the-fly coding
- Partial decoding
- Real-time adjustable density
- Symbol pruning
- File encoder
- Zero copy API
- Object pooling
- Hardware optimized
- Variable symbol length

• Platforms








Continuous Integration

- build on every commit
- buildbot.steinwurf.dk

Commercial Library Benchmarking (2013)

- Jerasure 1.2 by James Plank
 - Jerasure 2.0 by James Plank
 - OpenFEC by INRIA
 - ISA-L by INTEL
 - KODO by Steinwurf
-
- The intention is to make a fair comparison among them and start collaborative research on this topic!

Feature List (2013)

Library Capabilities	Kodo	Jerasure 1.2	Jerasure 2.0	ISA-L	Open FEC
Reed-Solomon Codes Supported	X	X	X	X	X
Network Coding Supported	X				
Updated with Novel Code Structures	X				(X)
Continuous Testing and Support	X				
Continuous Optimization of Algorithms	X				
Automatic Adaptation to CPU Features	X				
OS Support				FreeBSD 	
Compiler Support	GCC, Clang, MS VS	?	GCC	GCC	?
Date of Last Release	1/2014	8/2008 12/2011 ^x	1/2014	11/2013	4/2012
Hardware Acceleration on Intel Chipsets	SSSE3, CLMUL, AVX2		SSSE3	SSSE3, CLMUL	SSE
Hardware Acceleration on ARM chipsets	NEON				
Multi-core support	X				
Simulation support	Internal, NS3				

Comparison with State of the Art (2013)

Coding Speed [MB/s] for 1 MB per data segment

	F=GF(2 ⁸) P=1MB	Kodo 17 MT (sparse=0.5)	Kodo 17 (sparse=0.5)	ISA-L	Jerasure 2.0	OpenFEC
Industry trend ↓	G=8 (12)	3096/2980	3096/2980	2255/2635	1250/1365	353/292
	G=9 (13)	2542/2559	2752/2898	1961/2252	1096/1185	305/264
	G=10 (15)	2136/2227	2025/2126	1724/1796	997/1072	285/245
	G=16 (24)	1807/1496	1264/1239	1075/1180	628/644	179/160
	G=30 (45)	950/647	672/513	266/271	349/361	96/90
	G=60 (90)	594/329	359/256	123/122	184/184	48/46
	G=100 (150)	383/209	226/159	74/73	111/111	29/28
	G=150 (225)	266/141	153/107	47/46	74/74	19/19

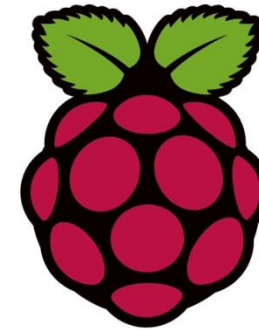
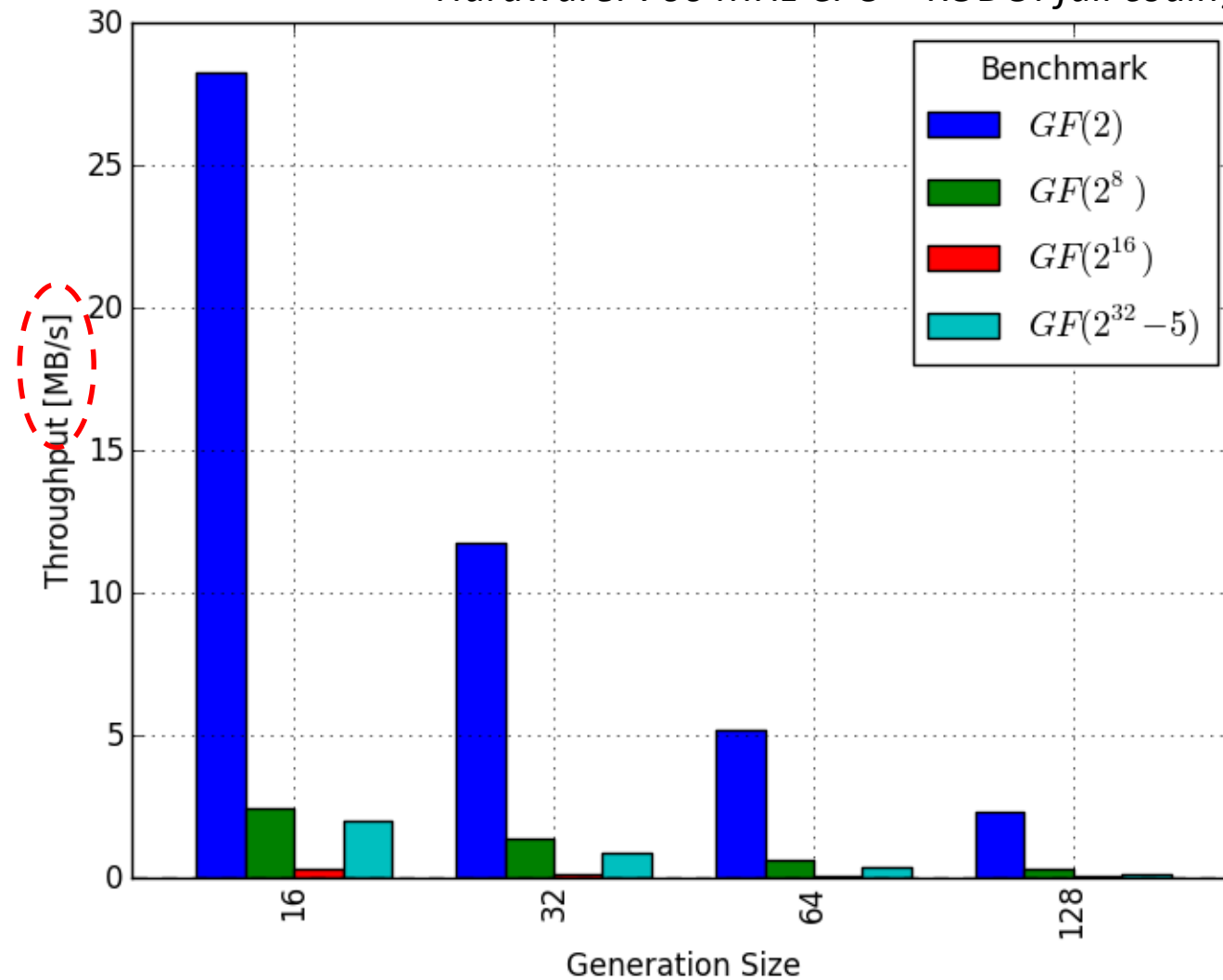
RLNC (rich feature set)
RS

RS use G times more memory than RLNC for data segment recovery

Measured on Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz

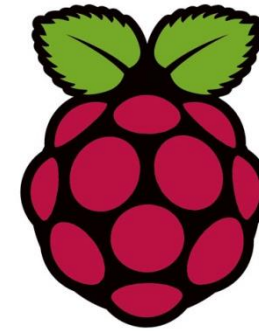
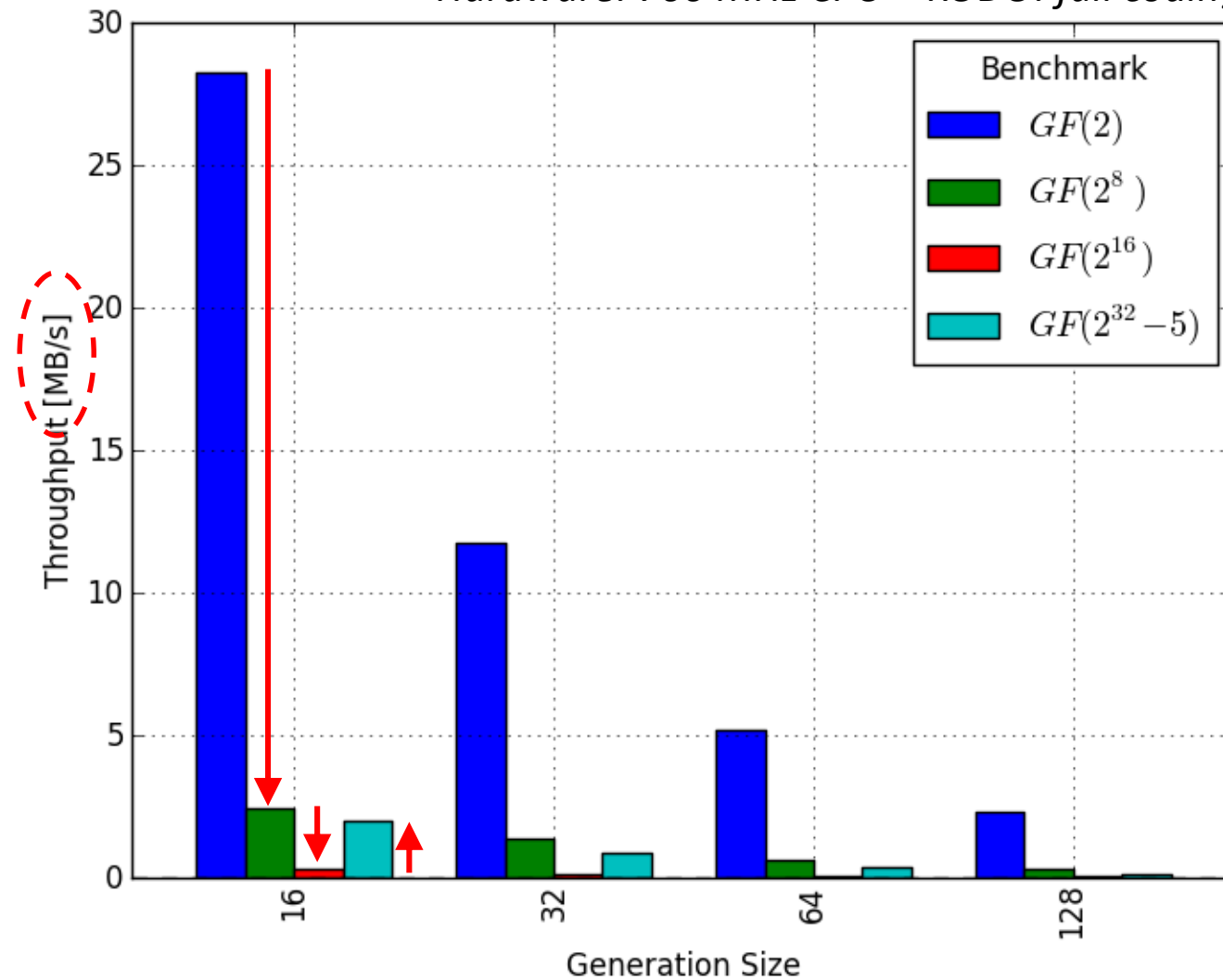
Raspberry Pi (2013)

Hardware: 700 MHz CPU – KODO: full coding



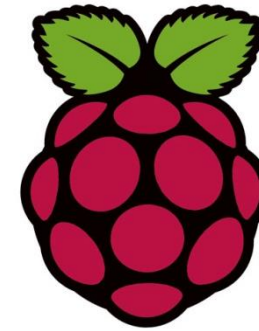
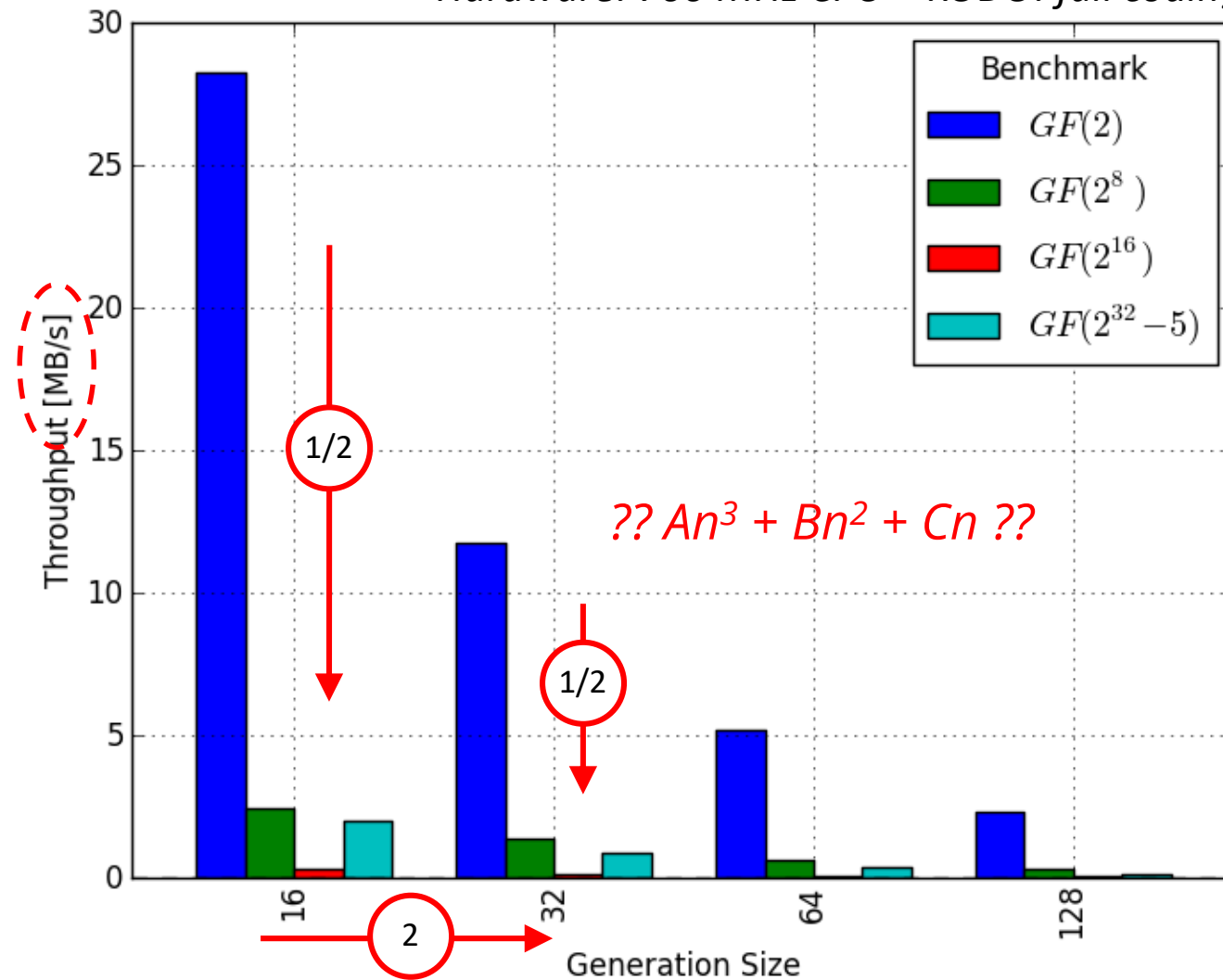
Raspberry Pi (2013)

Hardware: 700 MHz CPU – KODO: full coding



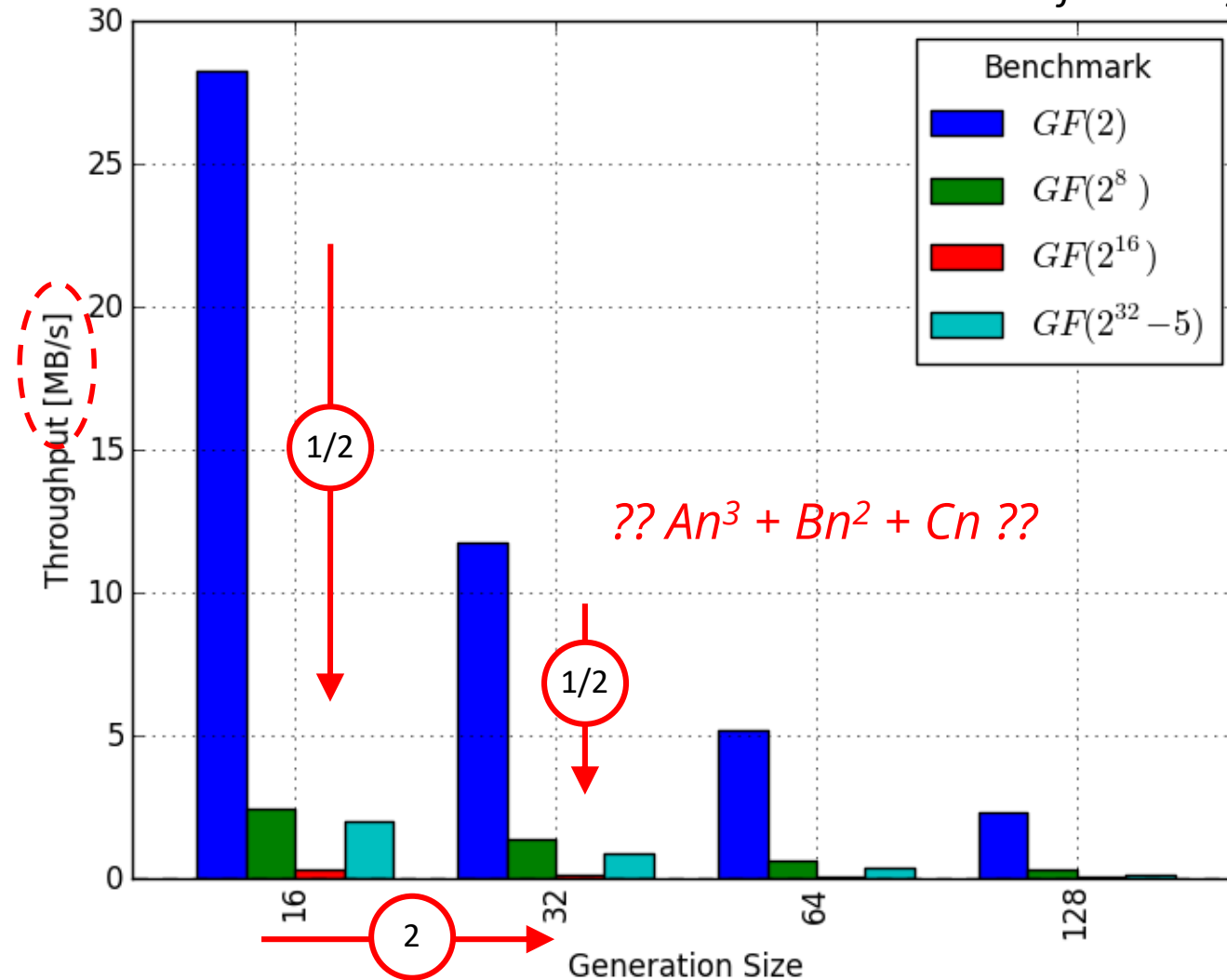
Raspberry Pi (2013)

Hardware: 700 MHz CPU – KODO: full coding



Raspberry Pi (2013)

Hardware: 700 MHz CPU – KODO: full coding



$$\text{Throughput} \propto \frac{g}{t}$$

but if

$$t \propto g^2$$

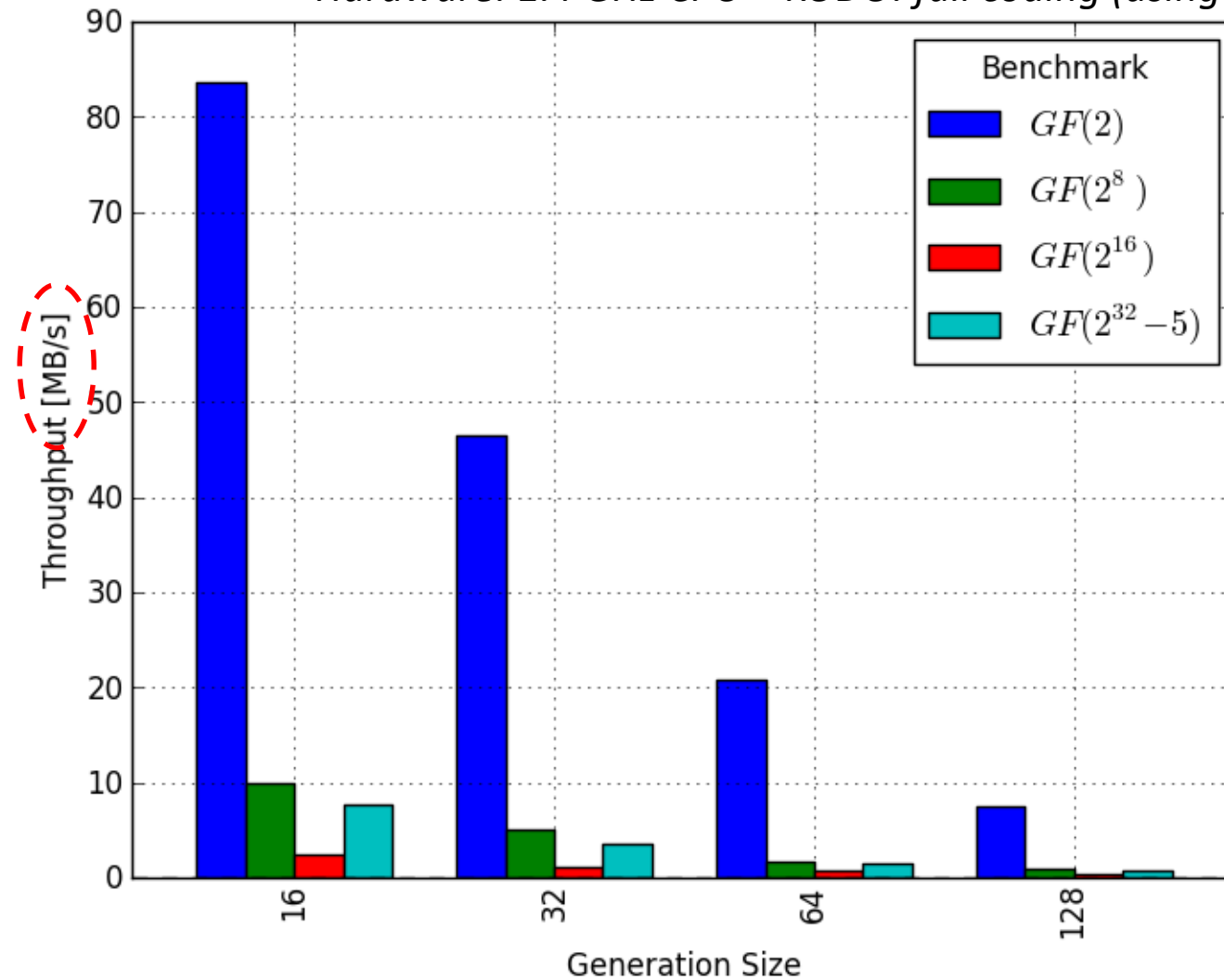
then

$$\text{Throughput} \propto \frac{g}{g^2} = \frac{1}{g}$$

If we double g , we half the throughput. This shows that the component Bn^2 is more significant for small n .

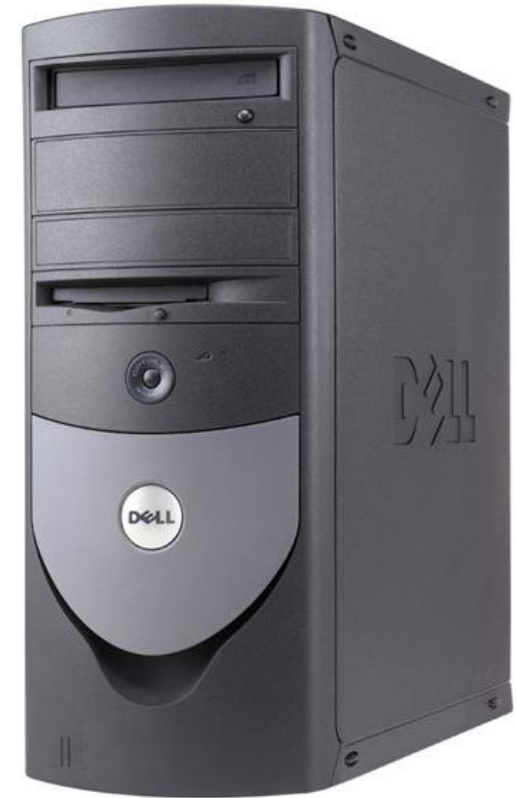
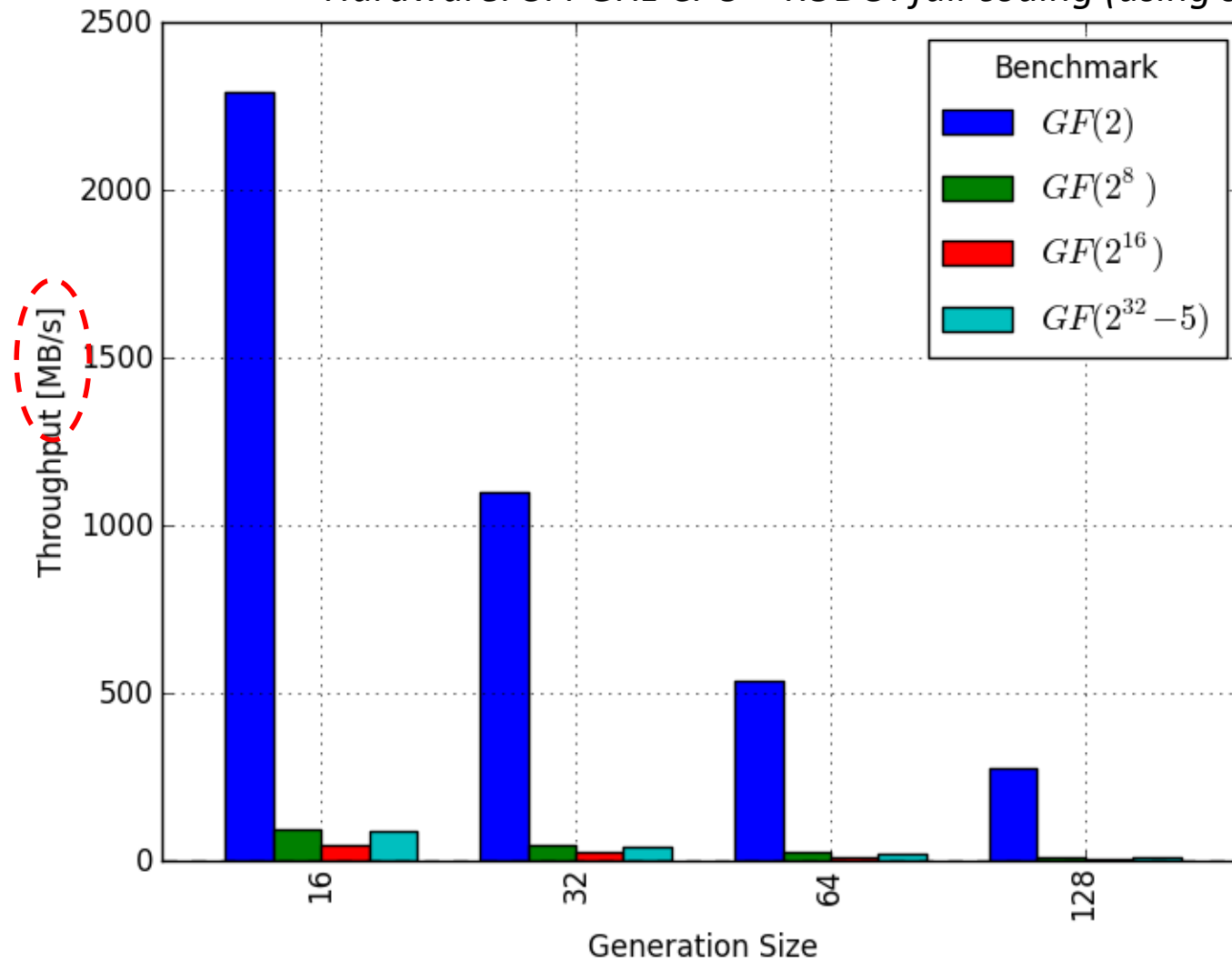
Android (2013)

Hardware: 1.4 GHz CPU – KODO: full coding (using one core)



PC (2013)

Hardware: 3.4 GHz CPU – KODO: full coding (using single processor)



KODO Modes

```
In [3]: # print all members containing "Factory" and "Encoder"  
print("\n".join([item for item in dir(kodo) if all([keyword in item for keyword in ["Factory", "Encoder"]])]))
```

```
FulcrumEncoderFactoryBinary  
FulcrumEncoderFactoryBinary16  
FulcrumEncoderFactoryBinary4  
FulcrumEncoderFactoryBinary8  
FullVectorEncoderFactoryBinary  
FullVectorEncoderFactoryBinary16  
FullVectorEncoderFactoryBinary4  
FullVectorEncoderFactoryBinary8  
NoCodeEncoderFactory  
OnTheFlyEncoderFactoryBinary  
OnTheFlyEncoderFactoryBinary16  
OnTheFlyEncoderFactoryBinary4  
OnTheFlyEncoderFactoryBinary8  
PerpetualEncoderFactoryBinary  
PerpetualEncoderFactoryBinary16  
PerpetualEncoderFactoryBinary4  
PerpetualEncoderFactoryBinary8  
SlidingWindowEncoderFactoryBinary  
SlidingWindowEncoderFactoryBinary16  
SlidingWindowEncoderFactoryBinary4  
SlidingWindowEncoderFactoryBinary8  
SparseFullVectorEncoderFactoryBinary  
SparseFullVectorEncoderFactoryBinary16  
SparseFullVectorEncoderFactoryBinary4  
SparseFullVectorEncoderFactoryBinary8
```

KODO Modes

```
In [3]: # print all members containing "Factory" and "Encoder"  
print("\n".join([item for item in dir(kodo) if all([keyword in item for keyword in ["Factory", "Encoder"]])]))
```

```
FulcrumEncoderFactoryBinary  
FulcrumEncoderFactoryBinary16  
FulcrumEncoderFactoryBinary4  
FulcrumEncoderFactoryBinary8  
FullVectorEncoderFactoryBinary  
FullVectorEncoderFactoryBinary16  
FullVectorEncoderFactoryBinary4  
FullVectorEncoderFactoryBinary8  
NoCodeEncoderFactory  
OnTheFlyEncoderFactoryBinary  
OnTheFlyEncoderFactoryBinary16  
OnTheFlyEncoderFactoryBinary4  
OnTheFlyEncoderFactoryBinary8  
PerpetualEncoderFactoryBinary  
PerpetualEncoderFactoryBinary16  
PerpetualEncoderFactoryBinary4  
PerpetualEncoderFactoryBinary8  
SlidingWindowEncoderFactoryBinary  
SlidingWindowEncoderFactoryBinary16  
SlidingWindowEncoderFactoryBinary4  
SlidingWindowEncoderFactoryBinary8  
SparseFullVectorEncoderFactoryBinary  
SparseFullVectorEncoderFactoryBinary16  
SparseFullVectorEncoderFactoryBinary4  
SparseFullVectorEncoderFactoryBinary8
```

KODO Modes

```
In [3]: # print all members containing "Factory" and "Encoder"  
print("\n".join([item for item in dir(kodo) if all([keyword in item for keyword in ["Factory", "Encoder"]])]))
```

```
FulcrumEncoderFactoryBinary  
FulcrumEncoderFactoryBinary16  
FulcrumEncoderFactoryBinary4  
FulcrumEncoderFactoryBinary8  
FullVectorEncoderFactoryBinary  
FullVectorEncoderFactoryBinary16  
FullVectorEncoderFactoryBinary4  
FullVectorEncoderFactoryBinary8  
NoCodeEncoderFactory  
OnTheFlyEncoderFactoryBinary  
OnTheFlyEncoderFactoryBinary16  
OnTheFlyEncoderFactoryBinary4  
OnTheFlyEncoderFactoryBinary8  
PerpetualEncoderFactoryBinary  
PerpetualEncoderFactoryBinary16  
PerpetualEncoderFactoryBinary4  
PerpetualEncoderFactoryBinary8  
SlidingWindowEncoderFactoryBinary  
SlidingWindowEncoderFactoryBinary16  
SlidingWindowEncoderFactoryBinary4  
SlidingWindowEncoderFactoryBinary8  
SparseFullVectorEncoderFactoryBinary  
SparseFullVectorEncoderFactoryBinary16  
SparseFullVectorEncoderFactoryBinary4  
SparseFullVectorEncoderFactoryBinary8
```

Network Coding GF(2)



$$\begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} \end{pmatrix}$$

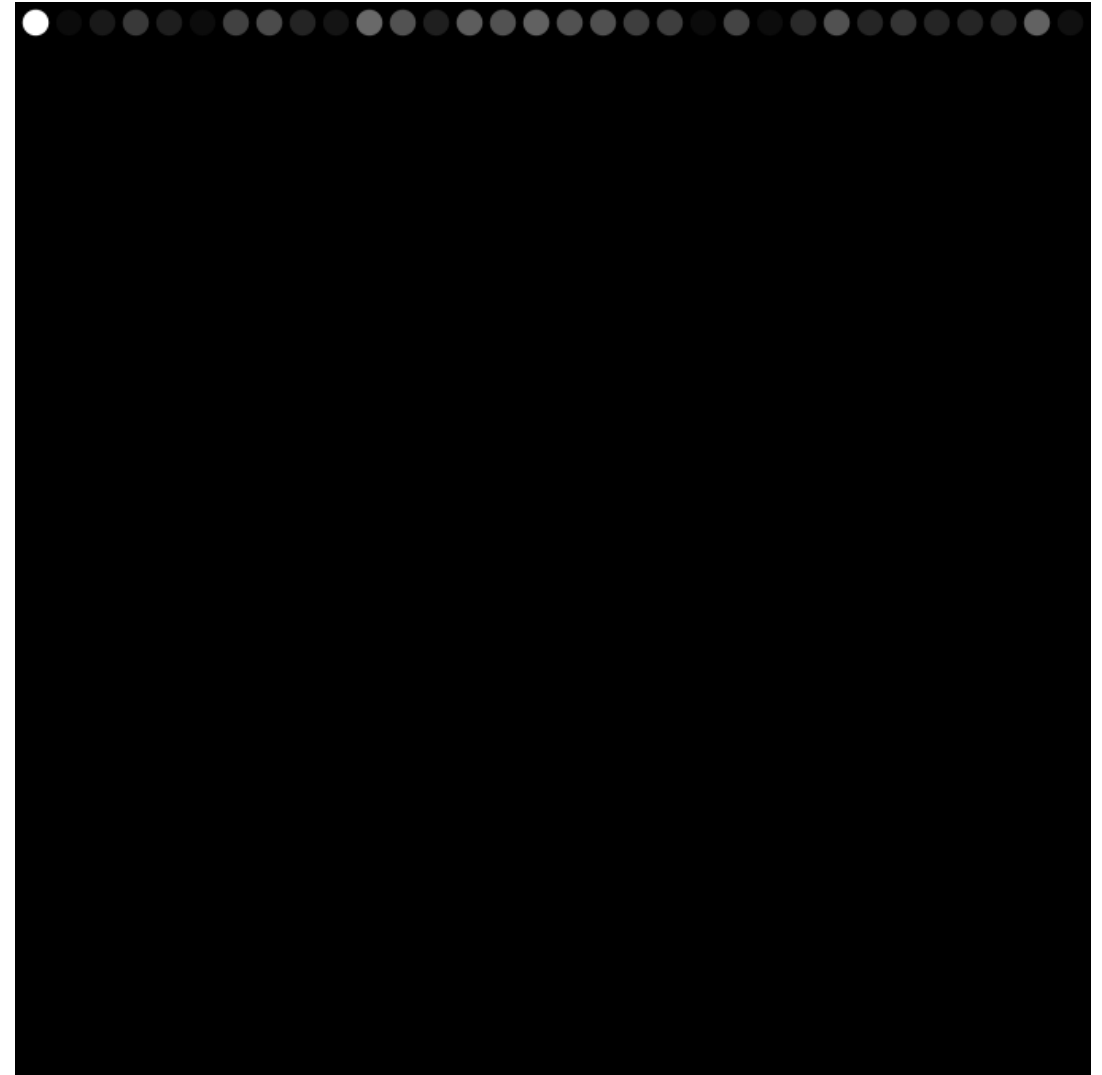
$$\begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} \end{pmatrix}$$

$$\begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} \end{pmatrix}$$

FullRLNC

Coding matrix is loaded with fully random elements of field size F

Probability of zero as field element is $1/F$



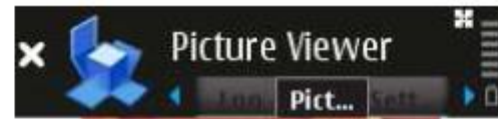
Systematic RLNC



Options Back

(d)

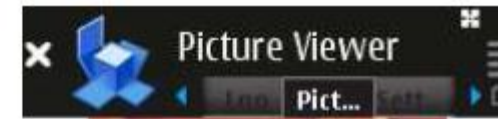
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$



Options Back

(e)

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$



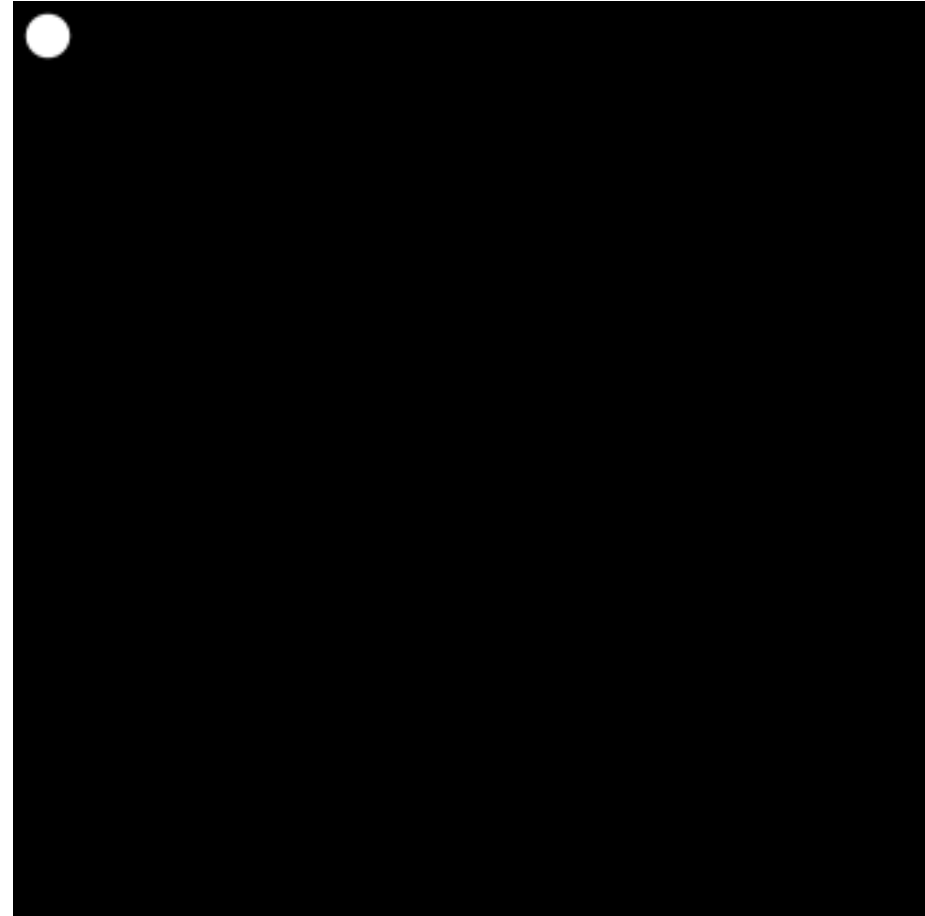
Options Back

(f)

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} \\ 0 & 0 & 1 & 0 \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} \end{pmatrix}$$

Systematic RLNC

Starting with uncoded packets and fill
wholes with fully encoded packets



KODO: RLNC Modes

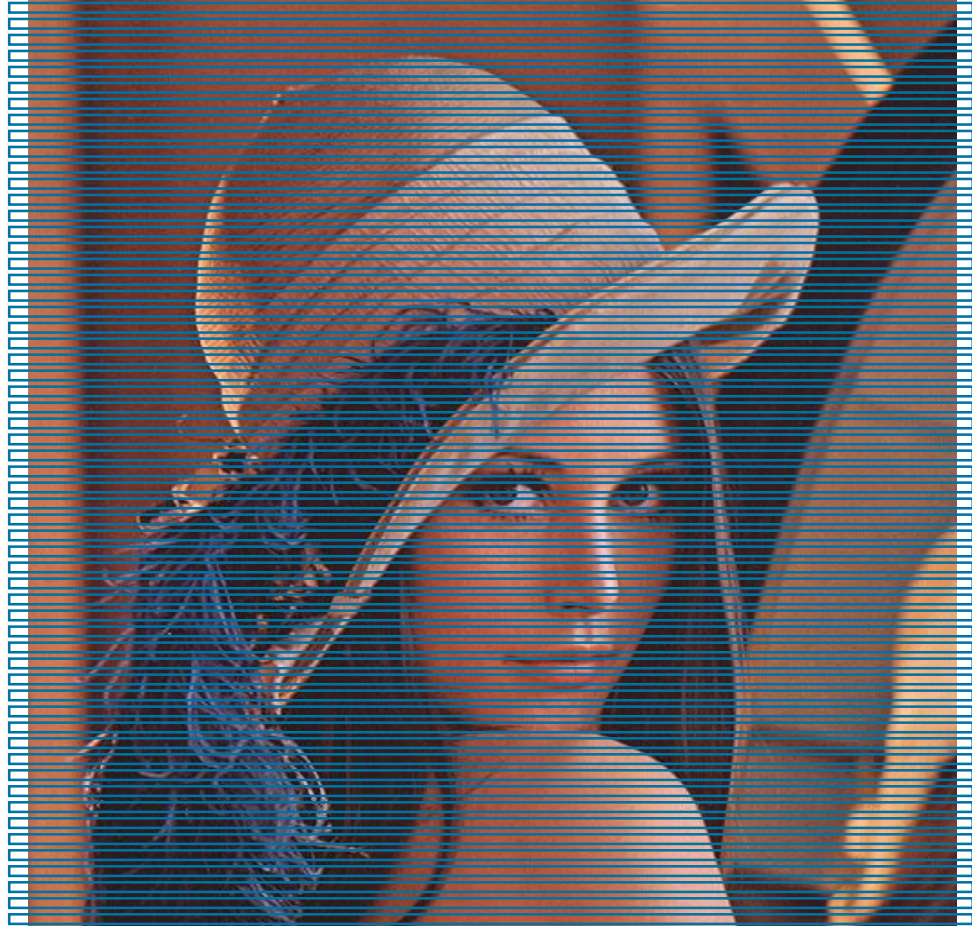
KODO: RLNC Modes

Input for Coding



KODO: RLNC Modes

Basic Idea

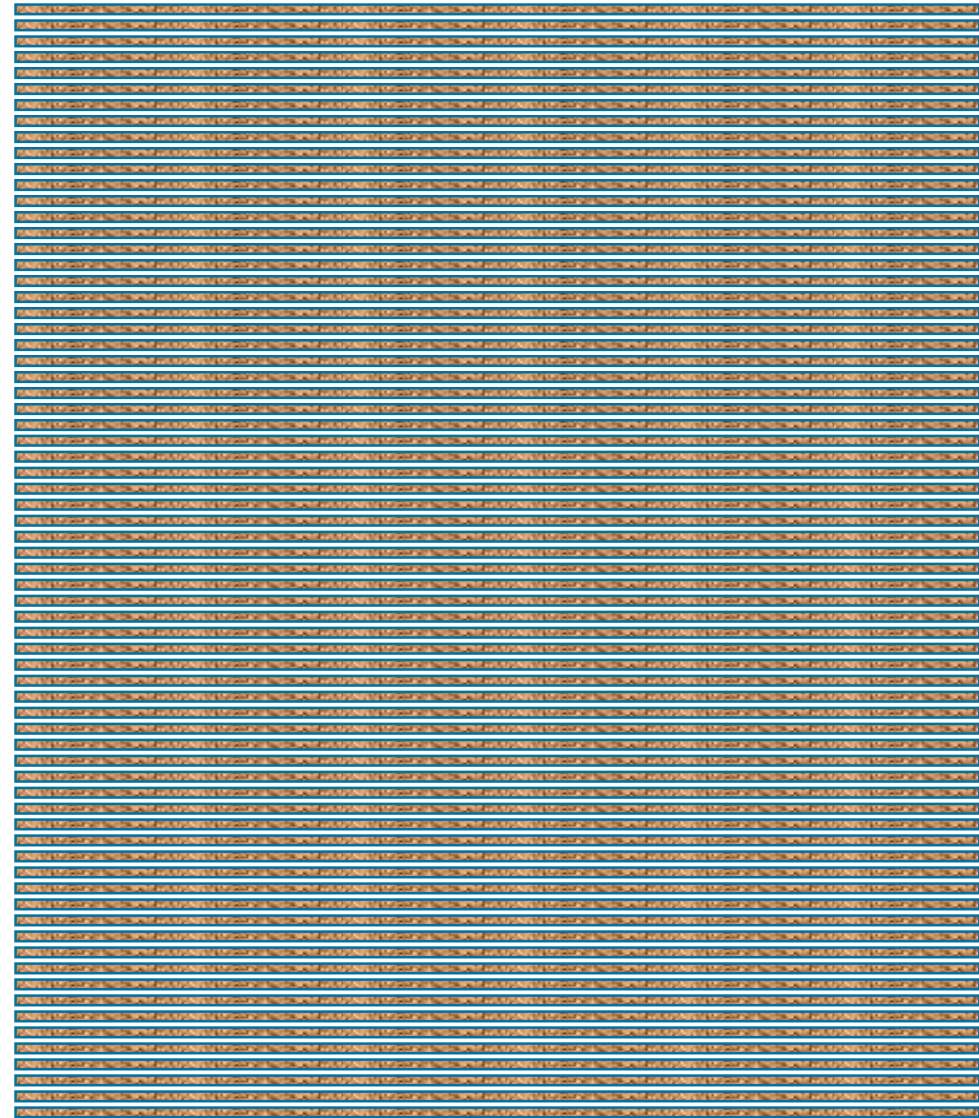


KODO: RLNC Modes

Basic Idea



KODO - Encoder

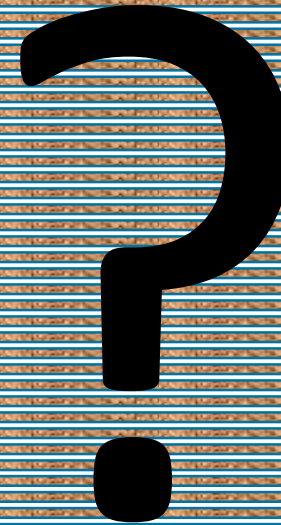


KODO: RLNC Modes

Block title

Lossy Medium

KODO - Decoder



KODO: RLNC Modes

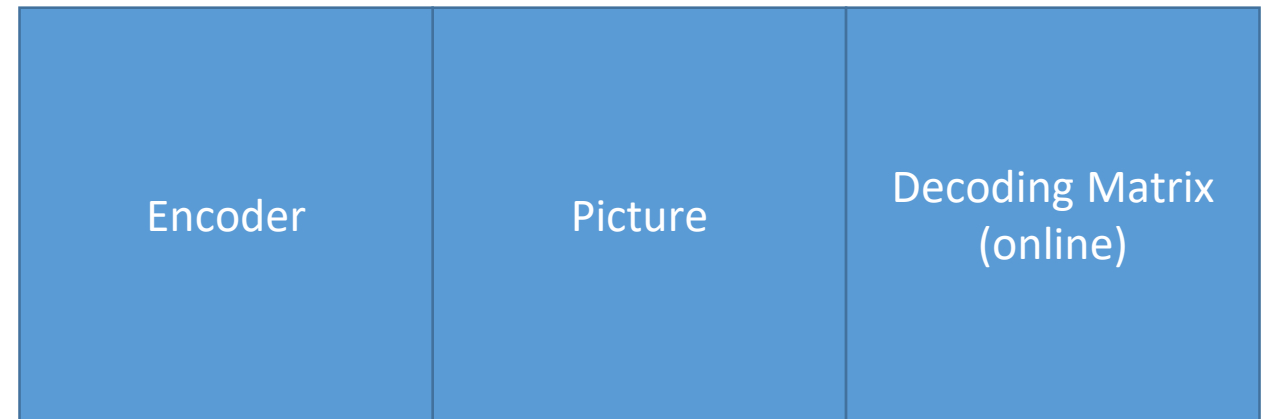
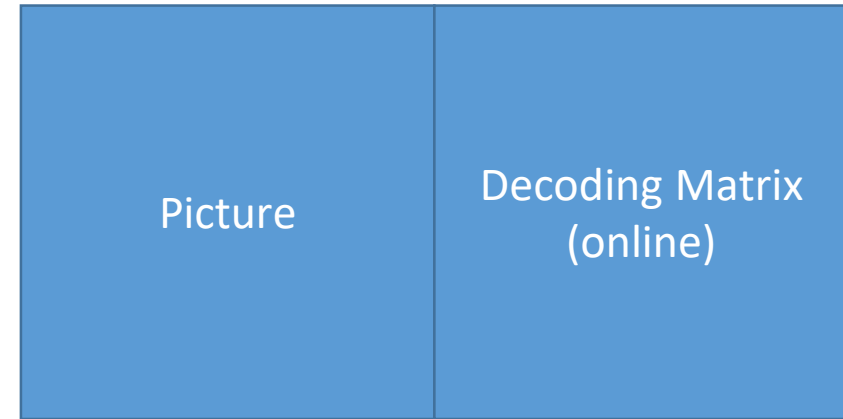
Basic Idea

Modes

- Systematic ON/OFF
- On the fly / Online
- Sliding Window

Parameters

- Field 2, 2^8 , 2^{16}
- Input (Salta vs. Lena vs. Pikachu)



Full RLNC

FullVectorEncoderFactoryBinary
FullVectorEncoderFactoryBinary16
FullVectorEncoderFactoryBinary4
FullVectorEncoderFactoryBinary8

Salta – Non Systematic Full RLNC

Finite Field : 2 - Binary

Salta – Non Systematic Full RLNC

Finite Field : Binary8

Systematic RLNC

Systematic RLNC

- In general, Gaussian elimination $n \times n$ matrix requires $An^3 + Bn^2 + Cn$ operations

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ \alpha_{5,1} & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Systematic RLNC

- Gaussian elimination $n \times n$ matrix requires $An^3 + Bn^2 + Cn$ operations

Nothing to do here!

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ \alpha_{5,1} & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Systematic RLNC

- Gaussian elimination $n \times n$ matrix requires $An^3 + Bn^2 + Cn$ operations
- Use D uncoded information to “clean” up the matrix

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ \alpha_{5,1} & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Nothing to do here!

Systematic RLNC

- Gaussian elimination $n \times n$ matrix requires $An^3 + Bn^2 + Cn$ operations
- Use D uncoded information to “clean” up the matrix

Subtract first row multiplied with $\alpha_{4,1}$ from fourth line!
 Subtract first row multiplied with $\alpha_{5,1}$ from fifth line!
 Subtract first row multiplied with $\alpha_{6,1}$ from sixth line!

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ \alpha_{5,1} & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Systematic RLNC

- Gaussian elimination $n \times n$ matrix requires $An^3 + Bn^2 + Cn$ operations
- Use D uncoded information to “clean” up the matrix

Subtract second row multiplied with $\alpha_{4,2}$ from fourth line!
 Subtract second row multiplied with $\alpha_{5,2}$ from fifth line!
 Subtract second row multiplied with $\alpha_{6,2}$ from sixth line!

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ 0 & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ 0 & \alpha_{6,2} & \alpha_{6,3} & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Systematic RLNC

- Gaussian elimination $n \times n$ matrix requires $An^3 + Bn^2 + Cn$ operations
- Use D uncoded information to “clean” up the matrix

Subtract third row multiplied with $\alpha_{4,3}$ from fourth line!
 Subtract third row multiplied with $\alpha_{5,3}$ from fifth line!
 Subtract third row multiplied with $\alpha_{6,3}$ from sixth line!

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ 0 & 0 & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ 0 & 0 & \alpha_{6,3} & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Systematic RLNC

- Gaussian elimination $n \times n$ matrix requires $An^3 + Bn^2 + Cn$ operations
- Use D uncoded information to “clean” up the matrix

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{matrix} & & & & \text{M-D} & & \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ 0 & 0 & 0 & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ 0 & 0 & 0 & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} & \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix} \end{matrix} \quad \begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{matrix} & & & & \text{M} & & \\ \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} & \alpha_{1,5} & \alpha_{1,6} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} & \alpha_{2,5} & \alpha_{2,6} \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} & \alpha_{3,5} & \alpha_{3,6} \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ \alpha_{5,1} & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} & \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix} \end{matrix}$$

$A(M-D)^3 + B(M-D)^2 + C(M-D)$
 $A(M)^3 + B(M)^2 + C(M)$

Lesson Learned

- Binary fields yield high coding rates (simple XOR calculations and 50% of the coefficients are zero)
- Systematic approach also yields high coding rates
- It seems the number of zeros in the coding matrix plays a role here. Why?
- Zeros allows for reordering of the coded packets rather than brute force calculating. Among other reasons.
- Remember some slides before:

0110
1000
1101

1000 ← Simple MOVE
0110
0011 ← 0101 (XOR 0110) ← 1101 (XOR 1000)

1000
0101 (0110 XOR 0011)
0011

Sparse RLNC

```
SparseFullVectorEncoderFactoryBinary  
SparseFullVectorEncoderFactoryBinary16  
SparseFullVectorEncoderFactoryBinary4  
SparseFullVectorEncoderFactoryBinary8
```

Sparse Network Codes

- Let's add zeros randomly into our coding matrix.

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & 0 & \alpha_{1,4} & \alpha_{1,5} & \alpha_{1,6} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} & \alpha_{2,5} & \alpha_{2,6} \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} & 0 & \alpha_{3,6} \\ \alpha_{4,1} & 0 & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ 0 & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & 0 \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & 0 & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Sparse Network Codes

- Let's add zeros randomly into our coding matrix. More!

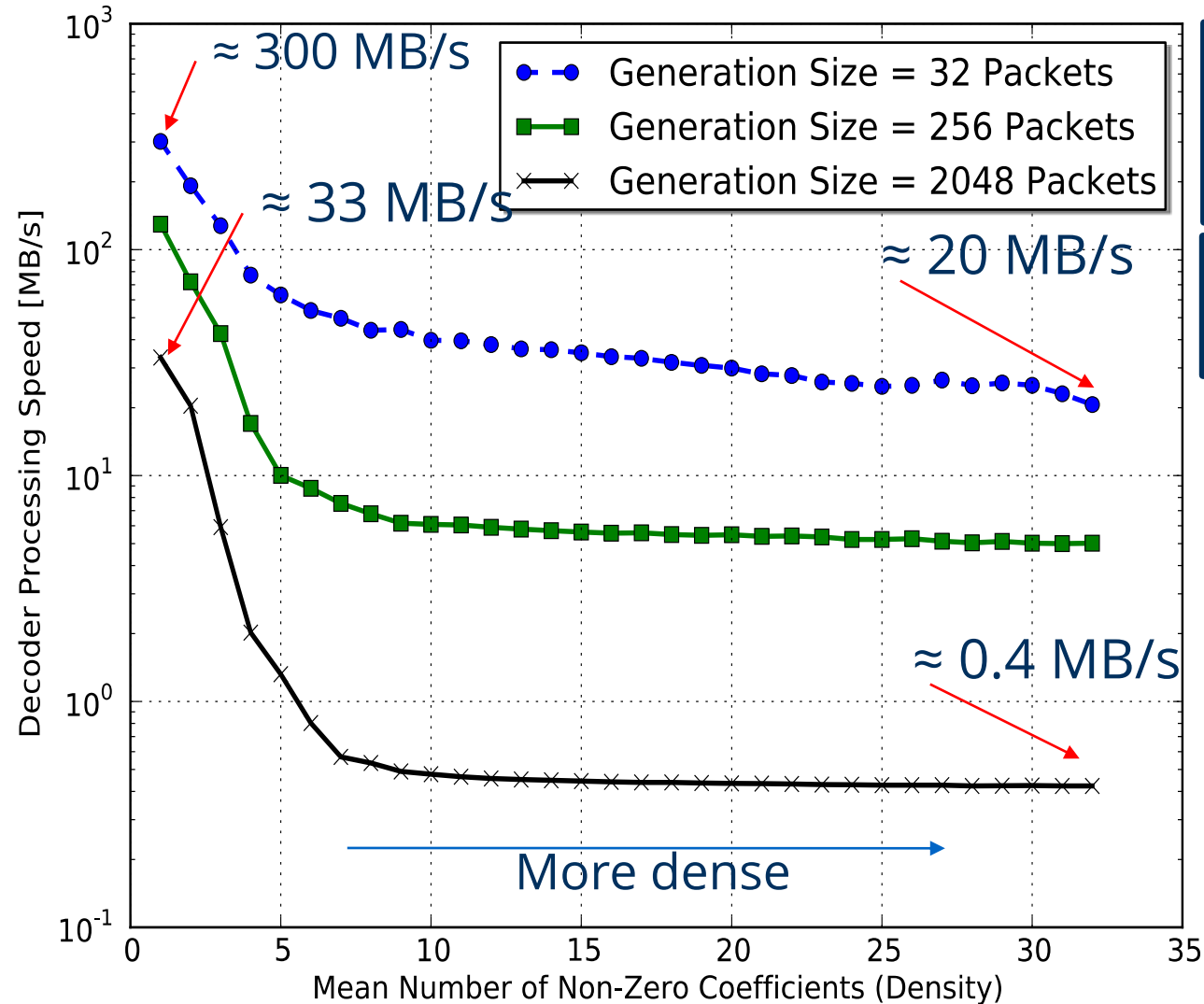
$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} 0 & \alpha_{1,2} & 0 & \alpha_{1,4} & \alpha_{1,5} & \alpha_{1,6} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} & \alpha_{2,5} & 0 \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} & 0 & \alpha_{3,6} \\ \alpha_{4,1} & 0 & 0 & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ 0 & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & 0 \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & 0 & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Sparse Network Codes

- Let's add zeros randomly into our coding matrix. More! More!

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} 0 & \alpha_{1,2} & 0 & \alpha_{1,4} & 0 & \alpha_{1,6} \\ \alpha_{2,1} & 0 & \alpha_{2,3} & 0 & \alpha_{2,5} & 0 \\ 0 & 0 & \alpha_{3,3} & 0 & 0 & \alpha_{3,6} \\ \alpha_{4,1} & 0 & 0 & \alpha_{4,4} & 0 & \alpha_{4,6} \\ 0 & \alpha_{5,2} & 0 & \alpha_{5,4} & 0 & 0 \\ \alpha_{6,1} & 0 & \alpha_{6,3} & 0 & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Sparse Network Codes



One or two orders of magnitude in the coding speed by

Dualism Theory and Implementation!



Sparse Network Codes

- Let's add zeros randomly into our coding matrix. More! More!
- The more zeros we add the faster the decoding process.
- BUT, what about linear dependency?

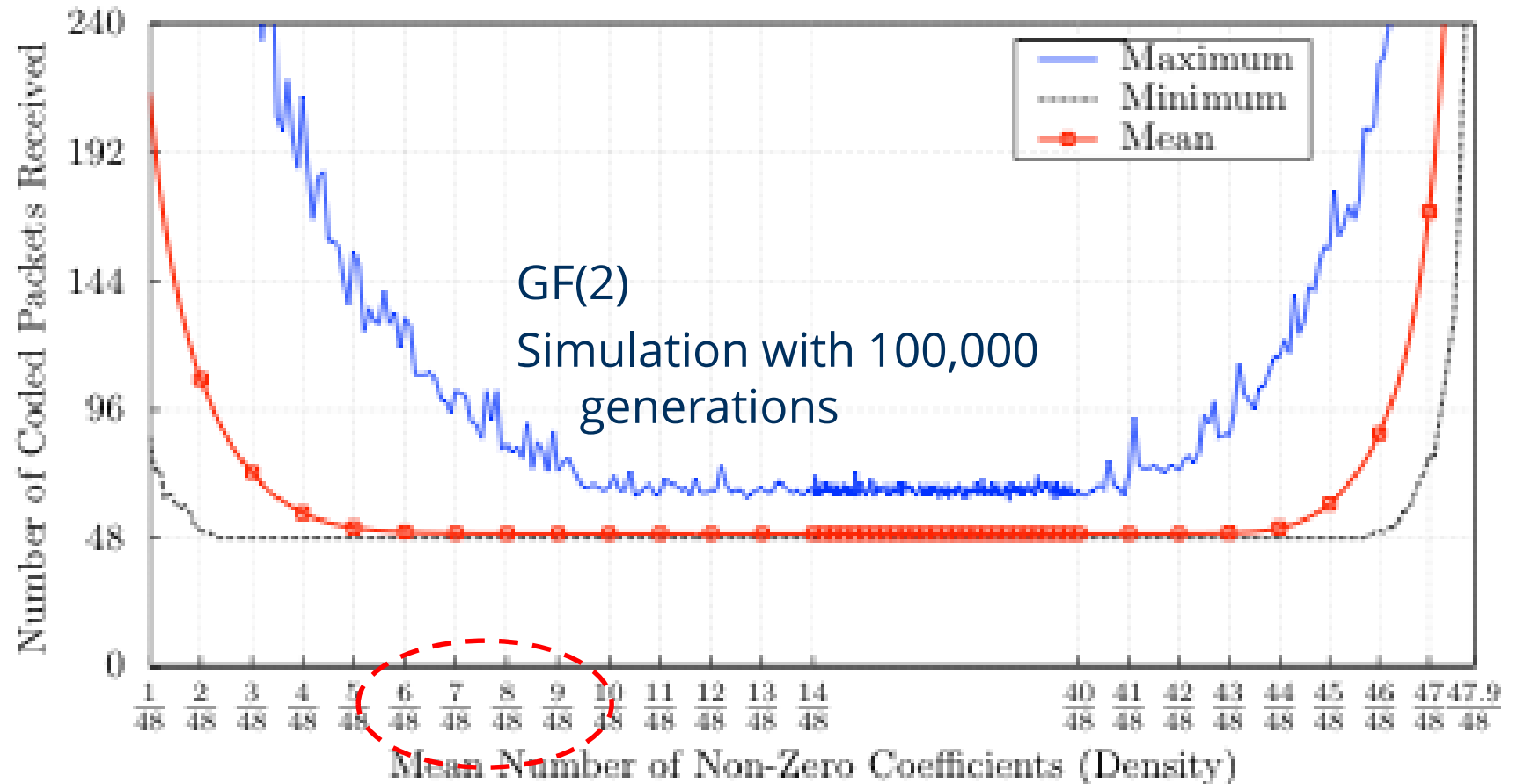
$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} 0 & \alpha_{1,2} & 0 & \alpha_{1,4} & 0 & \alpha_{1,6} \\ \alpha_{2,1} & 0 & \alpha_{2,3} & 0 & \alpha_{2,5} & 0 \\ 0 & 0 & \alpha_{3,3} & 0 & 0 & \alpha_{3,6} \\ \alpha_{4,1} & 0 & 0 & \alpha_{4,4} & 0 & \alpha_{4,6} \\ 0 & \alpha_{5,2} & 0 & \alpha_{5,4} & 0 & 0 \\ \alpha_{6,1} & 0 & \alpha_{6,3} & 0 & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Sparse Network Codes

Can we operate in the “speed up” region while maintaining good performance?

Yes, but the key is not to use a fixed density

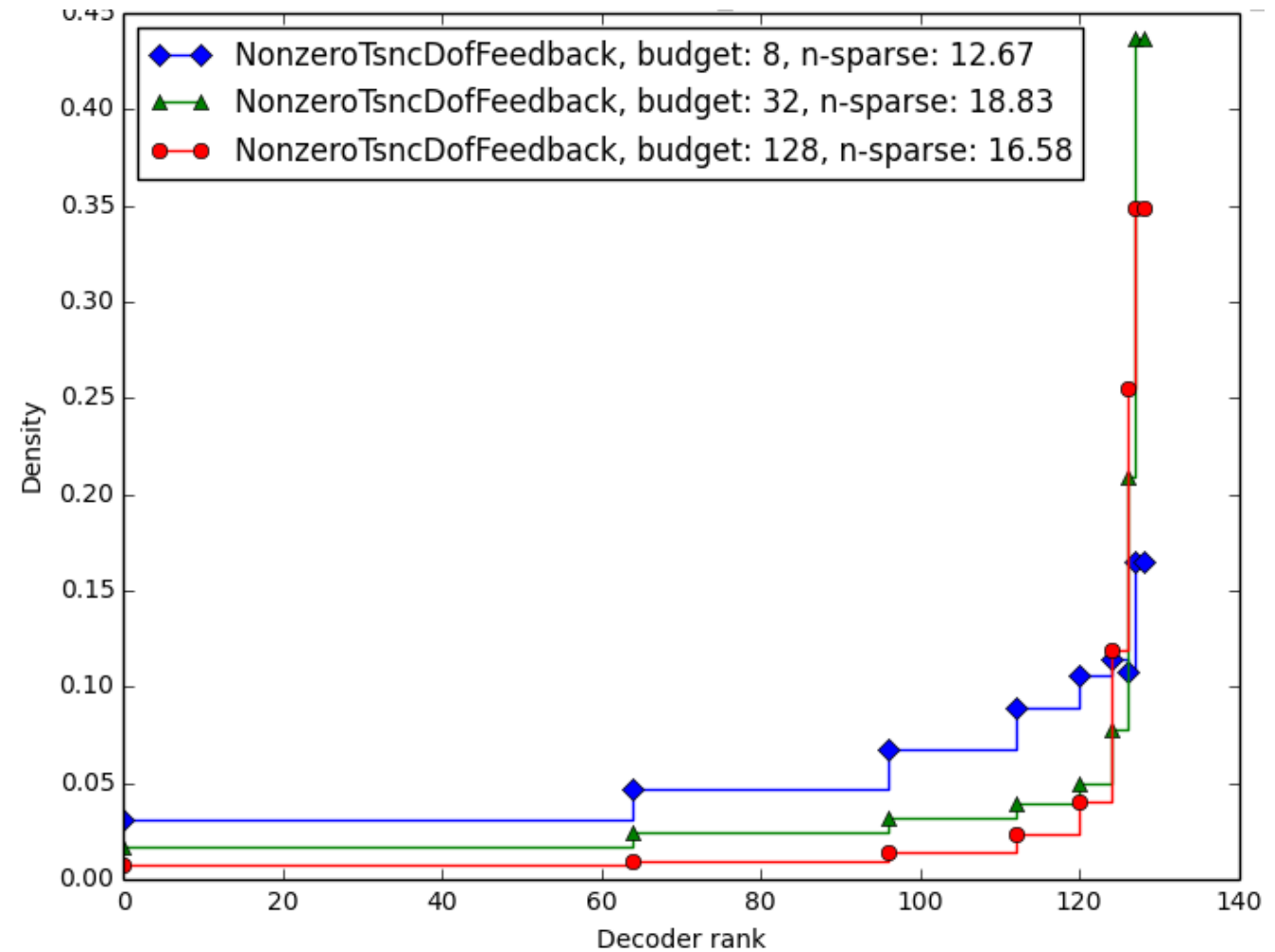
(Tunable sparse network coding, 2012)



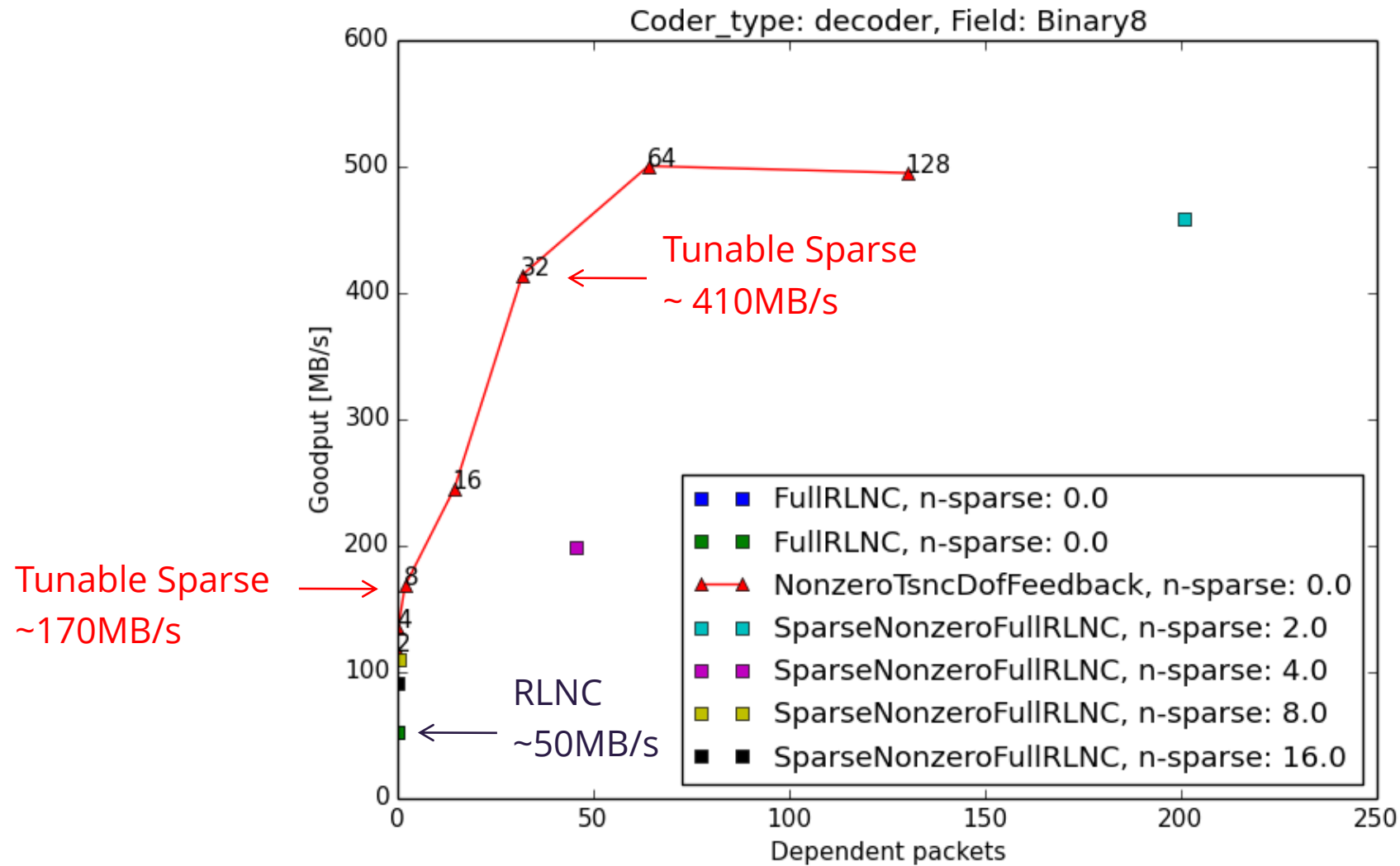
Tunabale Sparse RLNC

Tunable Sparse Network Codes

Scheme with feedback: targets specific performance degradation



Tunable Sparse Network Codes



Recoding for Sparse RLNC

- So what about recoding of sparse RLNC packets?
- In general yes it is possible.
- But by agnostic recoding the characteristics in terms of sparsity of the packets will change.

Perpetual RLNC

PerpetualEncoderFactoryBinary
PerpetualEncoderFactoryBinary16
PerpetualEncoderFactoryBinary4
PerpetualEncoderFactoryBinary8

Perpetual RLNC

- So increasing the number of the zeros in the coding matrix with the right amount can help in terms of complexity.
- The length of the encoding vector $G * \log_2(F)$ can be decreased if the encoding vector is sparse (larger number of zeros).
 - Variable Length Coding (VLC)
 - Huffmann Coding
- The idea of perpetual RLNC is to use the same number of non-zero coefficient but in a more planned manner instead of randomly spreading over the coding matrix.
 - Phase I: number of zeros
 - Phase II: number of non-zero elements
 - Phase III: number of zeros
- Encoding vector will be even smaller

Perpetual RLNC

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} & \alpha_{1,5} & \alpha_{1,6} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} & \alpha_{2,5} & \alpha_{2,6} \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} & \alpha_{3,5} & \alpha_{3,6} \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ \alpha_{5,1} & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Perpetual RLNC

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} & \alpha_{1,5} & \alpha_{1,6} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} & \alpha_{2,5} & \alpha_{2,6} \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} & \alpha_{3,5} & \alpha_{3,6} \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ \alpha_{5,1} & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Perpetual RLNC

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} & \alpha_{1,5} & \alpha_{1,6} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} & \alpha_{2,5} & \alpha_{2,6} \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} & \alpha_{3,5} & \alpha_{3,6} \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ \alpha_{5,1} & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Perpetual RLNC

Encoder

Content

Decoder

PerpetualEncoderFactoryBinary

~~PerpetualEncoderFactoryBinary16~~

PerpetualEncoderFactoryBinary4

PerpetualEncoderFactoryBinary8

On the Fly RLNC

OnTheFlyEncoderFactoryBinary
OnTheFlyEncoderFactoryBinary16
OnTheFlyEncoderFactoryBinary4
OnTheFlyEncoderFactoryBinary8

On the Fly RLNC

- Instead of a fixed corridor, On the Fly RLNC is encoding incoming packets on the fly.
- This could lead to instantaneous decodability at the receiver if no error occurs

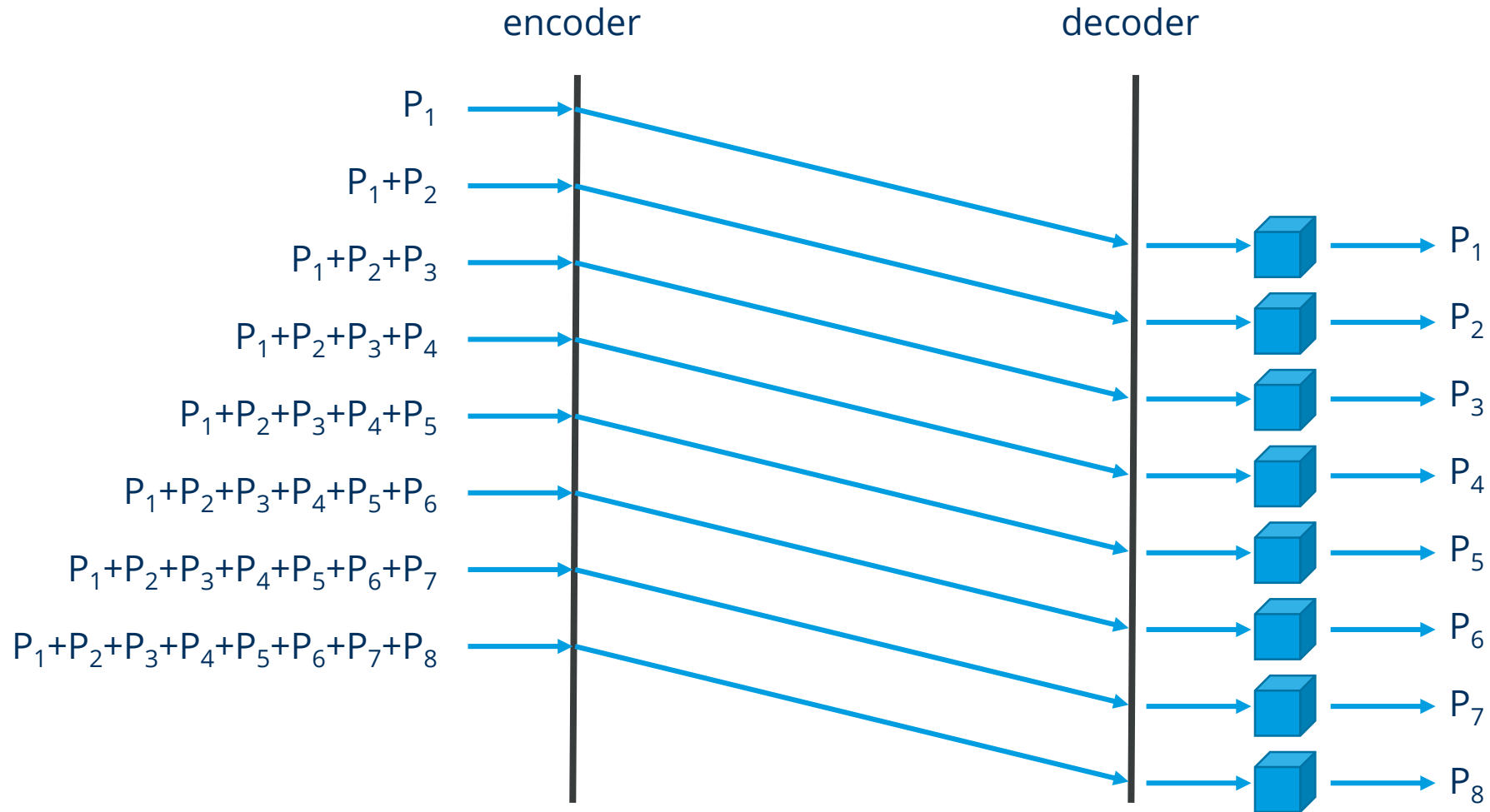
On the Fly RLNC

agnostic approach

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} & \alpha_{1,5} & \alpha_{1,6} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} & \alpha_{2,5} & \alpha_{2,6} \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} & \alpha_{3,5} & \alpha_{3,6} \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ \alpha_{5,1} & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

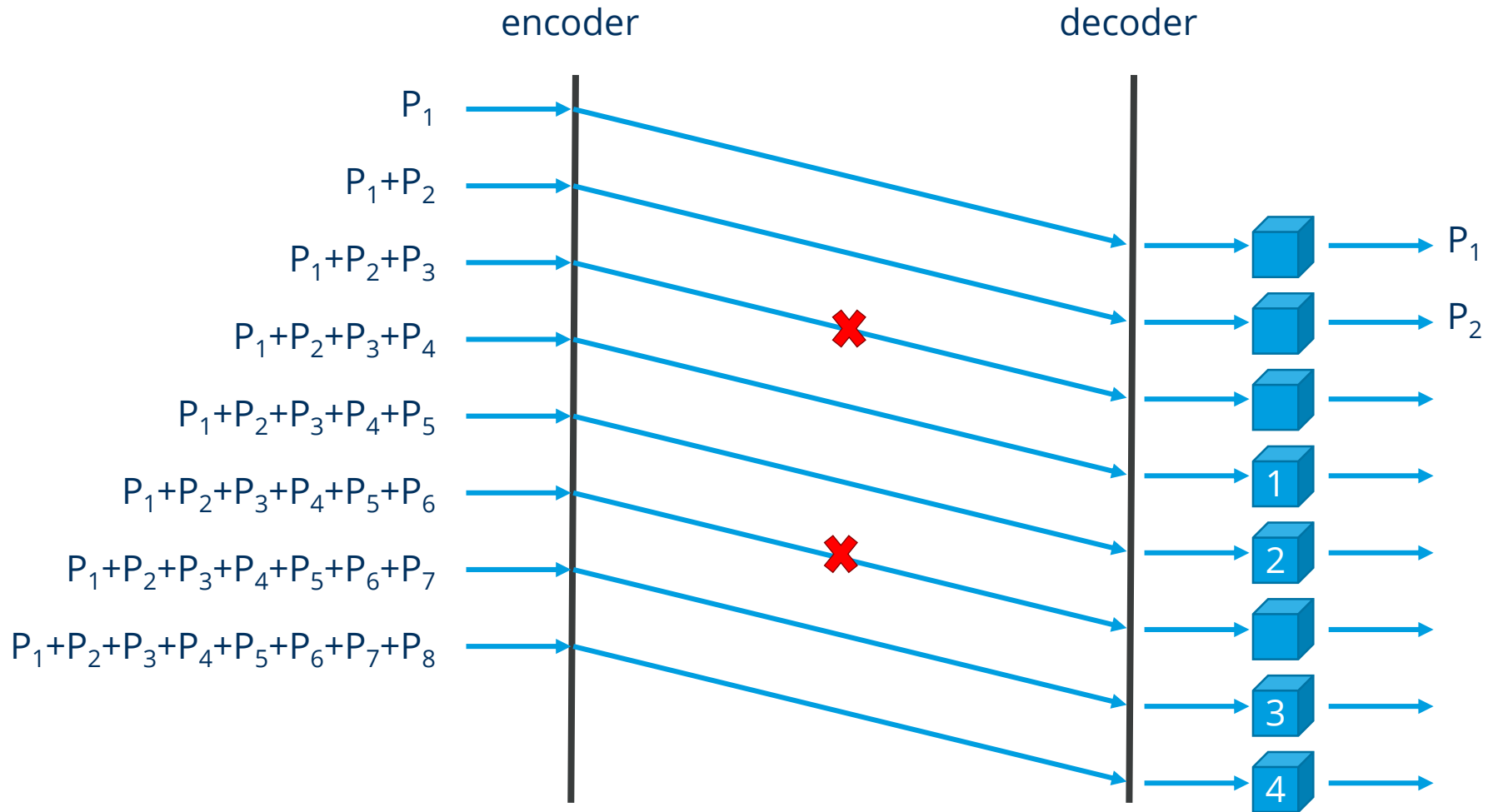
On the Fly RLNC

agnostic approach - error free



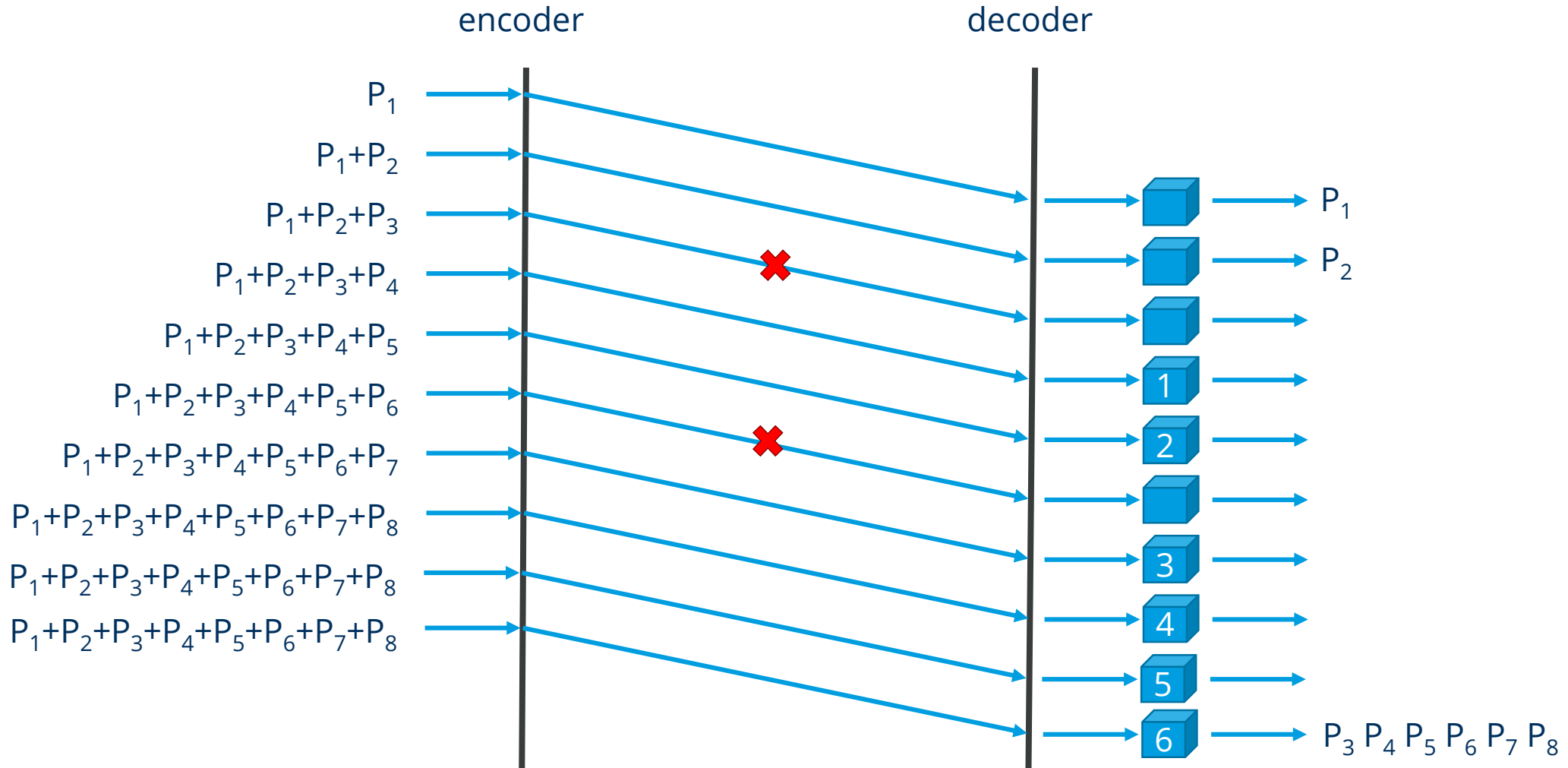
On the Fly RLNC

agnostic approach - error prone



On the Fly RLNC

agnostic approach - error prone



On the Fly RLNC

error aware approach

$$\begin{pmatrix} C_1 \\ \vdots \\ C_G \\ C_{G+1} \\ \vdots \\ C_K \end{pmatrix} = \begin{pmatrix} \alpha_{1,1} & \dots & \alpha_{1,G} \\ \vdots & \ddots & \vdots \\ \alpha_{G,G} \\ \vdots & \dots & \alpha_{G+1,G} \\ \vdots & \dots & \vdots \\ \vdots & \dots & \vdots \end{pmatrix} \begin{pmatrix} P_1 \\ \vdots \\ P_G \end{pmatrix}$$

On the Fly RLNC

Encoder

Content

Decoder

OnTheFlyEncoderFactoryBinary
~~OnTheFlyEncoderFactoryBinary16~~
OnTheFlyEncoderFactoryBinary4
OnTheFlyEncoderFactoryBinary8

Sliding Window RLNC

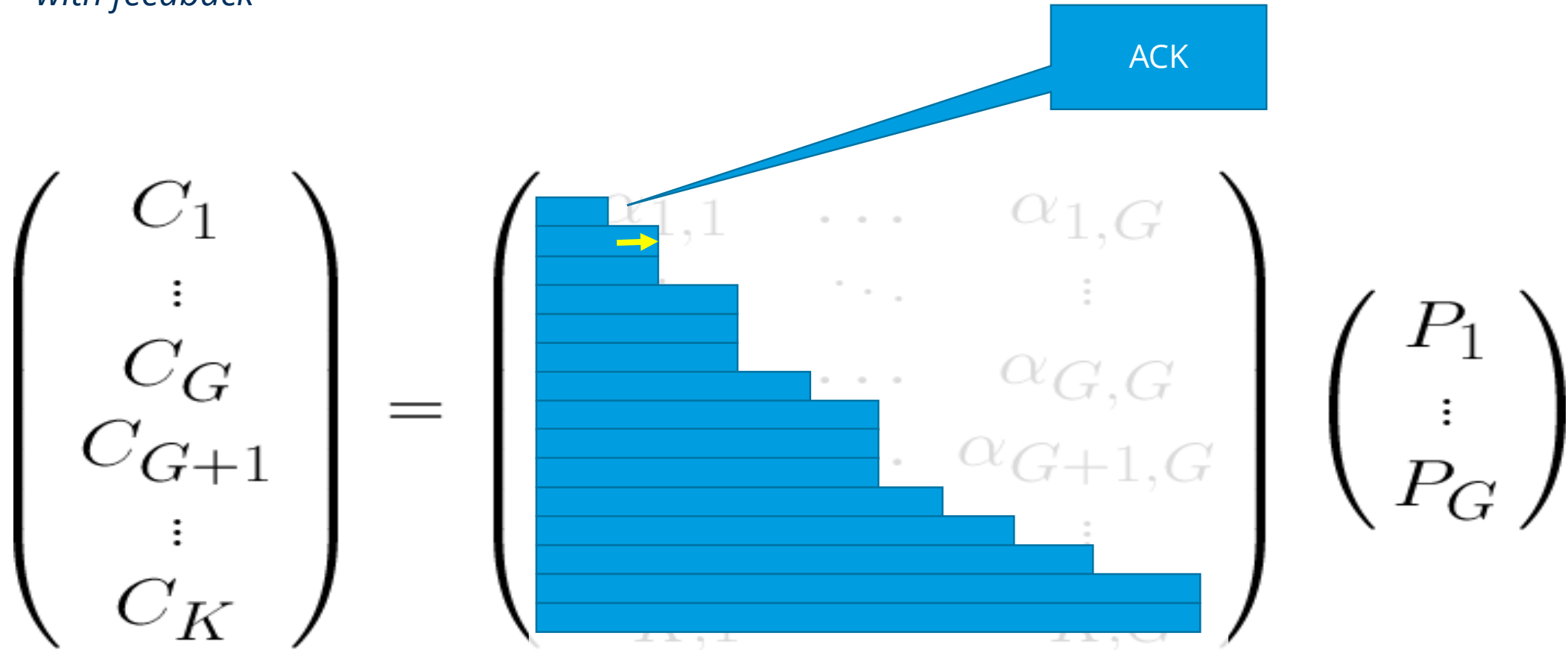
SlidingWindowEncoderFactoryBinary
SlidingWindowEncoderFactoryBinary16
SlidingWindowEncoderFactoryBinary4
SlidingWindowEncoderFactoryBinary8

Sliding Window

- So far we estimated the channel errors
- Would could we do if we would know about errors that happen?
- First of all we need a feedback

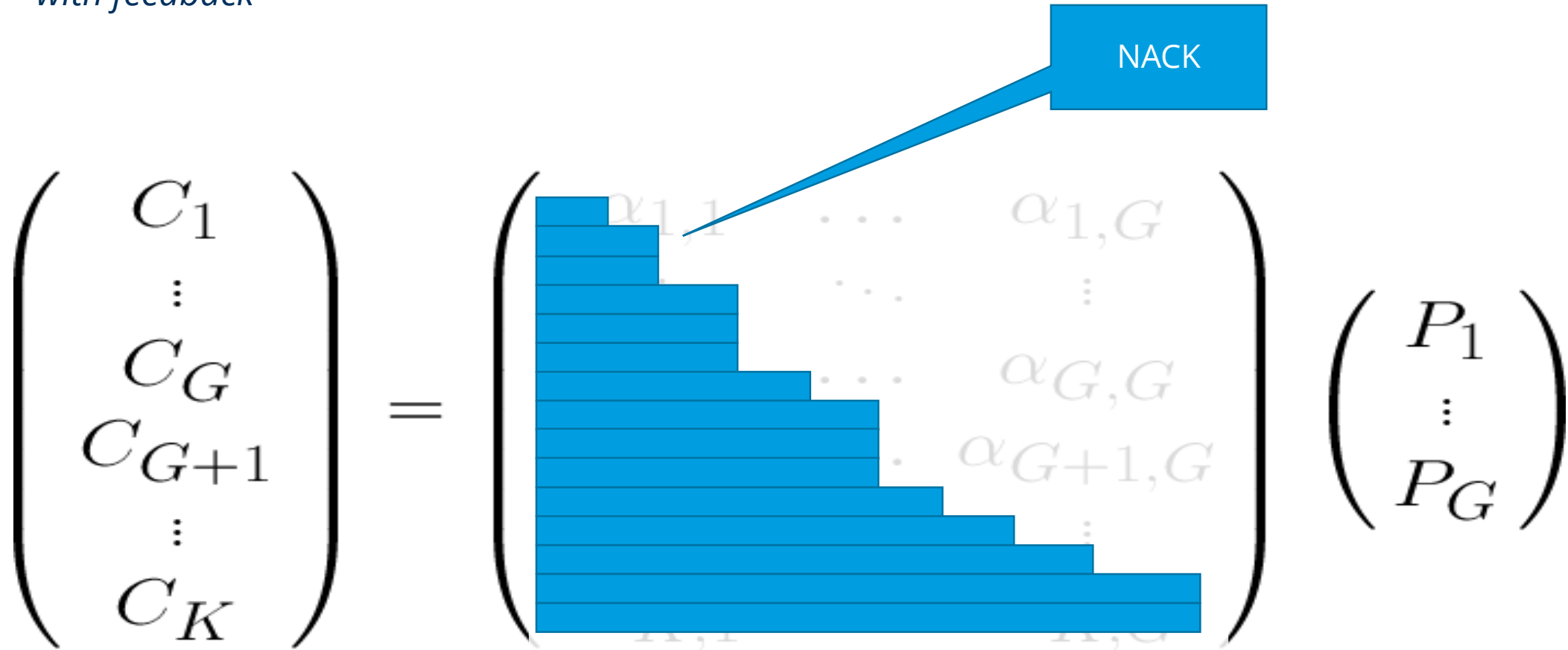
Sliding Window

with feedback



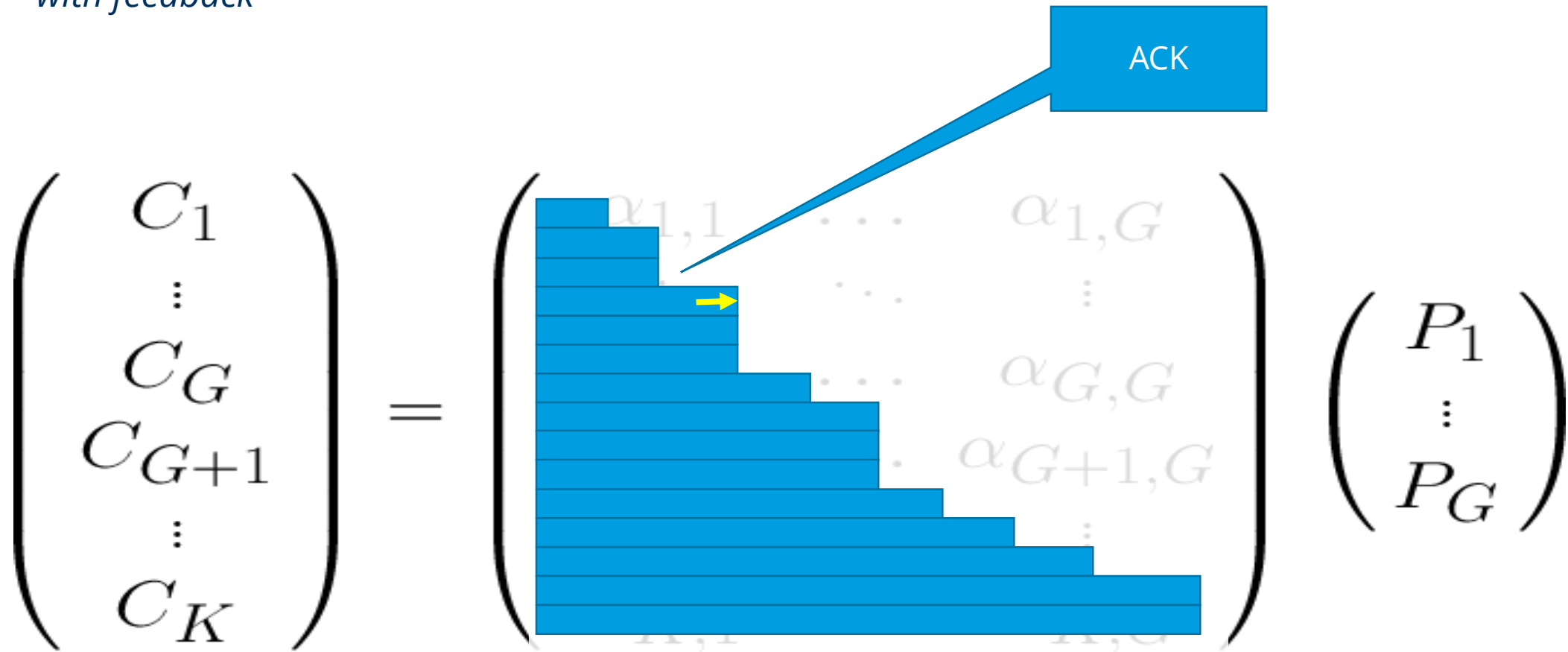
Sliding Window

with feedback



Sliding Window

with feedback



Sliding Window

- So far we estimated the channel errors
- Would could we do if we would know about errors that happen?
- First of all we need a feedback
- Opening the window on ACK reception
- Closing the window to reduce complexity

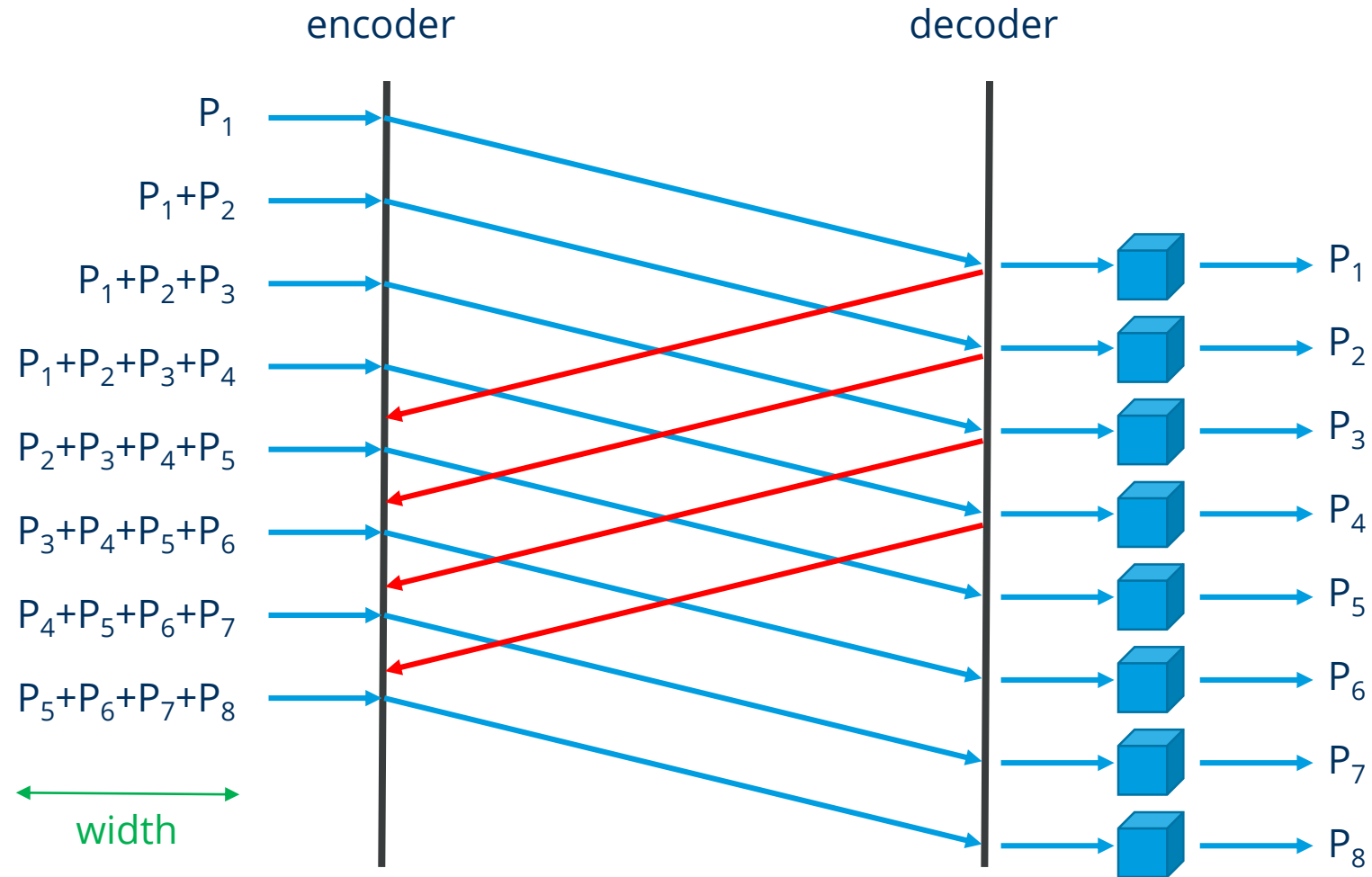
Sliding Window

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} & \alpha_{1,5} & \alpha_{1,6} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} & \alpha_{2,5} & \alpha_{2,6} \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} & \alpha_{3,5} & \alpha_{3,6} \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ \alpha_{5,1} & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

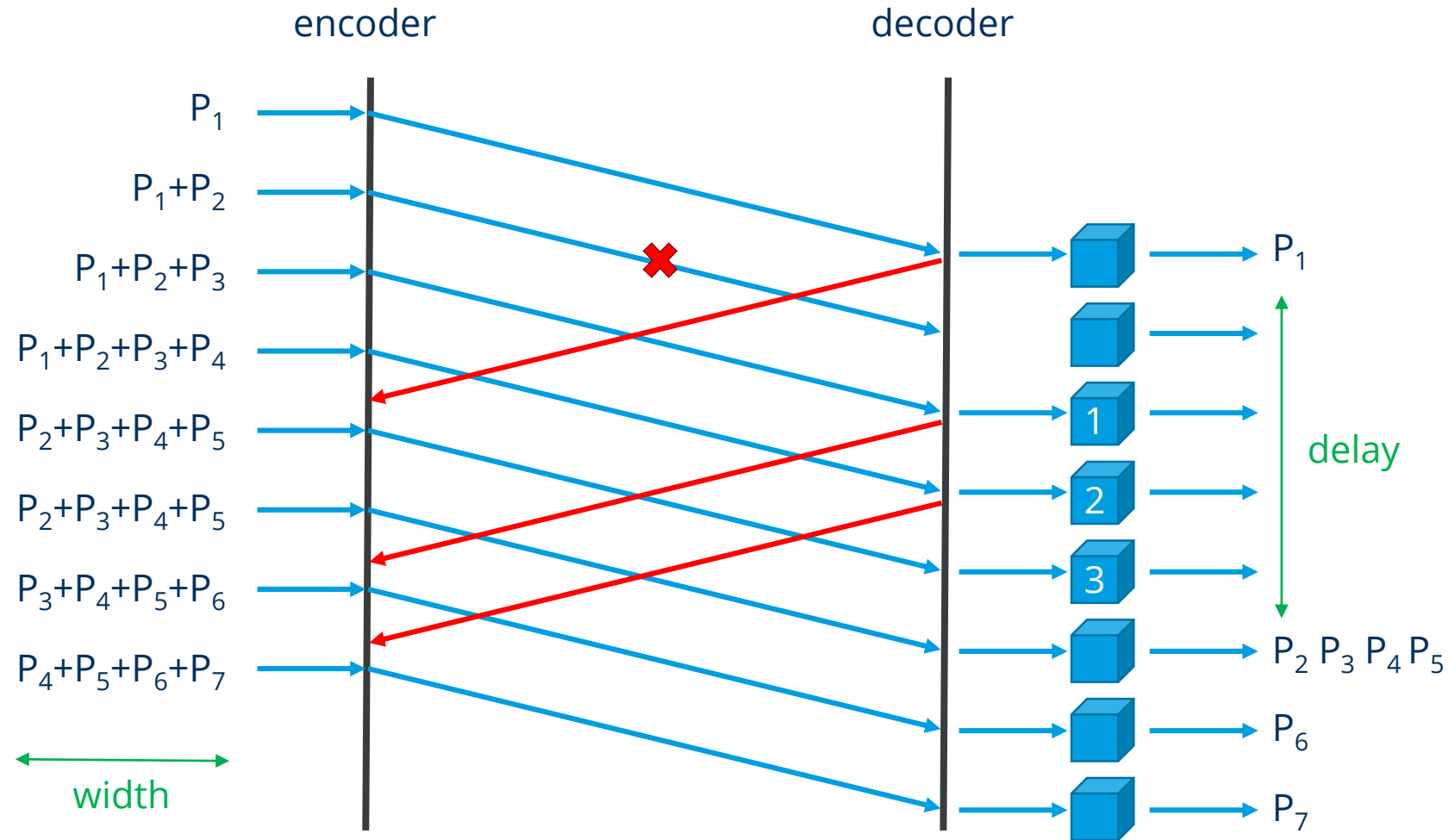
Sliding Window

- So far we estimated the channel errors
- Would could we do if we would know about errors that happen?
- First of all we need a feedback
- Opening the window on ACK reception
- Closing the window to reduce complexity
- Width of the window depends on propagation delay and error characteristics

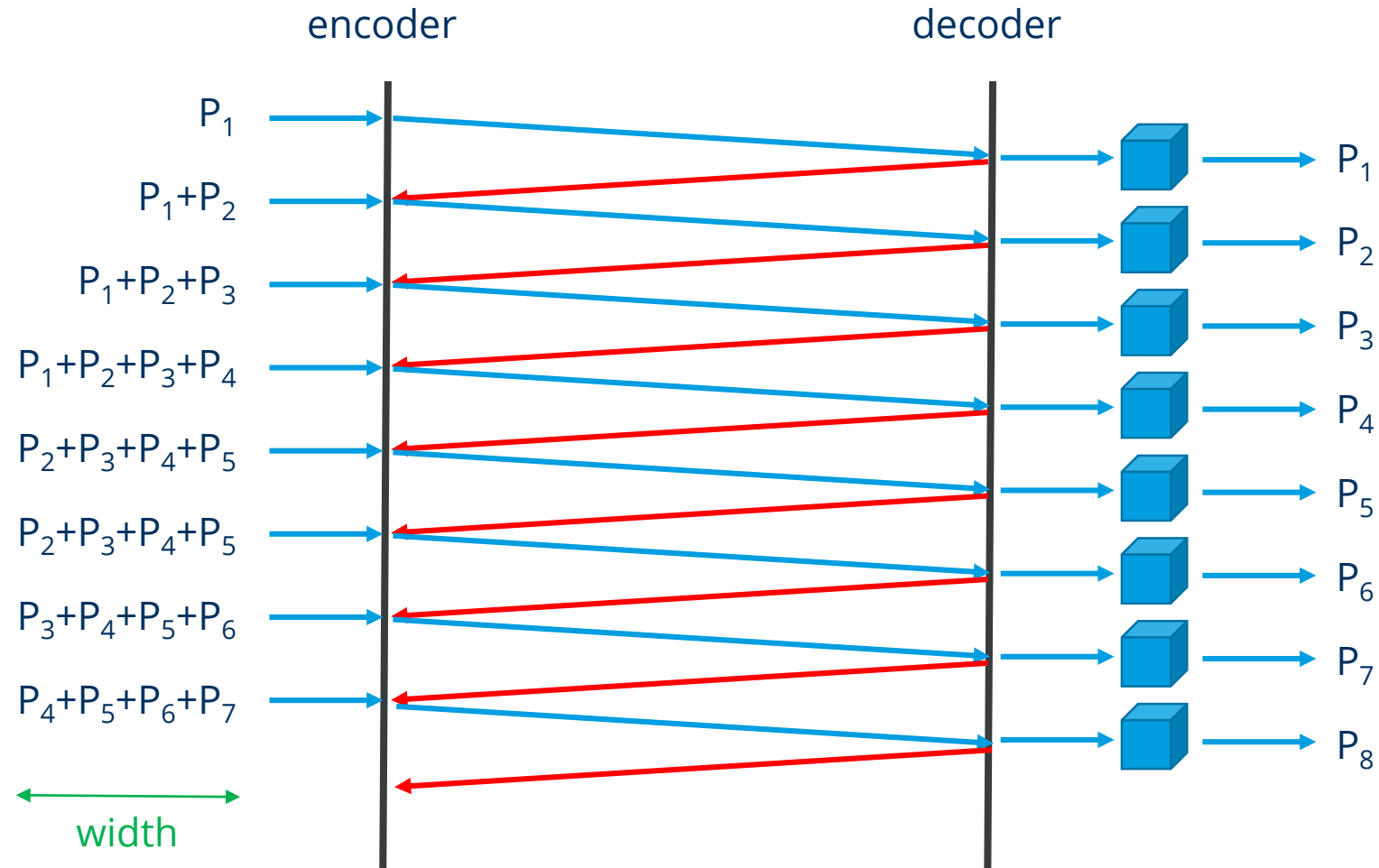
Sliding Window



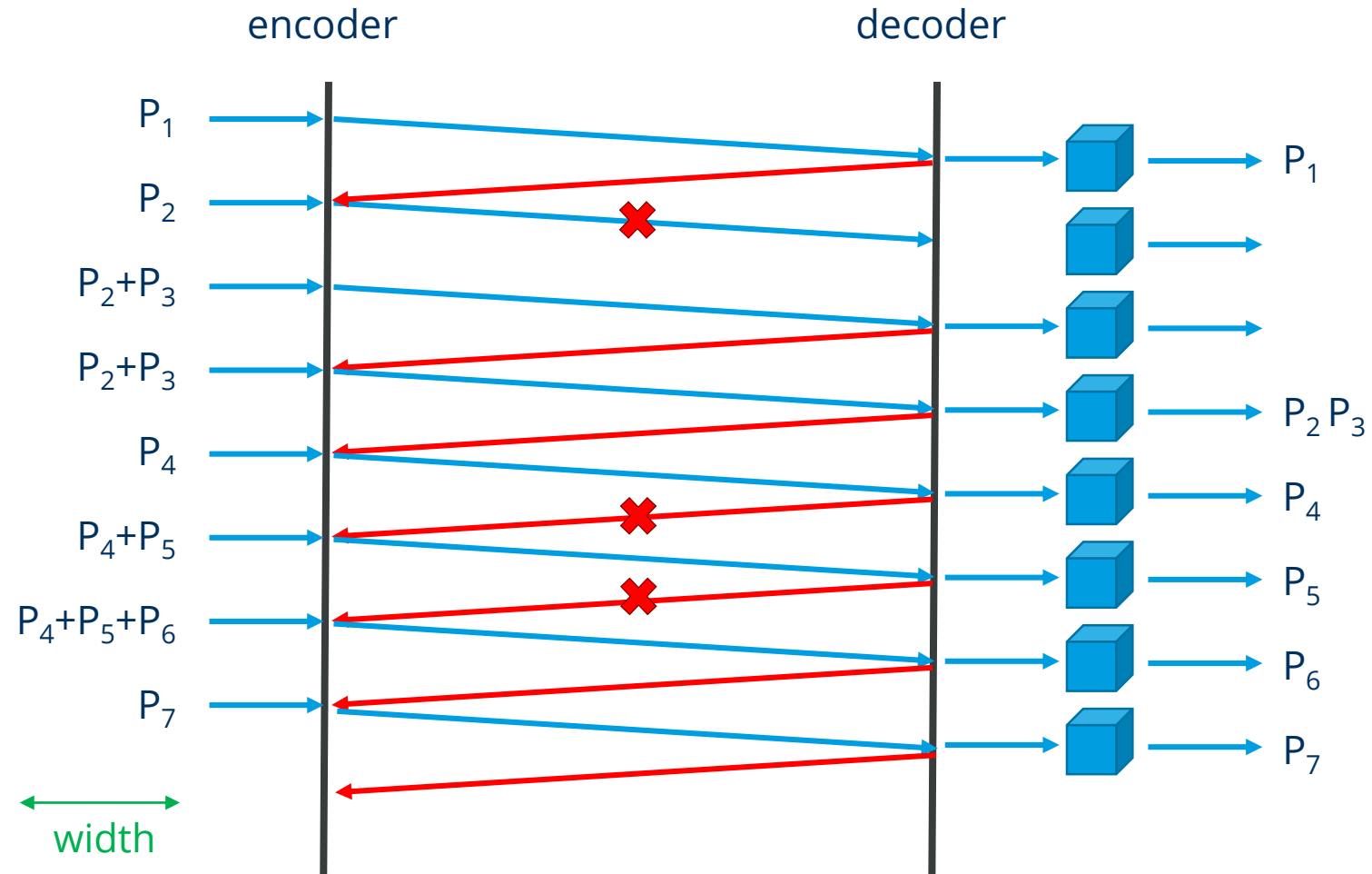
Sliding Window



Sliding Window

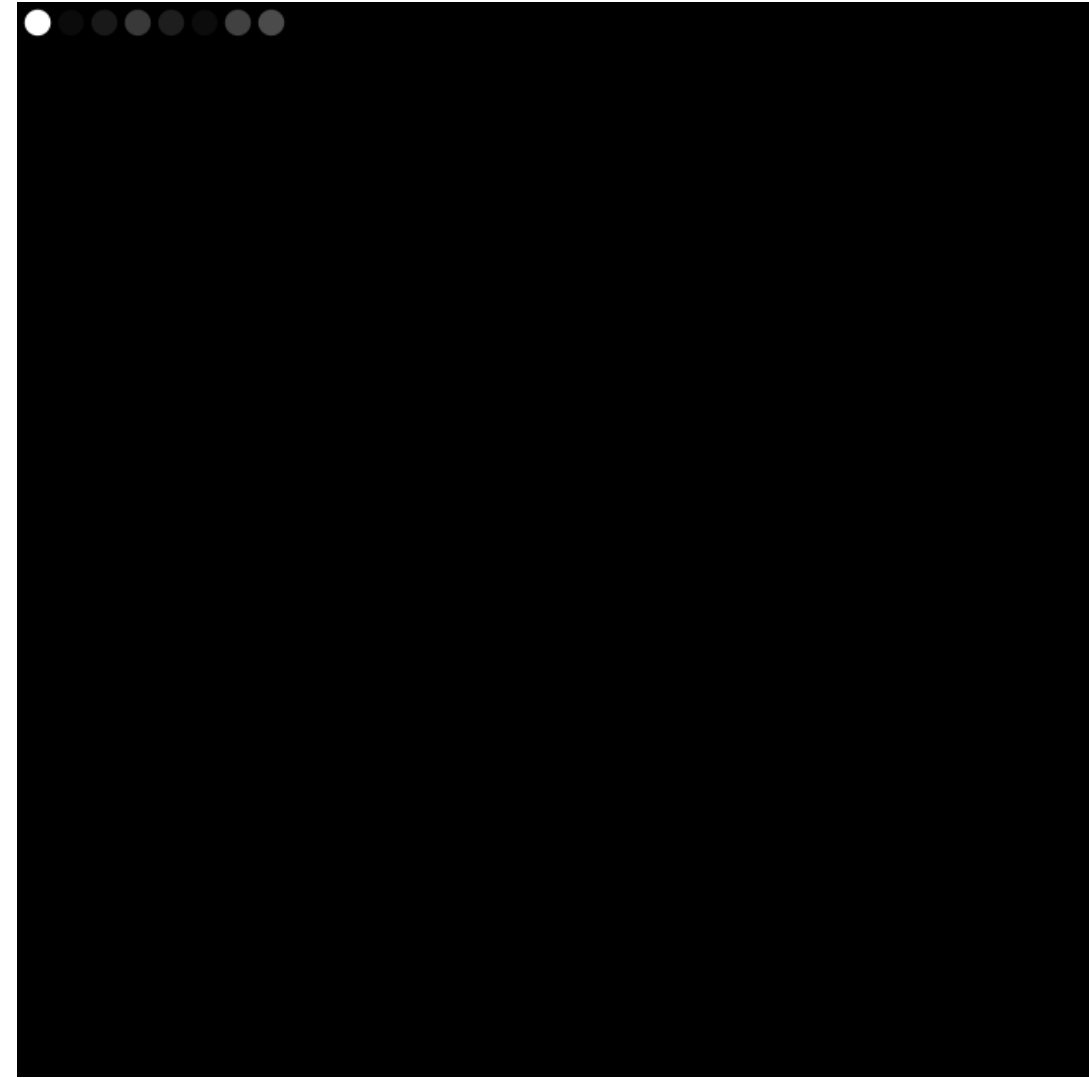


Sliding Window



Sliding Window

- Terminology
 - On the fly encoding/decoding: Newly incoming packets are added to the block.
 - Sliding window: Newly incoming packets are added to the block and acknowledged packets are removed.
 - Sliding window can work without blocks



Sliding Window

Encoder

Content

Decoder

SlidingWindowEncoderFactoryBinary
~~SlidingWindowEncoderFactoryBinary16~~
SlidingWindowEncoderFactoryBinary4
SlidingWindowEncoderFactoryBinary8

Sliding Window

Fulcrum RLNC

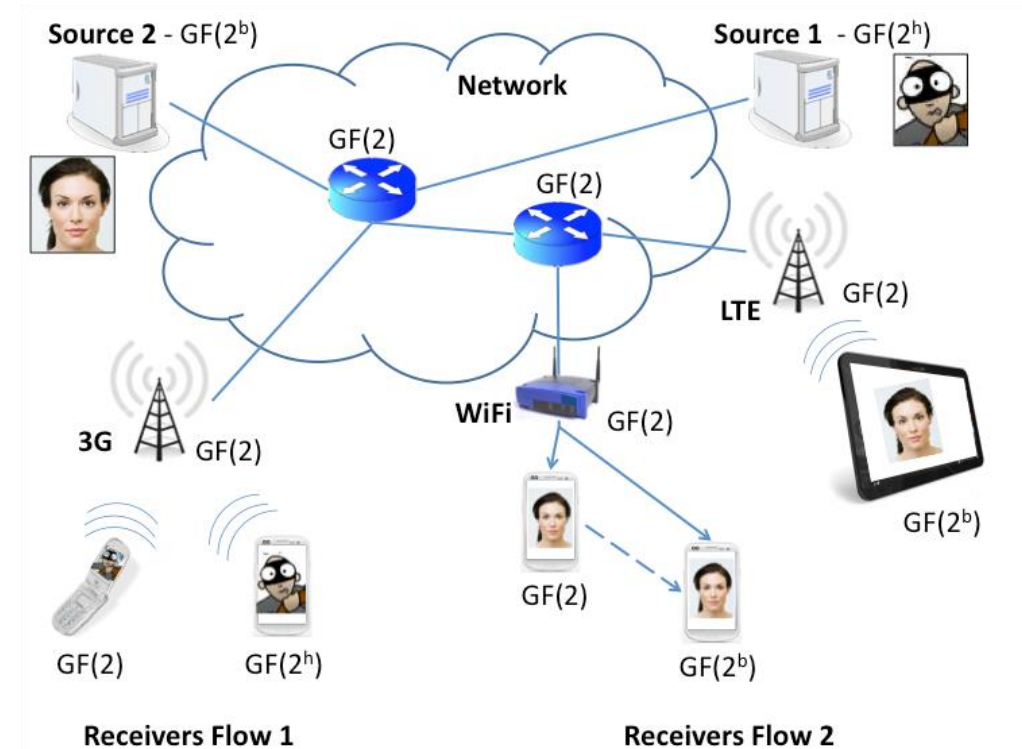
```
FulcrumEncoderFactoryBinary  
FulcrumEncoderFactoryBinary16  
FulcrumEncoderFactoryBinary4  
FulcrumEncoderFactoryBinary8
```

Problem: Heterogeneity in Display Size



General Ideas

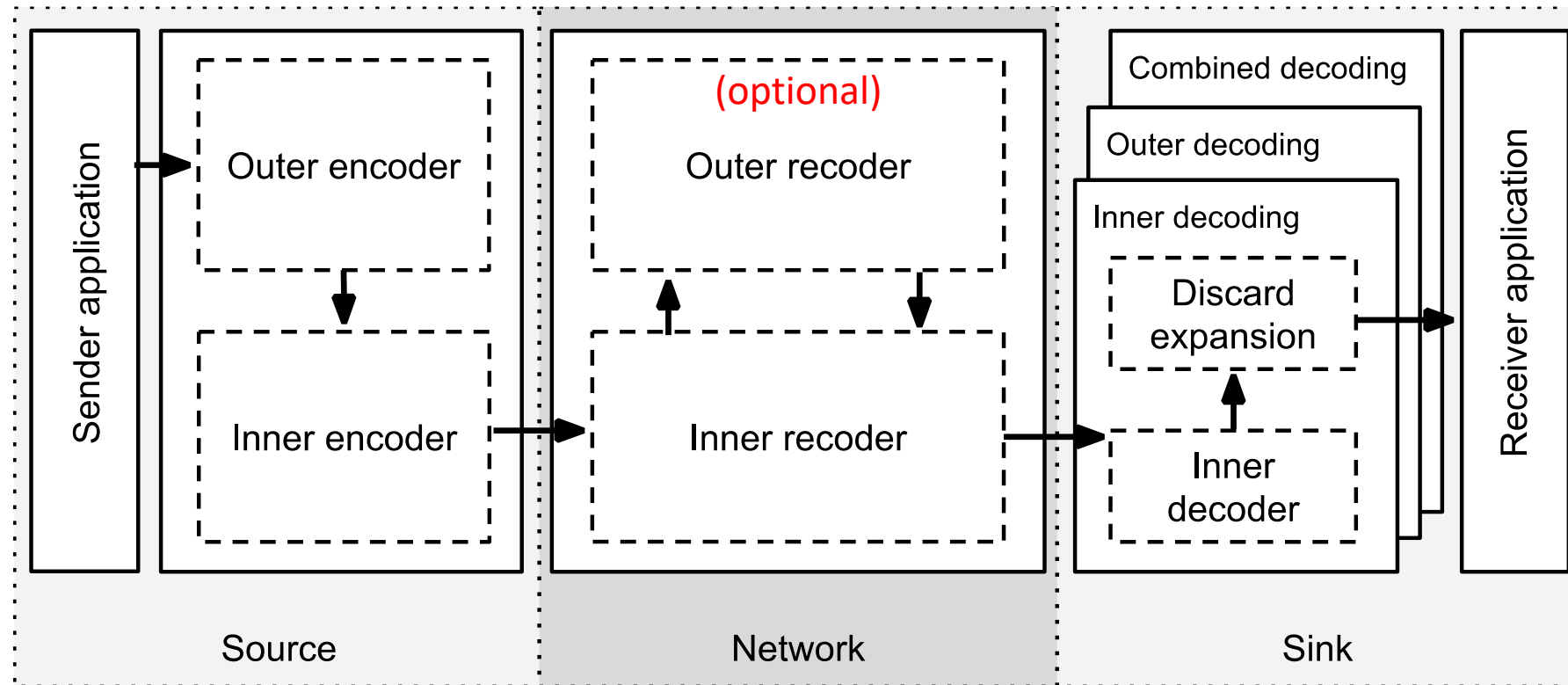
- Fluid allocation of complexity
- End devices agree on desired performance:
 - Independent from network
 - Chosen according to application requirements
- Network devices need only support a simple subset of functions
- Reduces overhead
 - Roughly 1 bit per coding coefficient
- Key: code concatenation with different field sizes



Benefits

- Simple is green, compatible, deployable
- Supports heterogeneous receivers
- Adaptive performance
- Practical recoding
- Spreads complexity to stronger devices
- Security: simple support
- New designs can be supported: backwards compatible

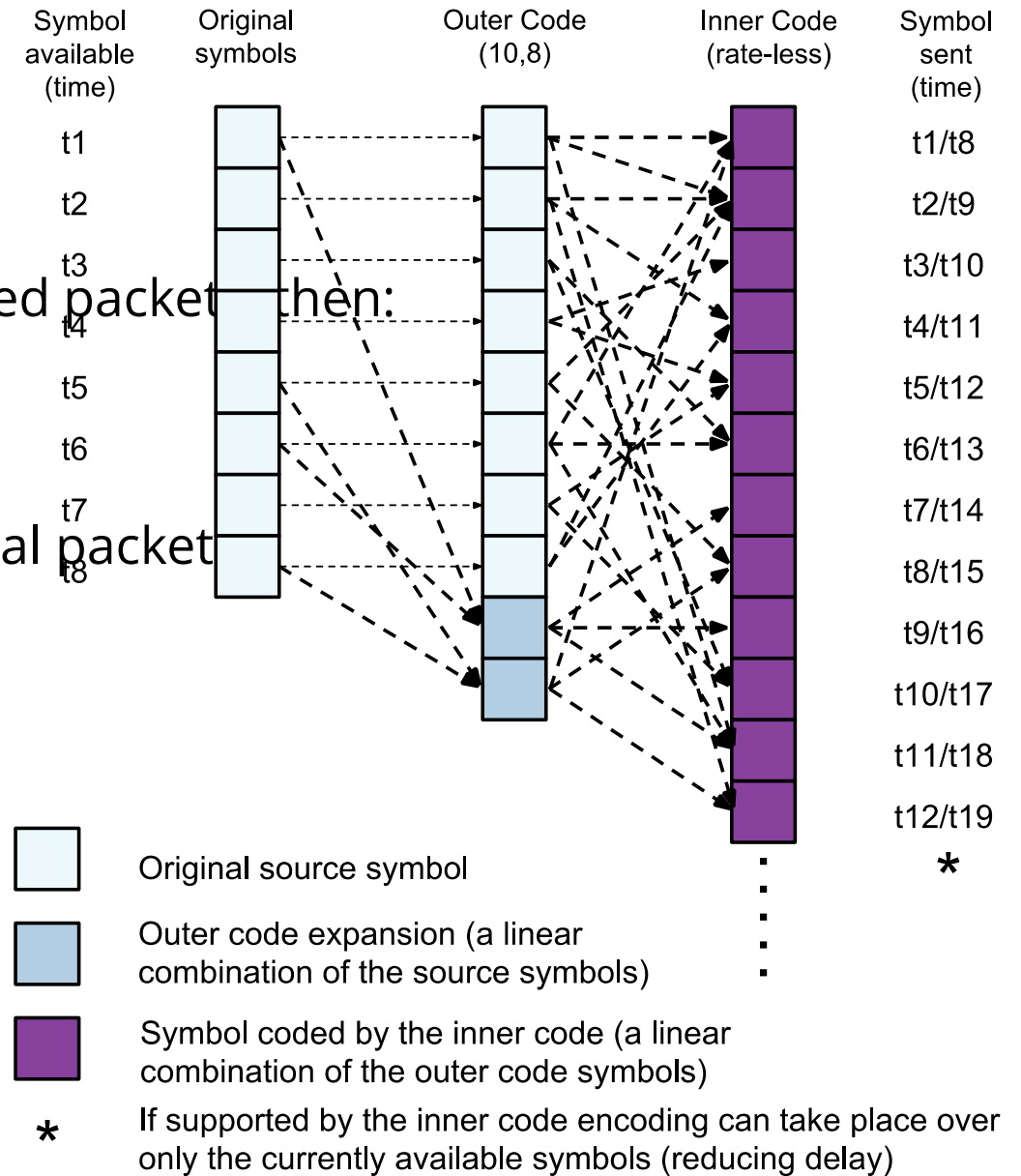
General Structure



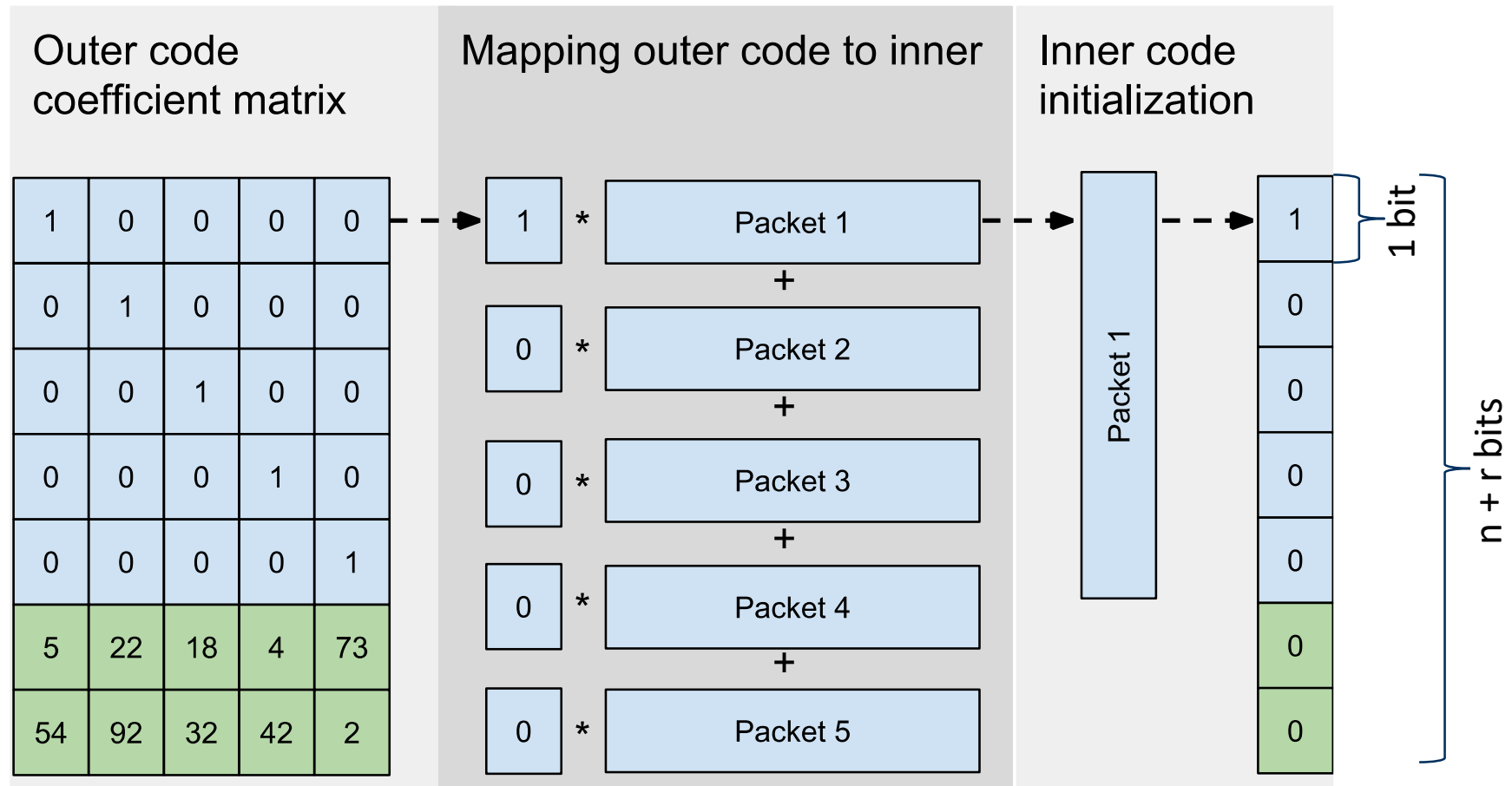
- Inner code: RLNC, Sparse RLNC, Perpetual, ... $GF(2)$
- Outer code: (systematic) RLNC, Reed-Solomon, ... $GF(2^h)$

Encoder

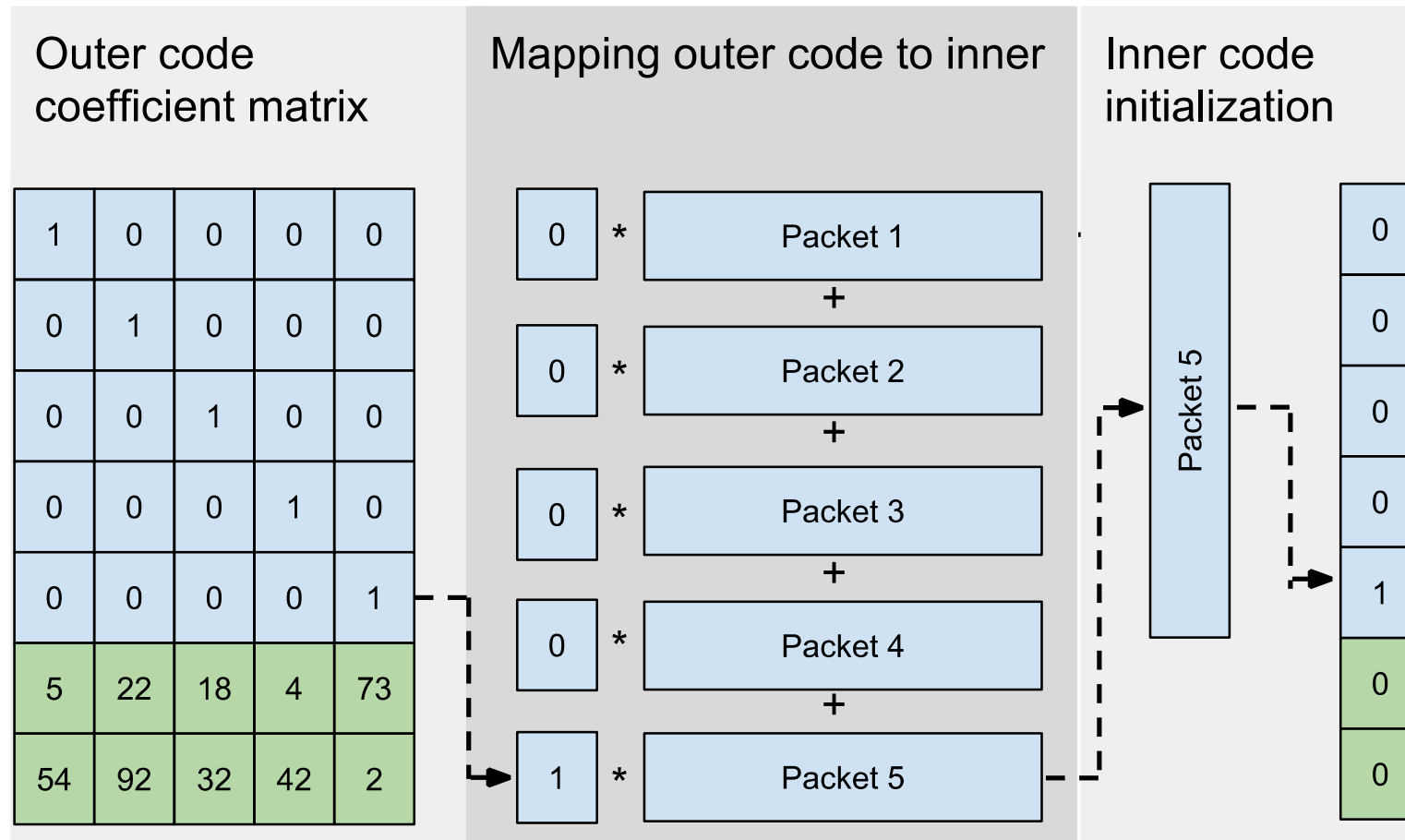
- Since inner code is created in GF(2) with pre-coded packet then:
- $n+r$ coefficients per inner coded packet
- This means: $1 + r/n$ bits per coefficient per original packet



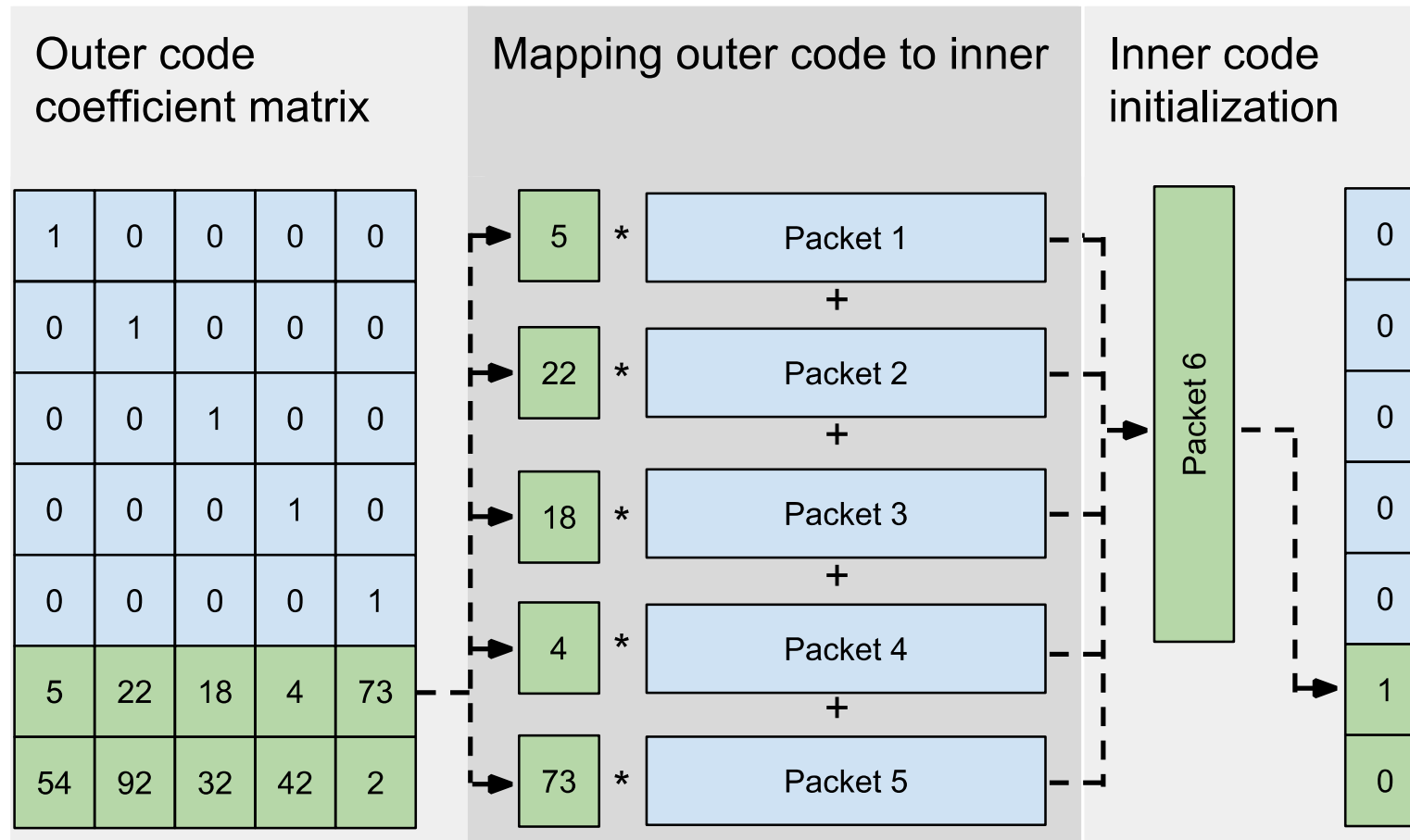
Encoder



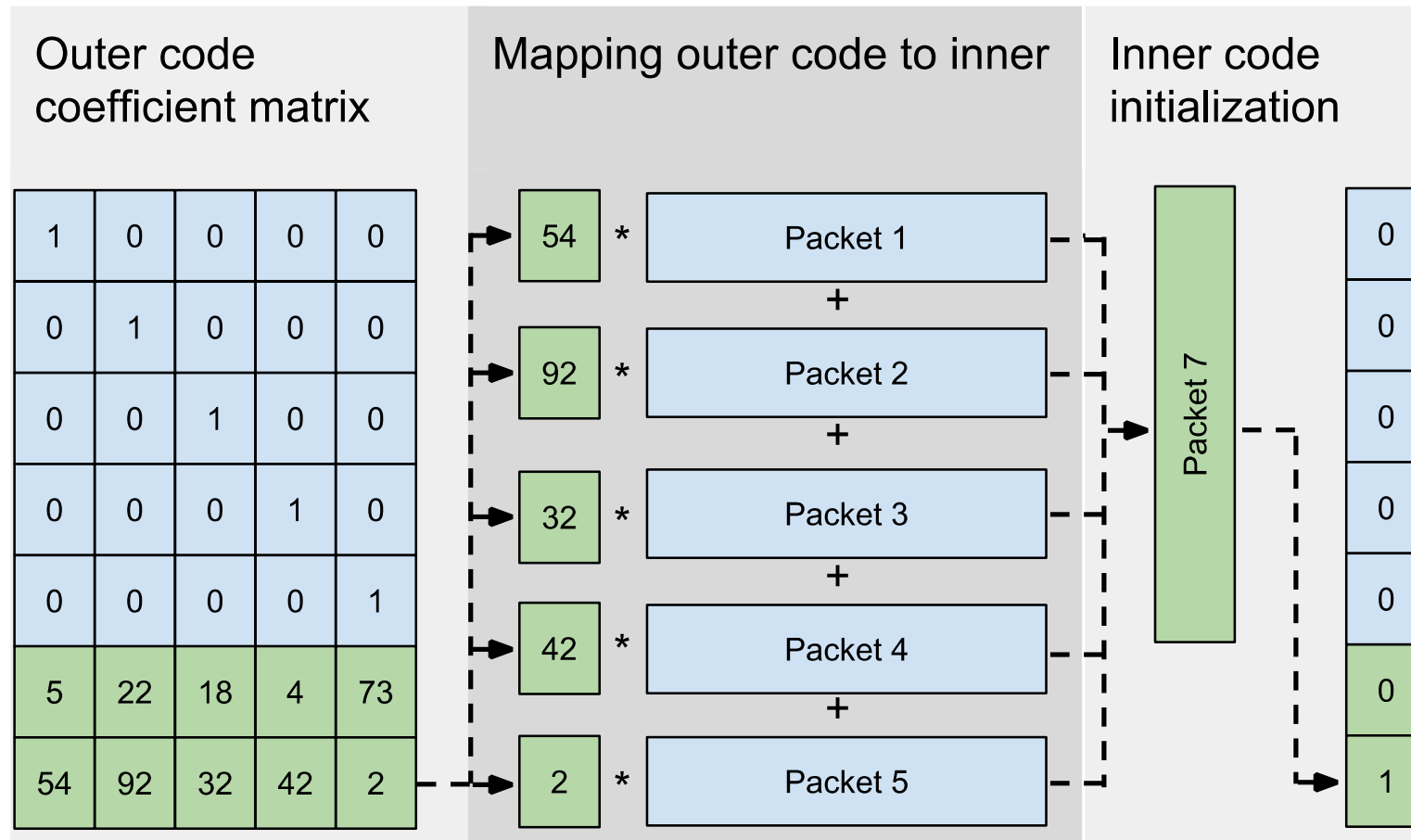
Encoder



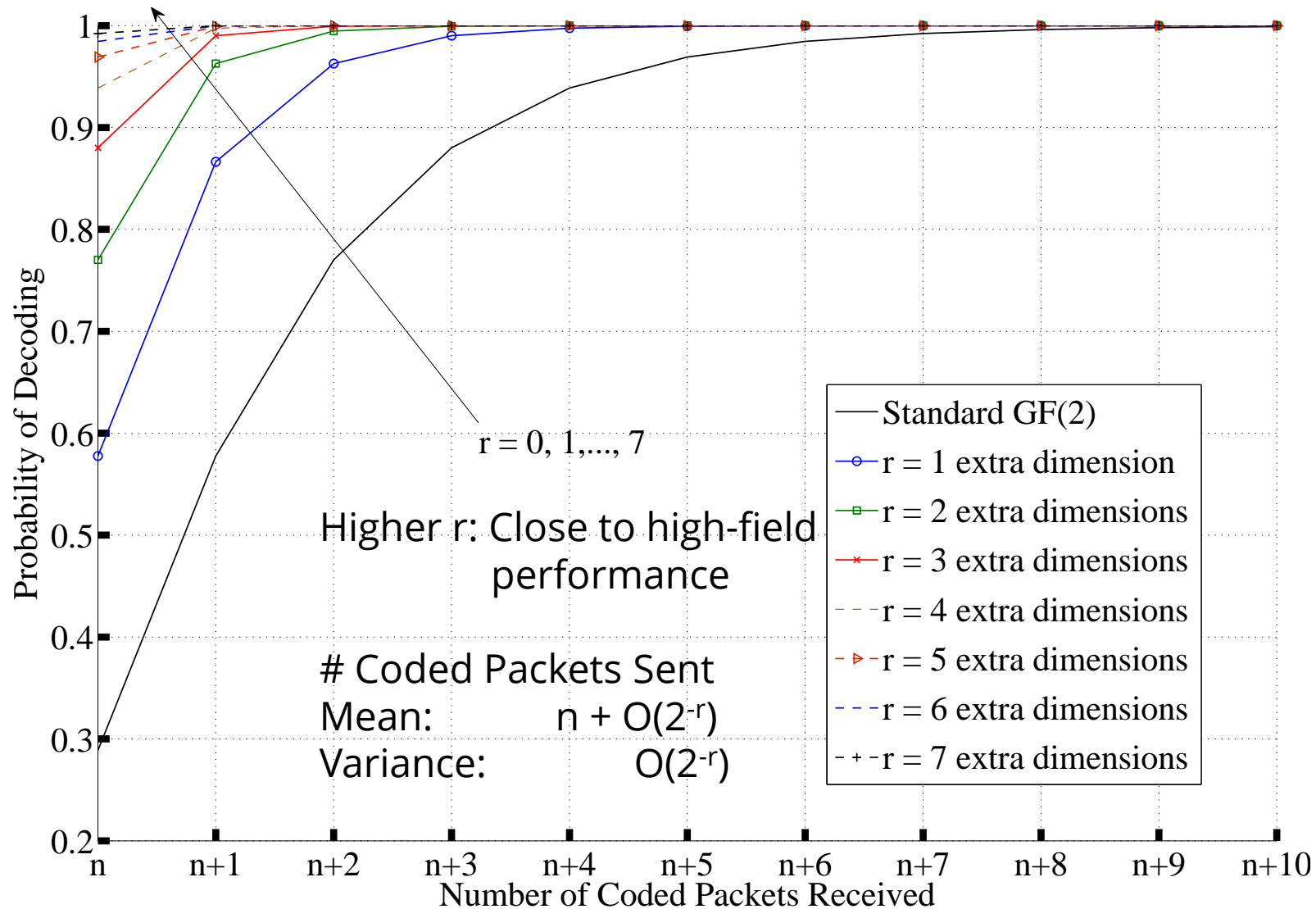
Encoder



Encoder



Received Packets before Decoding

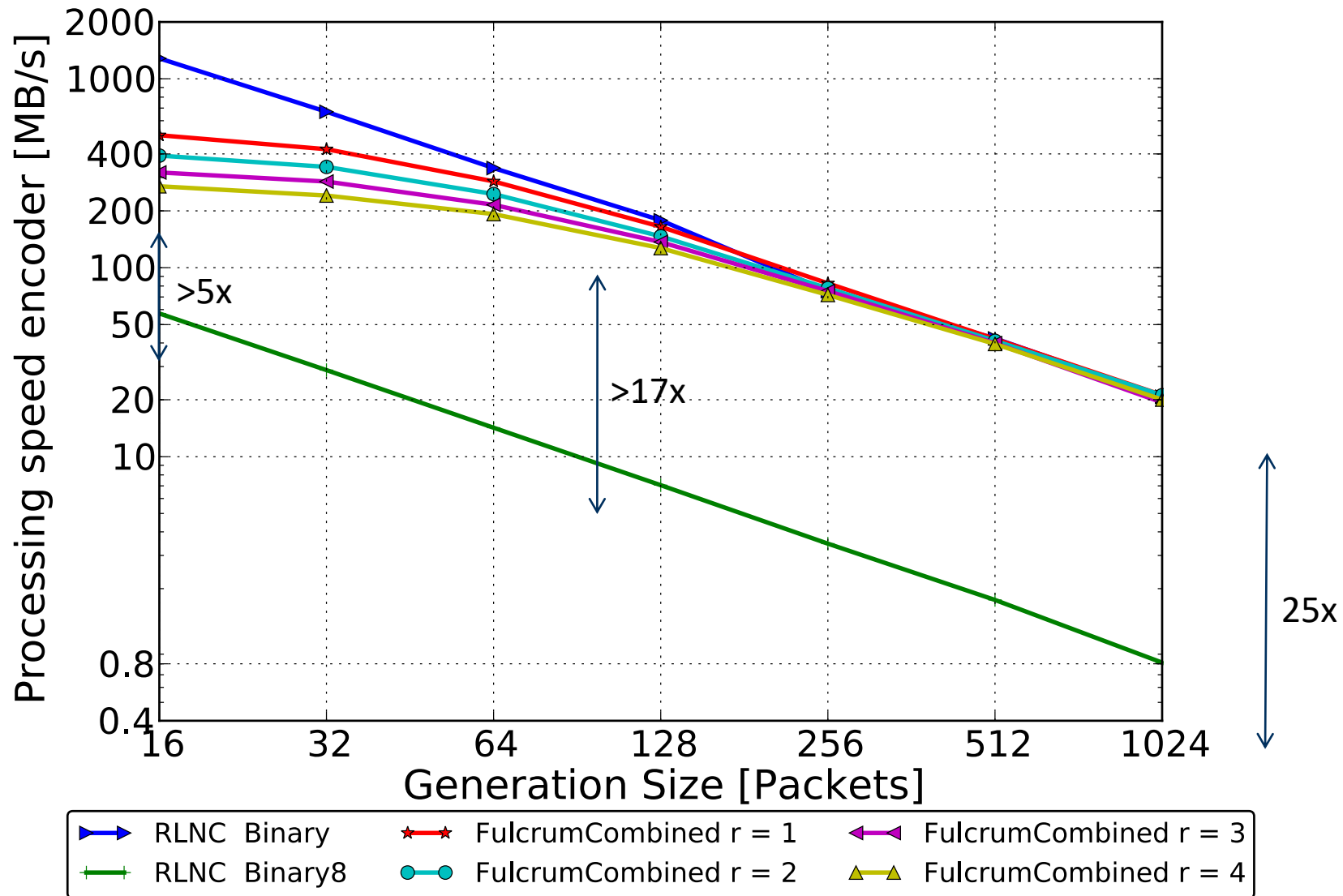


Received Packets before Decoding

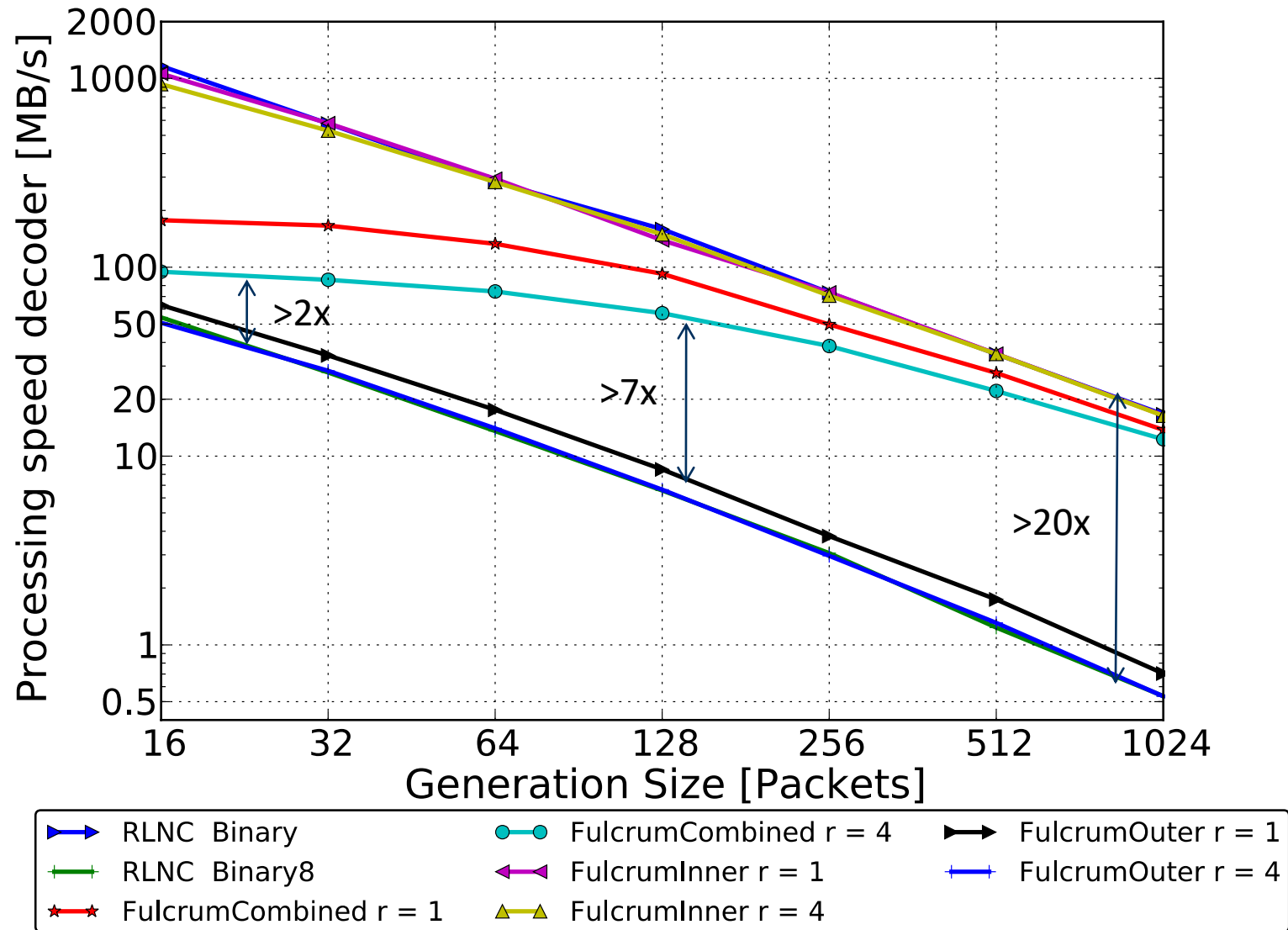
Decoding after receiving (coded packets)					
Code		n	n + 1	n + 2	n + 3
Fulcrum	r = 4	93.87%	99.75%	99.99%	99.9997%
	r = 7	99.22%	99.996%	99.99998%	99.99999992%
	r = 10	99.90%	99.9999%	99.99999996%	99.99999999998%
RaptorQ*		99%	99.99%	99.9999%	

* Qualcomm. (2013, Dec.) Raptorq - the superior fec technology
 Available: <http://www.qualcomm.com/media/documents/raptorq-data-sheet>

Performance Results: Encoder



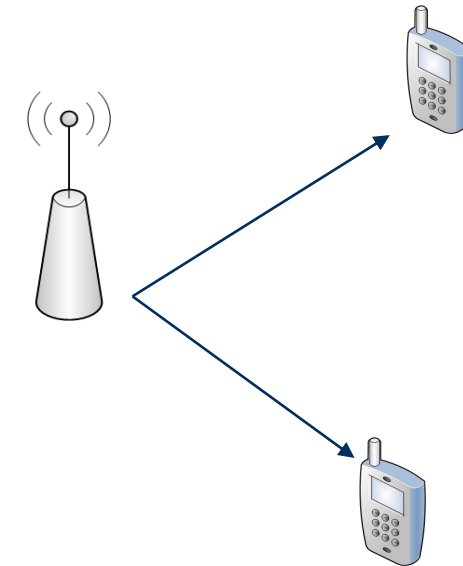
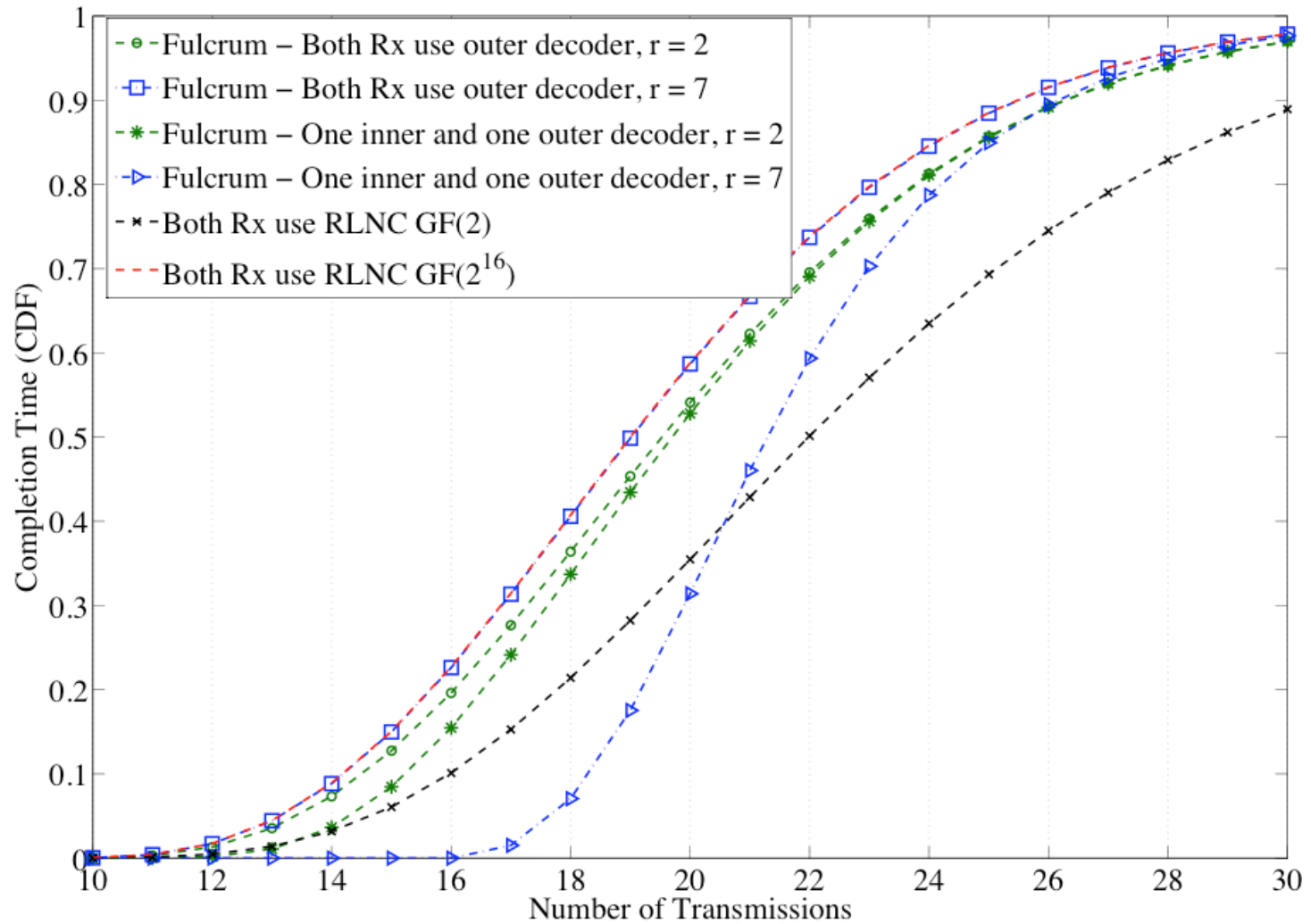
Performance Results: Decoder



Advantages

- Low overhead:
 - $1 + r/n \sim 1$ bit per coefficient per packet \rightarrow like GF(2)
 - Total transmitted packets: $n + O(2-r) \rightarrow$ like higher fields
- Processing speed (complexity) compared to GF(28):
 - Encoder 5x to 25x faster
 - Decoder 2x to $>20x$ faster
- Supports heterogeneous receivers
- Allows a fluid allocation of complexity
- Simple security support
- Network can implement a bare minimum:
 - Just XOR packets!

Heterogeneous Users



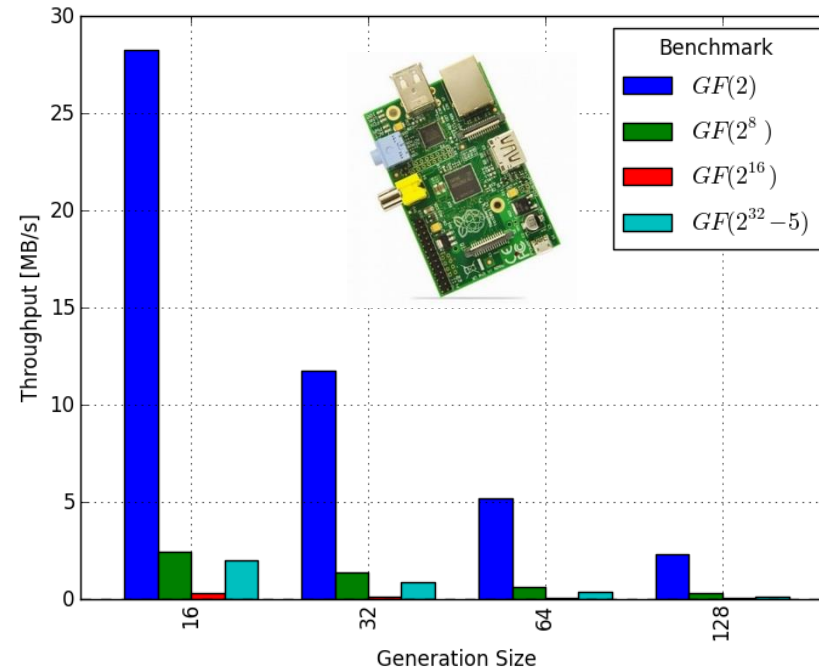
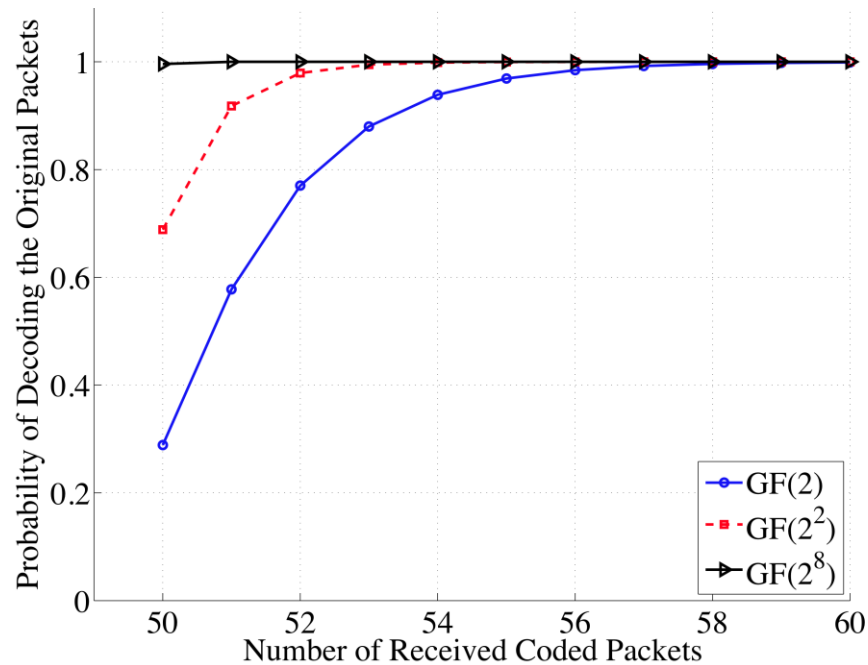
Composite Extension Finite Fields for Low Overhead Network Coding: Telescopic Codes

Nestor Hernandez, Janus Heide, Daniel Enrique Lucani Roetter, and Frank Hanns Paul Fitzek, "On the Overhead of Telescopic Codes in Network Coded Cooperation," I E E E V T S Vehicular Technology Conference. Proceedings, 2015.

Introduction

- Network coding is a promising paradigm
 - Store-code-forward instead of store-forward
 - Benefits in throughput, delay, robustness, energy
- Caveats:
 - Computational complexity
 - Overhead per packet, e.g., signaling of coding coefficients

Motivation

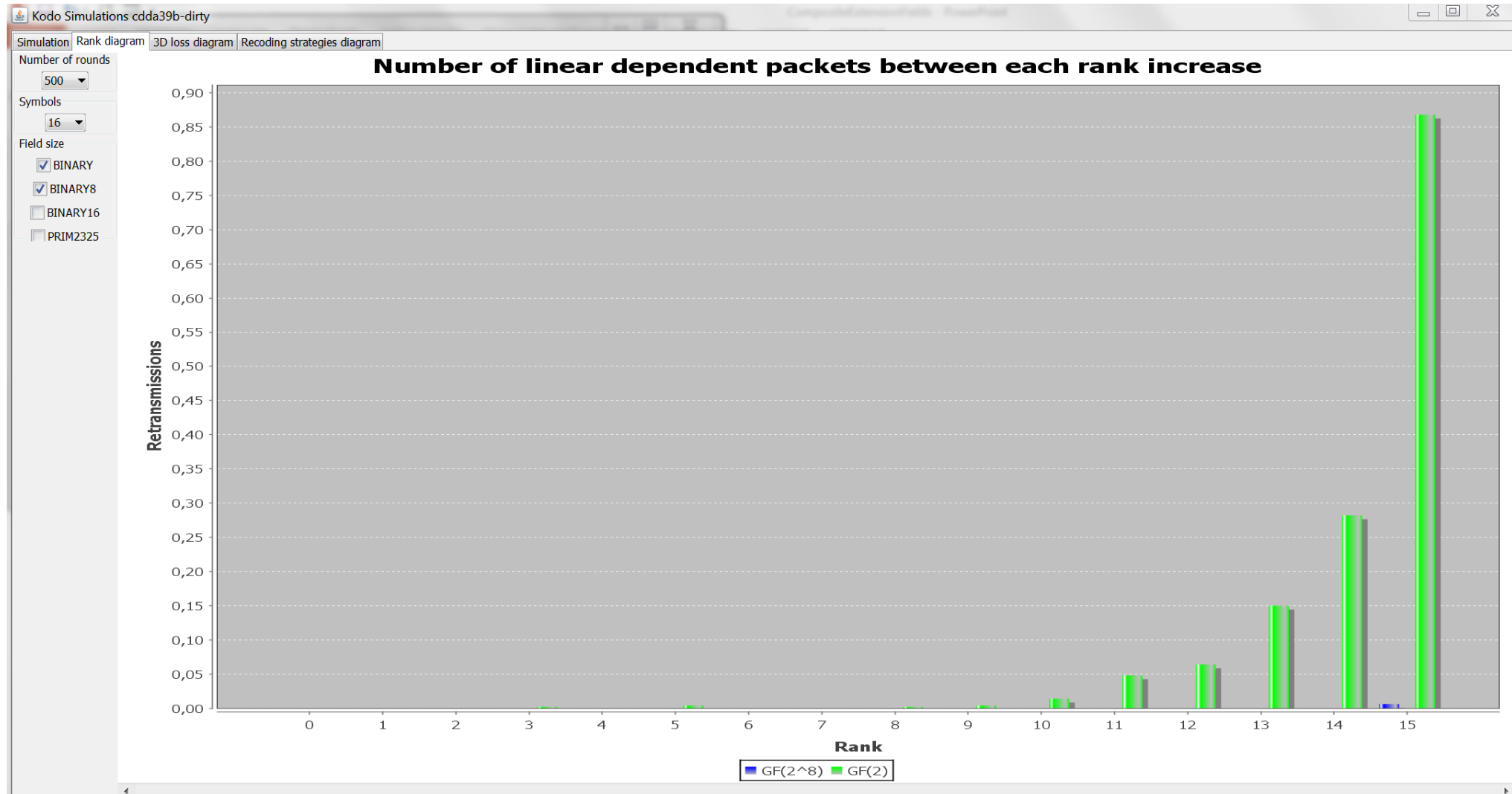


Larger fields
are more efficient!

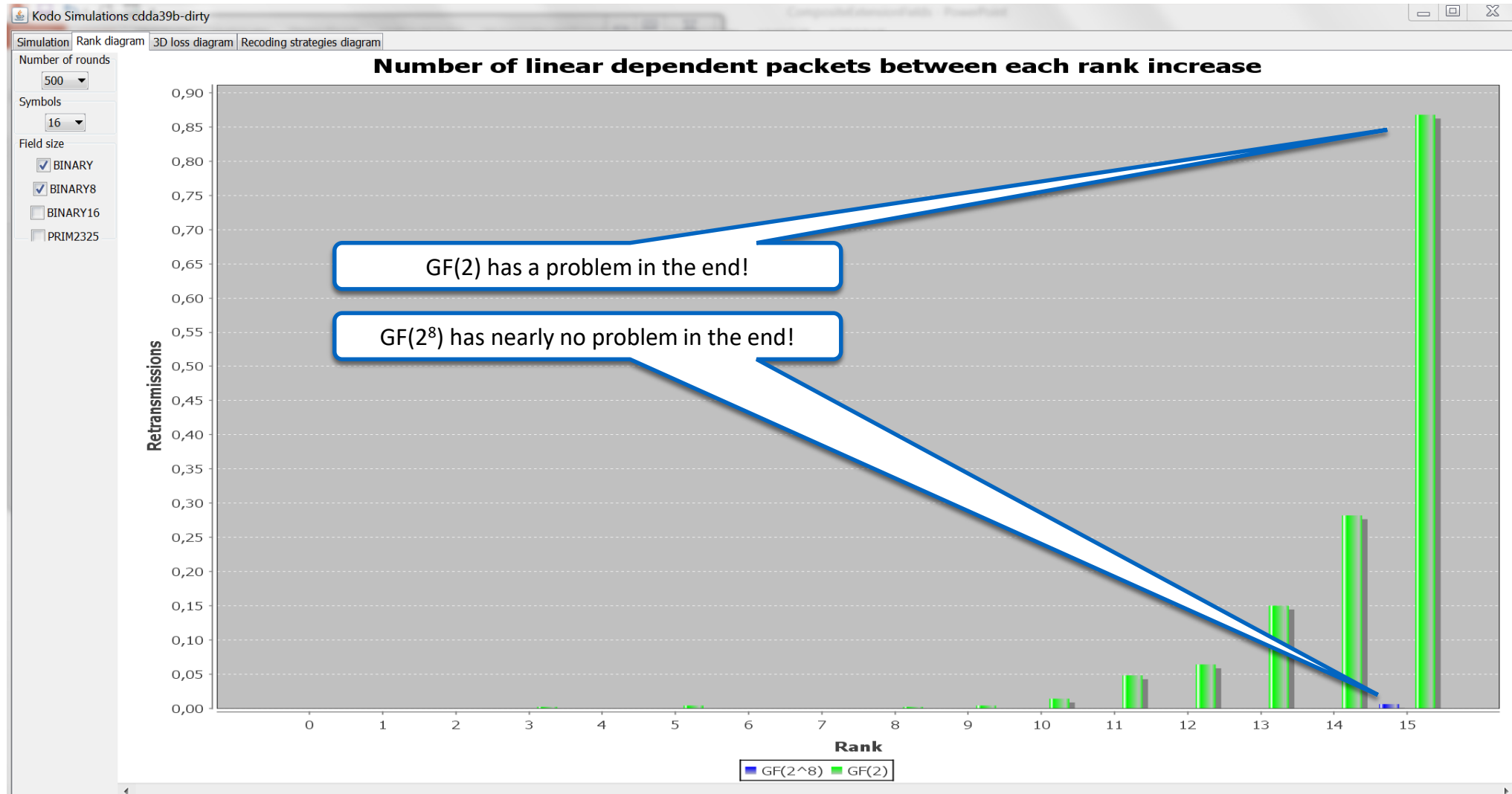
Smaller fields
are faster and need less
overhead!



Motivation



Motivation



Introduction

Previous work:

- Intrinsic information conveyance in network coding by Heide/Pedersen/Fitzek/Zhang
- WO2011114191A1
- Understanding single choice of field size, generation size on processing speed [Paramanathan et al 2013]
- Techniques that exploit sparsity: tunable sparse network coding, overlapping generations, BATS codes, ...
- Fulcrum network codes: low overhead, high processing speed, simple recoding, configurable performance [Lucani et al 2014]

Motivation

Fulcrum provides support using composite fields:

$GF(2)$ in the network and any $GF(2^k)$ at source and destinations

Key: operations performed in $GF(2)$ have an equivalent operation in any $GF(2^k)$

Example: $P1 + P2$ requires a bit by bit XOR in both cases

Can we expand this idea for more flexibility in code design?

Typically, $GF(2^k)$ is not compatible with a $GF(2^p)$ if $n \neq p$

Different primitive polynomials \rightarrow different mapping for product operations

Motivation

GF(2²) polynomial: X² + X + 1
 GF(2⁴) polynomial: X⁴ + X + 1

- Typically, GF(2^k) is not compatible with a GF(2^p) if n ≠ p
- Example: GF(2²) versus GF(2⁴)

	GF(2 ²)		
x	1	2	3
1	1	2	3
2	2	3	1
3	3	1	2

≠

	GF(2 ⁴)						
x	1	2	3	4	5	...	15
1	1	2	3	4	5	...	15
2	2	4	6	8	10	...	13
3	3	6	5	12	15	...	2
4	4	8	12	3	7	...	9
5	5	10	15	7	2	...	6
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
15	15	13	2	9	6	...	10

Motivation

GF(2²) polynomial: X² + X + 1
 GF(2⁴) polynomial: X⁴ + X + 1

- Typically, GF(2^k) is not compatible with a GF(2^p) if n ≠ p
- Example: GF(2²) versus GF(2⁴)

	GF(2 ²)			
×	1	2	3	
1	1	2	3	≠
2	2	3	1	
3	3	1	2	

	GF(2 ⁴)						
x	1	2	3	4	5	...	15
1	1	2	3	4	5	...	15
2	2	4	6	8	10	...	13
3	3	6	5	12	15	...	2
4	4	8	12	3	7	...	9
5	5	10	15	7	2	...	6
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
15	15	13	2	9	6	...	10

- Actually, we will need even stronger conditions
- Idea: Multiple composite extension fields

Multiple Composite Extension Fields

Use $GF(2)$ to construct $GF(2^2)$, use $GF(2^2)$ to construct $GF(2^{2^2})$

$GF(2^2)$			
x	1	2	3
1	1	2	3
2	2	3	1
3	3	1	2

=

		$GF(2^{2^2})$						
x		1	2	3	4	5	...	15
1		1	2	3	4	5	...	15
2		2	3	1	8		...	5
3		3	1	2	12		...	10
4		4	8	12	6		...	1
5		5					...	
⋮		⋮	⋮	⋮	⋮	⋮	⋮	⋮
15		15	5	10	1		...	9

Full compliance: product by 1, 2, 3 in $GF(2^{2^2})$ (4 bits)
Is equivalent to the product on two values of $GF(2^2)$ (2 bits)

Multiple Composite Extension Fields

Use $GF(2)$ to construct $GF(2^2)$, use $GF(2^2)$ to construct $GF(2^{2^2})$

Example: $15 \times 2 = 5$

$1111_b \times 0010_b$ in $GF(2^{2^2})$

But in $GF(2^2)$ we operate on pairs of bits

$11_b \times 10_b = 01_b$

Thus, on a vector

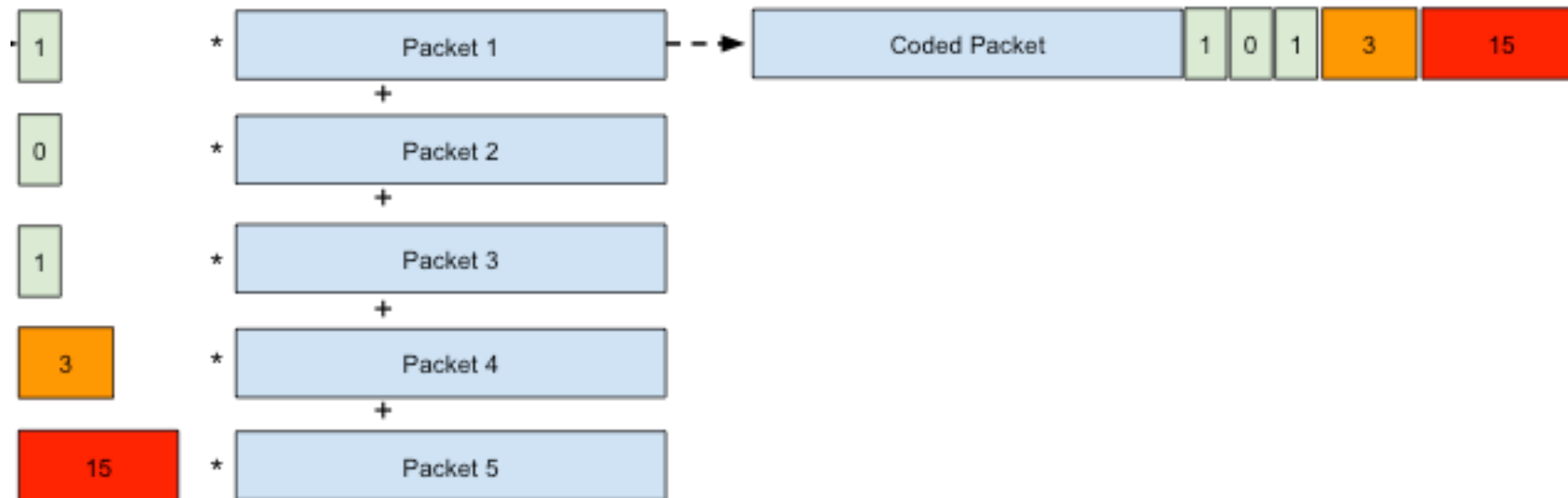
$11_b 11_b \times 10_b = 01_b 01_b$

Which results in “5” for the larger field

		$GF(2^{2^2})$						
x	1	2	3	4	5	...	15	
1	1	2	3	4	5	...	15	
2	2	3	1	8		...	5	
3	3	1	2	12		...	10	
4	4	8	12	6		...	1	
5	5					...		
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	
15	15	5	10	1		...	9	

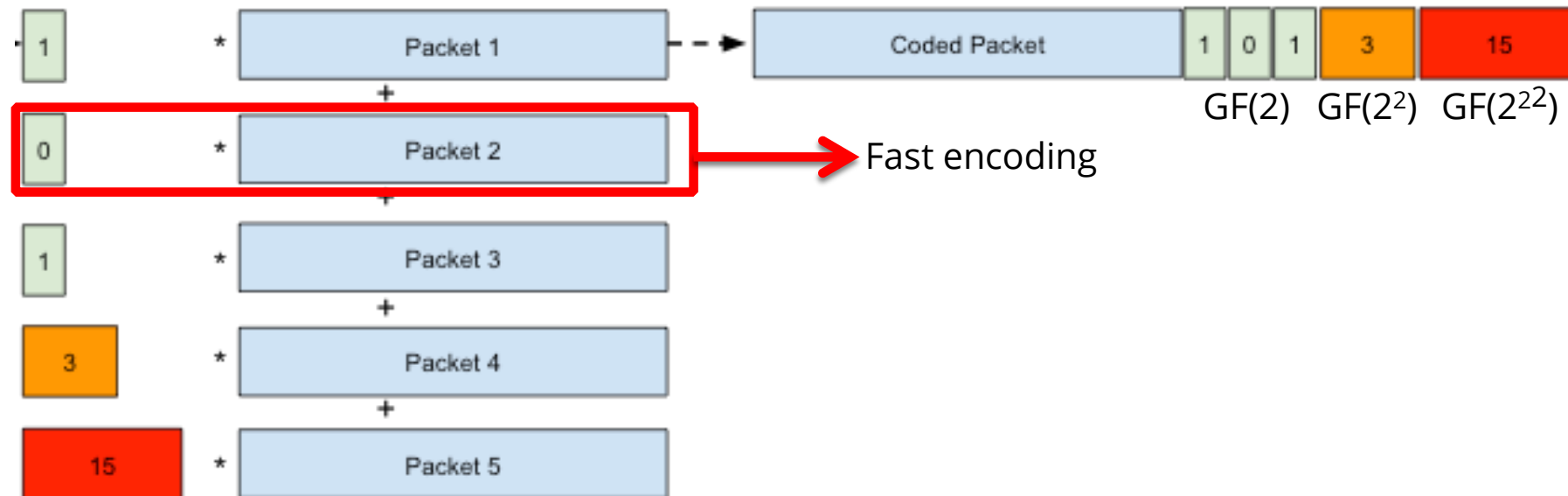
Telescopic Codes

- Design:
 - Multiple composite extension fields
- Goal: reduce overhead, maintaining high performance, faster encoding/decoding
 - Different packets are encoded using different field sizes



Telescopic Codes

- Design:
 - Multiple composite extension fields
- Goal: reduce overhead, maintaining high performance, faster encoding/decoding
 - Different packets are encoded using different field sizes



Telescopic Decoder

Coded Packet 1	1	0	1	3	15
Coded Packet 2	0	1	1	2	7
Coded Packet 4	0	0	1	0	3
Coded Packet 5	1	0	0	1	7
Coded Packet 7	1	0	0	0	13

----->

1	0	1	3	15
0	1	1	2	7
0	0	1	0	3
1	0	0	1	7
1	0	0	0	13

----->

1	0	1	3	15
0	1	1	2	7
0	0	1	0	3
0	0	1	2	8
0	0	1	3	2

GF(2) Elimination

First 2 columns

Only GF(2) operations

----->

1	0	0	3	12
0	1	0	2	4
0	0	1	0	3
0	0	0	2	11
0	0	0	3	1

GF(2) Elimination

Third column

Only GF(2) operations

----->

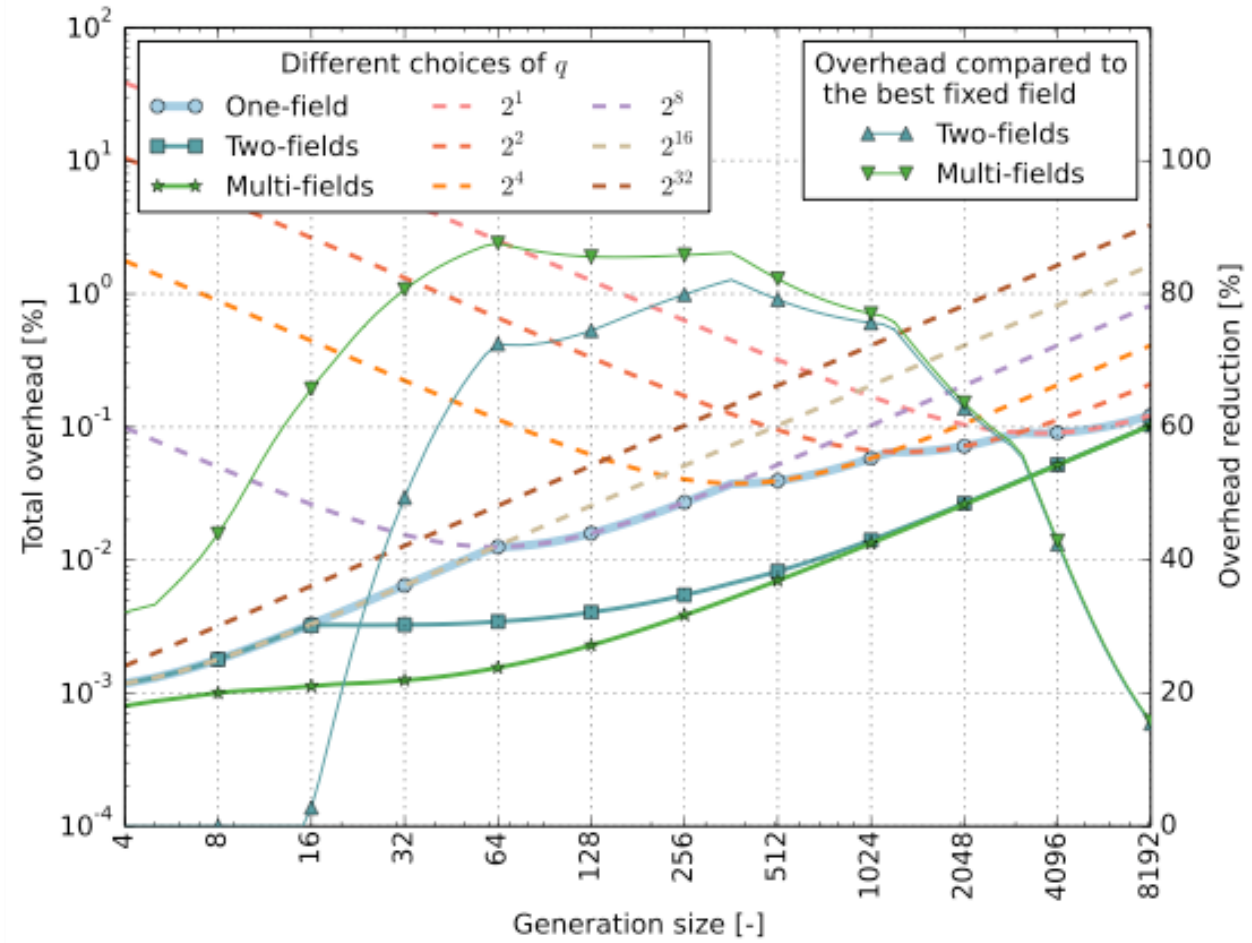
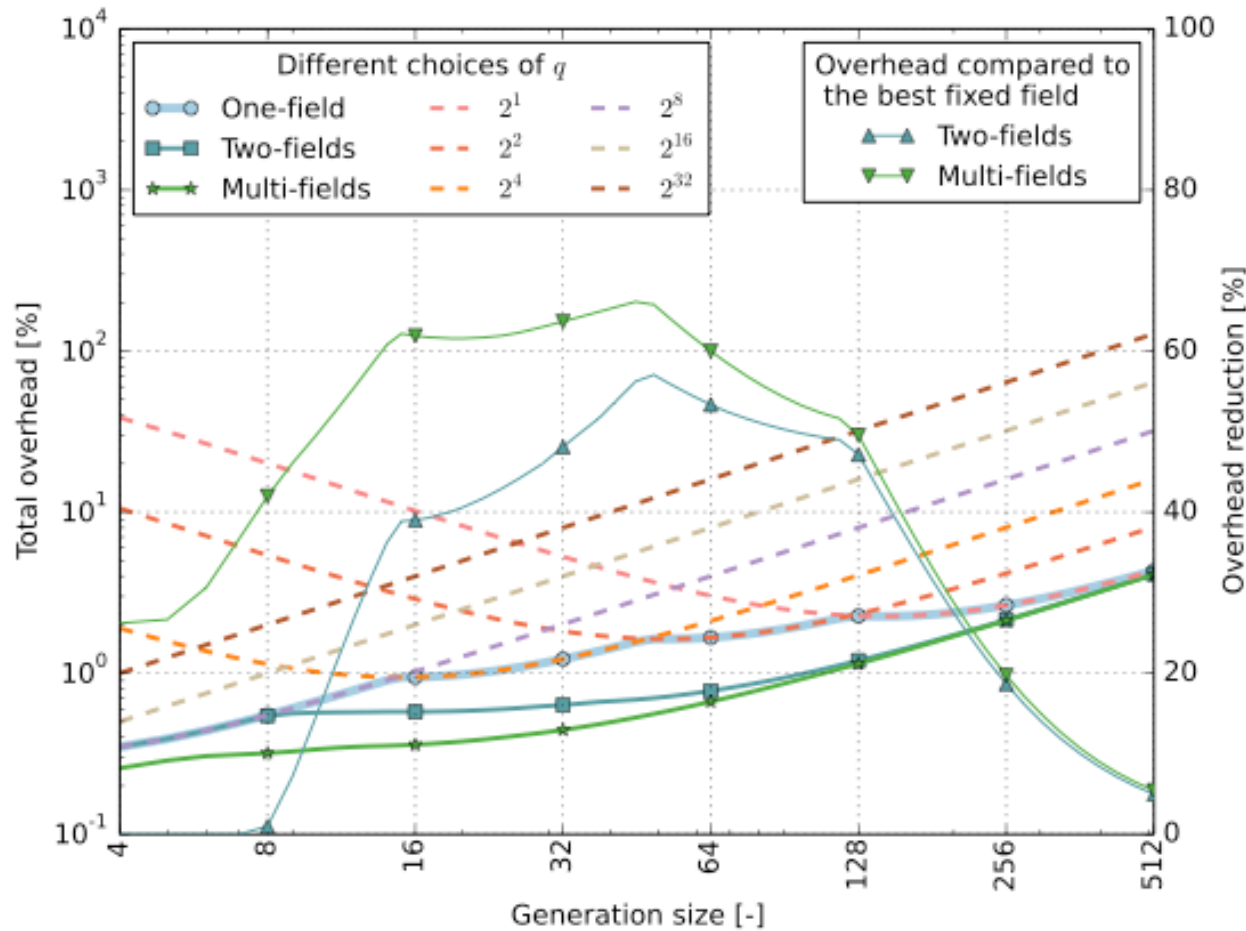
1	0	0	0	1
0	1	0	0	15
0	0	1	0	3
0	0	0	1	6
0	0	0	0	11

GF(2²) Elimination

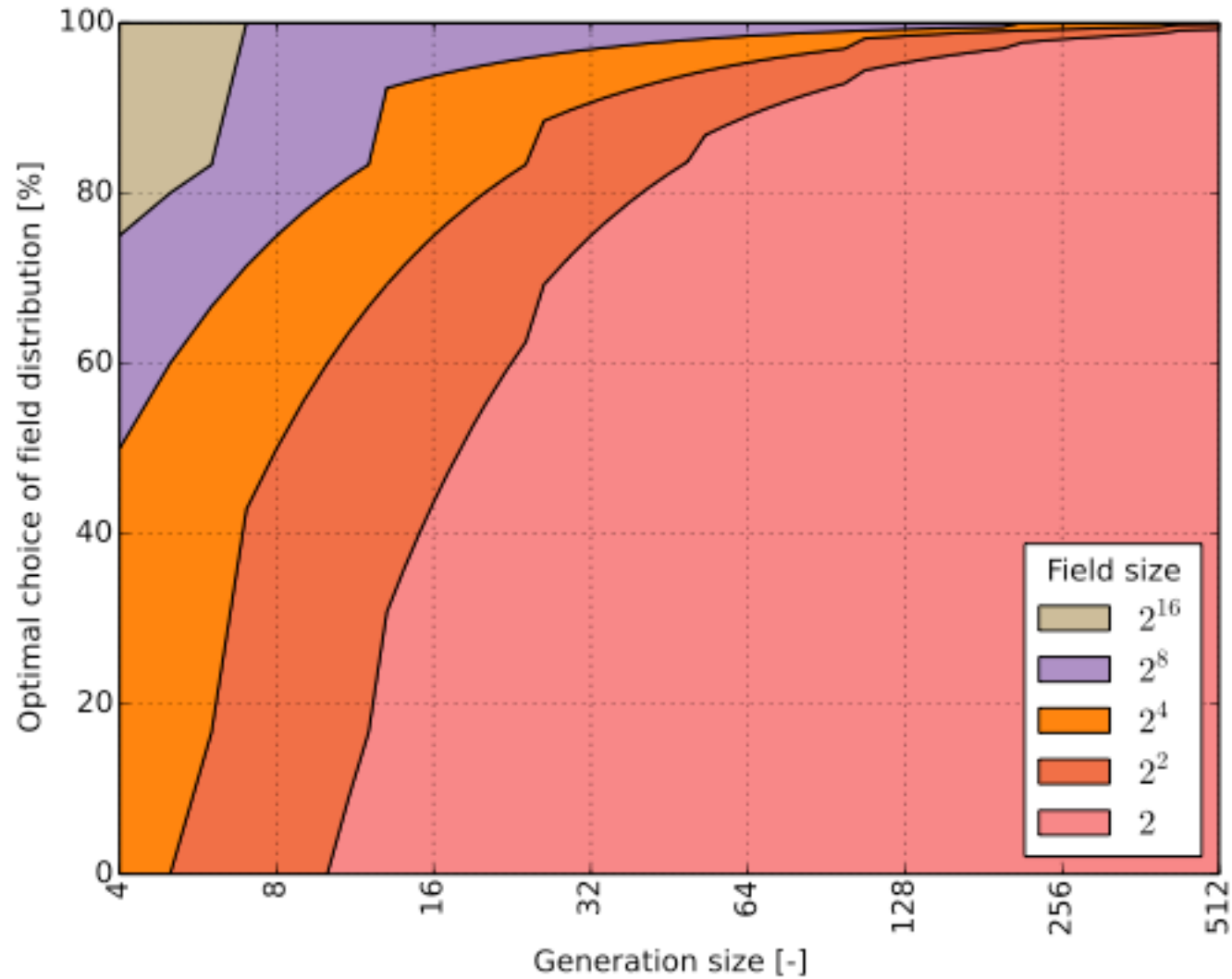
Fourth column

Only GF(2²) operations

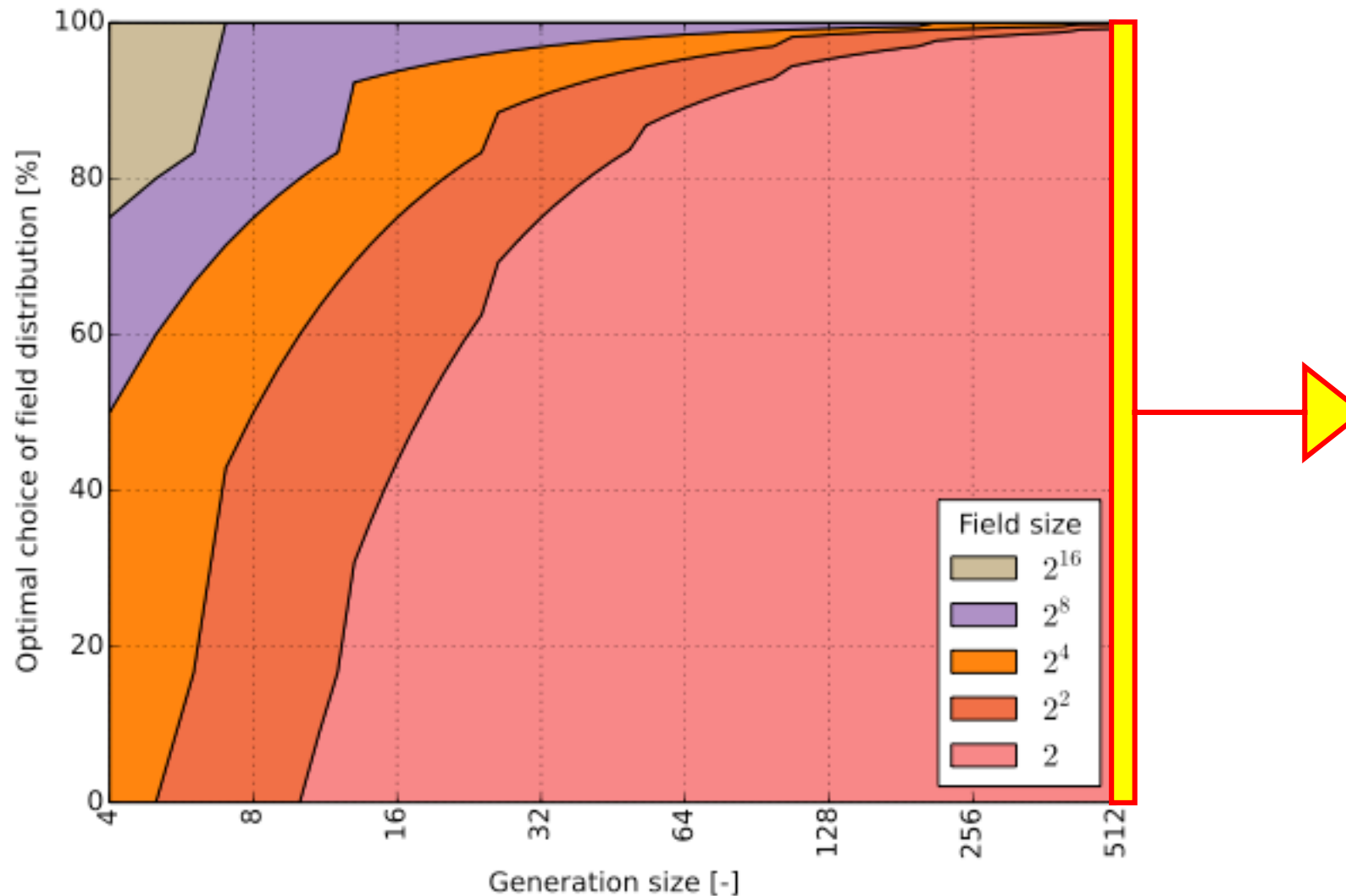
Telescopic Results



Telescopic Results



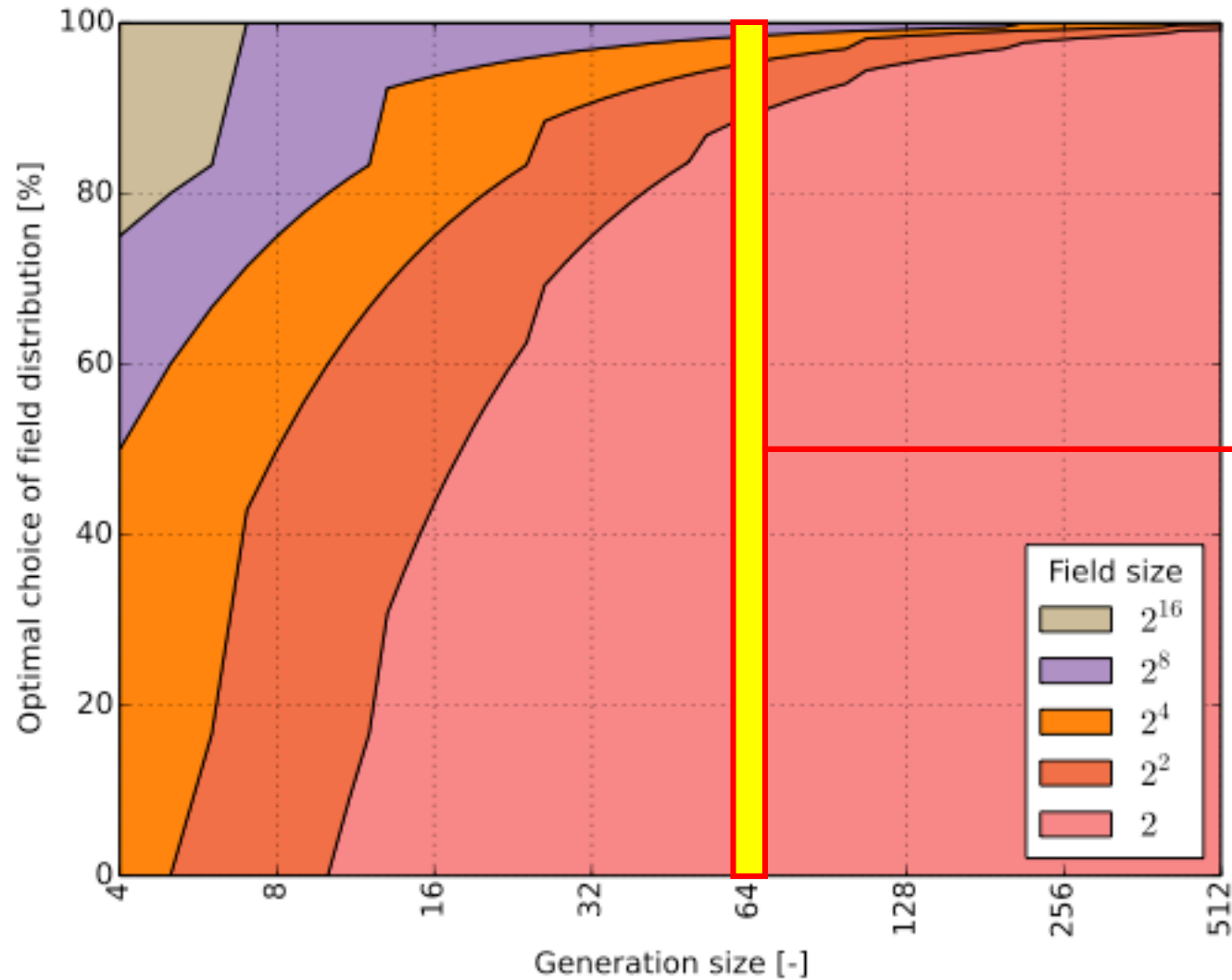
Telescopic Results



G=512

If generation size is huge, using binary fields is totally fine as the 1.6 extra packets per generation, here 512, is not really a problem.

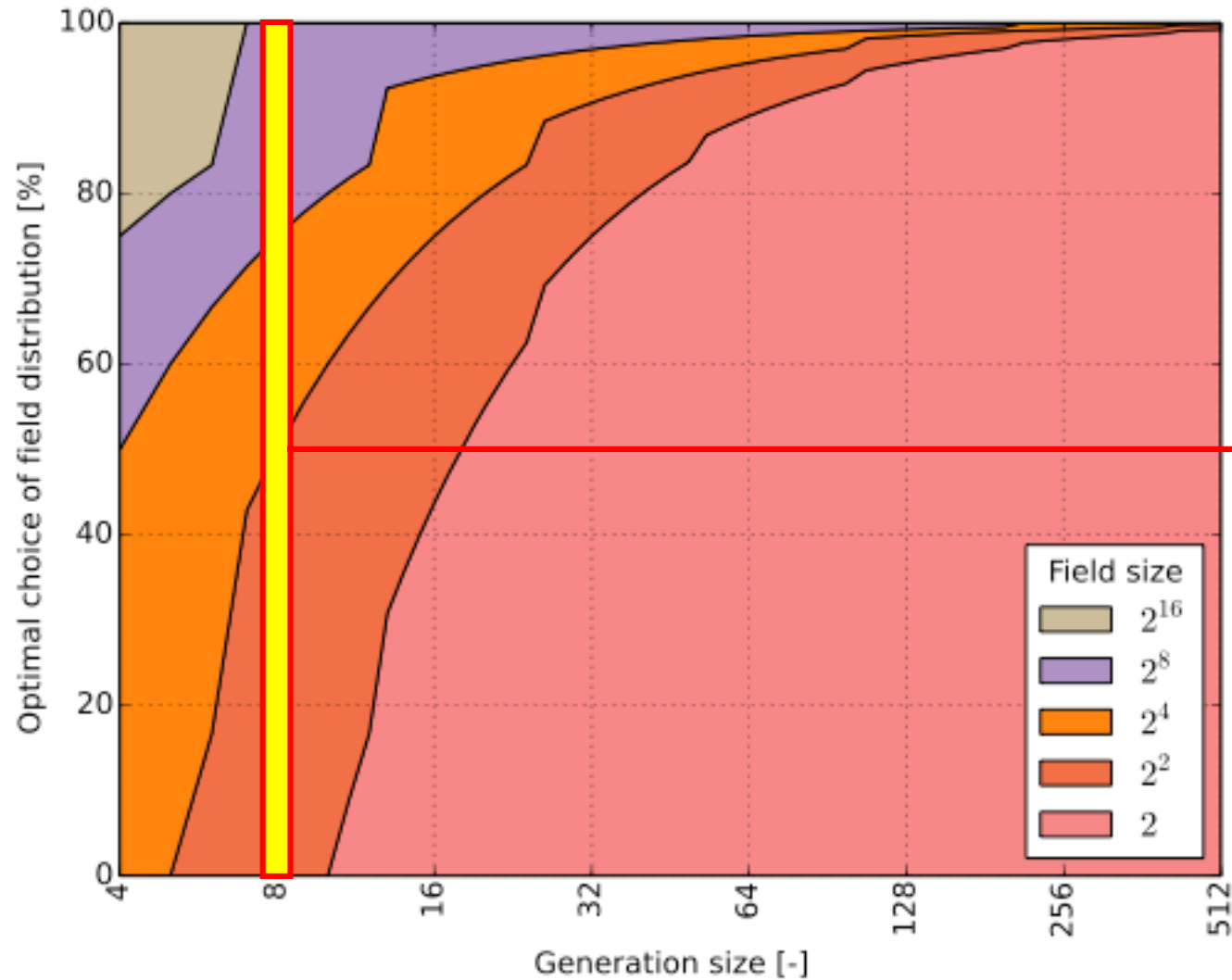
Telescopic Results



G=64

Some coded packets with higher fields in the end to improve the efficiency, but majority of the packets is still coded in the binary field.

Telescopic Results



G=8

Multi-field is used in order to achieve good efficiency.

Conclusions

- Simple design of composite extension fields
- Proposed telescopic codes
 - Reduce total overhead
 - Maintain high decoding probability
 - Potential for processing at high speed (simpler encoder/decoder)
- Future work
 - Applications in other codes
 - New code designs
 - Performance evaluation in real systems

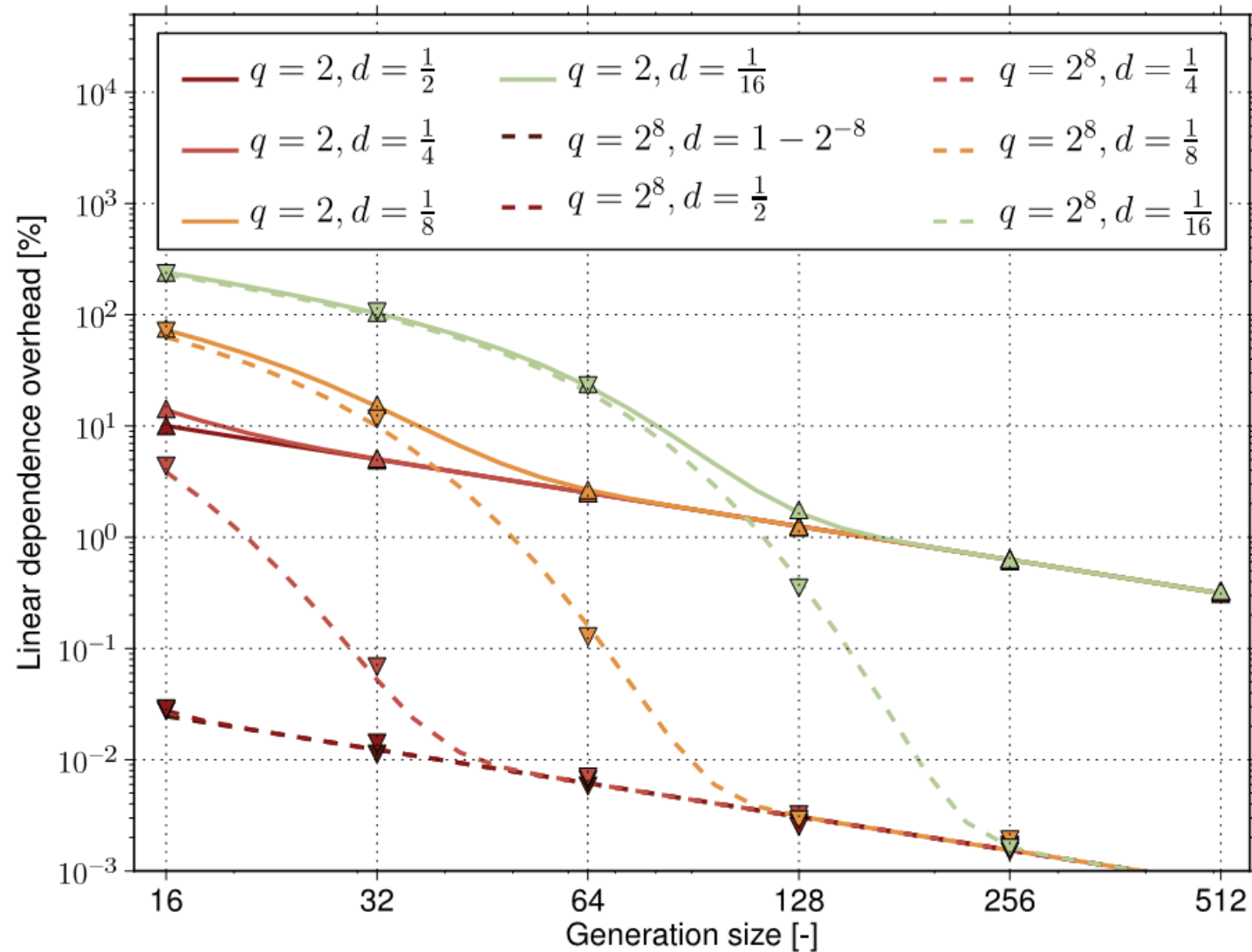
KODO: Overhead

Janus Heide and Morten V. Pedersen and Frank H.P. Fitzek and Muriel Medard. **On Code Parameters and Coding Vector Representation for Practical RLNC**. 2011. in *IEEE International Conference on Communications (ICC) - Communication Theory Symposium*. Kyoto, Japan.

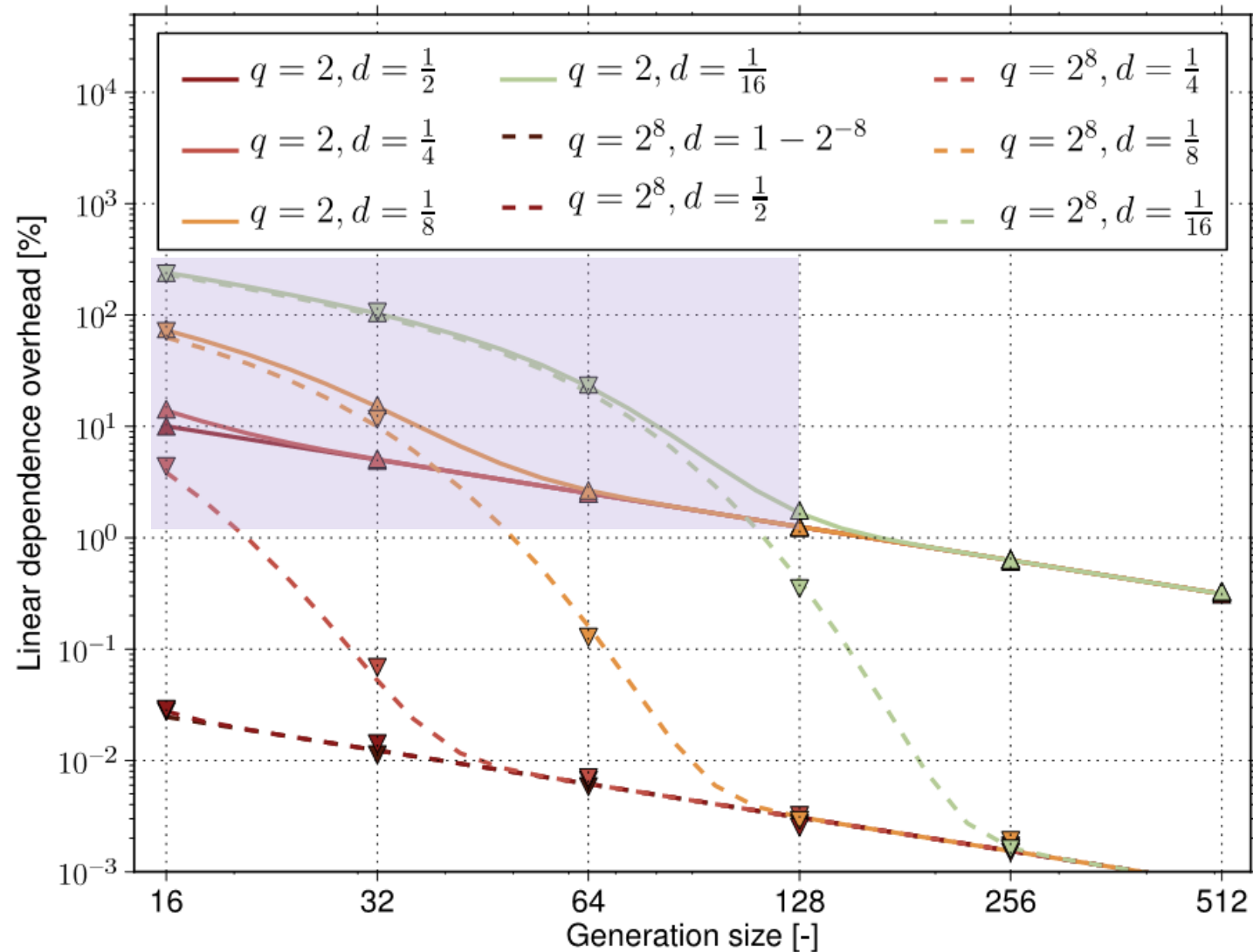
What is overhead?

- Overhead by encoding vector
 - Generation size G
 - Field size F
 - Payload P
 - Number of additional bits for each packet $A = G * \log_2(F)$ for Full RLNC
 - A can be smaller for Sparse RLNC if encoding vector is compressable
- Linear dependent retransmissions
 - In case a packet is linear dependent it has to be retransmitted (together with the header)
 - $A = P + G * \log_2(F)$

Linear Dependency for Different Density Levels and Field Sizes

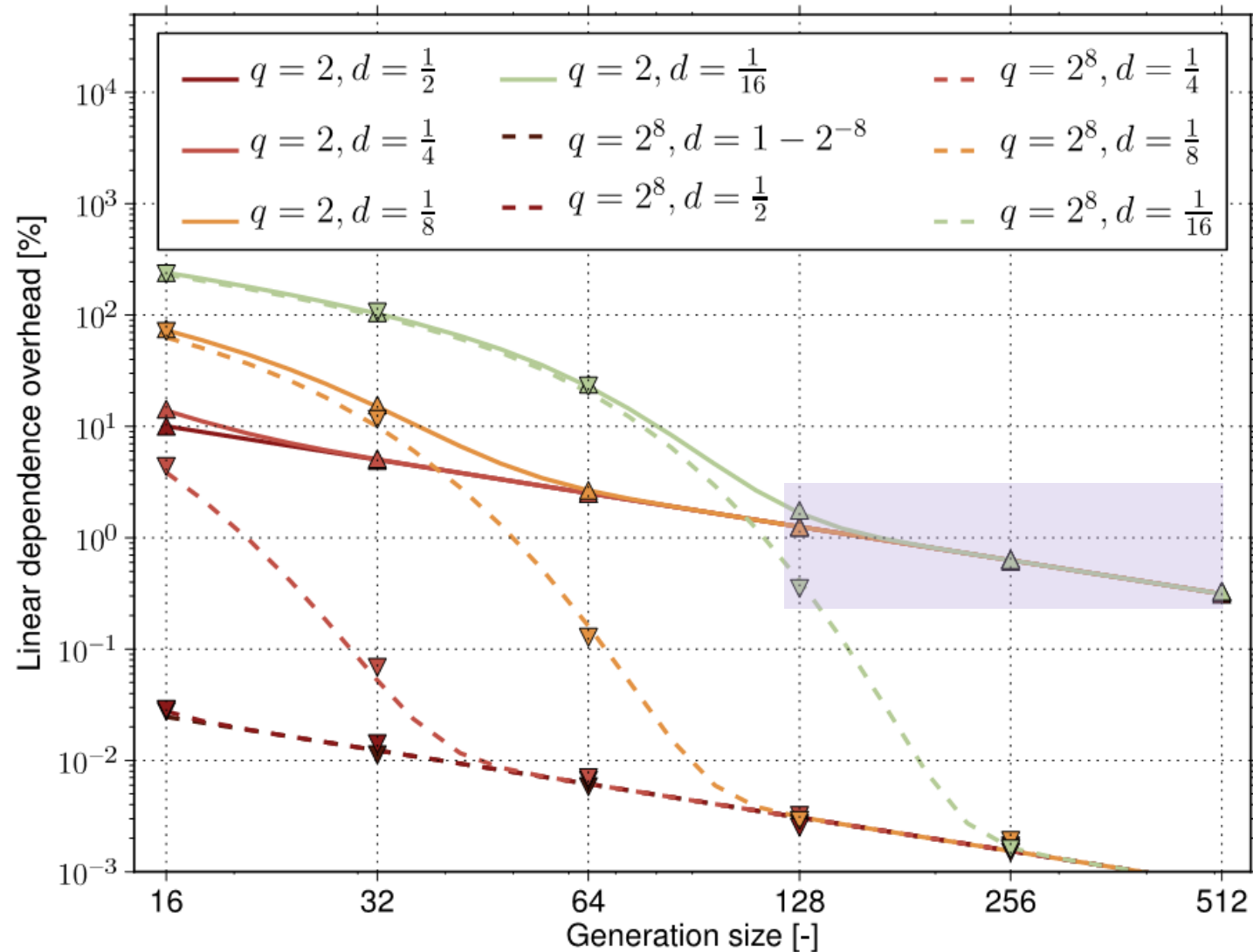


Linear Dependency for Different Density Levels and Field Sizes



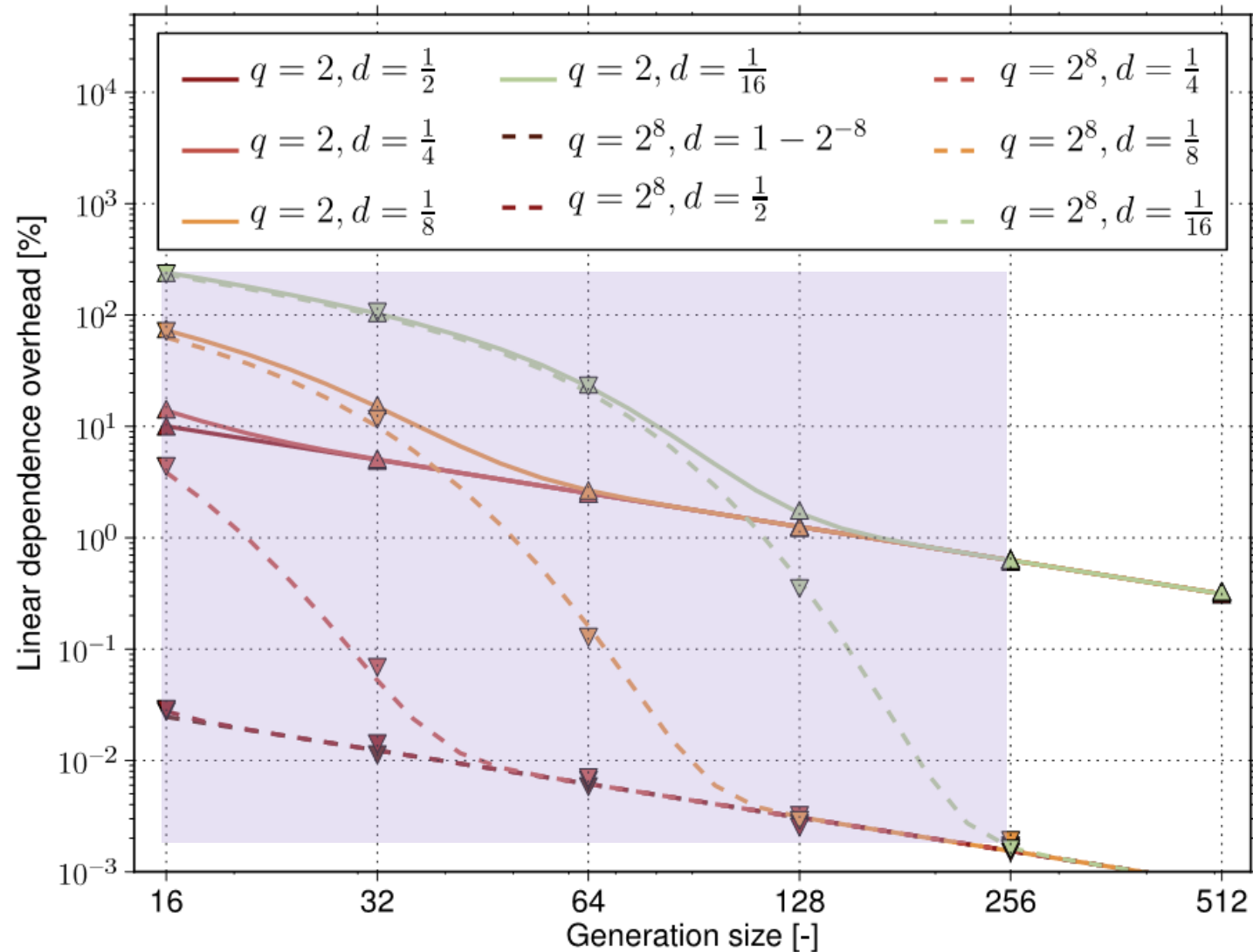
For the binary field and small generation sizes, the density level has an impact on the overhead.

Linear Dependency for Different Density Levels and Field Sizes



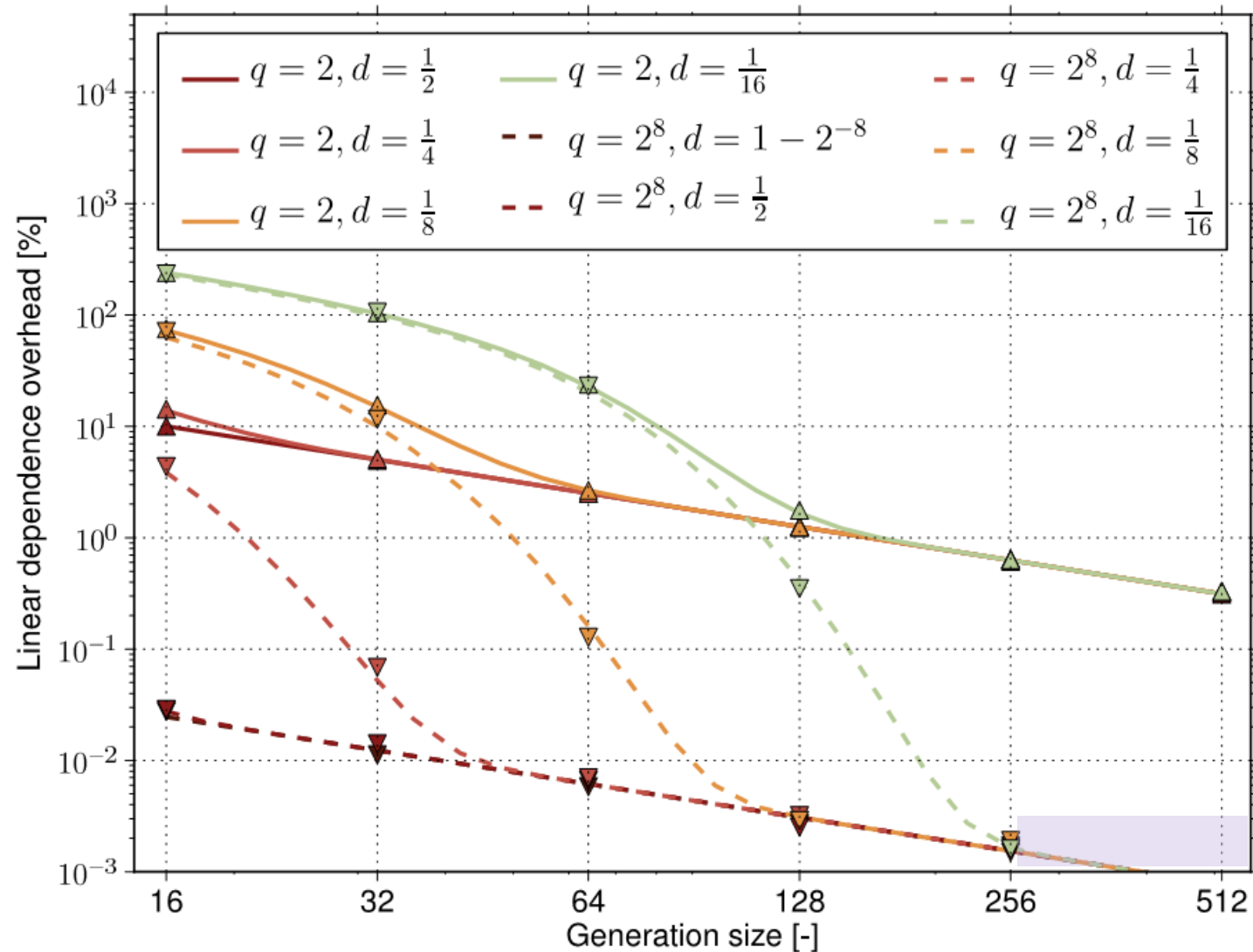
For the binary field and large generation sizes, the density level has no impact on the overhead.

Linear Dependency for Different Density Levels and Field Sizes



For the binary8 field and small to medium generation sizes, the density level has a huge impact on the overhead.

Linear Dependency for Different Density Levels and Field Sizes

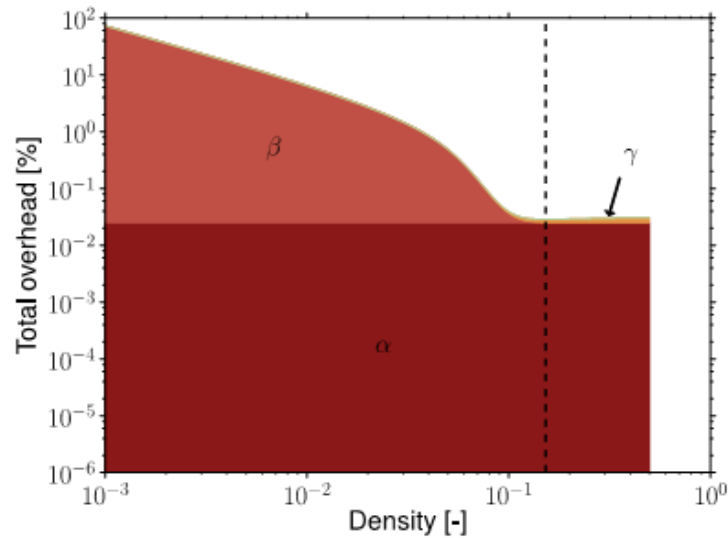


For the binary8 field and very large generation sizes, the density level has no impact on the overhead.

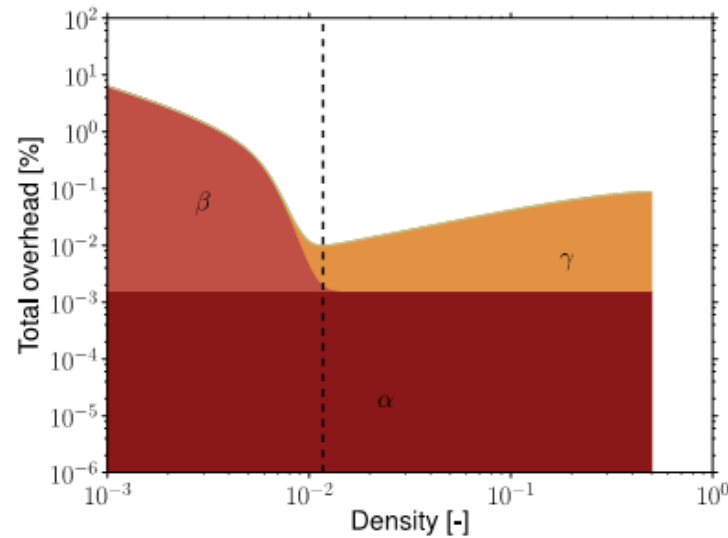
So where does the overhead comes from?

Linear Dependency for Different Density Levels

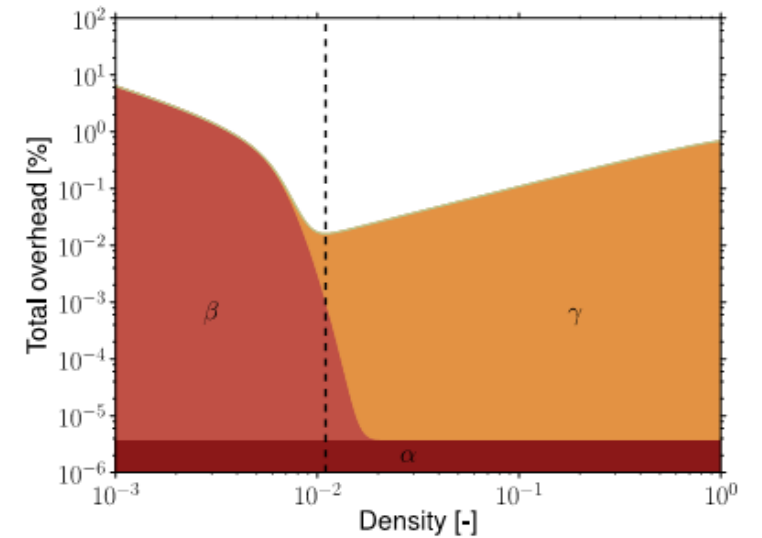
Designing rules ...



(a) $q = 2, g = 64$



(b) $q = 2, g = 1024$

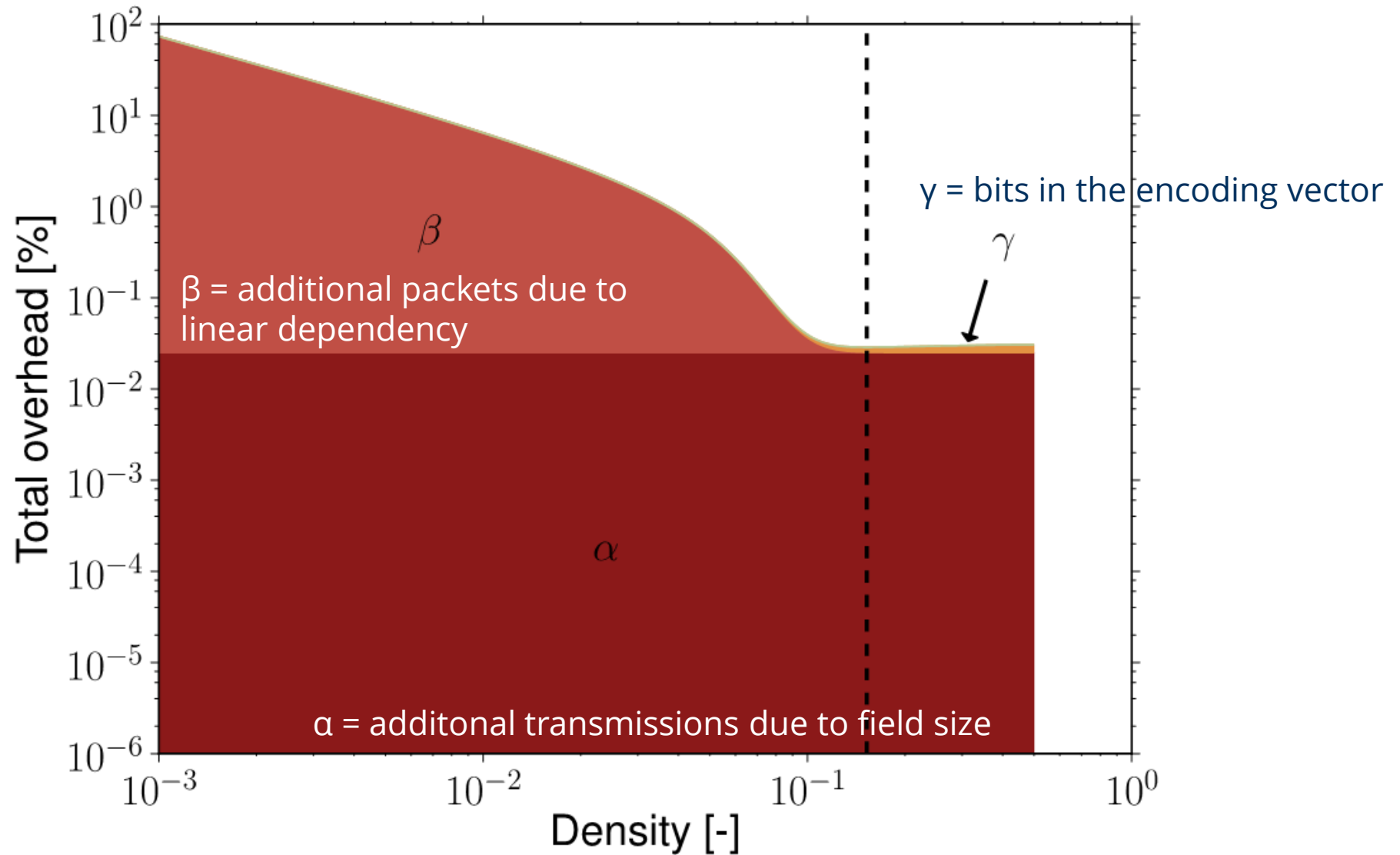


(c) $q = 2^8, g = 1024$

α = additional transmissions due to field size
 β = additional packets due to linear dependency
 γ = bits in the encoding vector

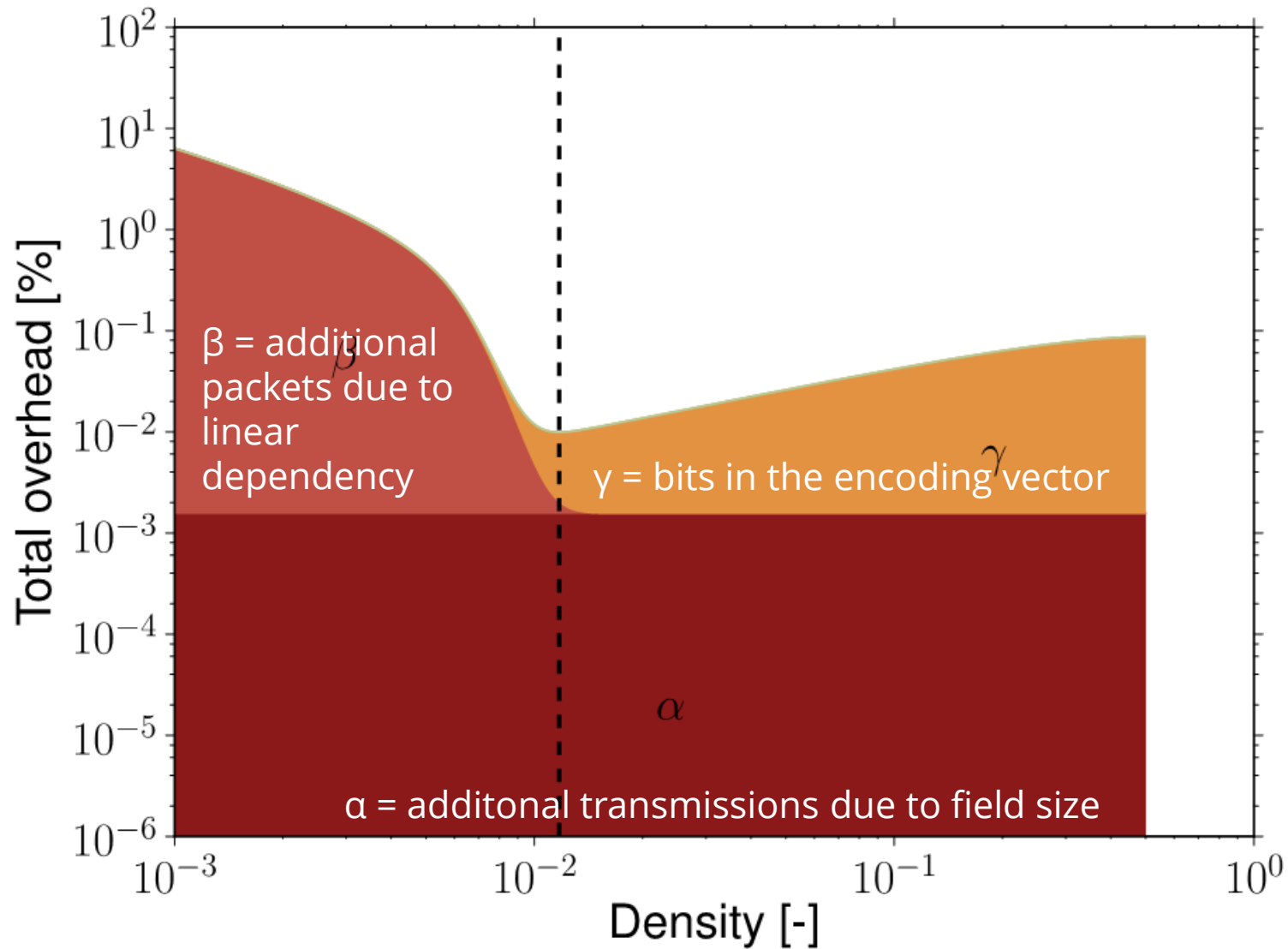
Linear Dependency for Different Density Levels

$F=2; G=64$



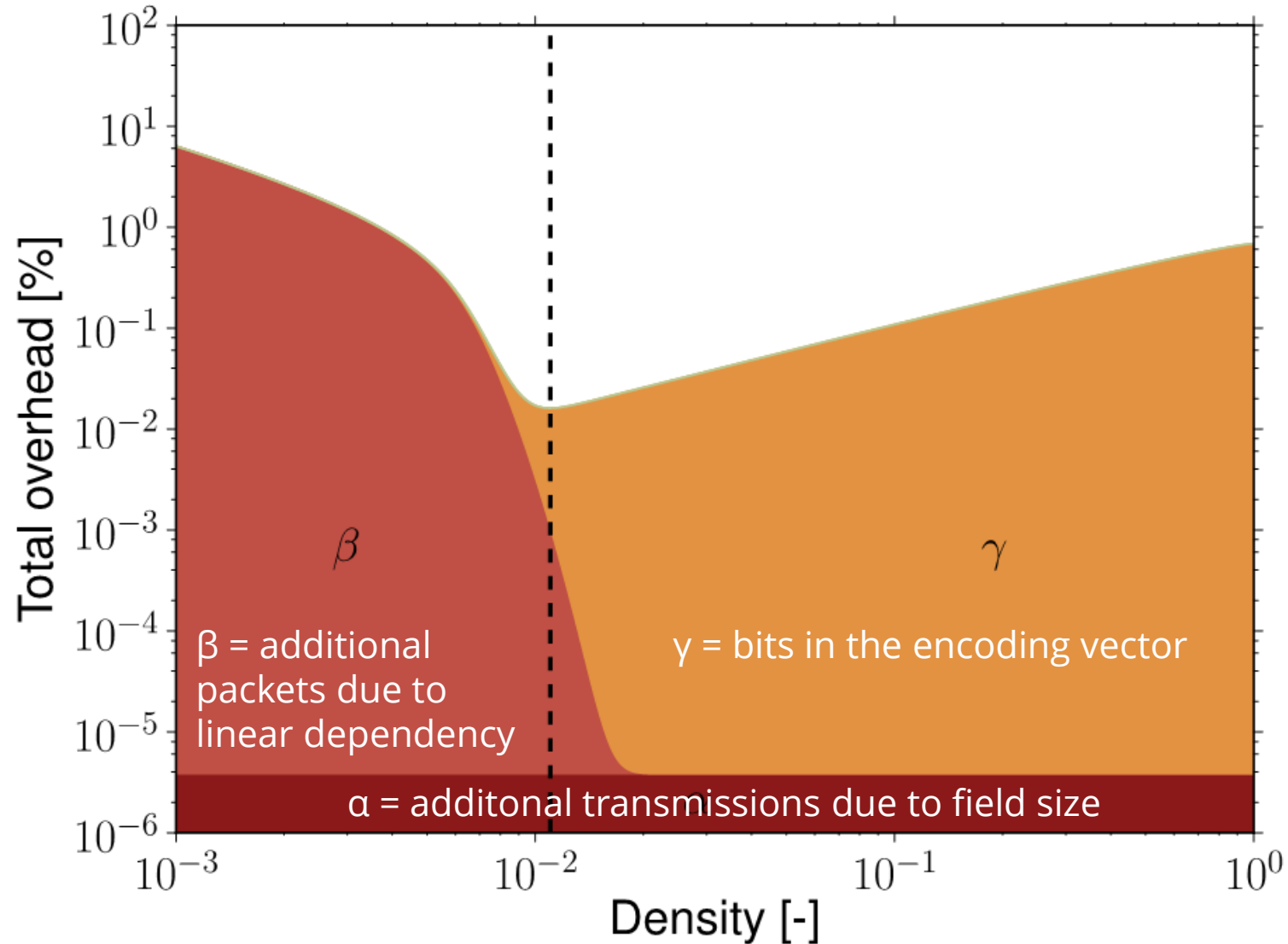
Linear Dependency for Different Density Levels

$F=2; G=1024$



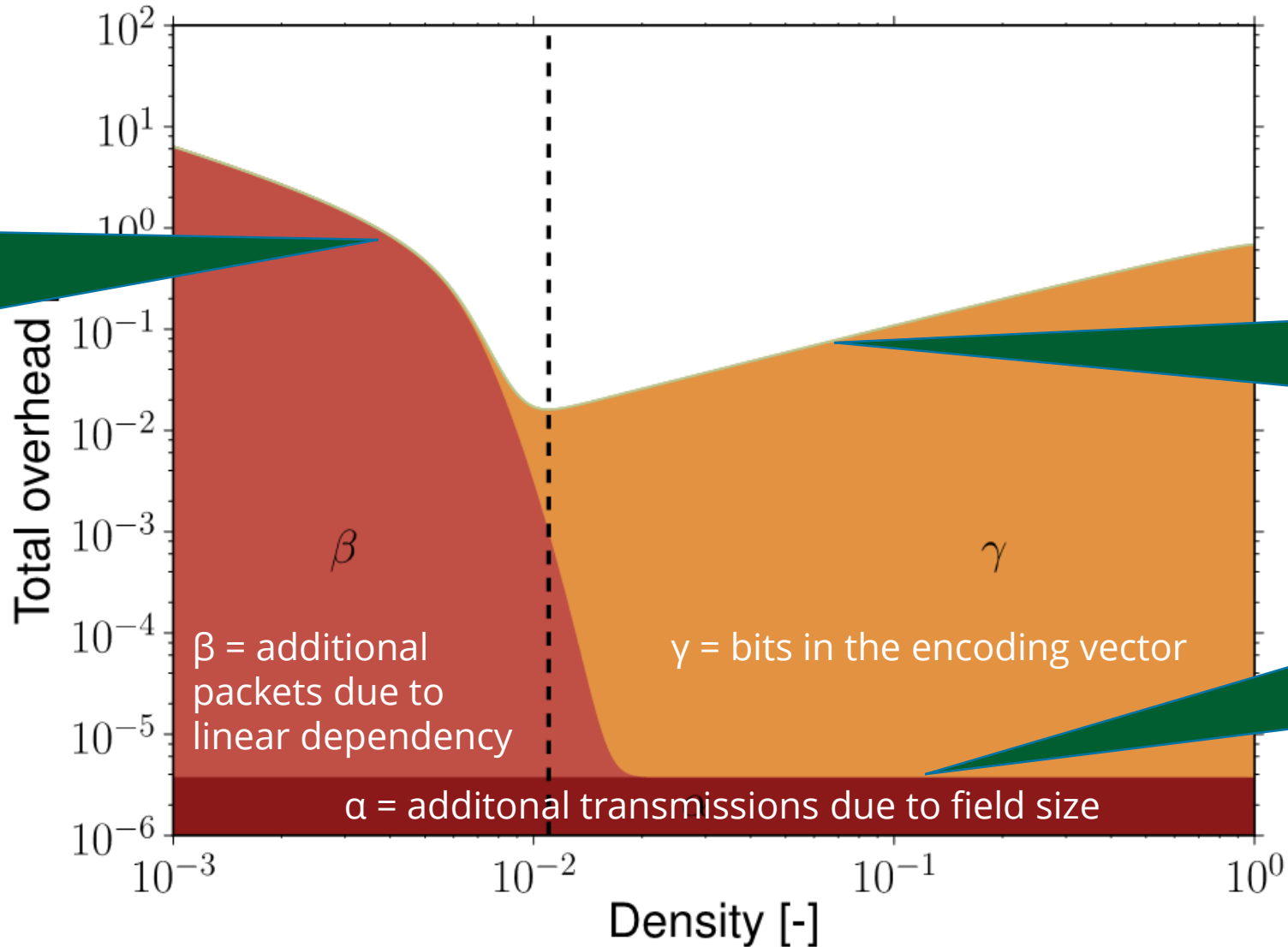
Linear Dependency for Different Density Levels

$F=2^8; G=1024$



Linear Dependency for Different Density Levels

$F=2^8; G=1024$



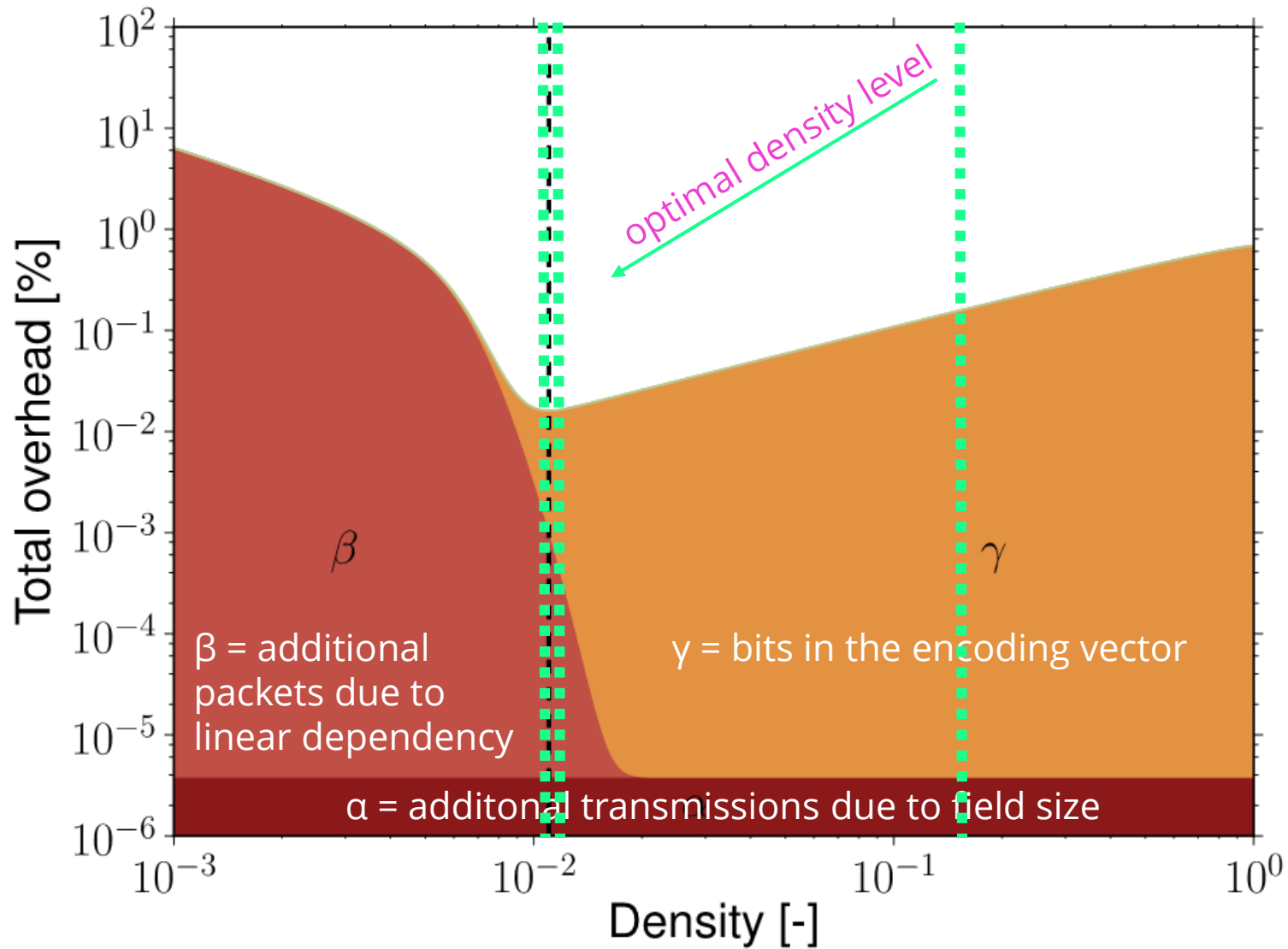
Risk of additional linear dependency mitigates for more dense encoding vectors.

Number of bits in encoding vector changes with the density level due to compression.

No dependency between density level and field size.

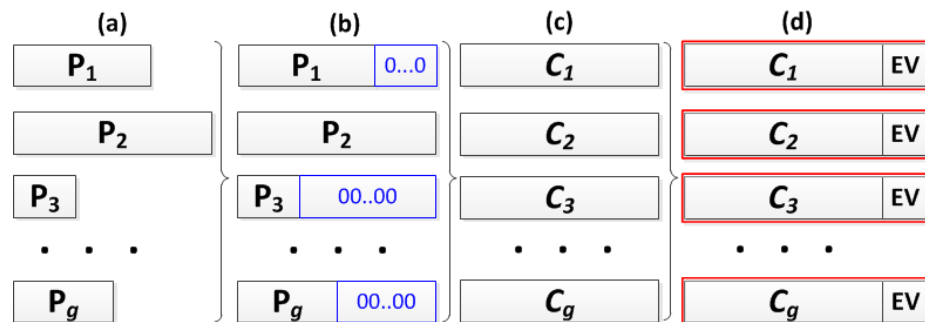
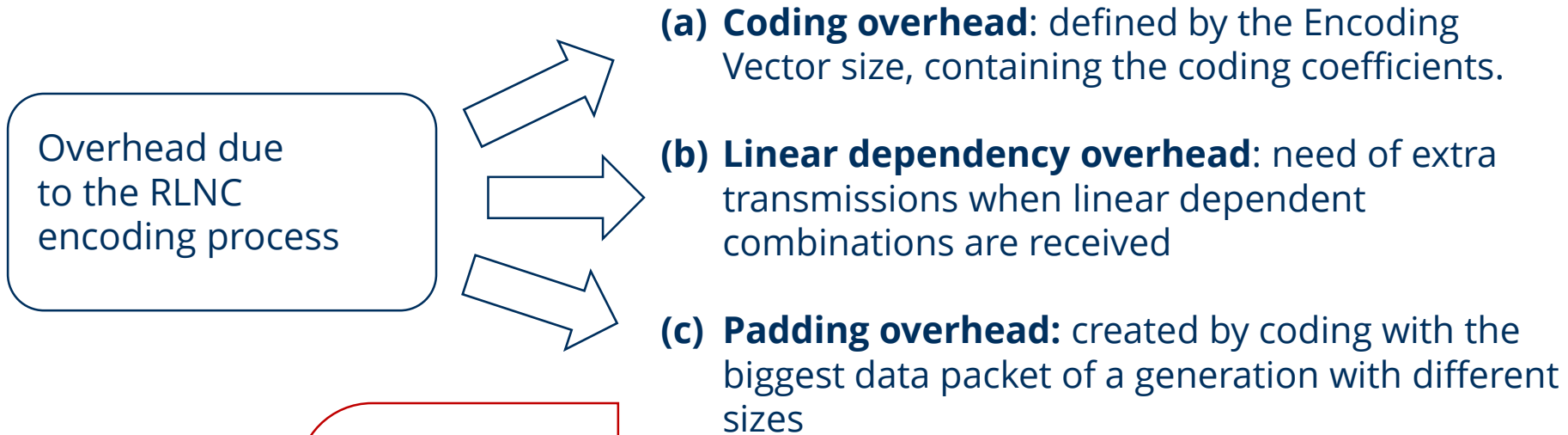
Linear Dependency for Different Density Levels

$F=2^8; G=1024$



Effects of Heterogeneous Packet lengths on Network Coding

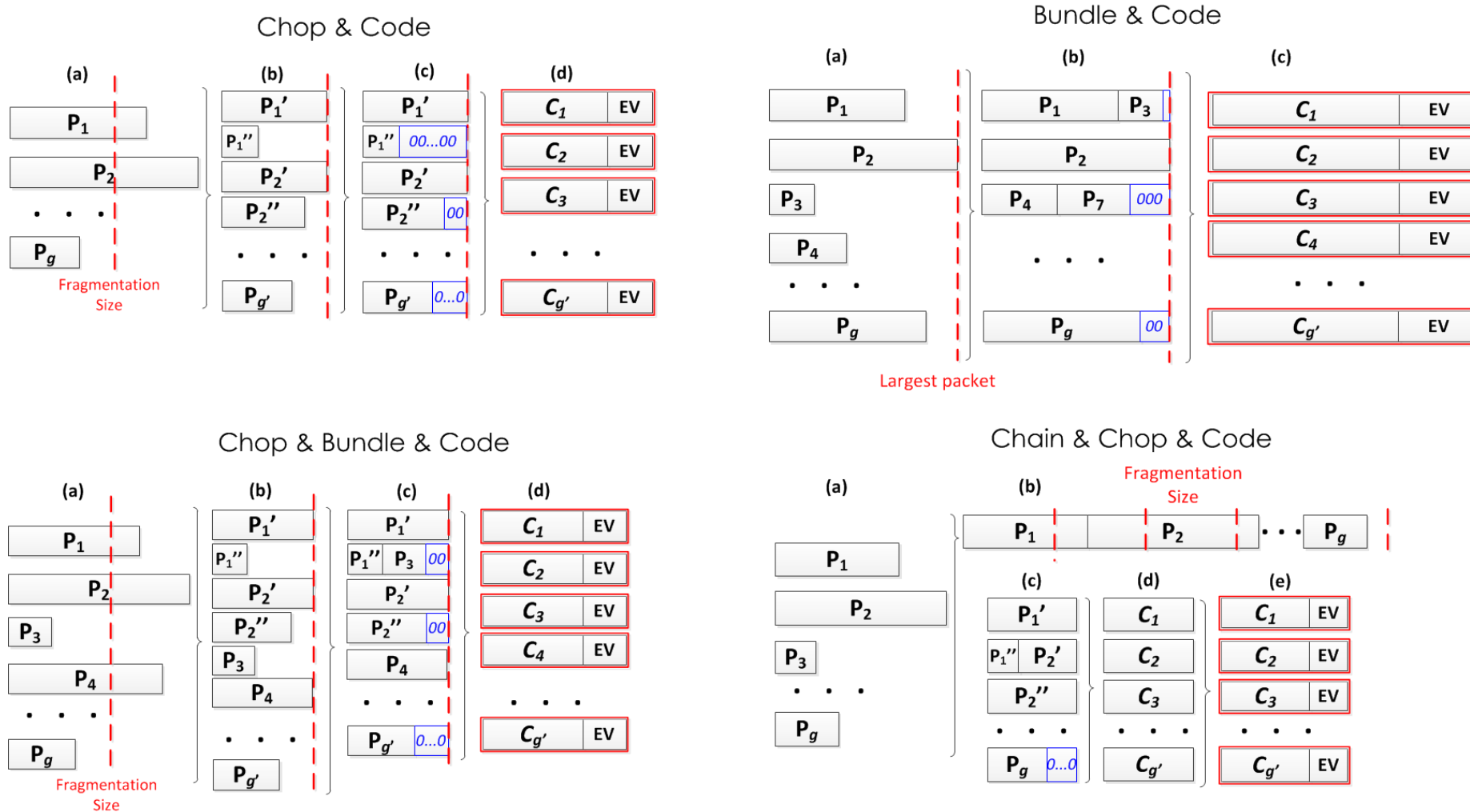
Effects of Heterogeneous Packet lengths on Network Coding



Naive assumption: all packets in a generation have the same size

Real life data: heterogeneity of packet lengths

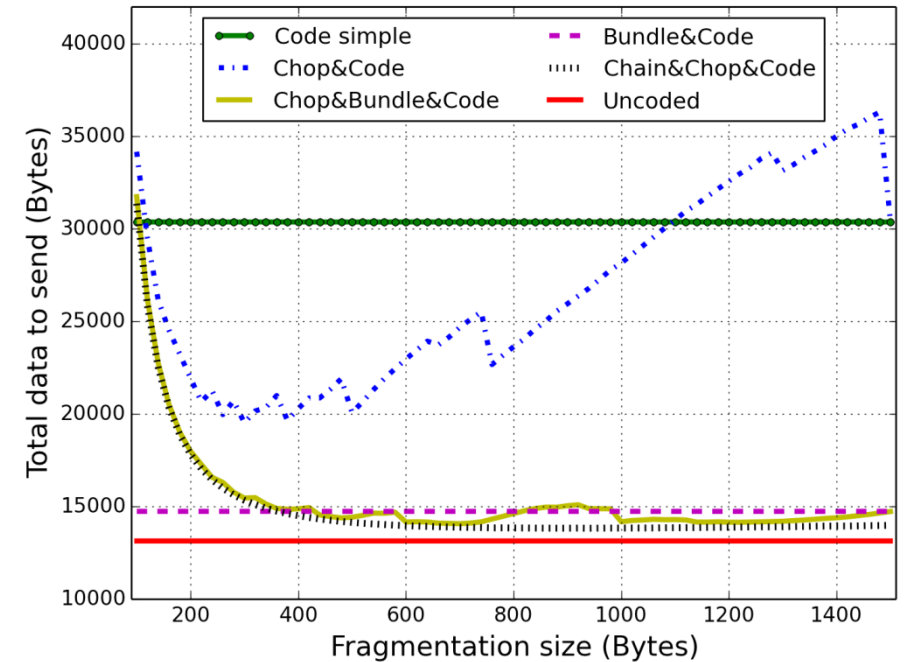
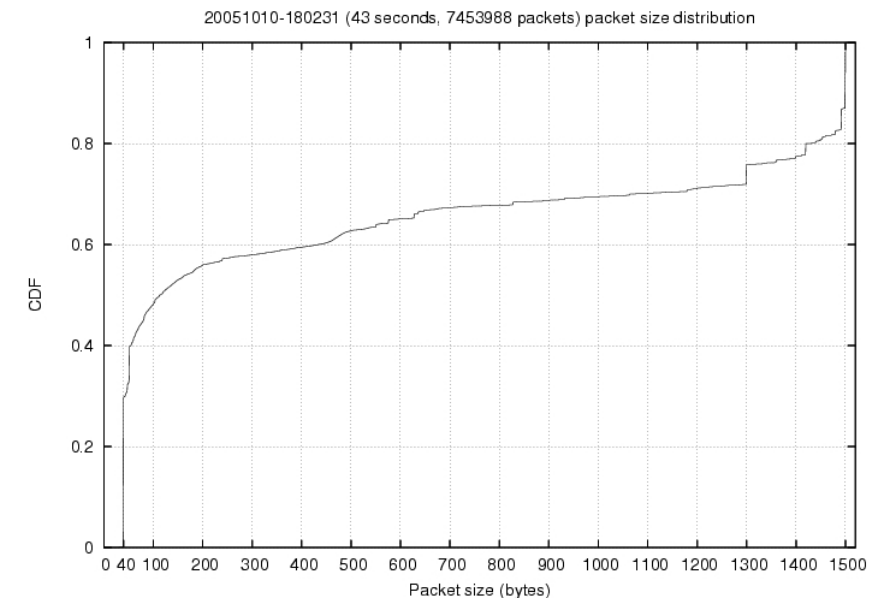
Effects of Heterogeneous Packet lengths on Network Coding



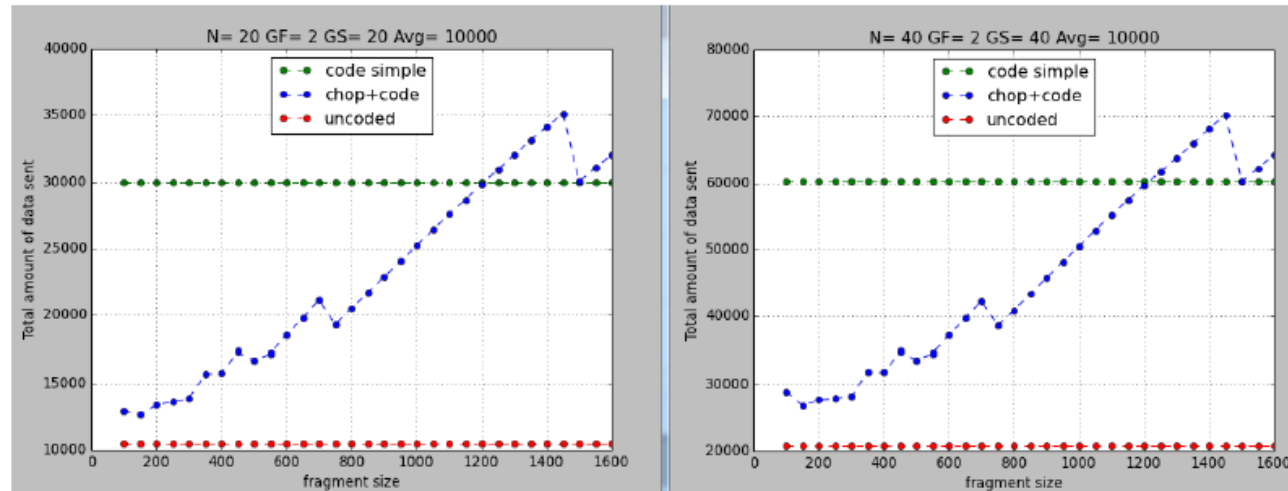
Internet data from CAIDA

- Generations of 20 packets with field size $q=2^8$
- Packets are chosen randomly following the CDF, and run the simulations 10000 times and averaged the results
- Internet packet size distribution is mostly of 40 Bytes and 1500 Bytes (with approximate probabilities of 40% and 20% respectively)¹

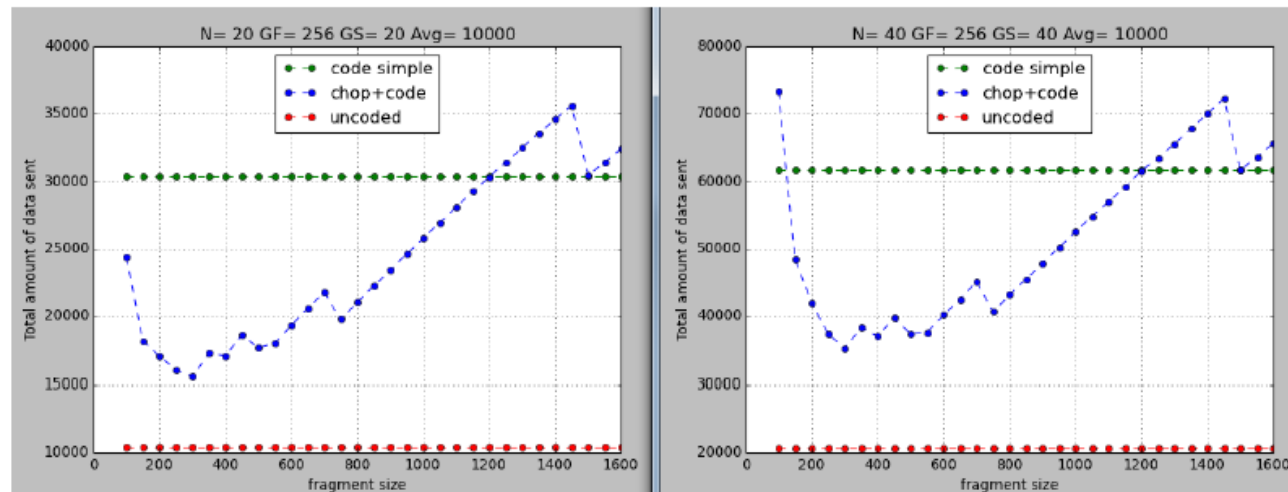
¹ <http://www.isi.edu/~johnh/PAPERS/Sinha07a.pdf>



Packet Size Comparison 1

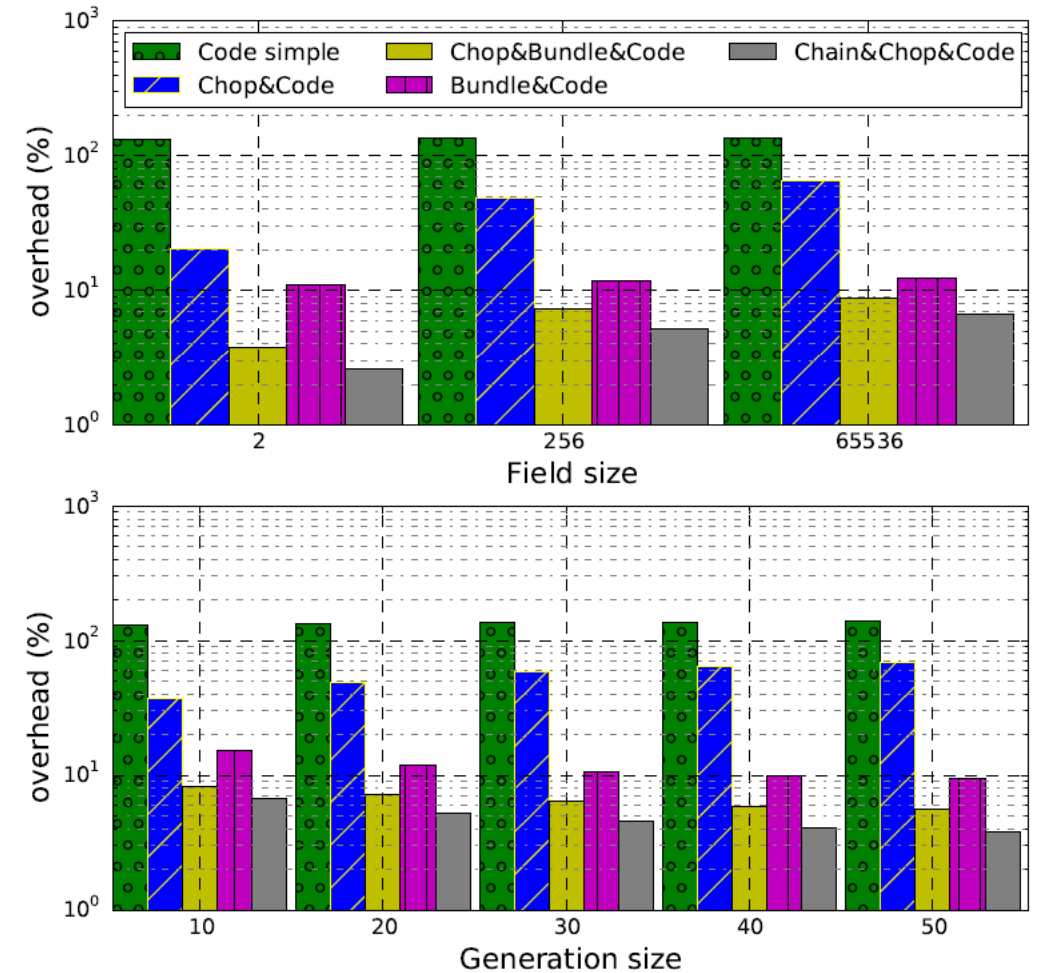


GF=2^8

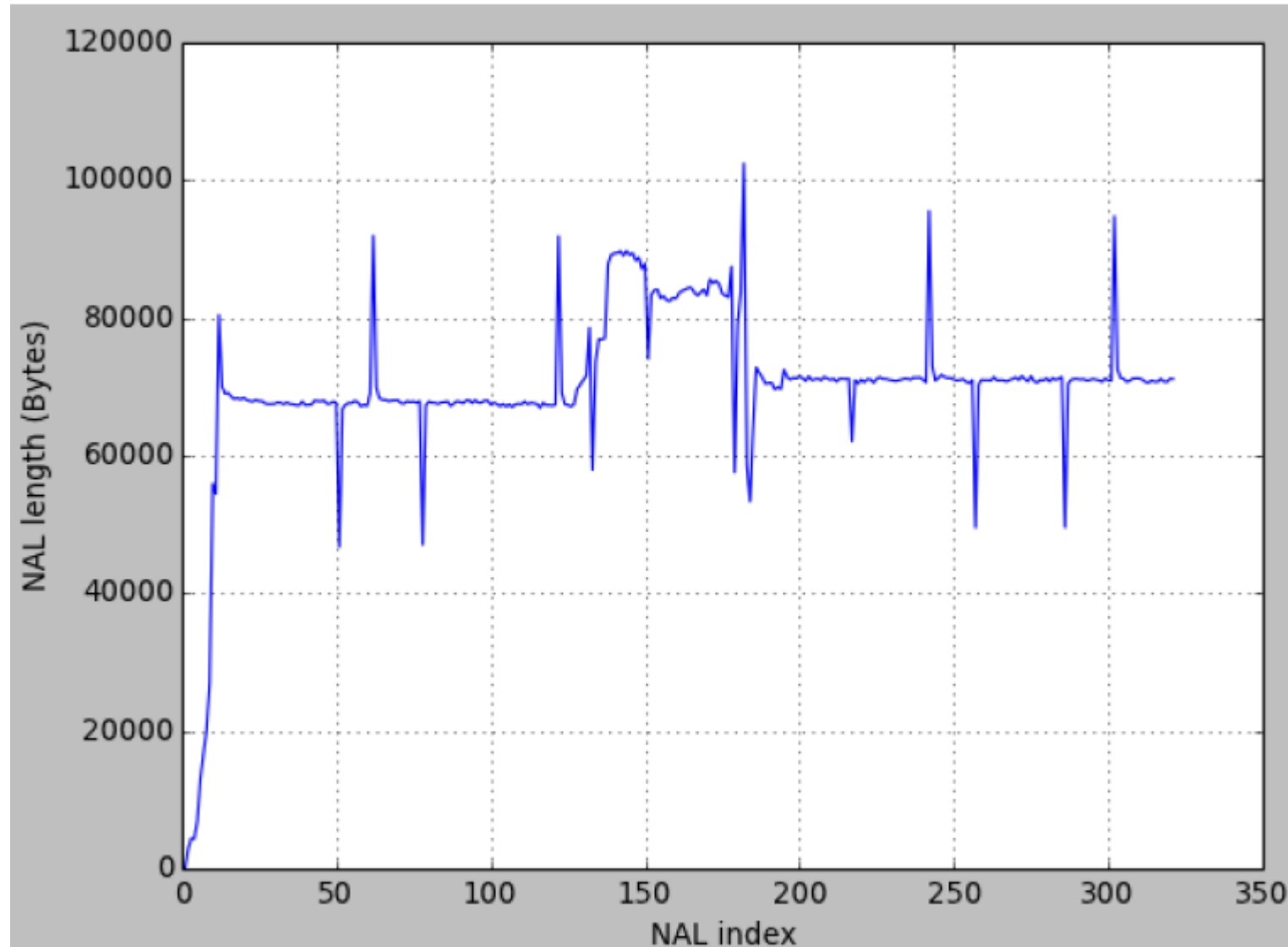


Internet data from CAIDA (2)

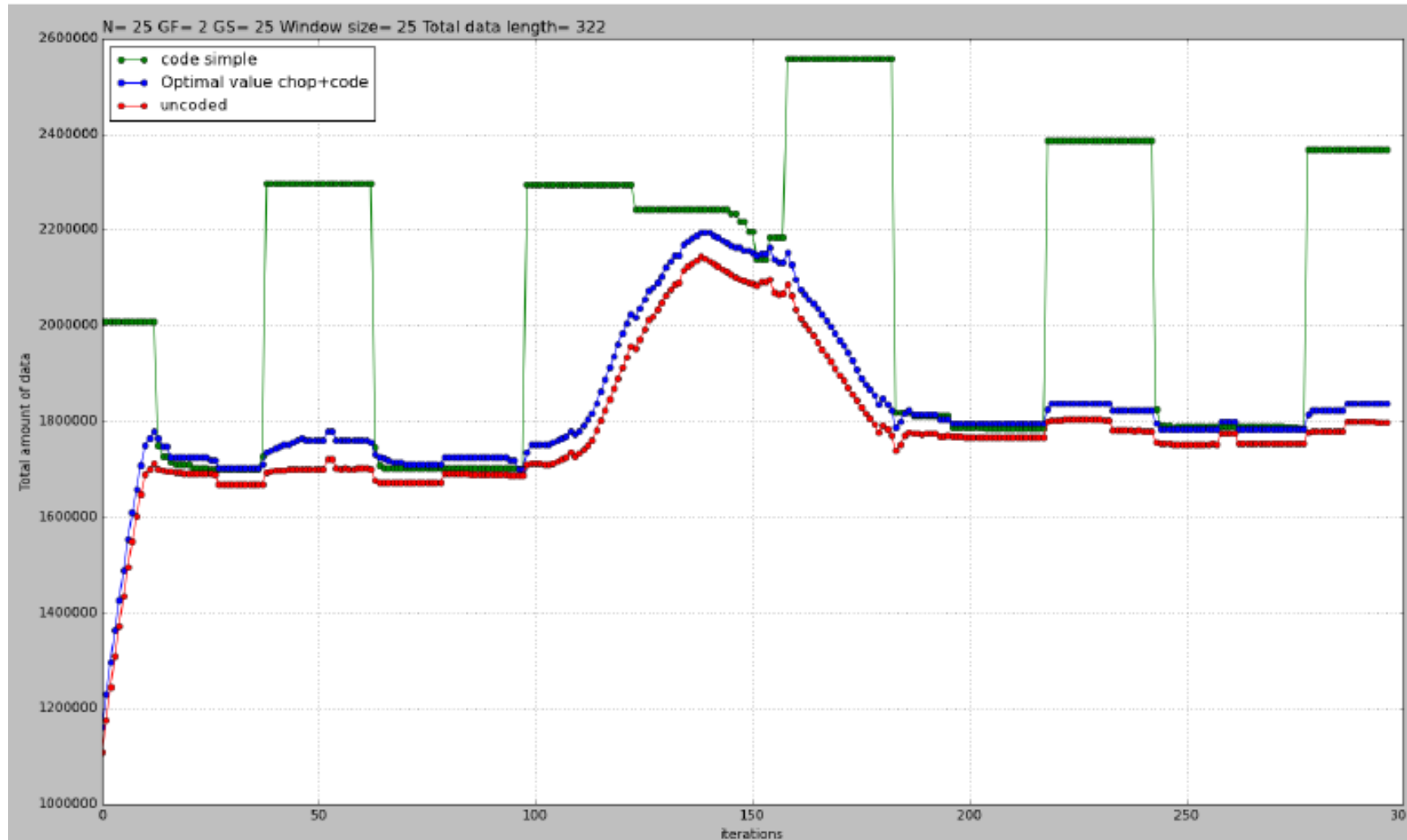
- Total coding overhead (%) added for each solution respect the uncoded data for:
 - different field sizes (and generation size constant of 20 packets)
 - different generation sizes (and field size constant $q=2^8$)
- Coding directly adds more than 100% of overhead.
- Chain& Chop& Code achieves to reduce to less than 5% for a large generation size.



Packet Size Example 2



Packet Size Comparison 2



KODO: Energy consumption and complexity

Energy consumption: KODO

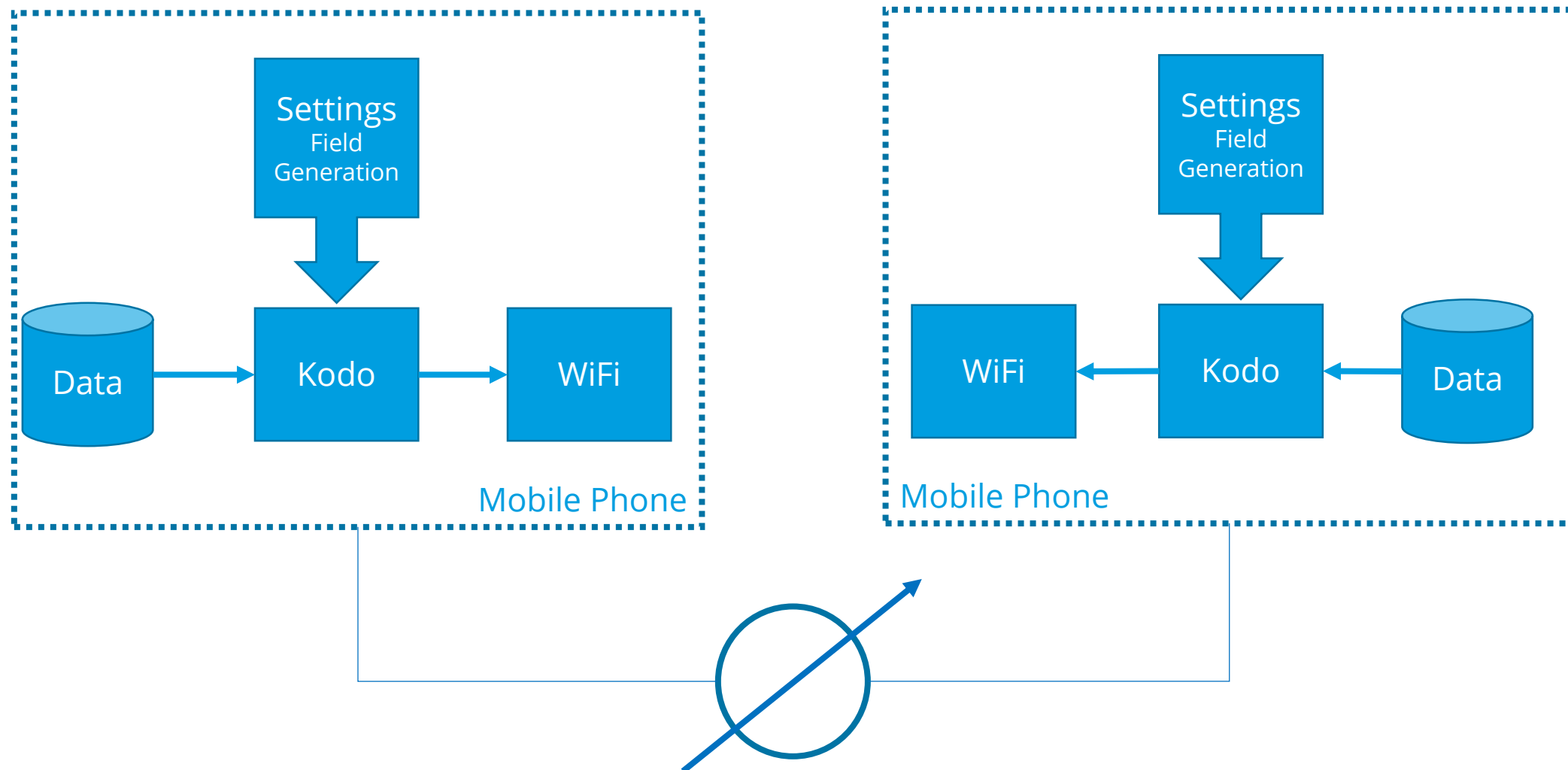


Energy consumption: KODO

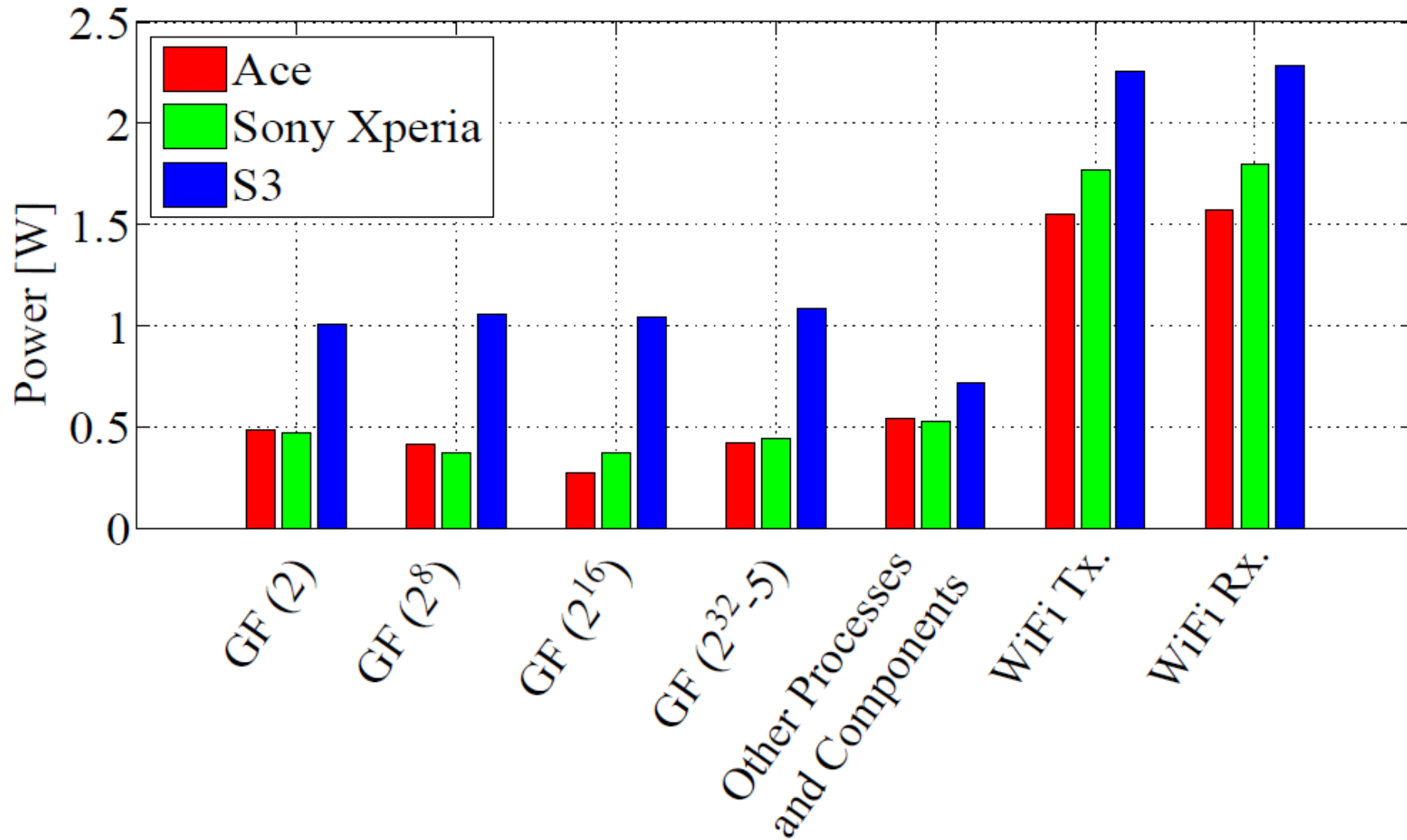
TABLE I
OVERVIEW OF THE FINITE FIELD IMPLEMENTATIONS USED.

$GF(2)$	The binary field $GF(2)$ was implemented using only the XOR operation, which typically yields very efficient implementations on most modern processors.
$GF(2^8)$	The binary extension field $GF(2^8)$ uses full multiplication and division lookup tables. Using this approach will replace the required polynomial multiplication or division of two field elements with a single table lookup. More details on these lookup tables are described in [17]. Addition and subtraction require only a bit-by-bit XOR.
$GF(2^{16})$	Computing a full multiplication and division table for $GF(2^{16})$ will typically require too much memory for most platforms (in the order of GBs). Instead, our implementation uses an extended logarithmic lookup, which reduces the memory requirements to the order of KBs. Addition and subtraction are performed as bit-by-bit XOR operation. Details are provided in [17].
$GF(2^{32}-5)$	This field is also known as an OPF (Optimal Prime Field). It has the advantage that it does not require any lookup tables and uses the standard arithmetic logic unit of the processor for all arithmetic operations. A detailed description of this field can be found in [18].

Measurement Setup



Energy consumption: KODO

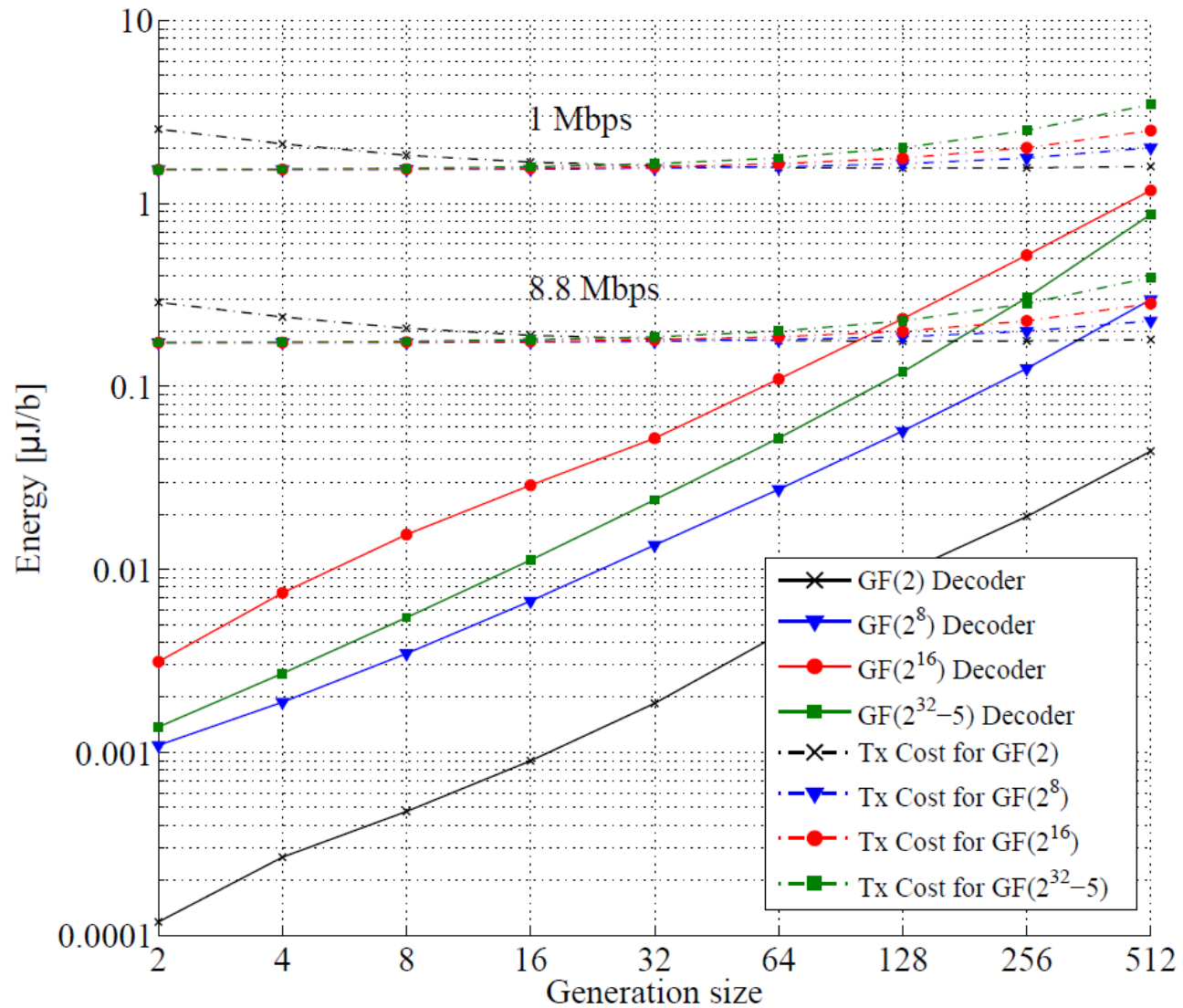


Energy consumption: KODO

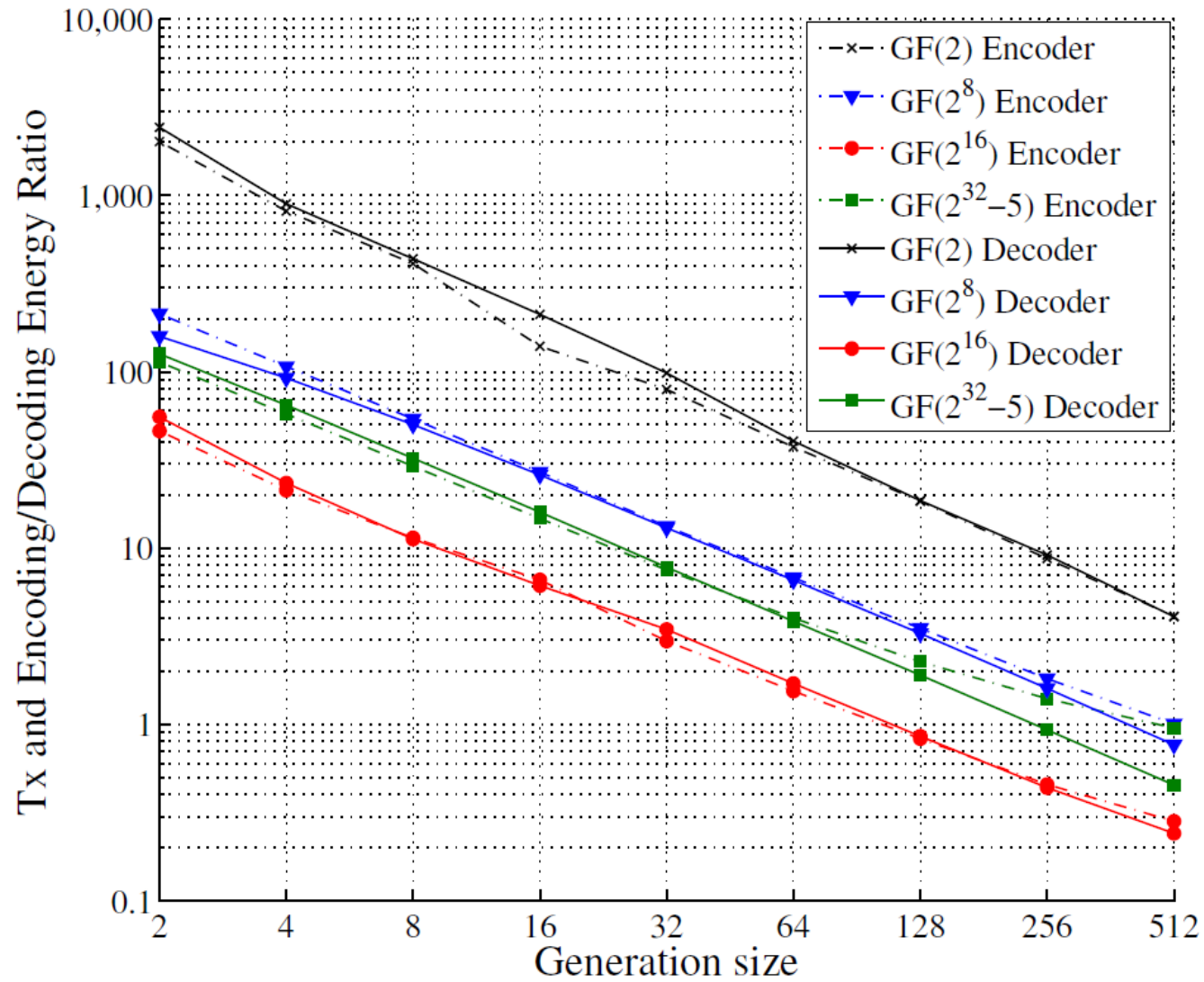
TABLE I
ENCODE AND DECODE PROCESSING SPEED (MBPS) AND AVERAGE PROCESSING POWER (W) FOR THREE MOBILE DEVICES.

Encoder Pkts/Gen.	$GF(2)$			$GF(2^8)$			$GF(2^{16})$			$GF(2^{32} - 5)$		
	Ace	Xperia	S3	Ace	Xperia	S3	Ace	Xperia	S3	Ace	Xperia	S3
2	1675.8	3328.1	3916.22	238.4	463.2	669.71	45.5	100.5	216.67	175.6	293.1	396.52
4	921.1	1615.3	2413.61	120.1	233.0	342.34	18.9	46.2	99.14	89.1	148.0	209.18
8	527.9	938.2	1394.60	59.4	117.4	172.91	9.1	24.8	36.04	44.9	74.0	80.76
16	253.1	347.1	640.2	30.5	58.3	85.6	4.7	14.1	27.1	22.7	36.7	52.5
32	132.6	206.9	338.6	15.3	28.5	41.6	2.4	6.2	13.9	11.3	18.0	25.8
64	66.5	99.8	152.3	7.5	14.3	20.7	1.2	3.1	7.0	5.6	9.0	12.8
128	30.3	49.5	77.6	3.6	7.1	10.3	0.6	1.6	3.5	2.8	4.5	6.4
256	11.7	23.3	38.7	1.6	3.4	5.1	0.3	0.8	1.6	1.3	2.2	3.2
512	4.8	10.8	18.5	0.8	1.7	2.5	0.1	0.4	0.8	0.6	1.1	1.6
Decoder	Ace	Xperia	S3	Ace	Xperia	S3	Ace	Xperia	S3	Ace	Xperia	S3
2	2079.0	4005.0	5545.00	148.6	345.3	477.57	53.5	120.7	203.53	187.6	325.5	440.20
4	944.6	1774.0	2499.21	92.2	200.7	294.45	17.9	50.8	102.71	96.8	166.1	231.63
8	561.2	998.8	1384.16	51.9	108.9	158.78	8.7	24.4	45.01	51.1	82.2	103.40
16	287.0	527.3	783.7	27.7	56.2	82.7	5.1	13.1	25.7	25.0	39.9	56.6
32	150.6	256.2	403.2	14.4	27.8	40.3	2.5	7.3	13.3	12.0	18.7	26.7
64	67.6	107.8	169.6	7.3	13.8	20.0	1.2	3.5	6.9	5.6	8.6	12.2
128	31.9	49.8	77.3	3.4	6.6	9.6	0.6	1.6	3.4	2.4	3.7	5.4
256	12.7	24.4	38.4	1.4	3.0	4.5	0.3	0.7	1.4	0.9	1.5	2.1
512	4.8	10.7	18.1	0.6	1.3	1.9	0.1	0.3	0.6	0.3	0.5	0.8
Power (W)	0.486	0.474	1.012	0.421	0.377	1.057	0.279	0.378	1.041	0.425	0.448	1.090

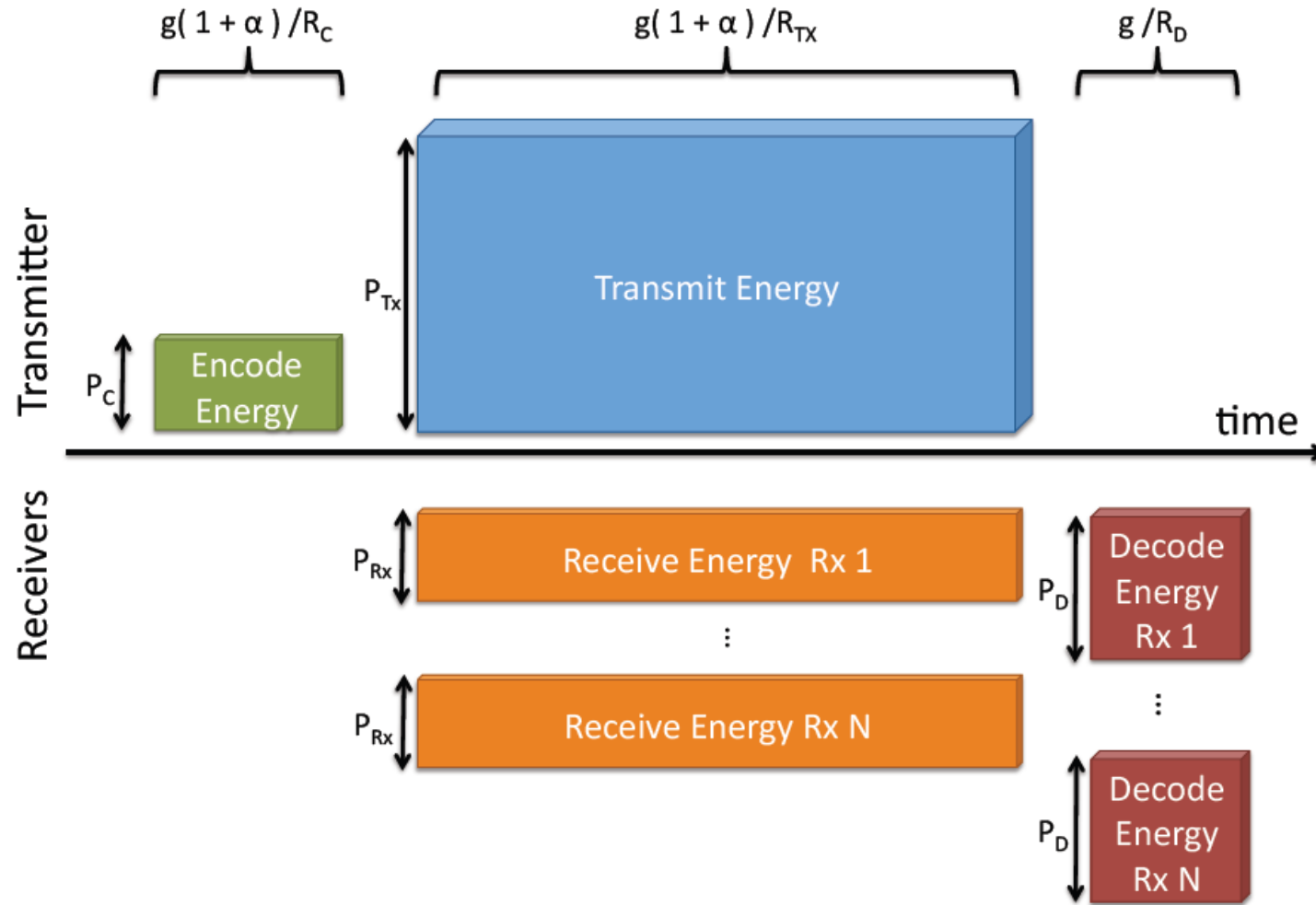
Energy consumption: KODO



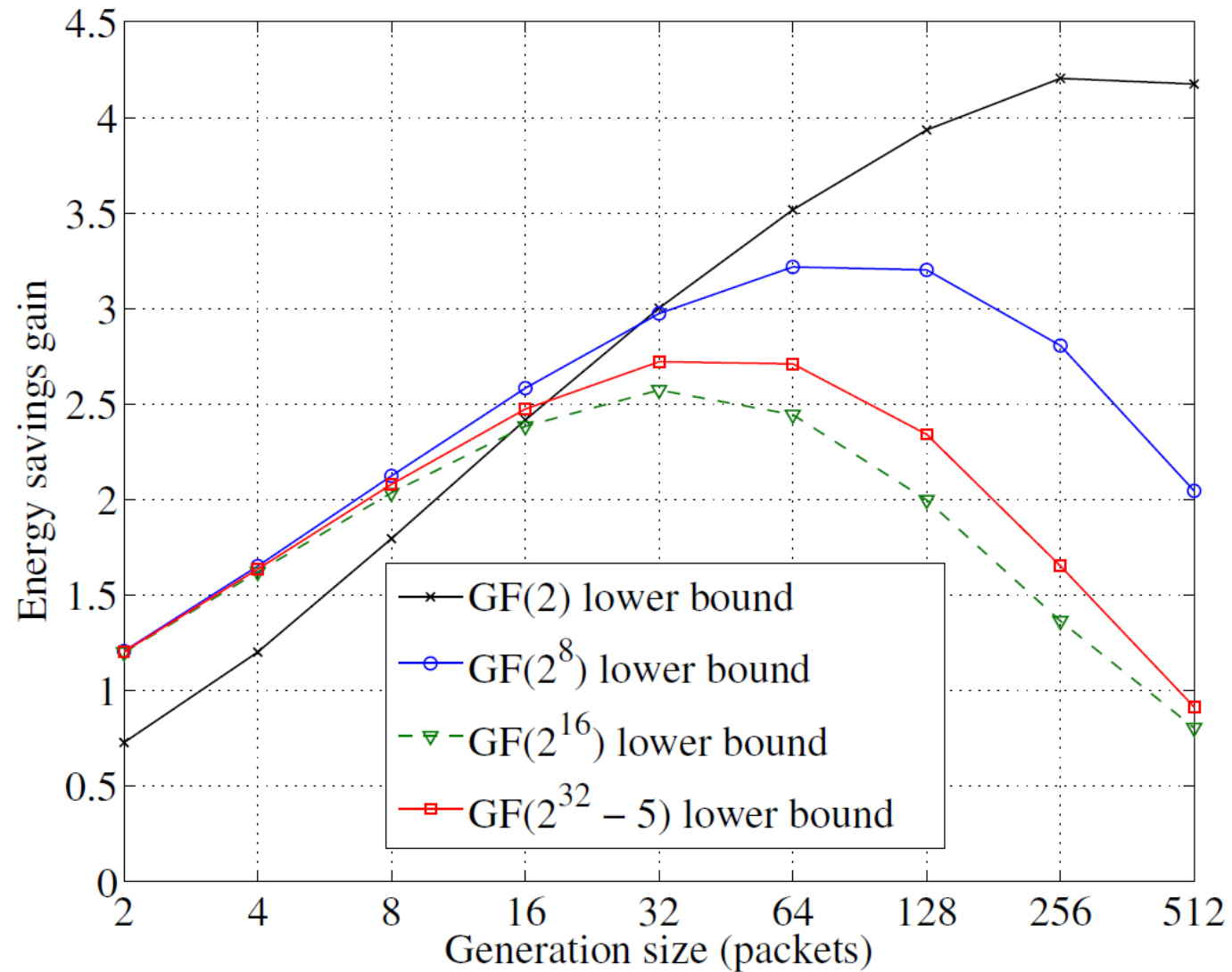
Energy consumption: KODO



Energy consumption: KODO



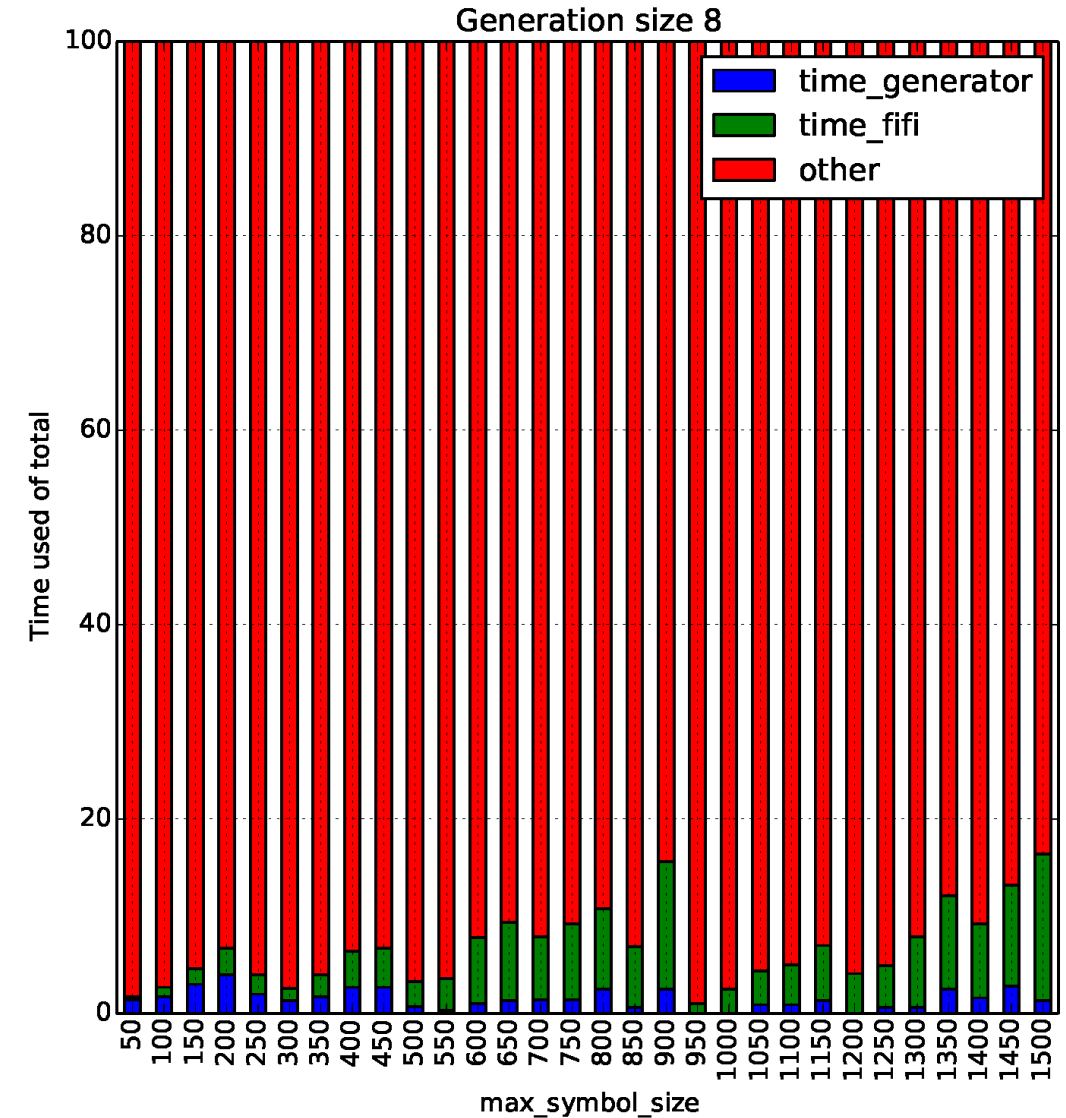
Energy consumption: KODO



Kodo: Practical Implementation

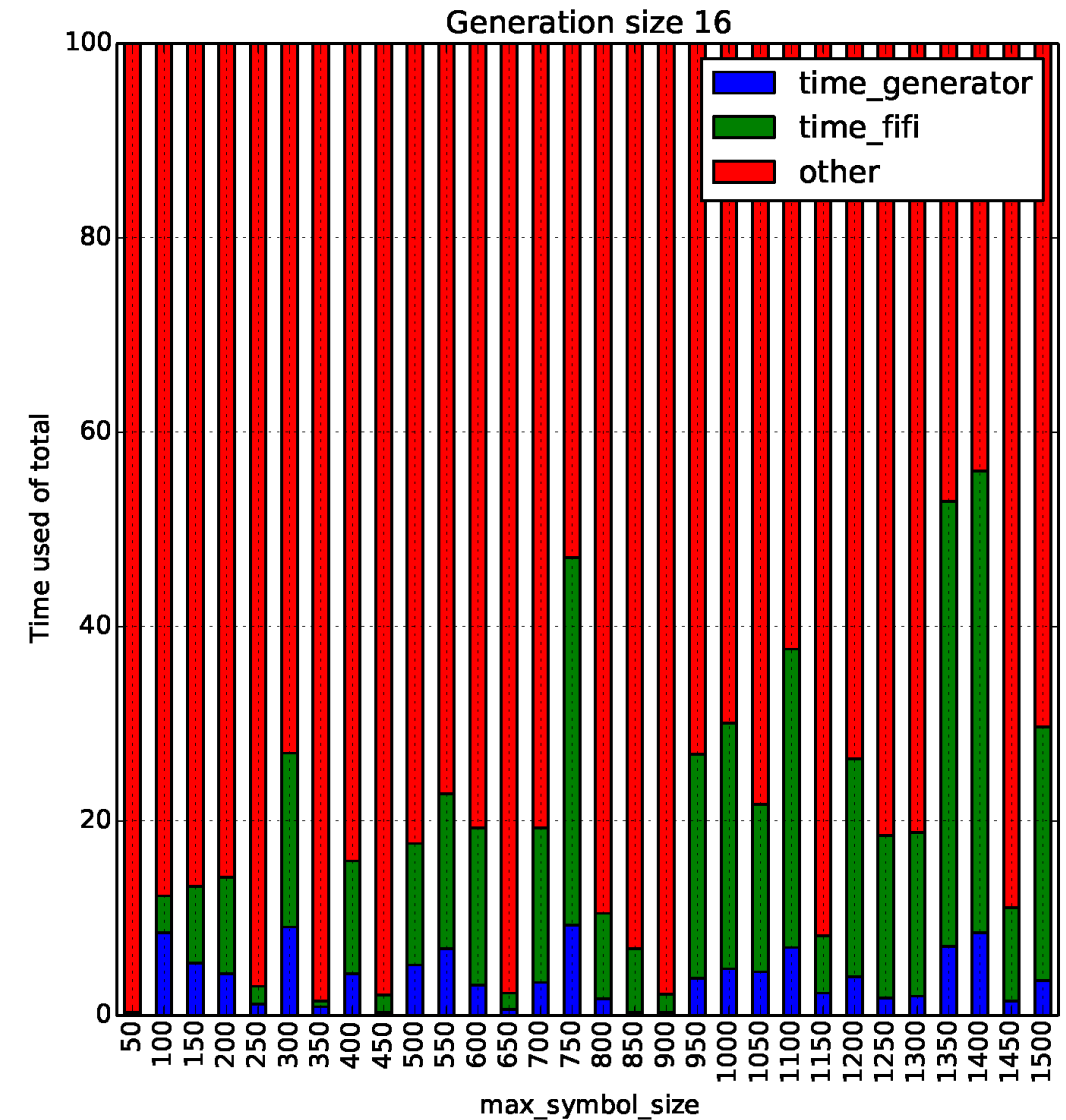
Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFO calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo



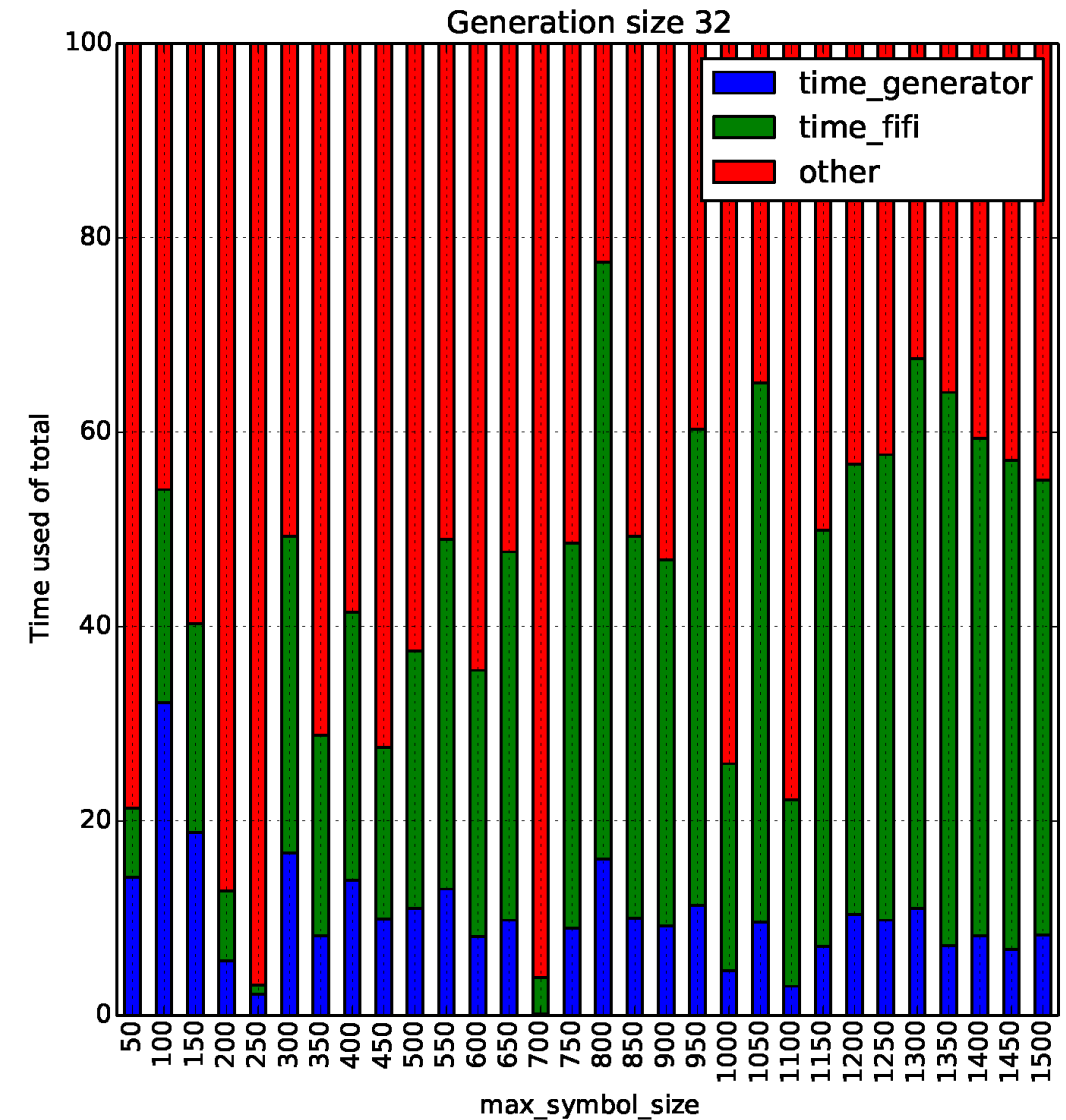
Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFO calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo



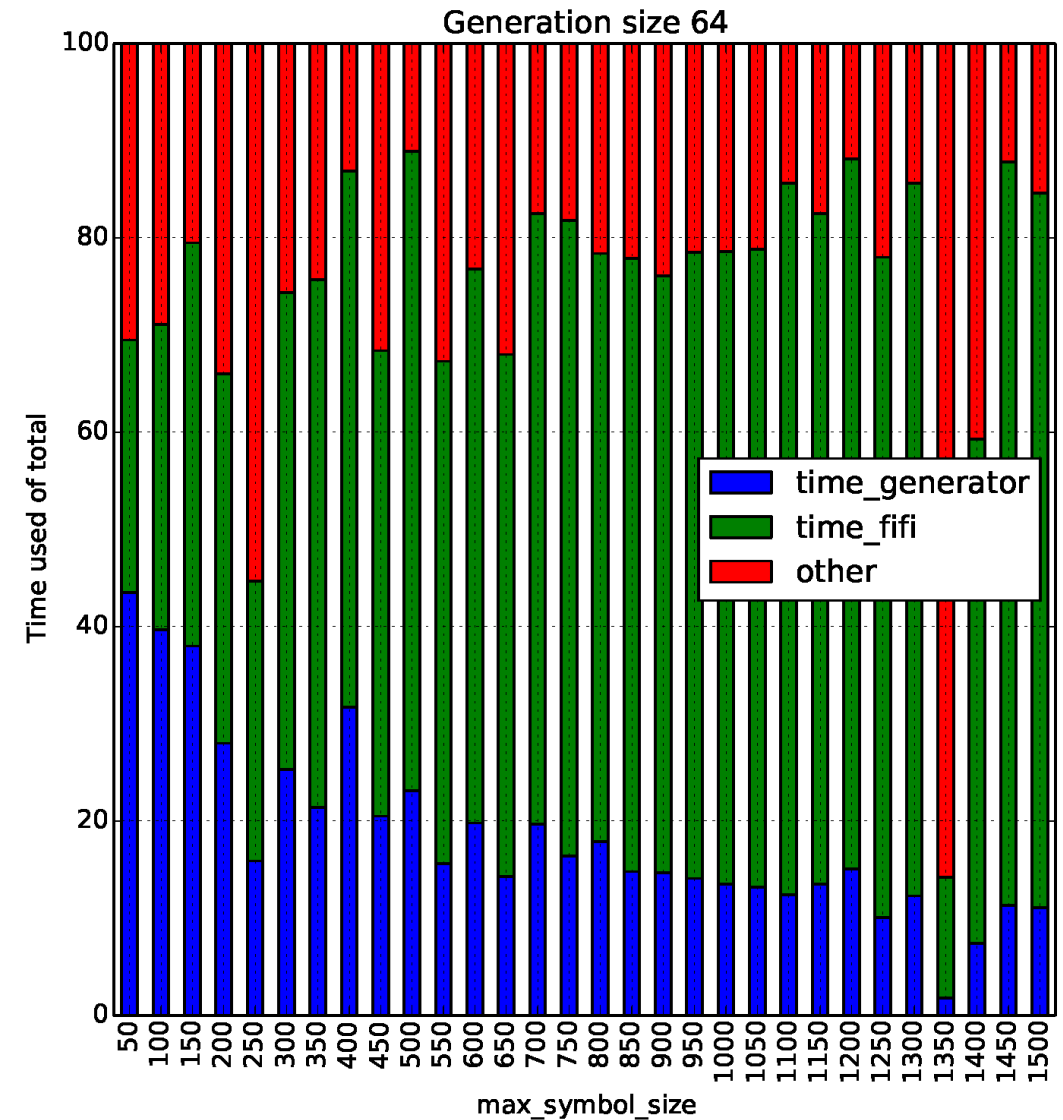
Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFI calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo



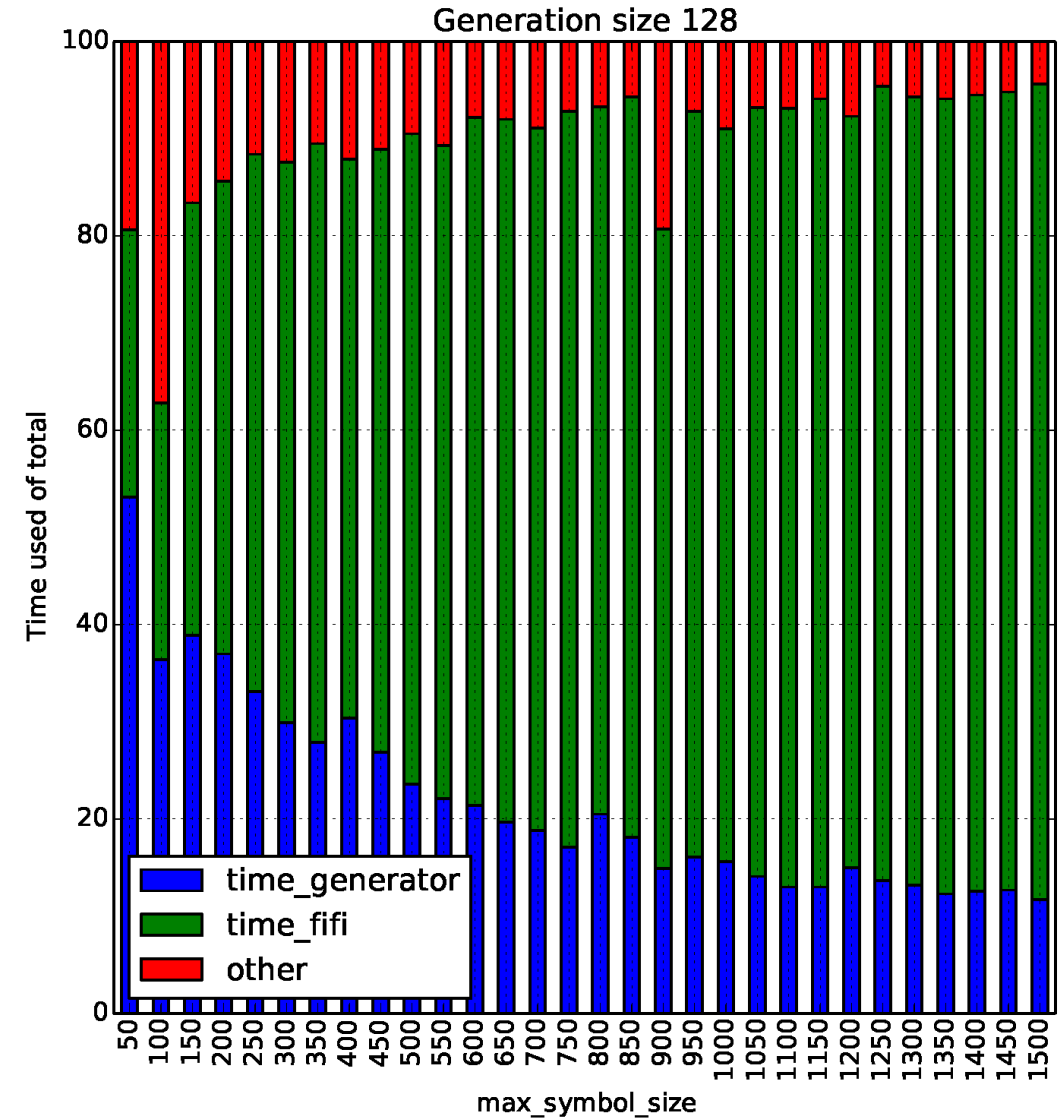
Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFI calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo



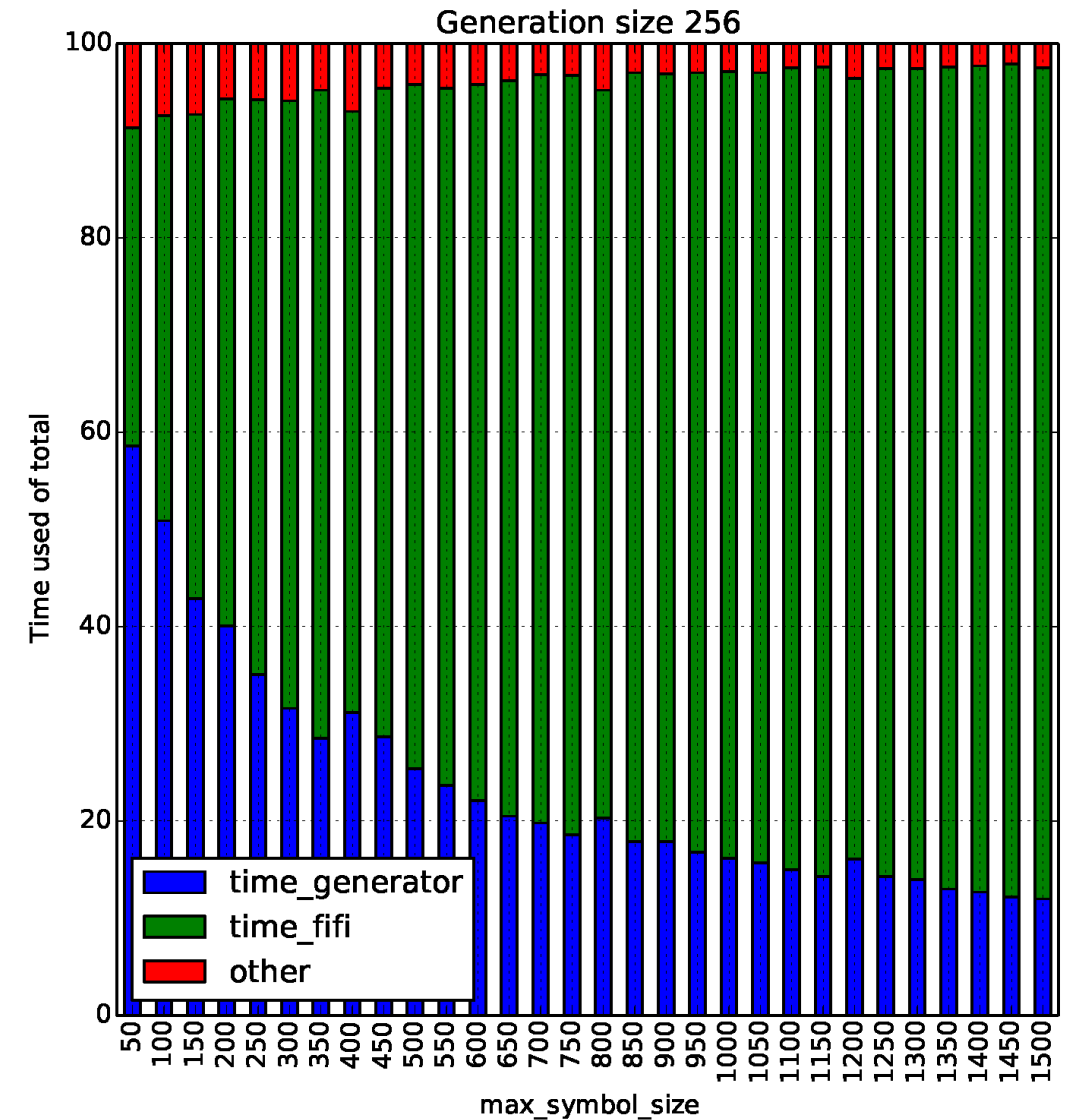
Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFI calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo



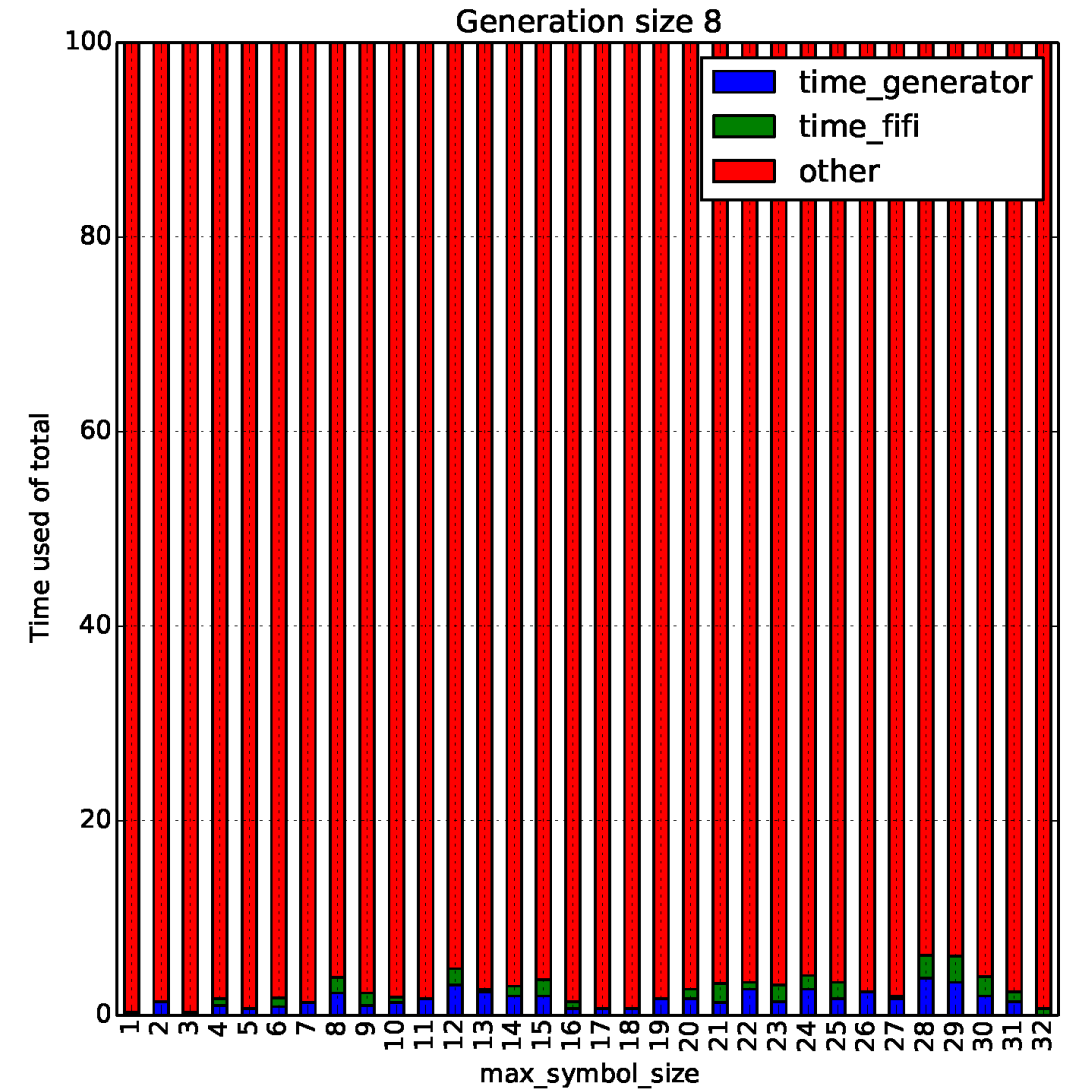
Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFI calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo



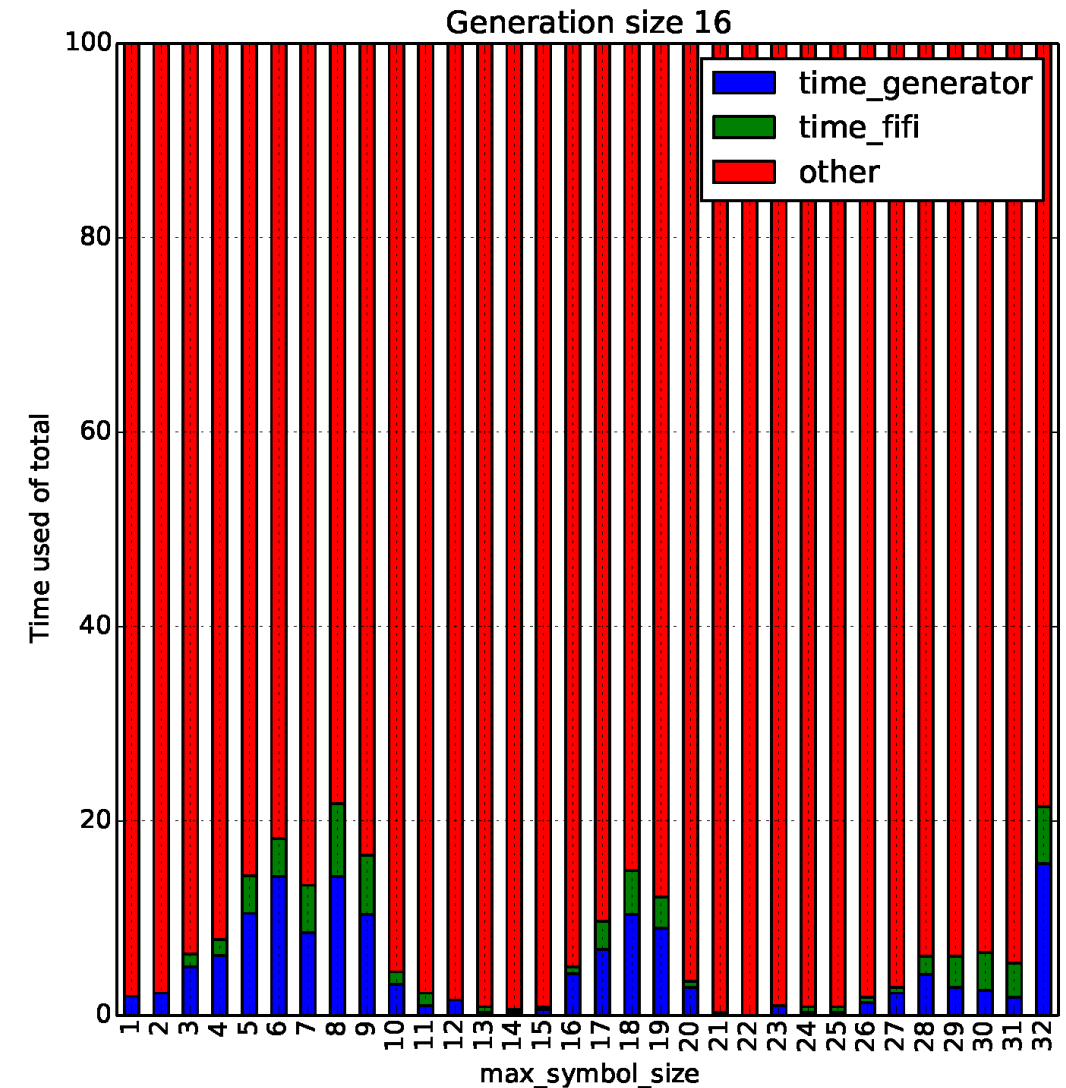
Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFI calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo
- SMALL packets, e.g. control



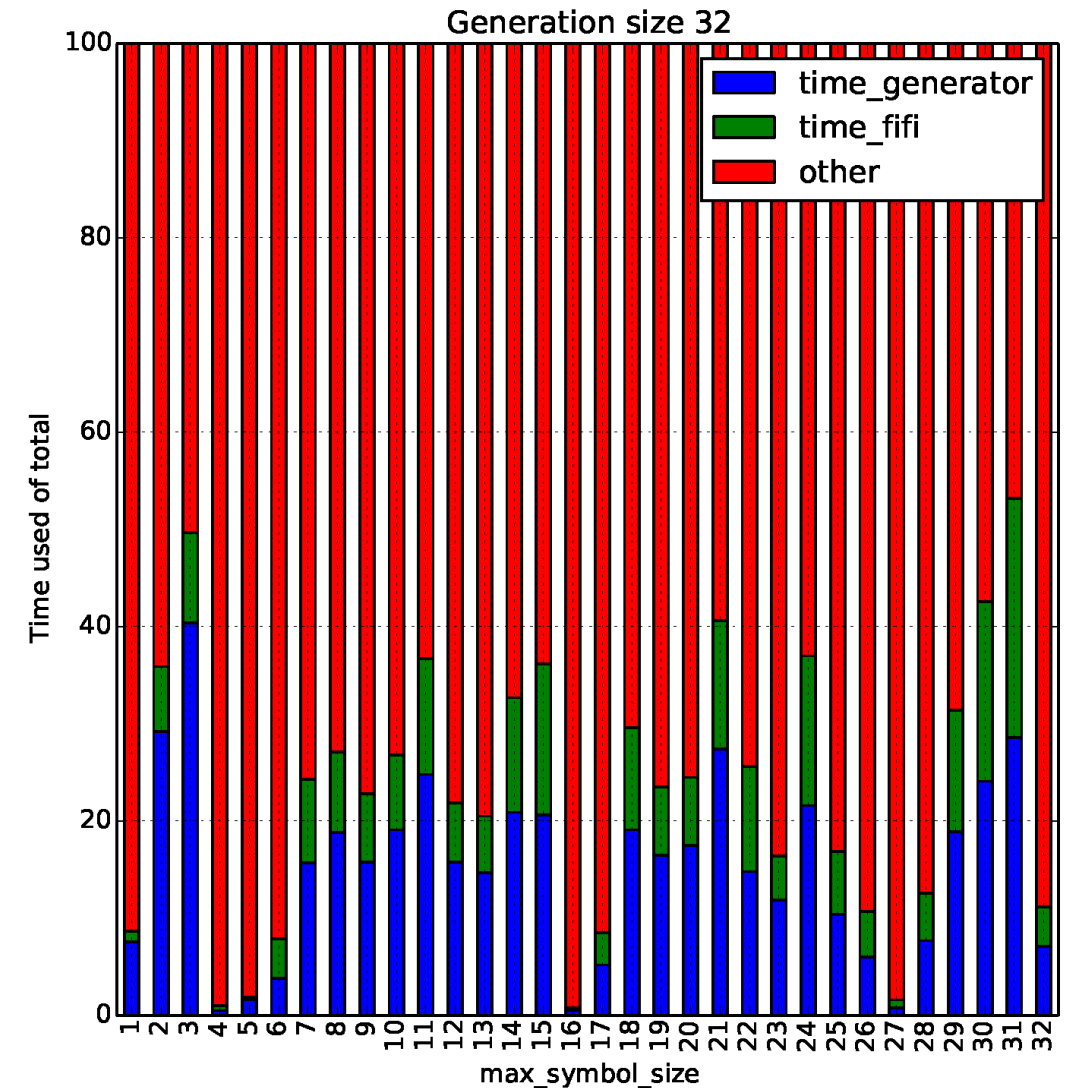
Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFI calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo
- SMALL packets, e.g. control



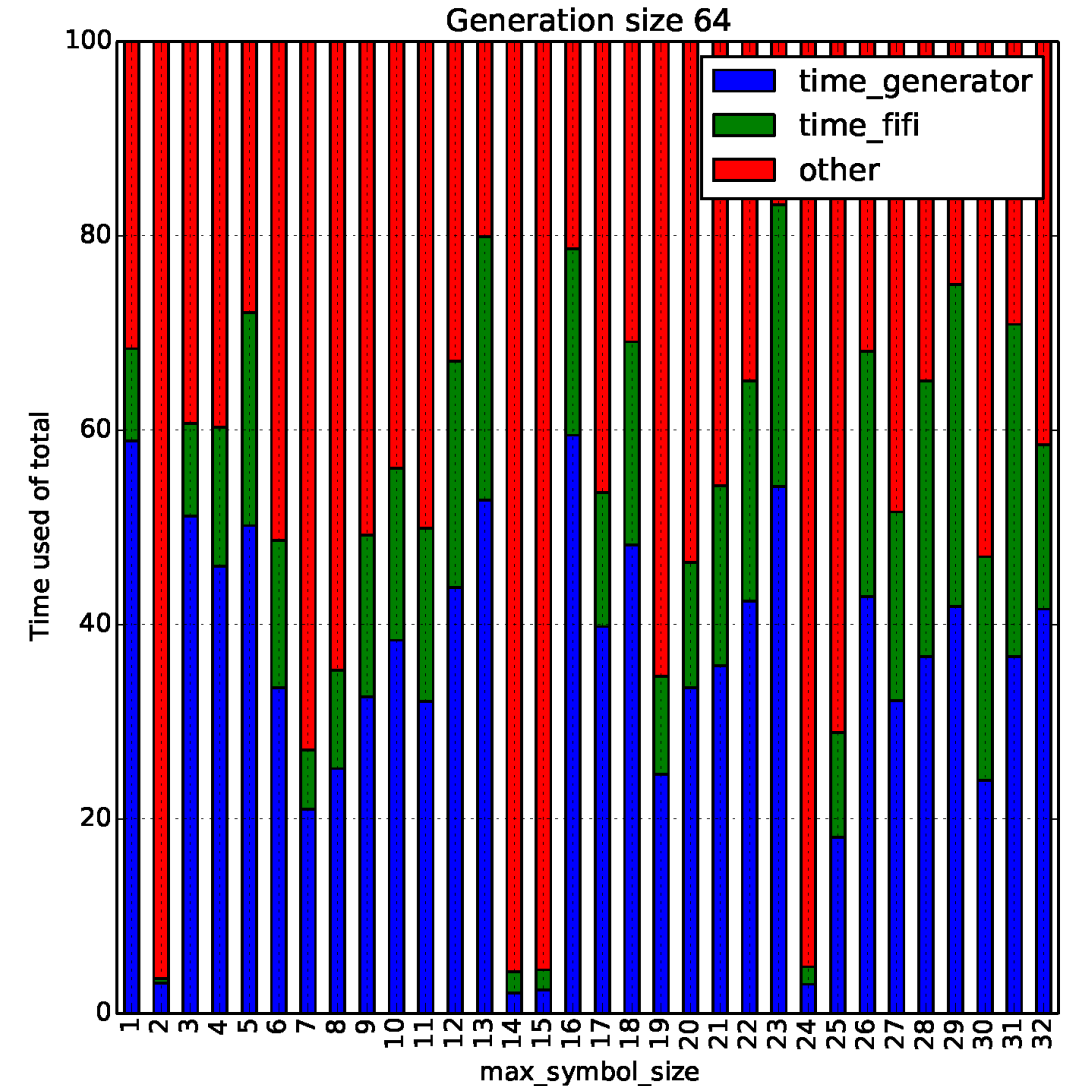
Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFO calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo
- SMALL packets, e.g. control



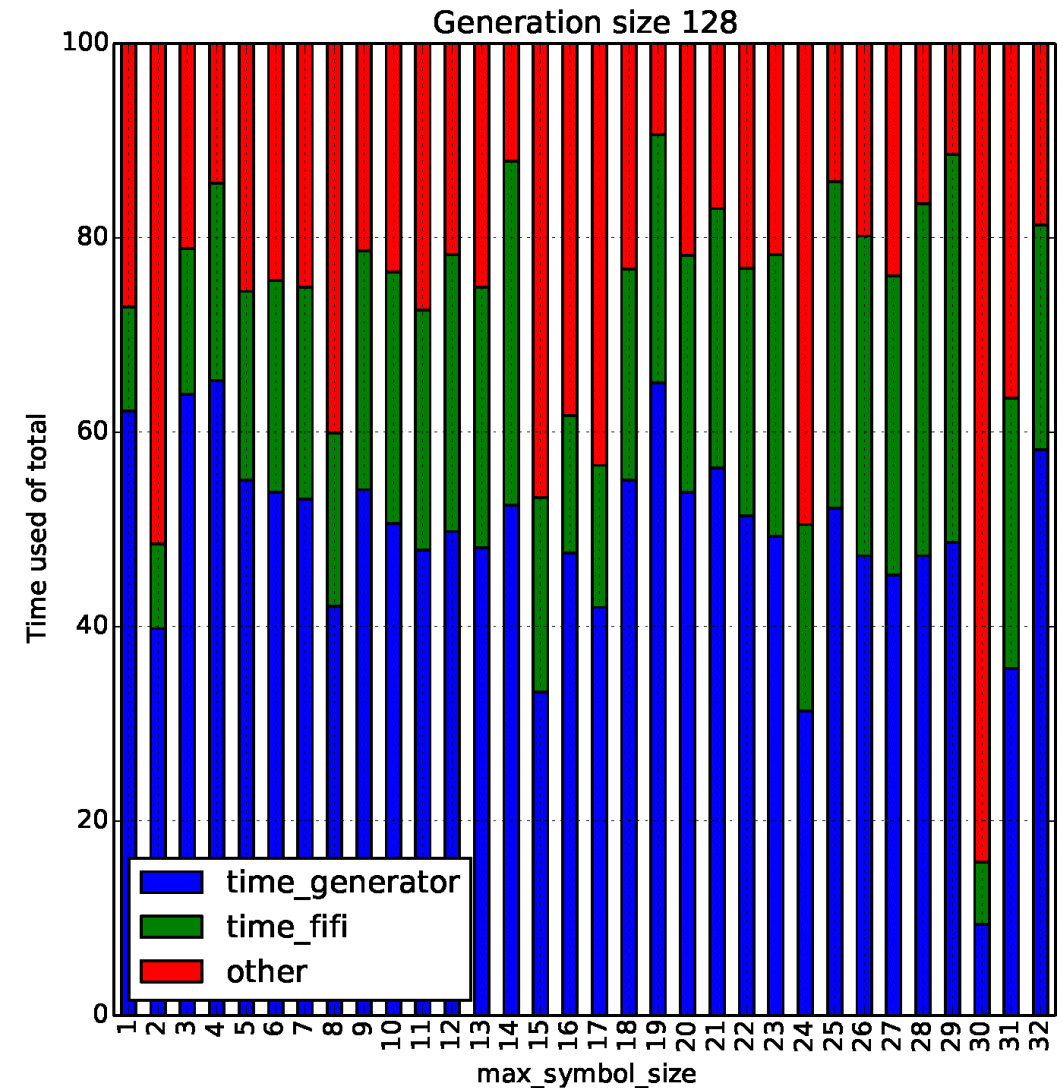
Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFI calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo
- SMALL packets, e.g. control



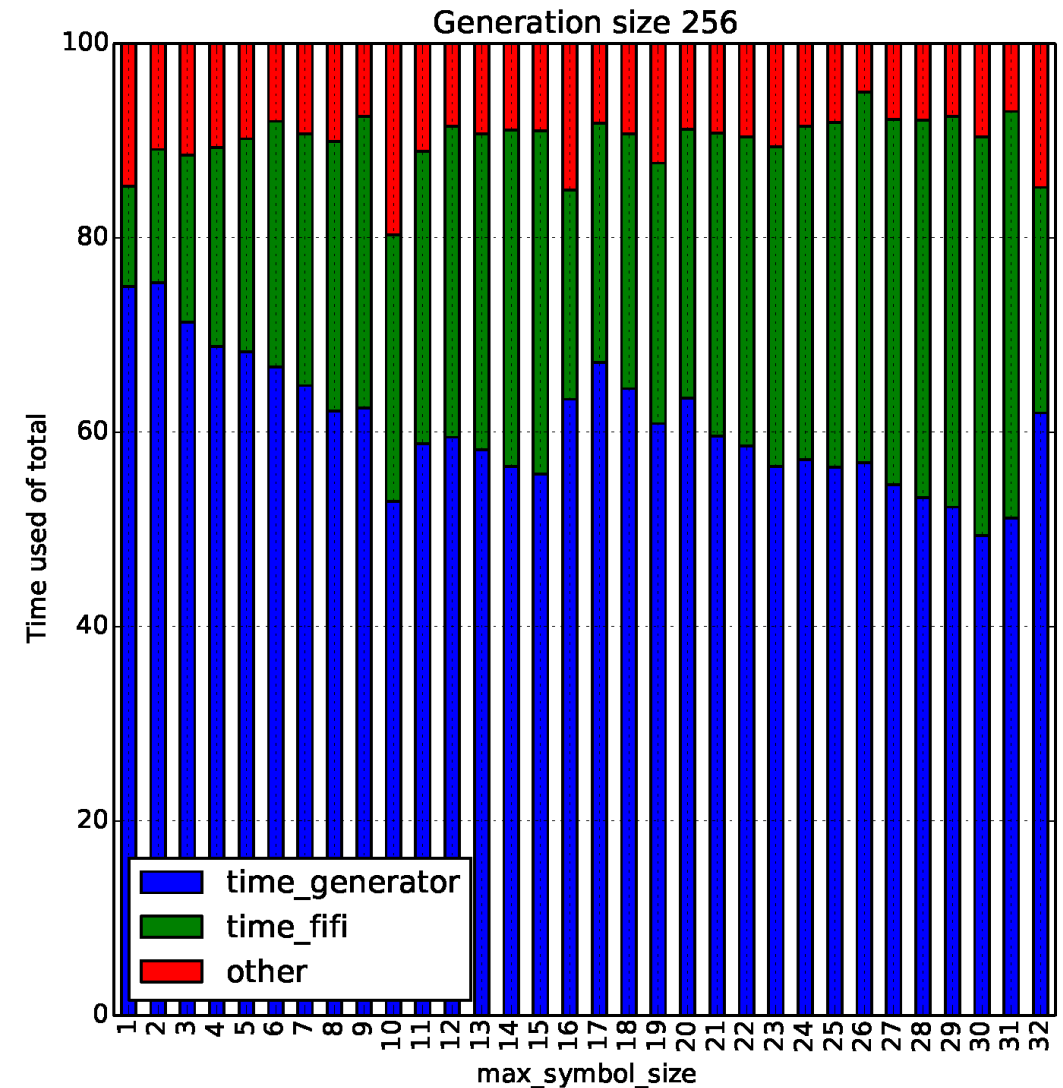
Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFI calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo
- SMALL packets, e.g. control



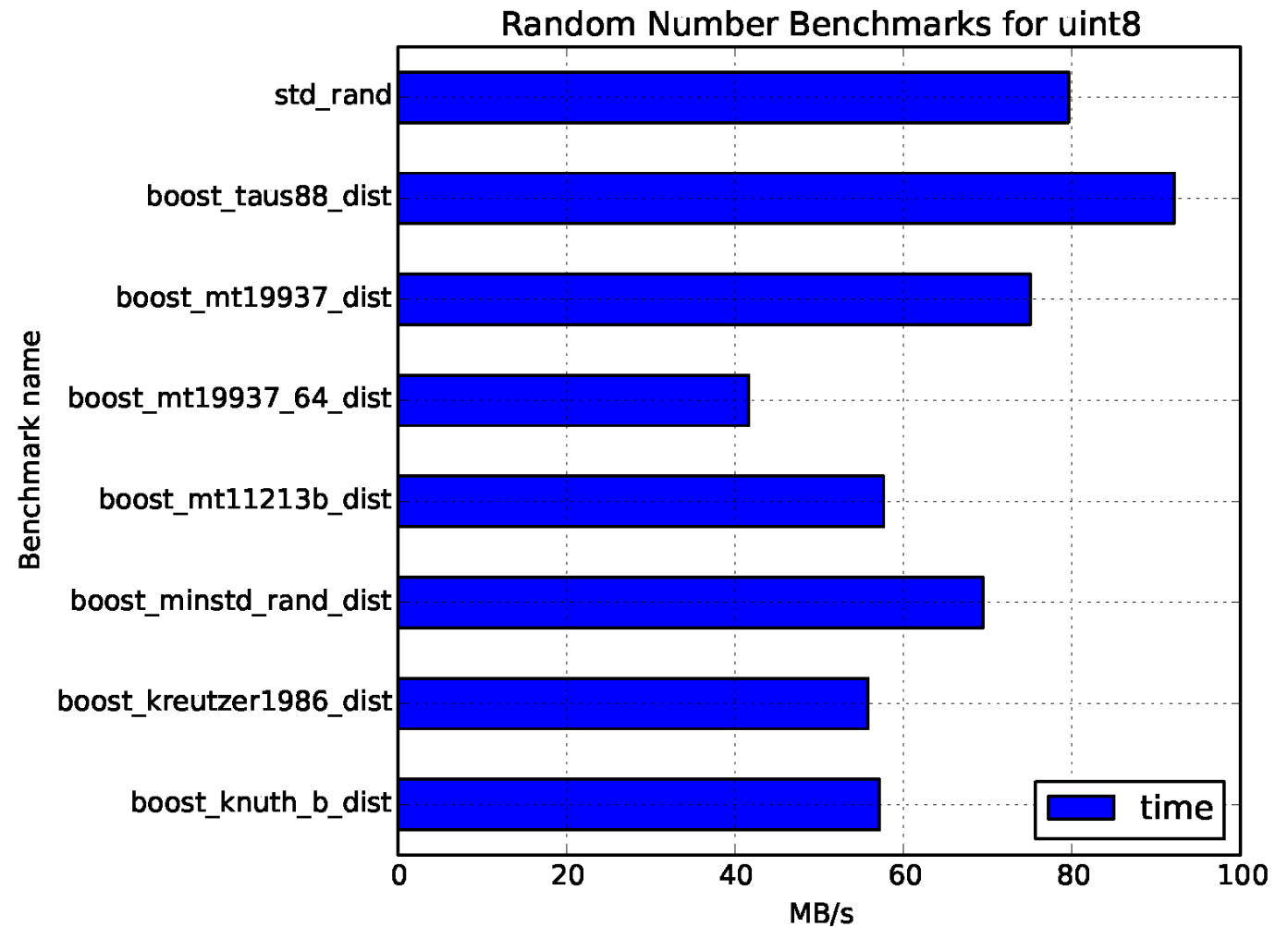
Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFI calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo
- SMALL packets, e.g. control



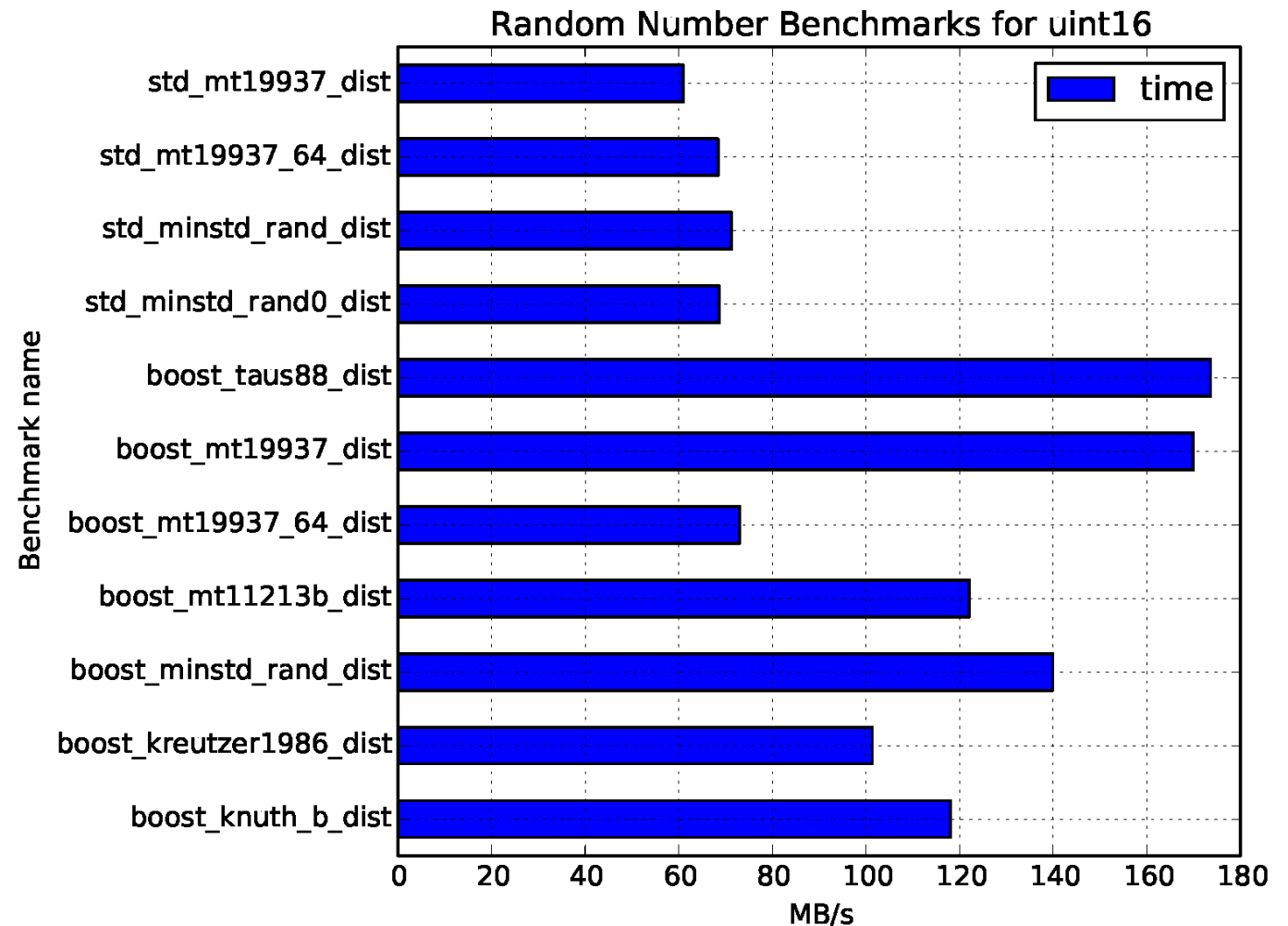
Random Generators

- The results show how many MB/s of random data can you generate with each algorithm
- *uintX* is how many X bits you generate at a time
- The *dist* and *raw* is whether just use then raw random number generator or feed it through a distribution (uniform)



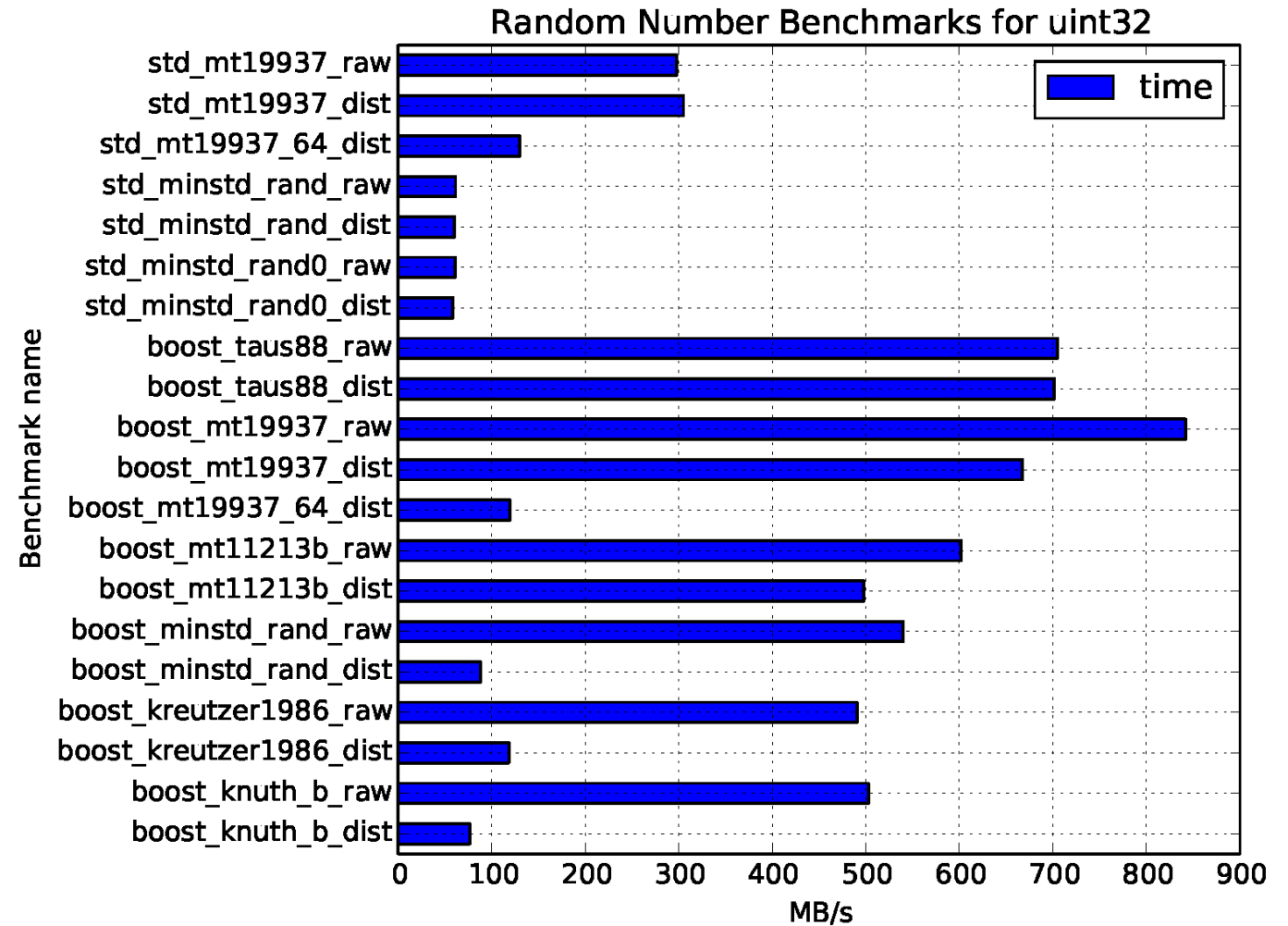
Random Generators

- The results show how many MB/s of random data can you generate with each algorithm
- *uintX* is how many X bits you generate at a time
- The *dist* and *raw* is whether just use then raw random number generator or feed it through a distribution (uniform)



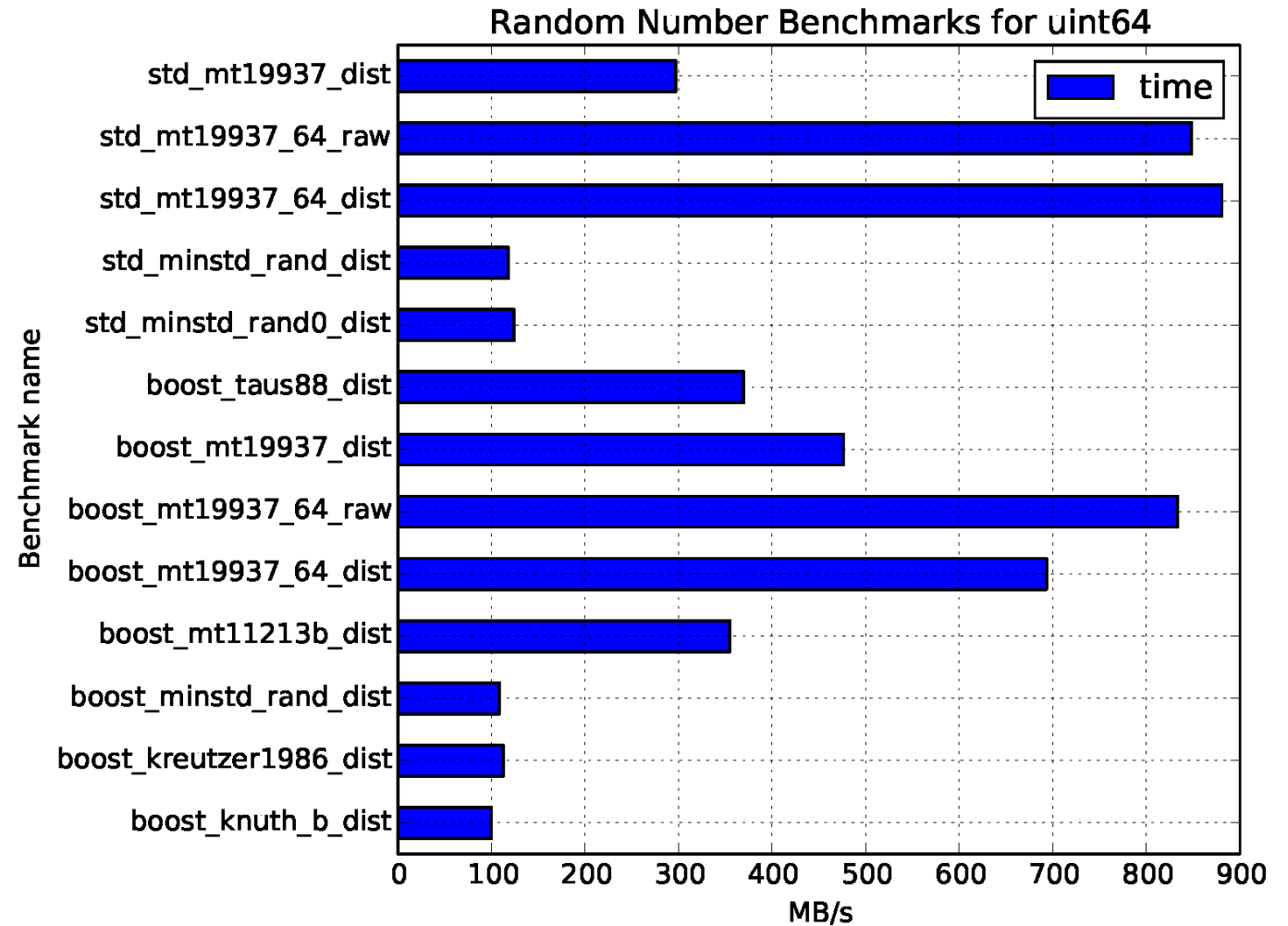
Random Generators

- The results show how many MB/s of random data can you generate with each algorithm
- *uintX* is how many X bits you generate at a time
- The *dist* and *raw* is whether just use then raw random number generator or feed it through a distribution (uniform)

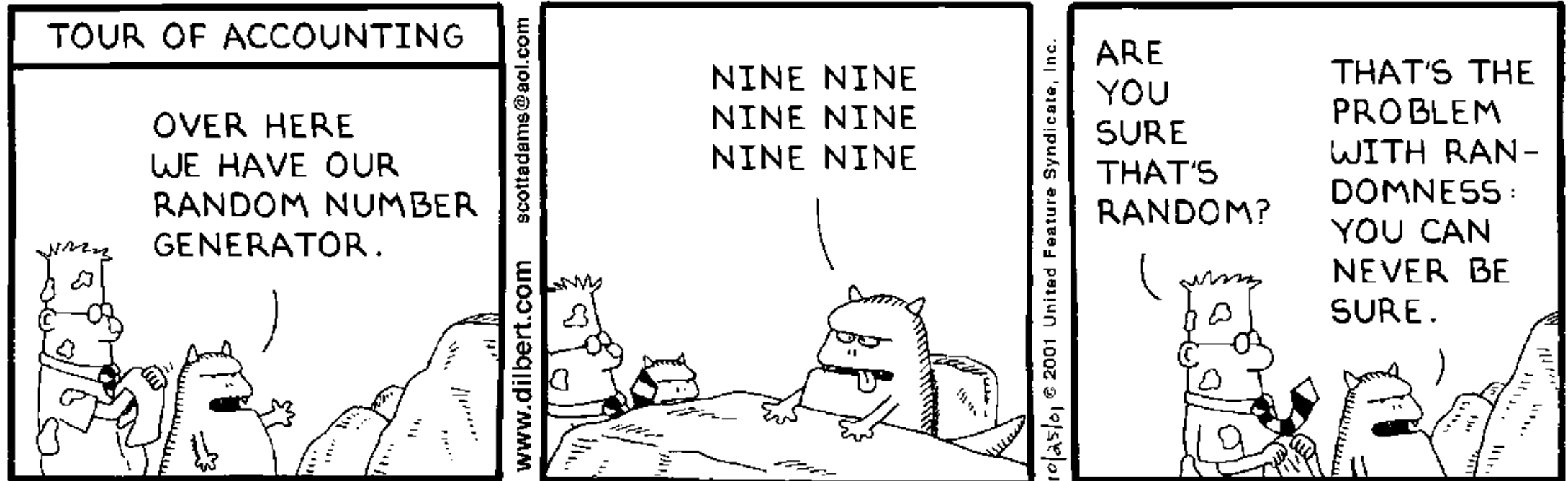


Random Generators

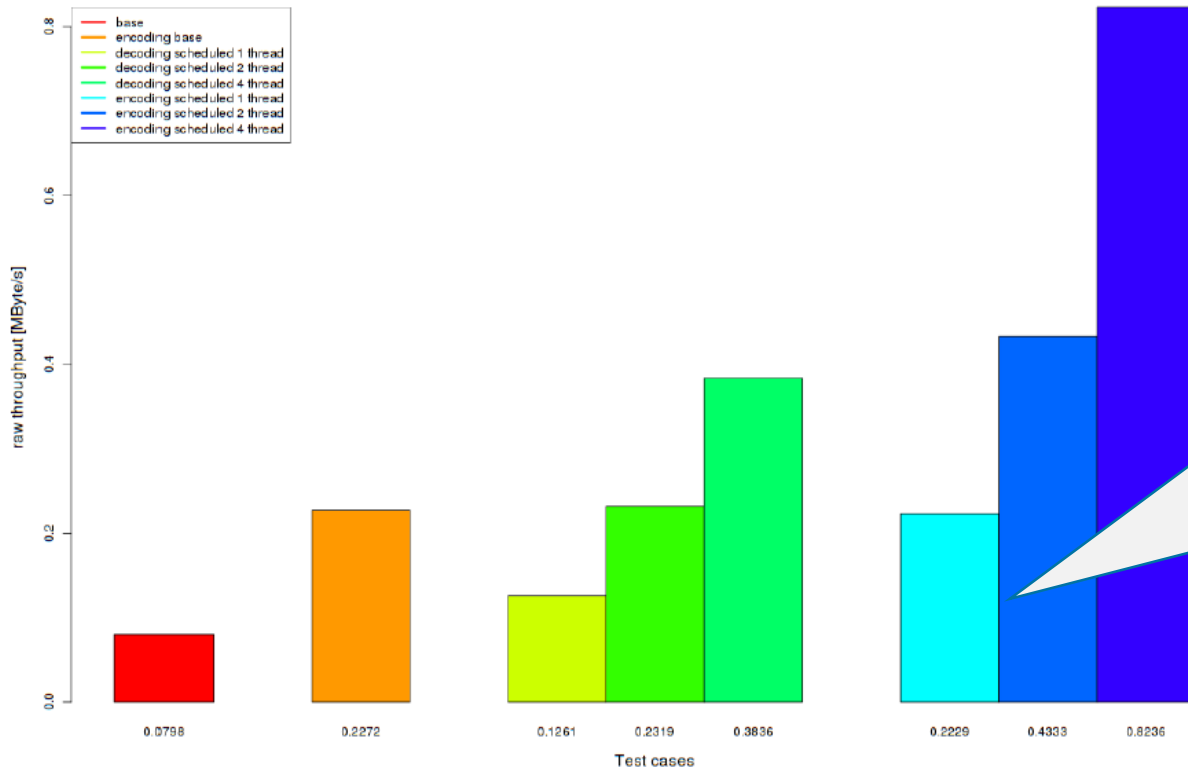
- The results show how many MB/s of random data can you generate with each algorithm
- *uintX* is how many X bits you generate at a time
- The *dist* and *raw* is whether just use then raw random number generator or feed it through a distribution (uniform)



Random Generators



Many-Core Implementation of Network Coding



Herb Sutter: „Free lunch is over!“

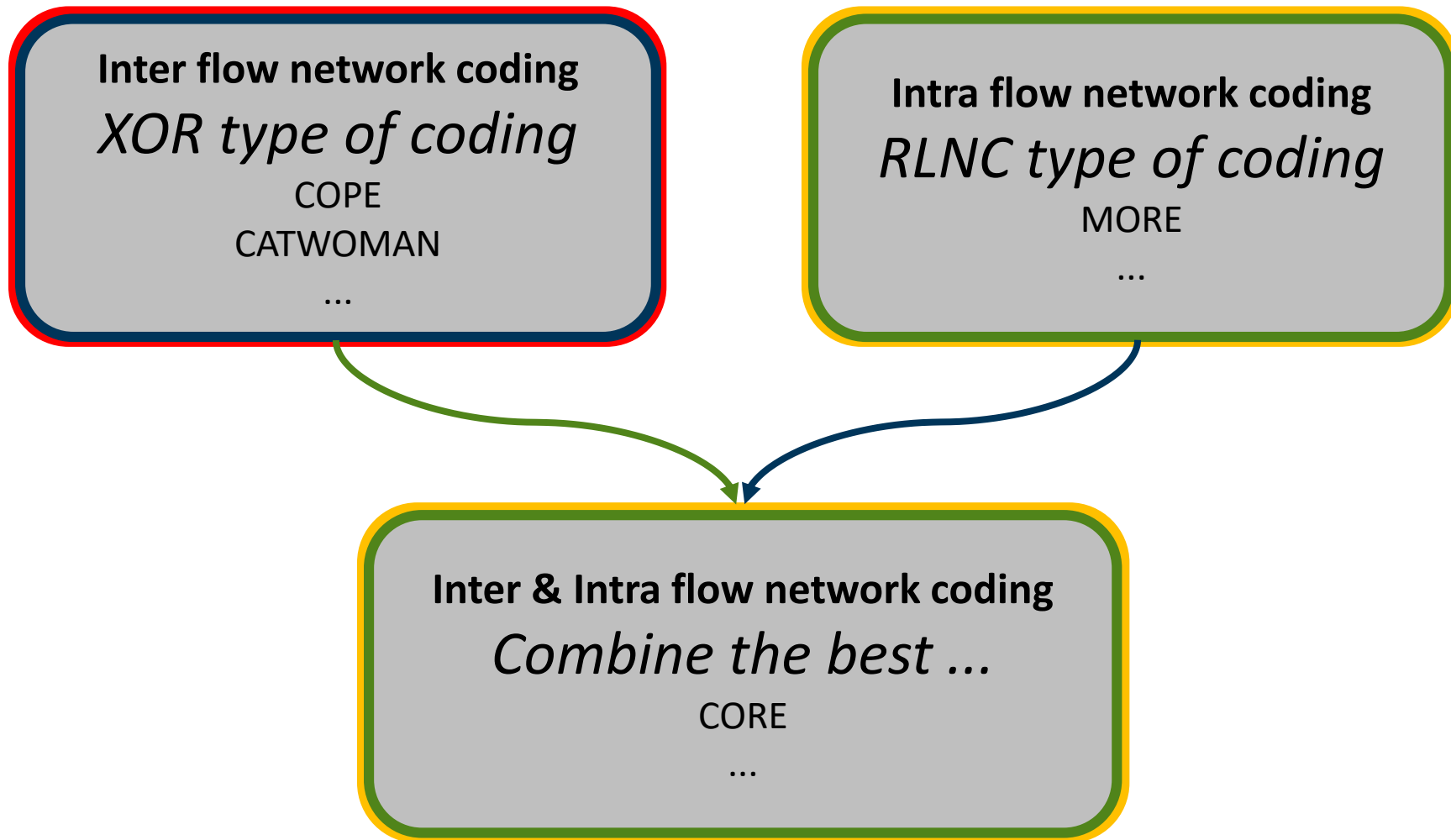
Lots of research in computer science and engineering focus on achieving low computational complexity (the big O). But perhaps in the future we need to consider algorithms with worse computational complexity but which are easy to parallelize.

On Raspberry Pi 2: 10x speed up over standard SIMD encoding by using 4 cores and cache optimization (generation size 1024)

Combination of Inter and Intra Flow Network Coding

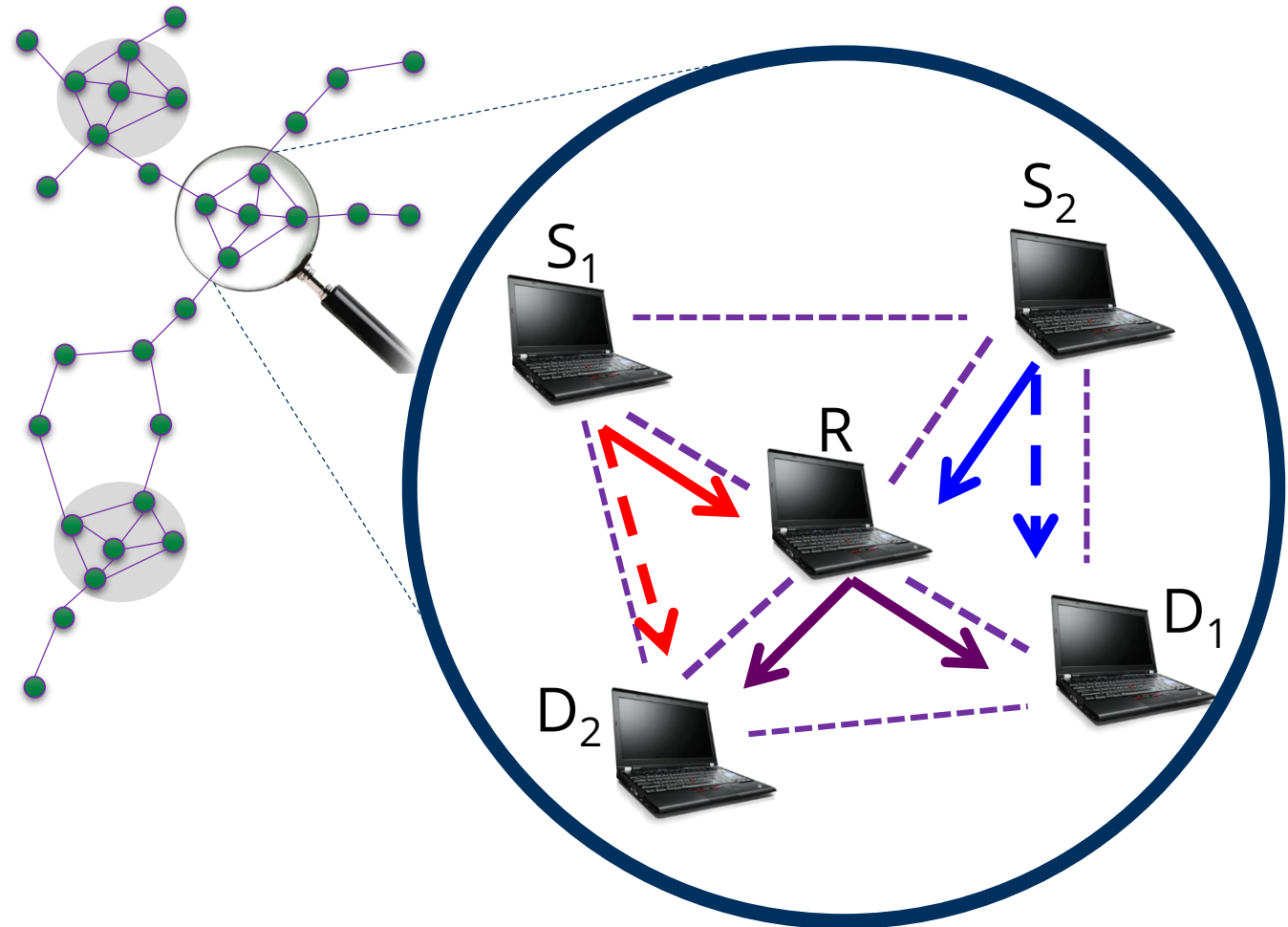
J. Krigslund, J. Hansen, M. Hundeboll, F.H.P. Fitzek, and T. Larsen, "Core: Cope with more in wireless meshed networks," in IEEE VTC2013-Spring: Cooperative Communication, Distributed MIMO and Relaying, Dresden, Germany, June 2013.

Combination of Inter and Intra Flow Network Coding

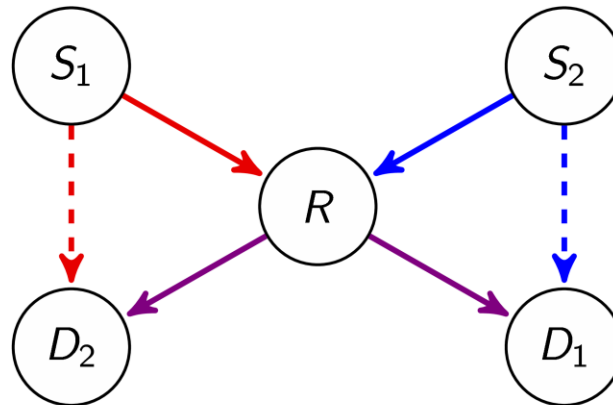


Setup under investigation

Challenge: channel losses
Overhearing is critical for inter-session



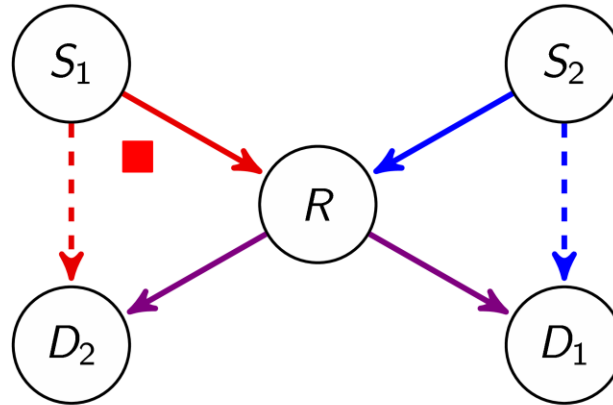
CORE



	P ₁	P ₂	P ₃	P ₁	P ₂	P ₃
1						
2						
3						
4						
5						
6						
7						

	P ₁	P ₂	P ₃	P ₁	P ₂	P ₃
1						
2						
3						
4						
5						
6						
7						

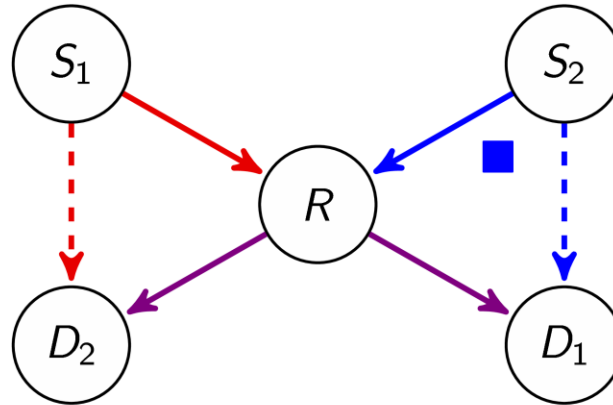
CORE



	P ₁	P ₂	P ₃	P ₁	P ₂	P ₃
1						
2						
3						
4						
5						
6						
7						

	P ₁	P ₂	P ₃	P ₁	P ₂	P ₃
1						
2						
3						
4						
5						
6						
7						

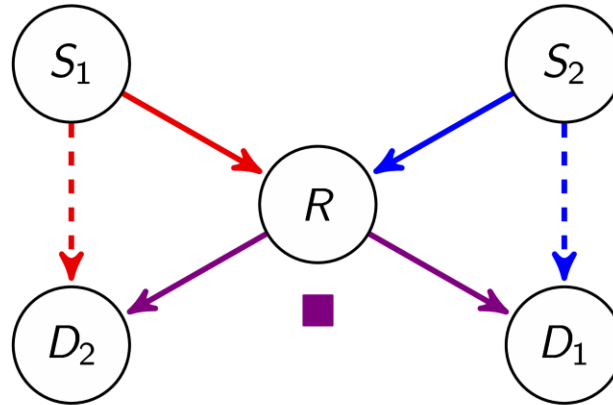
CORE



	P ₁	P ₂	P ₃	P ₁	P ₂	P ₃
1						
2						
3						
4						
5						
6						
7						

	P ₁	P ₂	P ₃	P ₁	P ₂	P ₃
1						
2						
3						
4						
5						
6						
7						

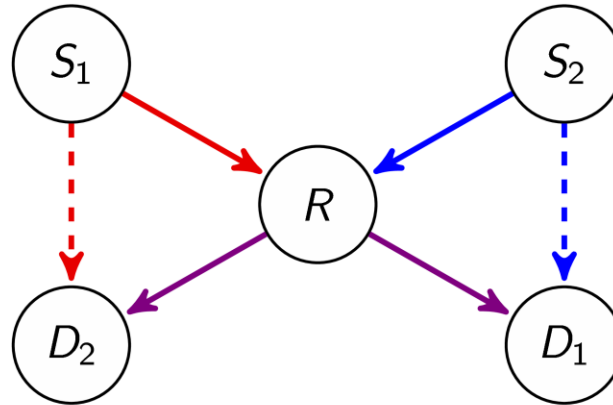
CORE



	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

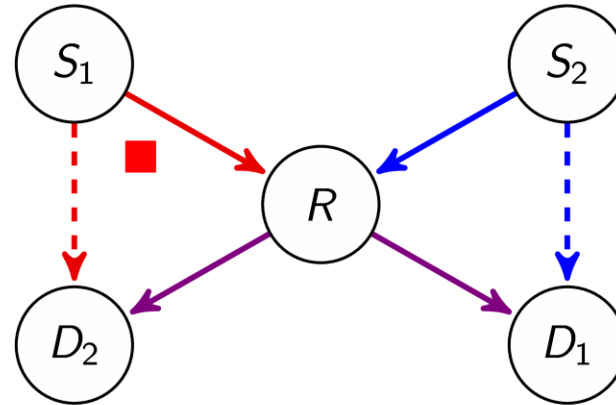
CORE



	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

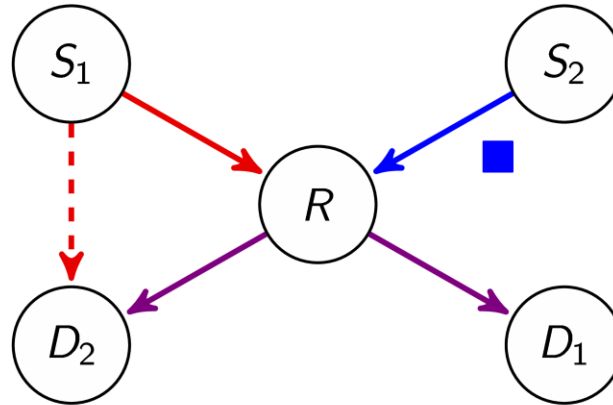
CORE



	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

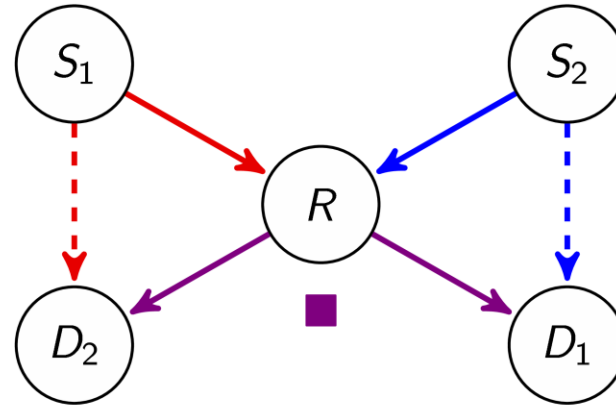
CORE



	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

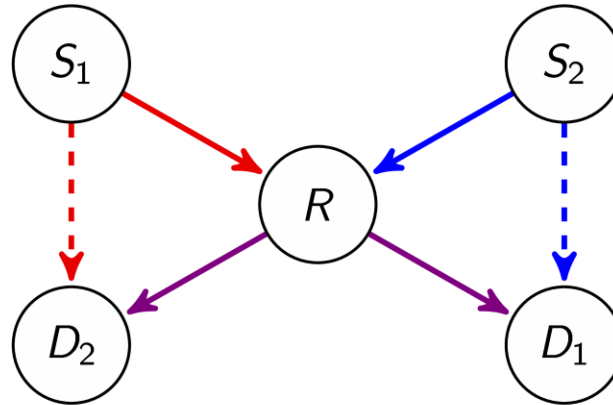
CORE



	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

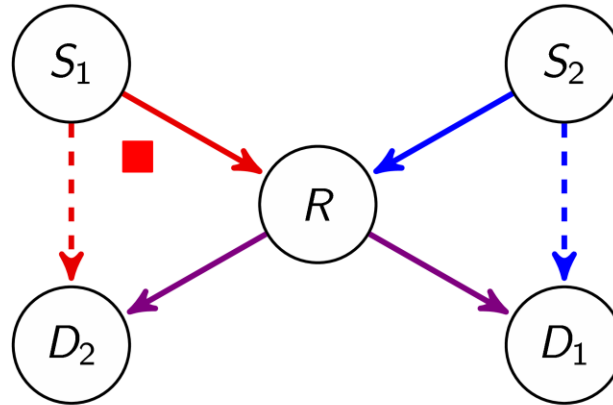
CORE



	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

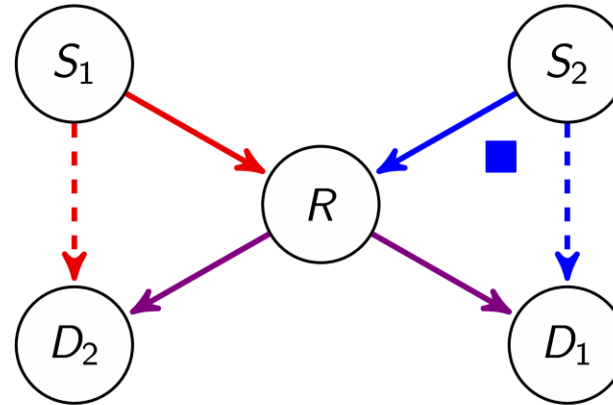
CORE



	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

CORE

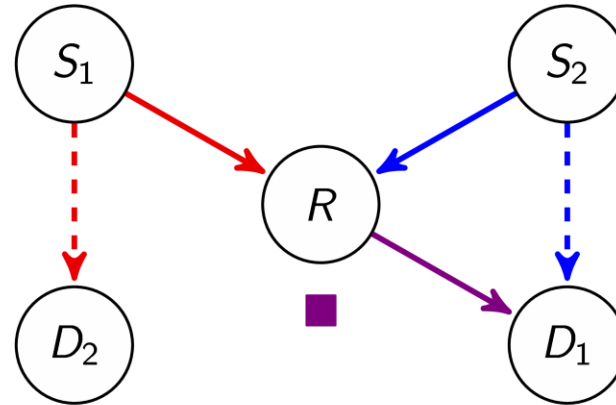


Relay has full rank!
Stop sources ...

	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

CORE

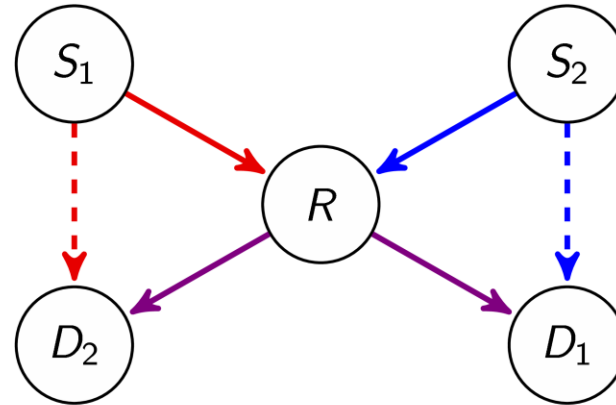


No retransmission

	P ₁	P ₂	P ₃	P ₁	P ₂	P ₃
1				Red	Red	Red
2	Blue	Blue	Blue			
3				Red	Red	Red
4	Blue	Blue	Blue			
5				Red	Red	Red
6						
7						

	P ₁	P ₂	P ₃	P ₁	P ₂	P ₃
1				Blue	Blue	Blue
2	Red	Red	Red			
3						
4	Purple	Purple	Purple	Purple	Purple	Purple
5				Blue	Blue	Blue
6	Purple	Purple	Purple	Purple	Purple	Purple
7						

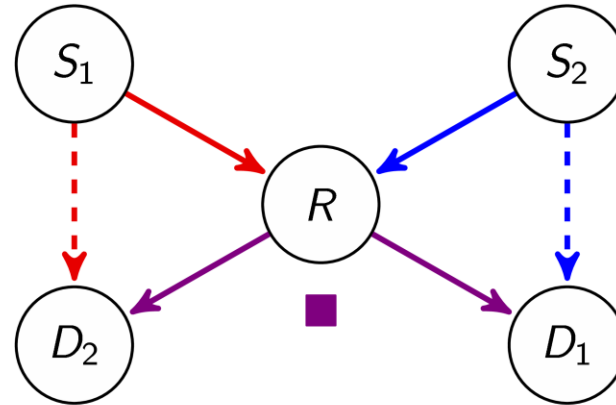
CORE



	P_1	P_2	P_3	P_1	P_2	P_3
1				Red	Red	Red
2	Blue	Blue	Blue			
3				Red	Red	Red
4	Blue	Blue	Blue			
5				Red	Red	Red
6						
7						

	P_1	P_2	P_3	P_1	P_2	P_3
1				Blue	Blue	Blue
2	Red	Red	Red			
3						
4	Purple	Purple	Purple	Purple	Purple	Purple
5				Blue	Blue	Blue
6	Red	Red	Red			
7						

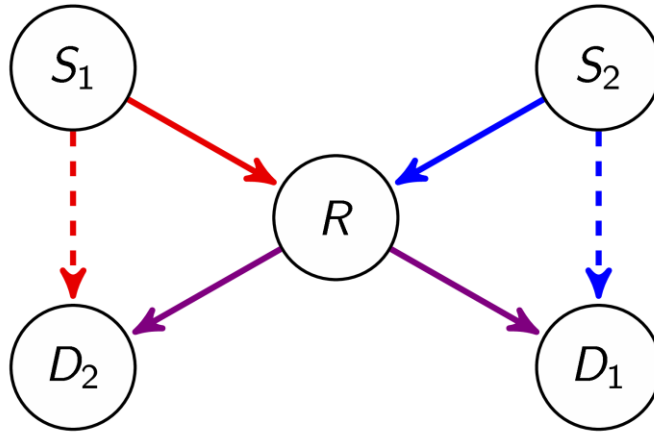
CORE



	P_1	P_2	P_3	P_1	P_2	P_3
1				Red	Red	Red
2	Blue	Blue	Blue			
3				Red	Red	Red
4	Blue	Blue	Blue			
5				Red	Red	Red
6						
7	Purple	Purple	Purple	Purple	Purple	Purple

	P_1	P_2	P_3	P_1	P_2	P_3
1				Blue	Blue	Blue
2	Red	Red	Red			
3						
4	Purple	Purple	Purple	Purple	Purple	Purple
5				Blue	Blue	Blue
6	Red	Red	Red			
7	Purple	Purple	Purple	Purple	Purple	Purple

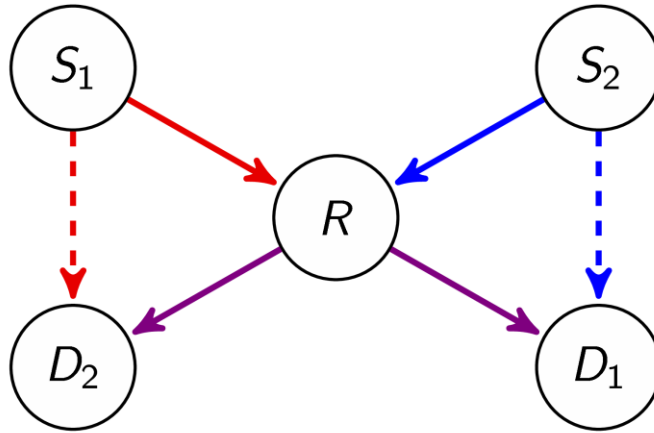
CORE



	P ₁	P ₂	P ₃	P ₁	P ₂	P ₃
1	Blue	Blue	Blue			
2	Blue	Blue	Blue			
3	Purple	Purple	Purple	Purple	Purple	Purple
4				Red	Red	Red
5				Red	Red	Red
6				Red	Red	Red

	P ₁	P ₂	P ₃	P ₁	P ₂	P ₃
1	Red	Red	Red			
2	Red	Red	Red			
3	Purple	Purple	Purple	Purple	Purple	Purple
4	Purple	Purple	Purple	Purple	Purple	Purple
5				Blue	Blue	Blue
6				Blue	Blue	Blue

CORE



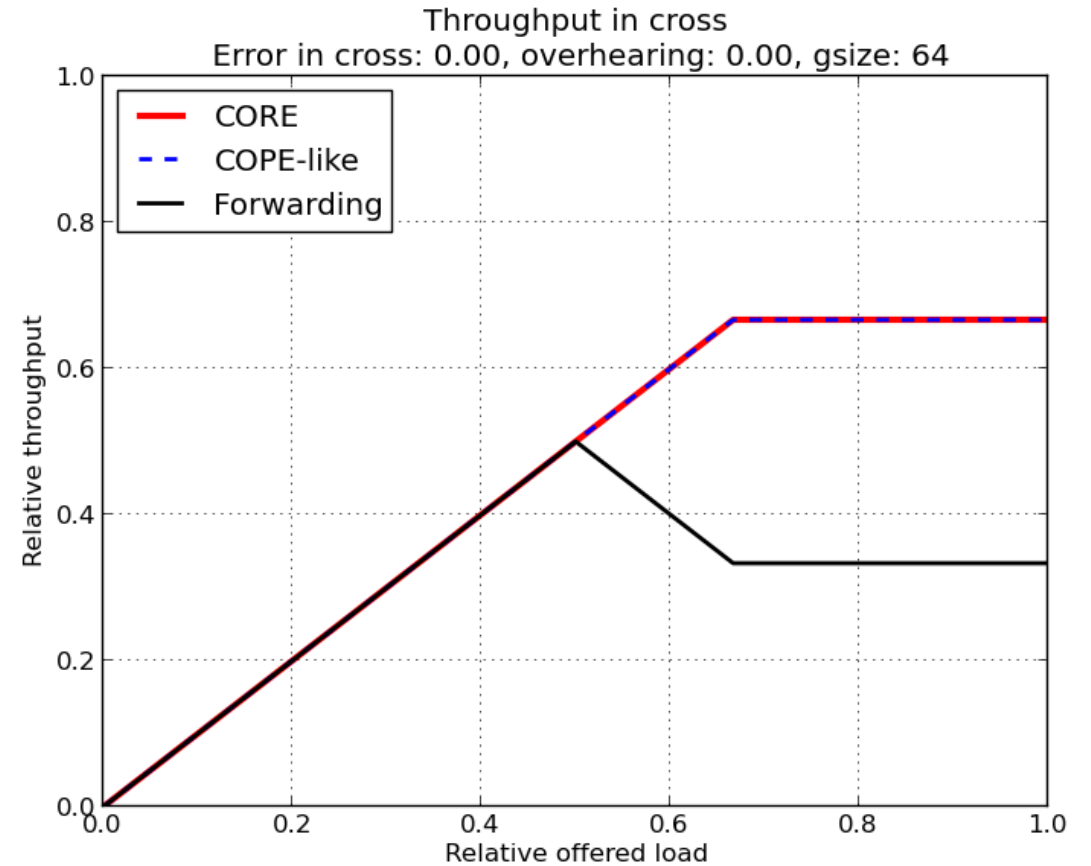
	P ₁	P ₂	P ₃		P ₁	P ₂	P ₃
1							
2							
3							
4							
5							
6							

	P ₁	P ₂	P ₃		P ₁	P ₂	P ₃
1							
2							
3							
4							
5							
6							

CORE Results

Throughput vs Offered Load

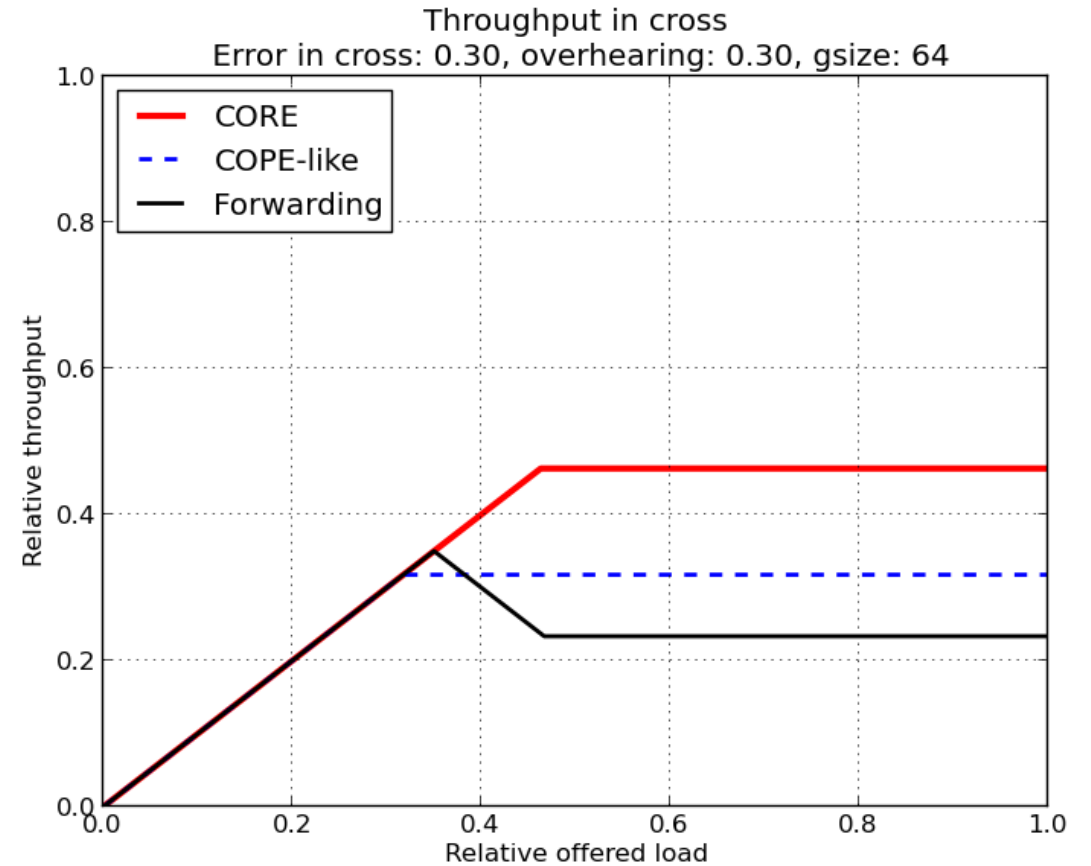
- Symmetric losses
- Loss probability = 0



CORE Results

Throughput vs Offered Load

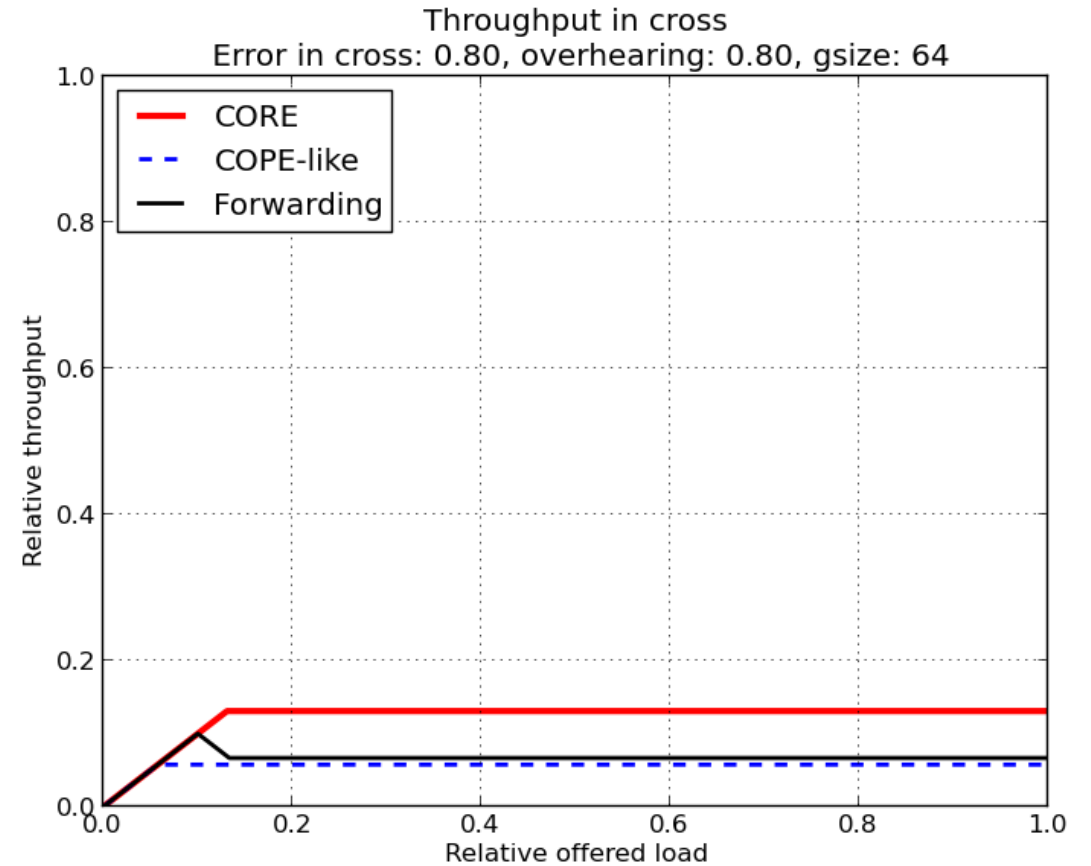
- Symmetric losses
- Loss probability = 0.3



CORE Results

Throughput vs Offered Load

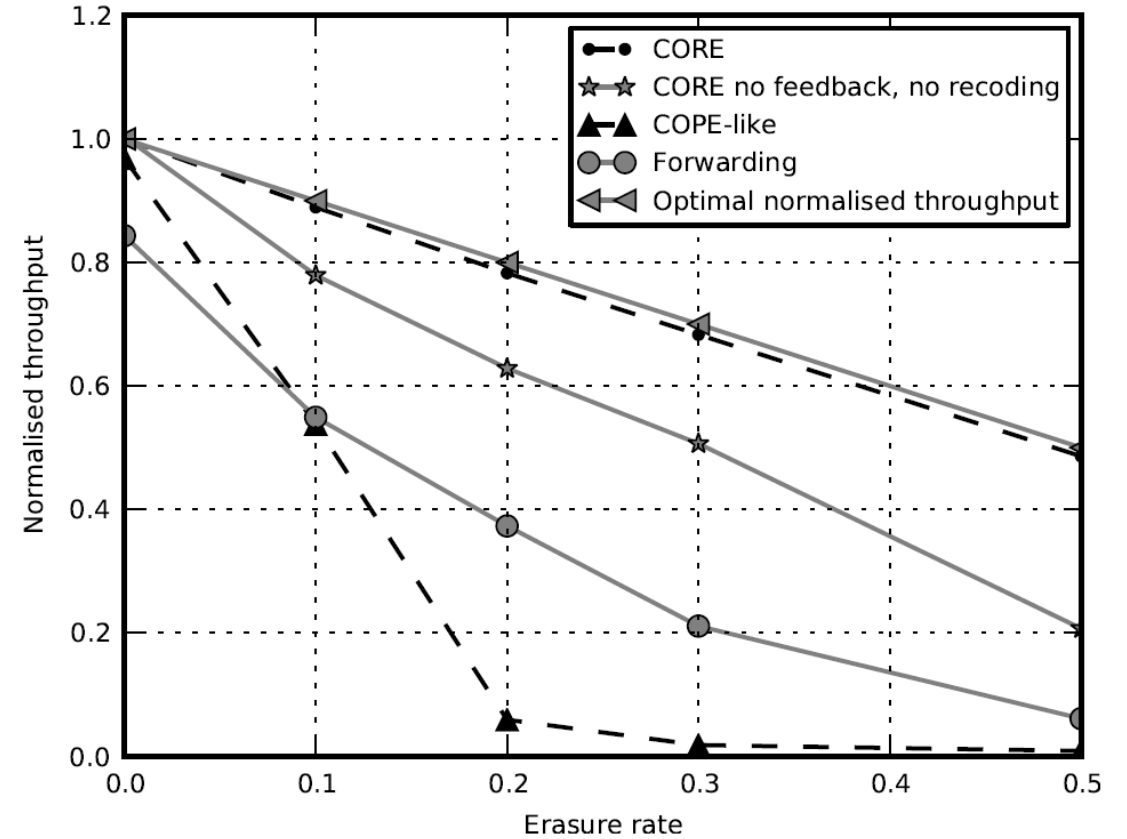
- Symmetric losses
- Loss probability = 0.8



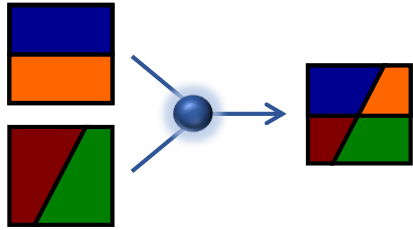
CORE Results

Throughput vs Losses

- Overall performance



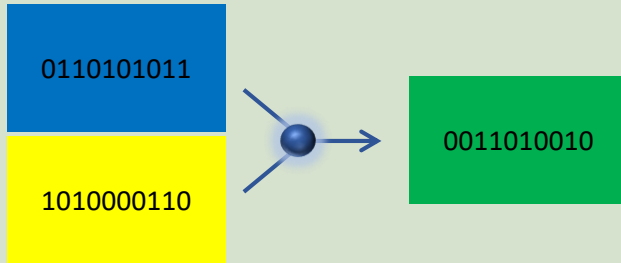
Different Forms of Network Coding



Static Networks
(planning possible)

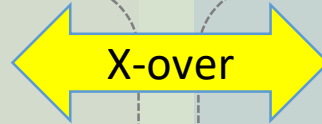
Dynamic Networks
(no planning possible)

Digital domain

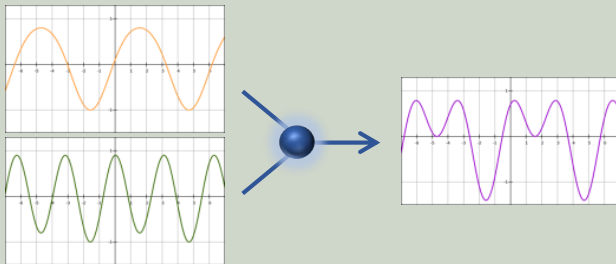


CRA I:
Digital Inter Session NC
e.g.: Internet of Things (IoT) protocols, ...

CRA III:
Digital Intra Session NC
e.g.: distributed storage complexity, heterogeneity, delay, ...

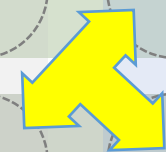


Analog domain



CRA II:
Analog Inter Session NC
e.g.: wireless mesh networks multiple signals, real world, ...

CRA IV:
Analog Intra Session NC
e.g.: cooperative beamforming multiple signals, real world, ...



Analog Intra Flow

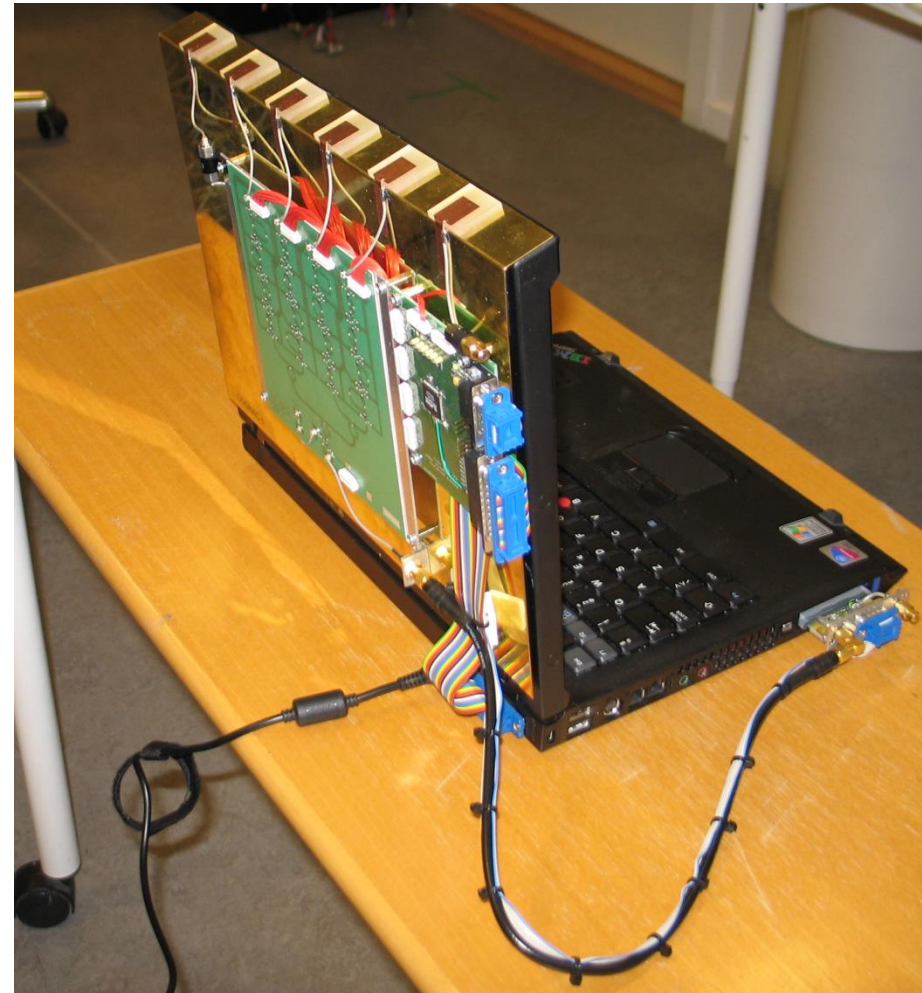
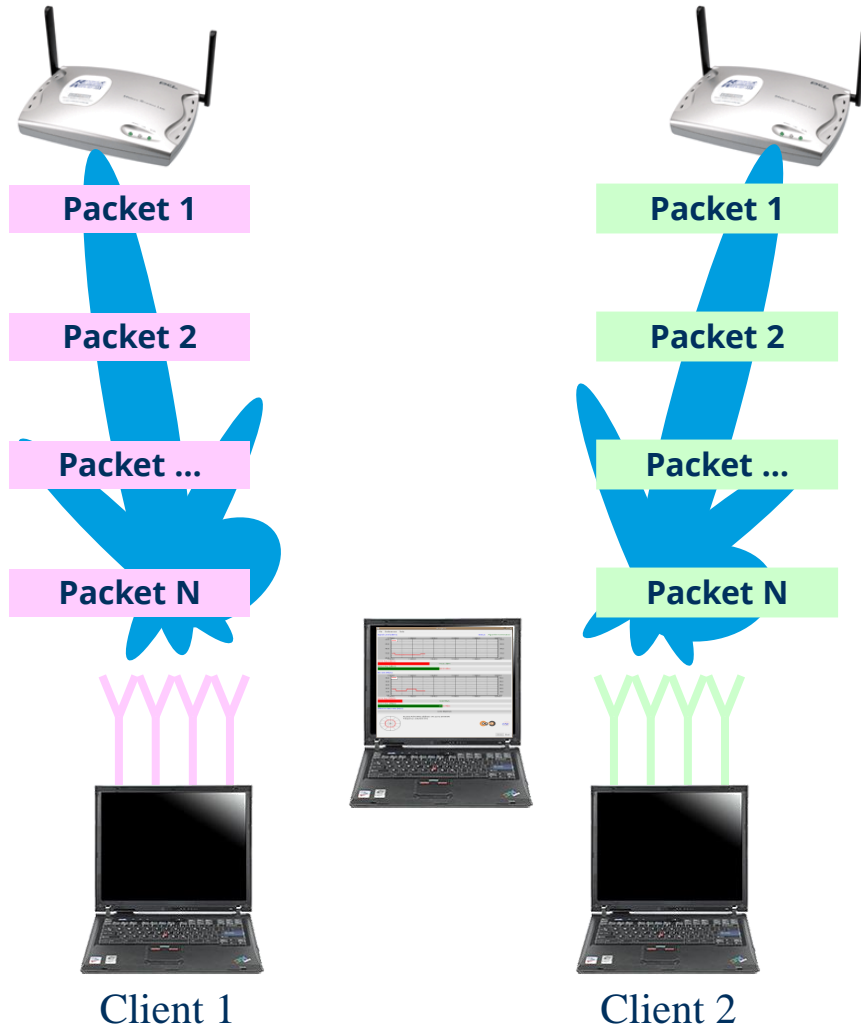
The missing link

Analog Intra Flow

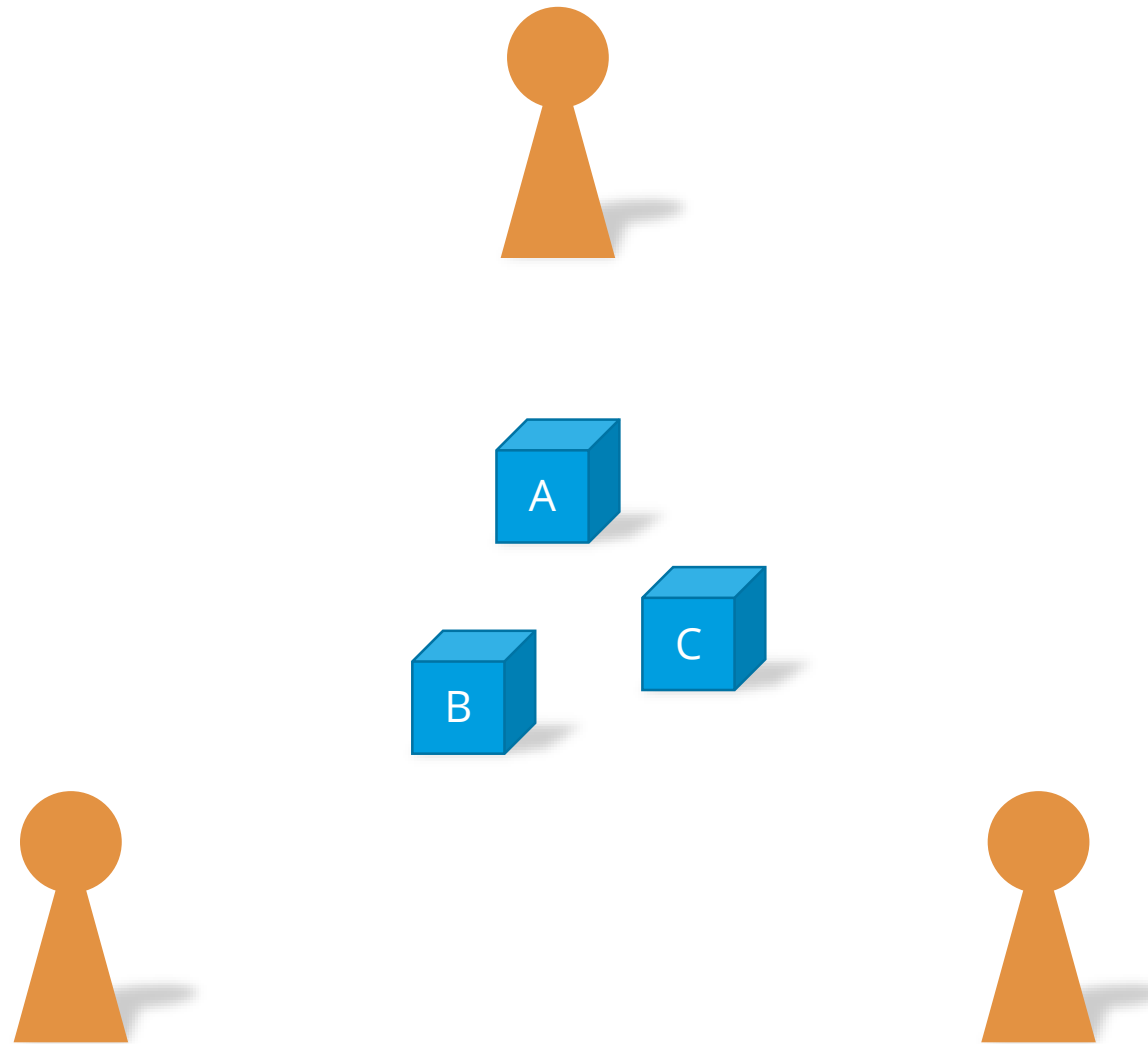
- Main idea to use mobile device beamforming to create random superposition of signals in the analog domain and treat this as coded information.
- Inspired by former work on beamforming for COTS devices.

<p>(12) United States Patent Fitzek et al.</p>	<p>(10) Patent No.: US 9,077,398 B2 (45) Date of Patent: Jul. 7, 2015</p>
<p>(54) NETWORK CODING BY BEAM FORMING</p> <p>(75) Inventors: Frank Fitzek, Aalborg (DK); Janus Heide, Aalborg (DK); Morten Pedersen, Aalborg (DK)</p> <p>(73) Assignee: NOKIA TECHNOLOGIES OY, Espoo (FI)</p> <p>(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.</p> <p>(21) Appl. No.: 13/984,892</p> <p>(22) PCT Filed: Feb. 22, 2011</p> <p>(86) PCT No.: PCT/IB2011/000362 § 371 (c)(1), (2), (4) Date: Aug. 12, 2013</p> <p>(87) PCT Pub. No.: WO2012/114141 PCT Pub. Date: Aug. 30, 2012</p> <p>(65) Prior Publication Data US 2013/0315338 A1 Nov. 28, 2013</p> <p>(51) Int. Cl. H04B 7/04 (2006.01) H04B 7/06 (2006.01) H04L 1/06 (2006.01) H04B 7/02 (2006.01) H04L 1/00 (2006.01)</p> <p>(52) U.S. Cl. CPC H04B 7/0408 (2013.01); H04B 7/0617</p>	<p>(56) References Cited U.S. PATENT DOCUMENTS</p> <p>2007/0149117 A1 6/2007 Hwang et al. 2007/0155336 A1 7/2007 Nam et al.</p> <p>(Continued)</p> <p>FOREIGN PATENT DOCUMENTS</p> <p>WO 2009026695 A1 3/2009 WO 2010019340 A1 2/2010 WO 2011035797 A1 3/2011</p> <p>OTHER PUBLICATIONS</p> <p>International Search Report and Written Opinion received for corresponding Patent Cooperation Treaty Application No. PCT/IB2011/000362, dated Dec. 1, 2011, 13 pages.</p> <p><i>Primary Examiner</i> — Leon Flores (74) <i>Attorney, Agent, or Firm</i> — Squire Patton Boggs (US) LLP</p> <p>(57) ABSTRACT Network coding by beam forming in networks, for example, in single frequency networks, can provide aid in increasing spectral efficiency. When network coding by beam forming and user cooperation are combined, spectral efficiency gains may be achieved. According to certain embodiments, a method includes operating a user equipment of a plurality of user equipment in a network comprising a plurality of access points. The method also includes the user equipment forming a beam. The method further receives processing received signals from at least one of the plurality of access points at the user equipment. The forming the beam is configured to let different user equipment of the plurality of user equipment to</p>

Cooperative Beamforming



Main Idea of Analog Network Coding with Beamforming



Finite Fields

Why finite Fields?

Linear Network Coding

Coded packets: linear combinations of original packets

Source

- Injects packets into the network (coded or uncoded)

Intermediate nodes

- Recombine with packets in their buffer

Receivers

- Decode if enough linear comb. are received

What are finite fields? Why do we need them?

Finite Fields - do we need them?

Data representation: linear combination of multiple packets could result in large payloads

Goal:

- Linear combinations of packets should result in packets

D	A	T	A	1
x	x	x	x	x

C ₁	C ₁	C ₁	C ₁	C ₁
+	+	+	+	+

D	A	T	A	2
x	x	x	x	x

C ₂	C ₂	C ₂	C ₂	C ₂
----------------	----------------	----------------	----------------	----------------

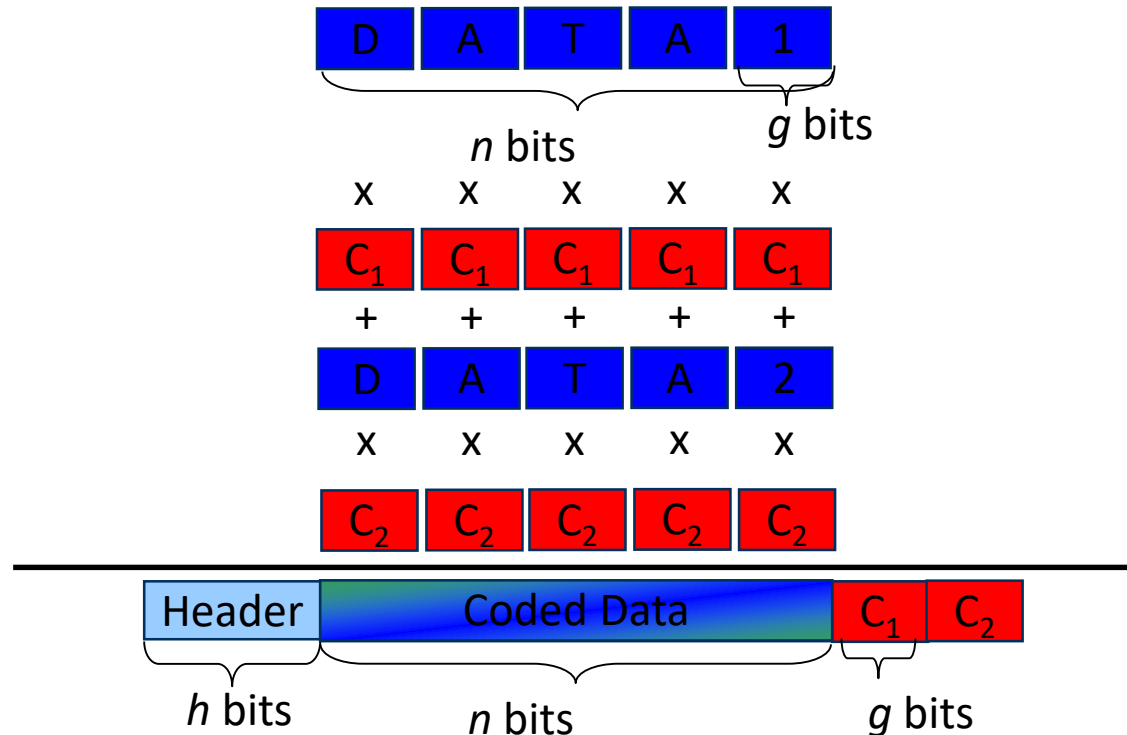
F	Z	H	A	K
---	---	---	---	---

Generating a Coded Packet

- Generating a linear network coded packet (CP)

$$CP_j = \sum_i C_i P_i$$

- Operations over finite field of size. e.g. $g = 8$ bits, $q = 256$

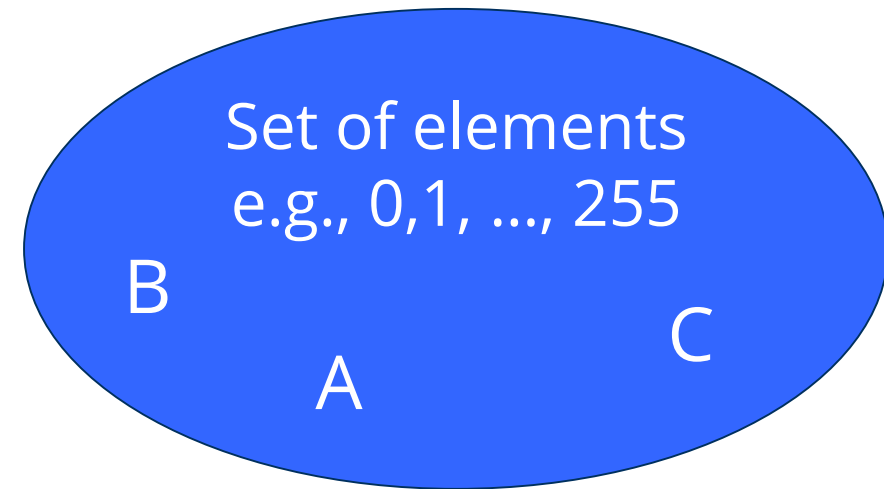


Finite Fields - What are they?

Operations:

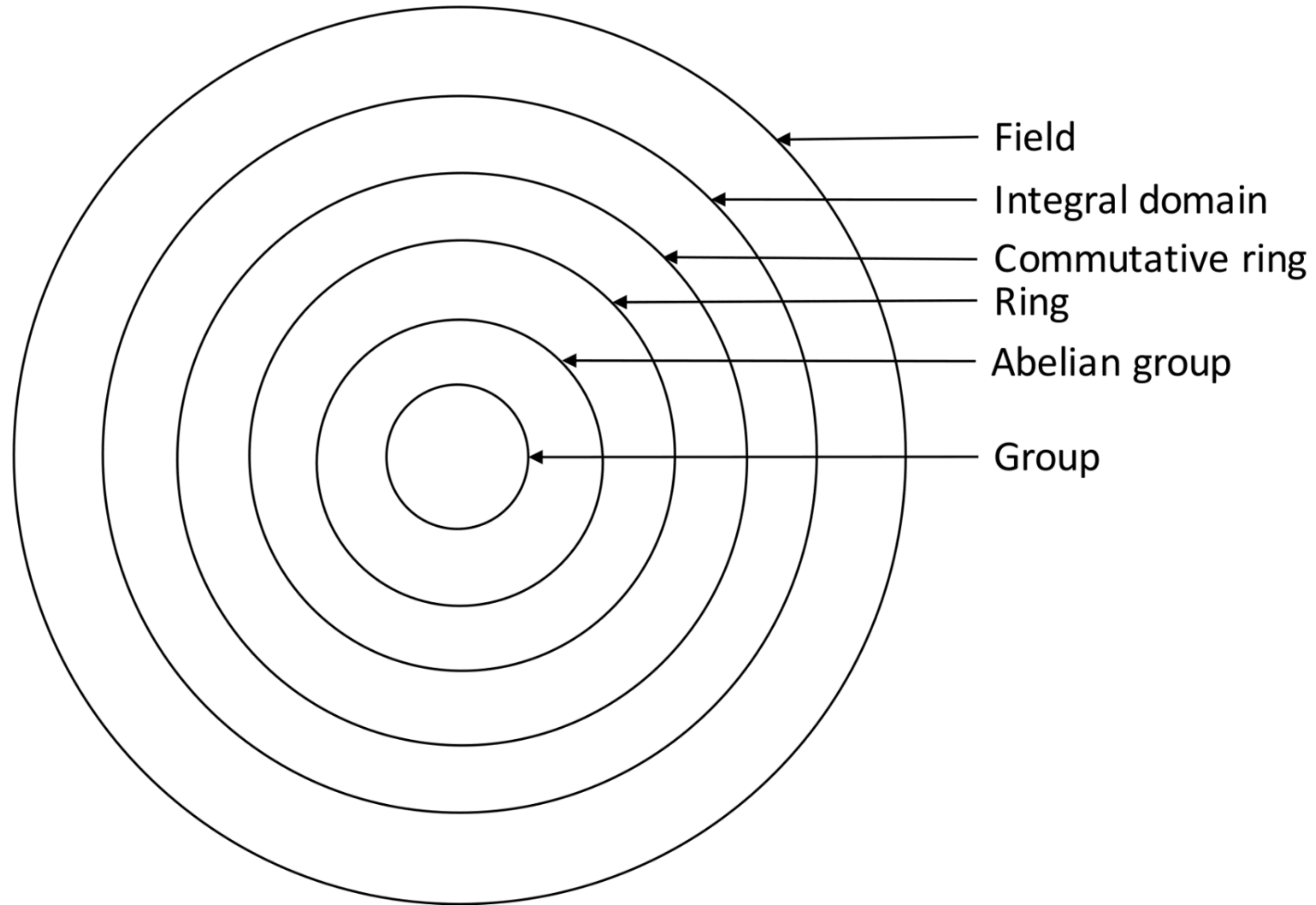
Addition, Multiplication

Property: closure



$+$, \times

Groups, Rings, Fields



Groups, Rings, Fields

Groups

- A group G , denoted $\{G, * \}$ is a set of elements with a binary operation $*$ that associates each ordered pair (a,b) of elements in G to an element $(a*b)$ in G following

Axioms

- *Closure*: If a and b in G , then $a * b$ is also in G
- *Associative*: $a * (b * c) = (a * b) * c$ for all a,b,c in G
- *Identity*: Exists e in G , s.t. $a*e = e*a = a$ for all a in G
- *Inverse*: for each a in G , exists a' in G , s.t. $a*a' = a'*a = e$

Finite group: finite number of elements

Groups, Rings, Fields

Abelian Group

- Group that satisfies
- *Commutative*: If a and b in G , then $a * b = b * a$

Rings

- A ring R , denoted by $\{R, +, \cdot\}$ is a set of elements with two binary operations: addition, multiplication. For all a, b, c in R the following axioms are satisfied
- R is **abelian group** with respect to the addition
- *Closure* under multiplication: ab in R
- *Associativity* of multiplication: $a(bc) = (ab)c$
- *Distributive laws*: $a(b+c) = ab + ac$
 $(a + b)c = ac + bc$

Groups, Rings, Fields

Commutative Ring

- A **ring** that also satisfies
- *Commutativity of multiplication*: $ab = ba$ in R

Integral Domain

- R is a commutative **ring** that also satisfies
- *Multiplicative identity*: exists $\mathbf{1}$, s.t. $a\mathbf{1} = \mathbf{1}a = a$
- *No zero divisors*: $ab = 0$, implies either $a = 0$ or $b = 0$

Field

- F is a field, $\{ F, +, \times \}$ that satisfies
- F is an **integral domain**
- *Multiplicative inverse*: for each a in F , except 0 , exists an element a^{-1} , s.t. $a(a^{-1}) = (a^{-1})a = 1$

Finite Fields GF(p)

Can write fields of the form GF(p), where p is prime

Addition and multiplication over GF(p) are mod p

Focus on $p = 2$

Example:

GF(2) addition: equivalent to XOR

multiplication: equivalent to AND

How to divide? Multiply by multiplicative inverse

Finding the multiplicative inverse

1.- Can look for a^{-1} such that $(a^{-1} \cdot a) \equiv 1$

2.- Can use the extended Euclidean algorithm

Finite Fields - Applying GF(2) to NC

Example:

GF(2) addition: XOR

multiplication: AND

Given 2 data packets

P1: 01011001 P2: 10001001

calculate the content of the coded packet P1+P2.

$$\begin{array}{r} P1 + P2 = \\ 01011001 \text{ (XOR bit by bit)} \\ 10001001 \\ \hline 11010000 \end{array}$$

What are the coefficients?

Modular Arithmetic

Modulus

- If a is an integer, $n > 0$ integer, we define $a \bmod n$ to be the remainder when a is divided by n
- The integer n is called the modulus
- For any integer a , we can write
$$\mathbf{a = qn + r, \text{ with } 0 \leq r < n, \text{ and } q = \lfloor a/n \rfloor}$$
- E.g., $11 \bmod 7 = 4$, $-11 \bmod 7 = 3$

Congruent modulo n

- If $(a \bmod n) = (b \bmod n)$, and it's expressed
$$\mathbf{a \equiv b \pmod{n}}$$
- E.g., $20 \equiv 6 \pmod{7}$

Modular Arithmetic

Properties of congruencies

- $a \equiv b \pmod{n}$ if $n \mid (a-b)$
- $a \equiv b \pmod{n}$ implies $b \equiv a \pmod{n}$
- If $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$, then $a \equiv c \pmod{n}$

Modular arithmetic operations

$$[(a \bmod n) + (b \bmod n)] \bmod n = (a+b) \bmod n$$

$$[(a \bmod n) - (b \bmod n)] \bmod n = (a-b) \bmod n$$

$$[(a \bmod n) \times (b \bmod n)] \bmod n = (a \times b) \bmod n$$

Rules of ordinary arithmetic involving addition, subtraction, multiplication carry over

Modular Arithmetic

Properties of modular arithmetic

- Define $Z_n = \{ 0, 1, \dots, n-1 \}$ as the set of **residues** or **residue classes** mod n .
- Each element of Z_n is a residue class and can define it as
$$[j] = \{a : a \text{ is integer, } a \equiv j \pmod{n}\}$$
- Reducing k mod n : finding smallest non-negative integer a , such that $k \equiv a \pmod{n}$

Modular Arithmetic

Properties of modular arithmetic

- If $(a + b) \equiv (a + c) \pmod{n}$, then $b \equiv c \pmod{n}$
- If $(a \times b) \equiv (a \times c) \pmod{n}$, then $b \equiv c \pmod{n}$ if a is relatively prime to n , i.e., $\gcd(a,n) = 1$

Why is the last property important?

What happens when “p” is not a prime?

Will modular arithmetic still work?

Example 1:

If $\gcd(a,n) \neq 1$, the last equation does not hold

e.g. $6 \times 3 = 18 = 2 \pmod{8}$

and $6 \times 7 = 42 = 2 \pmod{8}$ but

$3 \pmod{8} \neq 7 \pmod{8}$

What happens when “p” is not a prime?

Will modular arithmetic still work?

+	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

x	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	0	2
3	0	3	2	1

What about $\text{GF}(2^n)$?

Since 2^n is not a prime, operations are defined in a different way

— polynomial arithmetic

Ordinary polynomial arithmetic:

— A polynomial of degree n

— $F(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_0 x^0 = \sum a_i x^i$

— a_i are the coefficients, chosen from a set

Operations:

Addition $f(x) + g(x) = \sum (a_i + b_i) x^i$

Multiplication $f(x) \times g(x) = \sum C_i x^i$

with $c_k = a_0 b_k + a_1 b_{k-1} + \dots + a_k b_0$

What about $\text{GF}(2^n)$?

Polynomial arithmetic in $\text{GF}(2^n)$:

Arithmetic follows rules of polynomial arithmetic

Arithmetic of coefficients is performed modulo 2

- i.e., using $\text{GF}(2)$ addition/multiplication for coefficients of the same order
- e.g., $(a_i + b_i) \bmod 2$

If multiplication results in a polynomial greater than $n-1$, then the polynomial is reduced modulo an irreducible polynomial $p(x)$

- Think of it as a $\bmod p(x)$ operation: divide by $p(x)$, keep the remainder

Example GF(2²)

How about 2 + 3?

$2 = (10)_b$ and $3 = (11)_b$

Thus, 2 + 3 becomes

$$x + (x + 1) = 1$$

+	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

Irreducible polynomial

$$p(x) = x^2 + x + 1$$

(111)_b

Example GF(2²)

How about 2 x 3?

$$x(x+1) = (x^2 + x) \bmod p(x)$$

$$\begin{array}{r} x^2 + x + 1 \quad \Big| \quad \begin{array}{r} 1 \\ \hline x^2 + x \\ x^2 + x + 1 \\ \hline 1 \end{array} \end{array}$$

x	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

Irreducible polynomial

$$p(x) = x^2 + x + 1$$

(111)_b

How to implement multiplication $GF(2^n)$?

A.- Product (shifts+ XORs)

- 1) Pick one as multiplier (M) and another as multiplicand (m)
- 2) For each "1" in "M", left shift the "m" by the position of the "1"
- 3) XOR shifted versions

B.- Modulo irreducible polynomial (long division)

- 1) Initialize: $F(0)$ = Result of part A
- 2) Take irreducible polynomial $p(x)$ left shift until first "1" of polynomial and of value F match
- 3) $F(i+1) = (\text{shifted } p(x)) \text{ XOR } (F(i))$
- 4) Stop if first "1" of $F(i+1)$ occurs in the $(n-1)$ -th bit

How to implement multiplication $GF(2^n)$?

$$M = (00010001)_b \text{ and } m = (10100111)_b$$

$$\begin{array}{r}
 101001110000 \quad (m \text{ shifted 4 times}) \\
 \text{(XOR)} \quad 10100111 \quad (m \text{ shifted 0 times}) \\
 \hline
 101011010111
 \end{array}$$

B.- Modulo irreducible polynomial

$$p(x) = x^8 + x^4 + x^3 + x + 1 \quad (100011011)_b$$

$$\begin{array}{r}
 101011010111 \quad \text{mod } p(x) \\
 \text{(XOR)} \quad \mathbf{100011011000} \\
 \hline
 001000001111 \\
 \text{(XOR)} \quad \mathbf{001000110110} \\
 \hline
 000000\mathbf{111001} \quad \rightarrow \mathbf{(00111001)_b}
 \end{array}$$

Take Away Points

Use of finite fields necessary to maintain packet size

Addition is simple operation

Multiplication operation becomes more complex with larger field size

Why use (or not) look up tables?

- For small field size, it is efficient and requires small storage capability
- For larger field size, large look-up tables

fifi-python

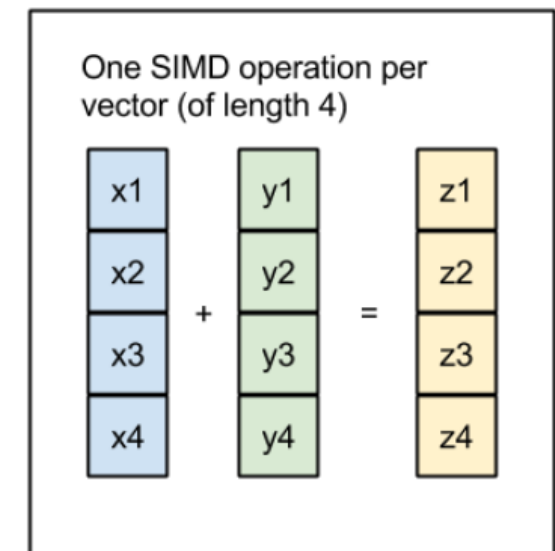
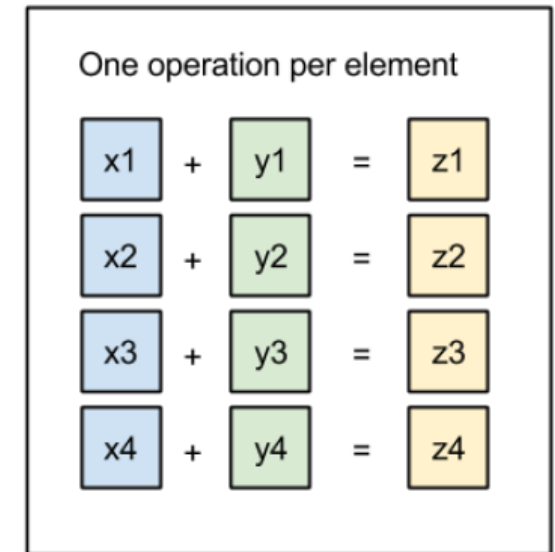
Input by morten v. pedersen

Finite field implementation

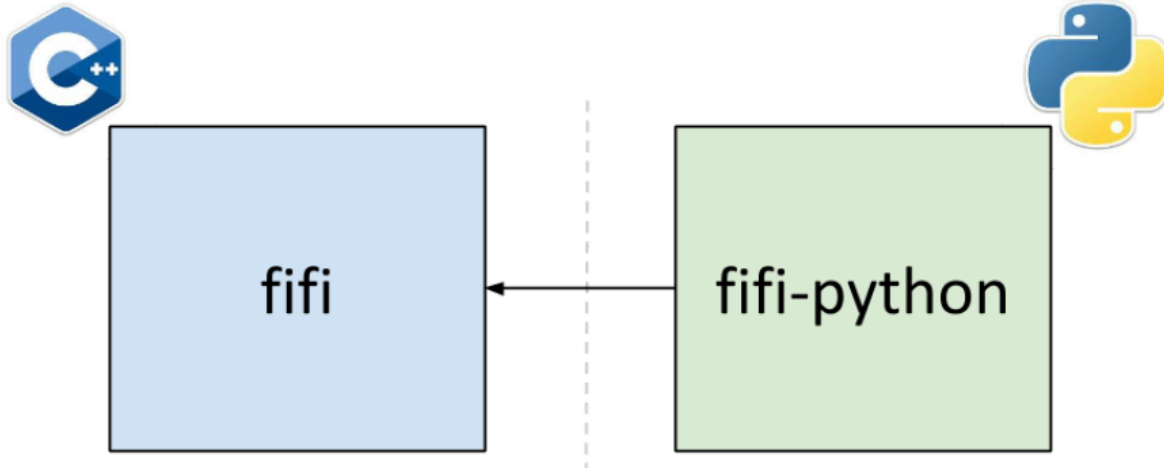
- Finite fields are a key component in many important applications:
 - Cryptography.
 - Error/Erasure correcting codes.
 - Error detection e.g. CRC (Cyclic Redundancy Checks).
- Having an efficient implementation is important to evaluate system critical parameters:
 - Throughput, latency, energy consumption, memory consumption etc.
- Today we will use the `fifi-python` library to perform finite field computations.

A note on performance

- In the past few year performance has increase significantly
- This has happened because we understand how to utilize specialized instructions on modern CPUs called SIMD (Single Instruction Multiple Data).
- fifi and fifi-python uses this extensively (on both x86 and ARM CPUs)



About fifi and fifi-python



```
fifi-python/  
├── config.py  
├── examples  
│   ├── fifi_simple_api.py  
│   ├── fifi_simple_api_table_example.py  
│   ├── hello_fifi.py  
│   └── hello_simple_api.py  
├── LICENSE.rst  
├── NEWS.rst  
├── README.rst  
├── src  
│   └── fifi_python  
├── test  
│   ├── data.json  
│   └── fifi_python_tests.py  
├── waf  
└── wscript
```

Exploring the API

- To use the library `import` it into your scripts.

```
In []: import fifi
```

- You can explore the API using the built-in `help()` function

```
In [2]: help(fifi)
```

Help on module fifi:

NAME

fifi

FILE

/home/mvp/Dropbox/work_code/notebooks/slides_fifi_python/fifi.so

CLASSES

Boost.Python.instance(__builtin__.object)

extended_log_table_binary16

extended_log_table_binary4

extended_log_table_binary8

full_table_binary4

full_table_binary8

log_table_binary16

log_table_binary4

log_table_binary8

optimal_prime_prime2325

simple_online_binary

simple_online_binary16

simple_online_binary4

simple_online_binary8

```
class extended_log_table_binary16(Boost.Python.instance)
```

```
| A finite field implementation
```

```
|
```

Using help()

- You can also get more specific help, by passing a specific class or function.

```
In [4]: import fifi
        help(fifi.simple_online_binary4.add)
```

Help on method add:

```
add(...) unbound fifi.simple_online_binary4 method
        Returns the sum of two field elements.
```

```
        :param a: The augend.
```

```
        :param b: The addend.
```

```
        :returns: The sum of a and b.
```

A small example

```
In [8]: import fifi

field = fifi.simple_online_binary4()
order = 2**4

table = ''

for i in range(order):
    for j in range(order):
        table += '{:02d} '.format(field.multiply(i,j))
    table += '\n'

print(table)
```

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
00 02 04 06 08 10 12 14 03 01 07 05 11 09 15 13
00 03 06 05 12 15 10 09 11 08 13 14 07 04 01 02
00 04 08 12 03 07 11 15 06 02 14 10 05 01 13 09
00 05 10 15 07 02 13 08 14 11 04 01 09 12 03 06
00 06 12 10 11 13 07 01 05 03 09 15 14 08 02 04
00 07 14 09 15 08 01 06 13 10 03 04 02 05 12 11
00 08 03 11 06 14 05 13 12 04 15 07 10 02 09 01
00 09 01 08 02 11 03 10 04 13 05 12 06 15 07 14
00 10 07 13 14 04 09 03 15 05 08 02 01 11 06 12
00 11 05 14 10 01 15 04 07 12 02 09 13 06 08 03
00 12 11 07 05 09 14 02 10 06 01 13 15 03 04 08
00 13 09 04 01 12 08 05 02 15 11 06 03 14 10 07
00 14 15 01 13 03 02 12 09 07 06 08 04 10 11 05
00 15 13 02 09 06 04 11 01 14 12 03 08 07 05 10
```

Hands-On

Make sure `fifi-python` is installed (lecture slides)

Create a script (e.g. `print_table.py`) and write the code to print both the multiplication and division tables.

Run the script `python print_table.py`.

Using the Simple API

- The previous examples use the field object directly.
- It is also possible to standard arithmetic operations (+, -, *, /, ~) with field elements.
- To use it copy the `fifi-python/examples/fifi_simple_api.py` to your scripts folder.

```
In [3]: #!/usr/bin/env python

# Copyright Steinwurf ApS 2011-2013.
# Distributed under the "STEINWURF RESEARCH LICENSE 1.0".
# See accompanying file LICENSE.rst or
# http://www.steinwurf.com/licensing

from fifi_simple_api import B4

def main():
    a = B4(13)
    b = B4(7)

    print("{a} + {b} = {result}".format(a=a, b=b, result=a + b))
    print("{a} - {b} = {result}".format(a=a, b=b, result=a - b))
    print("{a} * {b} = {result}".format(a=a, b=b, result=a * b))
    print("{a} / {b} = {result}".format(a=a, b=b, result=a / b))
    print("~{a} = {result}".format(a=a, result=~a))

if __name__ == '__main__':
    main()

13 + 7 = 10
13 - 7 = 6
13 * 7 = 5
13 / 7 = 8
~13 = 4
```