UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

COMMUNICATION WITH RAPTORQ ERASURE CODES IN MALICIOUS ENVIRONMENTS

José Manuel Sá Lopes

DISSERTAÇÃO
MESTRADO EM SEGURANÇA INFORMÁTICA

2013

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

COMMUNICATION WITH RAPTORQ ERASURE CODES IN MALICIOUS ENVIRONMENTS

José Manuel Sá Lopes

DISSERTAÇÃO

MESTRADO EM SEGURANÇA INFORMÁTICA

Dissertação orientada pelo Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves

2013

Acknowledgments

To my advisor, Professor Nuno Neves, for his comments and suggestions. This research would never have been possible without his invaluable support, guidance and motivation. It is an honor, and also a responsibility, to work closely with him.

To all my colleagues in lab 25, and in the LaSIGE research group. A special thanks to my friends and colleagues João Alpuim, Guilherme Rodrigues, Fábio Botelho and Emanuel Alves, for all the work we have done together in the past five years. To João for carrying me on his shoulders when I needed, and to Fábio for being my brother at war.

To all my close friends, for being always there for me, offering comprehension and motivation whenever needed.

To my parents, grandfather, and all my family for their constant and unconditional support. A special thanks to my life companion Filipa, for her patience, compassion, enthusiasm and, most of all, for believing in me when even I did not.

I would also like to thank the support by the EC through project FP7-257475 (MASSIF), and by the FCT through the Multiannual program (LaSIGE) and project PTDC/EIA-EIA/113729/2009 (SITAN).

"If I have seen further it is by standing on ye sholders of Giants." — Isaac Newton

Resumo

A teoria de códigos é o estudo das propriedades dos códigos e sua adequação para uma aplicação específica. Um dos usos dos códigos é a correcção de erros. A técnica de *Forward Error Correction* (FEC) é utilizada para recuperar de erros na transmissão de dados, quando canais de comunicação não fiáveis são utilizados. A ideia central de FEC é o transmissor codificar a sua mensagem de uma maneira redundante, utilizando um código *error-correcting code* (ECC), conhecido por código auto-corrector. Como exemplo, temos os códigos de Hamming, desenvolvidos por Richard Hamming na década de 40.

A redundância permite (idealmente) ao receptor detectar erros que ocorreram durante a transmissão e corrigi-los. Assim, o receptor pode corrigir os erros sem a necessidade de retransmissões (com um custo adicional de largura de banda). Deste modo, a técnica de FEC é normalmente utilizada em cenários onde as retransmissões não são admissíveis em termos de custos ou mesmo impossíveis, como em ligações unidireccionais ou quando se transmite para vários receptores em multicast. O FEC também é utilizado em sistemas de armazenamento, para recuperar informação corrompida.

Os *fountain codes* representam uma classe de códigos com a propriedade de produzir uma sequência potencialmente infinita de símbolos codificados, a partir dos símbolos originais (i.e., os dados a serem transmitidos). Para explicar os *fountain codes* é normalmente feita uma analogia com uma fonte de água: qualquer pessoa pode encher um copo na fonte, não importa quais as gotas de água que enchem o copo, apenas quantas gotas estão no copo, porque no final o resultado é o mesmo – um copo cheio de água. Analogamente, o mesmo se passa numa tranmissão que use *fountain codes*: não importa o conjunto de símbolos codificados que são recebidos, apenas a quantidade de símbolos recebidos: após a descodificação, o resultado são os símbolos originais.

Um *fountain code* é *ideal* se os K símbolos originais podem ser recuperados a partir de quaisquer K símbolos codificados. Geralmente, na prática, os *fountain codes* são conhecidos por terem algoritmos de codificação e descodificação muito eficientes, e por conseguirem recuperar os K símbolos originais a partir de qualquer conjunto de K' símbolos codificados com alta probabilidade (com K' apenas ligeiramente superior a K).

Estes códigos foram idealizados como a codificação ideal para transferir ficheiros (especialmente ficheiros grandes) para mais do que um receptor, provando ser uma maneira muito mais escalável do que por exemplo usando TCP. Os *LT codes* representam a primeira realização practicamente viável de *fountain codes*. Subsequentemente, os *Raptor codes* foram desenvolvidos, baseados em parte nos *LT codes*, para melhorar (diminuir) a complexidade computacional e a probabilidade de falha. Para tal, aplicam um "pré-código" aos símbolos originais antes de codificá-los.

Os *Raptor codes* já foram usados em vários standards, nomeadamente de *streaming* de vídeo em redes broadcast, e também são utilizados em sistemas militares e de comunicação de emergência após desastres. O primeiro *Raptor code* a ser adoptado em vários standards, foi o R10 [1]. Entretanto, na vanguarda dos *Raptor codes* está o standard *RaptorQ* [2].

Dada a natureza crítica dos sistemas onde estes códigos são utilizados, nós achámos que seria relevante estudar a sua resiliência perante faltas maliciosas. Estes códigos foram conceptualizados para corrigirem faltas acidentais, e fazem-no incrivelmente bem: os *RaptorQ*, por exemplo, têm uma probabilidade de falha (i.e., não conseguirem recuperar os símbolos originais após a operação de descodificação) na ordem dos 10^{-5} para um *overhead* de apenas 1 símbolo (i.e., $K' = K + 1$).

Nesta dissertação nós relatamos a nossa investigação sobre a robustez do código *RaptorQ* perante faltas maliciosas injectadas por um atacante com controlo da rede (i.e., que pode eliminar pacotes, por exemplo através de um router infectado). Para além disso descrevemos, tanto quanto sabemos, a primeira concretização do RaptorQ, além da empresa[1] que os desenvolveu originalmente. Tencionamos transformar a nossa implementação num projecto de código aberto.

Começamos por contextualizar os cenários onde a utilização de *fountain codes* é relevante, e por vezes quase que necessária. A seguir abordamos a evolução dos *fountain codes*, culminando numa descrição mais detalhada do código *RaptorQ*.

Prosseguimos para a nossa implementação de uma biblioteca completamente compatível com o standard do IETF RFC 6330 (onde o *RaptorQ* está especificado). Testámos a sua resiliência, primeiro contra faltas acidentais, para verificar que os valores da probabilidade de falha obtidos na prática, estavam congruentes com os valores disponíveis na literatura.

De seguida, estabelecemos um ataque de prova de conceito que permite que, escolhendo os pacotes que passam, mas perdendo relativamente muitos pacotes, consigamos forçar 100% de probabilidade da descodificação falhar. Entretanto, visto ser necessário perder um grande número de pacotes, o ataque pode ser facilmente detectado, pois para a maioria dos valores de K testados seria quase um ataque de *Denial-of-Service* (DoS).

Com base no raciocínio do nosso ataque inicial, nós aperfeiçoamos o ataque, reduzindo o número de pacotes perdidos para vários valores de K para apenas entre 1% e 2% dos pacotes a transmitir. Estes valores tornam o ataque muito viável, pois dificultam

[1]*Digital Fountain*, que foi adquirida pela *Qualcomm Incorporated* em Fevereiro de 2009.

muito a sua detecção. Também discutimos como este ataque poderia ser efectuado quando a comunicação é feita através de um canal seguro, onde as mensagens são cifradas. Isto é possível visto o ataque ser directamente ao desenho do standard, e independente do conteúdo das mensagens.

Por fim, discutimos as implicações prácticas deste ataque, e propomos algumas possíveis soluções, que dificultariam o ataque, tornando-o inexiquível na práctica. Estas soluções podem ser facilmente adaptadas às implementações existentes e ao próprio standard.

As contribuições principais do nosso trabalho, podem ser resumidas em:

1. Uma implementação do standard do IETF RFC 6330, que especifica o código *RaptorQ*, e uma avaliação dos valores de probabilidade de falha do código *RaptorQ*, comparando os nossos resultados com os disponíveis na literatura;

2. Uma prova de conceito de que o código *RaptorQ* pode ser quebrado se as faltas forem arbitrariamente maliciosas, e um algoritmo que permite refinar esse ataque, reduzindo ao mínimo o número de pacotes que têm de ser eliminados;

3. Algumas ideias e tácticas para ajudar a execução do ataque quando canais cifrados são utilizados;

4. Um conjunto de possíveis soluções que podem ser adaptadas ao standard e as implementações para tornar o ataque inexequível.

Do nosso trabalho, nomeadamente da nossa prova de conceito de que o código *RaptorQ* pode ser atacado, resultou uma publicação: J. Lopes and N. Neves, “Robustness of the RaptorQ FEC Code Under Malicious Attacks”, in *INForum*, Évora, September 2013. Entretanto, ainda há material para ser publicado, nomeadamente o nosso ataque aperfeiçoado e as soluções propostas, que pretendemos submeter para publicação a curto prazo.

Palavras-chave: Códigos de Erro, *Forward Error Correction*, *Fountain Codes*, Resiliência, *RaptorQ*.

Abstract

Forward Error Correction (FEC) is a technique used to recover from erasures that might occur during the transmission of packets. The central idea is for the sender to encode its data in a redundant way using an error-correcting code (ECC). Fountain codes is a class of ECC that allows a potentially limitless sequence of encoded packets to be created from the original data, allowing the recovery of arbitrary losses (with high probability) with small overheads.

The most recent fountain code to be standardized by the Internet Engineering Task Force (IETF) is called RaptorQ. It offers enviable decoding complexity and has an overhead-failure curve that puts it closest to the ideal fountain code. Given that RaptorQ was conceived with accidental faults in mind, we decided to investigate its robustness in a malicious environment. The motivation is that RaptorQ will be used not only for media delivery but also in critical systems, such as in military and defense scenarios, and as such it might become the target of an attack.

The thesis presents our implementation of RaptorQ, which we intend to make public in the near future (to our knowledge, the first for this code). It also evaluates the decoding failure probabilities of RaptorQ and compares them to the ones available in the literature. An attack to the RaptorQ standard was also investigated: first, as a proof of concept, resulting in an inelegant and easily detectable attack; then, it was refined, making the attack much more effective and harder to detect. Finally, we also discuss some possible solutions that could easily be adopted into the standard and its implementations, which would render our attack much harder to execute (or even unfeasible).

Keywords: Erasure Codes, Forward Error Correction, Fountain Codes, Resilience, RaptorQ.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter motivates the work of the thesis, and presents the main goals and most important achievements. In the end of the chapter, we analyze the planning presented on the preliminary report and the actual task accomplishment, and we also describe the organization of the rest of the document.

1.1 Motivation and Goals

In telecommunication, information theory, and coding theory, forward error correction (FEC) - or channel coding - is a technique used for recovering from errors in data transmission over unreliable or noisy communication channels. The central idea is that the sender encodes the message in a redundant way by applying an error-correcting code (ECC).

The redundancy allows the receiver to detect a limited number of errors that may occur anywhere in the message, and often to correct these errors without retransmission. FEC gives the receiver the ability to correct errors without needing a reverse channel to request the retransmission of data, but at the cost of a fixed, higher forward channel bandwidth. FEC is therefore applied in situations where retransmissions are costly or impossible, such as one-way communication links or when transmitting to multiple receivers in a multicast. FEC information is usually added to storage devices to enable recovery of corrupted (or lost) data.

Fountain codes are a class of erasure codes with an attractive property when employing forward error correction: the original source symbols (i.e., the data to be transmitted) can ideally be recovered with high probability from *any* subset of the encoding symbols of size equal to or only slightly larger than the number of source symbols. The most recent and efficient fountain codes are called Raptor codes, which were standardized under the names R10 [1] and RaptorQ [2].

Figure 1.1 shows a typical use case scenario for fountain codes. It corresponds to an application where a single sender transmits a file to multiple receivers. In such a scenario,

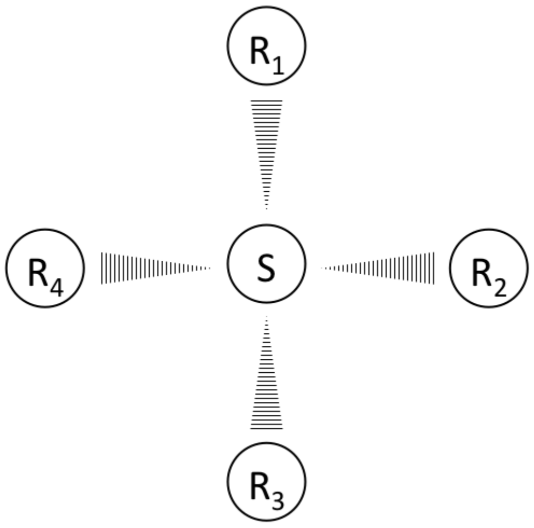


Figure 1.1: Point-to-multipoint transmission, a typical use case for fountain codes.

using TCP channels would not be a scalable solution because the sender needs to keep track of which packets were received at each receiver. Resorting to UDP would solve this problem, but would lack the reliability offered by TCP. If the sender was to "manually" do the necessary retransmissions, and determine which packets were delivered to each receiver, the complexity would be high and would create scalability issues. However, coding the file with a fountain code and transmitting over UDP solves the scalability issue and provides the necessary reliability: each receiver would be able to recover from the errors affecting its own channel, without the need for retransmissions.

RaptorQ is the most recent fountain code to be described. Its decoding properties have suggested that it could be deployed in mission critical applications. Its computational complexity has been evaluated on different platforms with all kinds of parameter settings.

The thesis describes an implementation of the RaptorQ standard [2], which we are in the process of making an open source project (to our knowledge, the first open project). The results from testing our implementation's probability of decoding failure confirm the robustness claimed by the literature on RaptorQ. Even for small amounts of extra redundant information (called overhead), it is possible to reach decoding failure probabilities in the order of $1{\times}10^{-7}$.

However, these codes were conceived with *benign* environments in mind. Given the critical nature of the many systems that employ these technologies, it is relevant to consider the impact that an adversary could have in their robustness by introducing malicious faults. Even though the probability for decoding failure is very low, it still exists. Therefore, an attacker could try to force these rare failure scenarios more often, for example, by selecting which packets reach the receiver and which packets are dropped by the network.

Our goal was also to investigate to what extent a malicious adversary could affect RaptorQ's resilience. In particular, we studied if it was possible to hinder the decoding process, thus preventing the recovery of the original message, and the cost of executing such attack (i.e., how viable can the attack be). Our results demonstrate that the RaptorQ standard can be successfully attacked with a reasonably small effort. Furthermore, we discuss one or more ways to try to prevent the attack, or at least make it more difficult to perform in practice.

1.2 Contributions and Publications

The main contributions of this thesis can be summarized as:

1. A fully-compliant implementation of IETF's RFC 6330 [2], which specifies the RaptorQ code. This implementation will be put on public domain over the next months. In addition, a study is presented that confirms the low failure probabilities previously claimed by other sources;

2. A proof of concept attack forcing a decoding failure probability of 100% is described, where an attacker intelligently selects certain packets to be eliminated in the network. Additionally, the rationale behind a brute force algorithm is explained, which refines the attack and makes it extremely hard to detect (just by looking at the average packet loss). A set of suggestions and techniques is also suggested to help executing this attack even when communication is made through a secure channel;

3. A set of solutions that could be easily adapted in implementations and the standards, which would greatly increase the difficulty of executing such an attack, or even render it impossible.

From the described work, namely from the proof of concept that the RaptorQ code can be attacked, resulted one paper: J. Lopes and N. Neves, "Robustness of the RaptorQ FEC Code Under Malicious Attacks", in *INForum*, Évora, September 2013. However, there is still research material that should be published, which we intend to do over the next months.

1.3 Planning

In this section we analyze the planning presented in the preliminary report and the actual task accomplishment.

In the preliminary report we presented the project schedule shown in Figure 1.2. In practice what we observed is that we spent less time in the "Investigation" part and a lot more time in "Development" part, which consequently reduced the available time for the "Evaluation" and "Dissertation" parts. We had envisioned that the implementation of the RaptorQ standard would be very time-consuming, given its non-trivial nature. However,

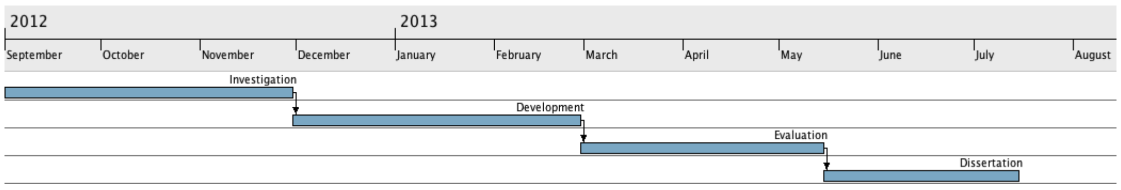


Figure 1.2: Gantt chart illustrating the original project schedule.

it seems we underestimated the complexity of the standard, and the magnitude of the undertaking (a relatively short period of time was given). Fortunately, we were able to still accomplish all the tasks with a small delay. Moreover, the original work was extended by studying efficient ways to attack the code and evaluating them in practice.

1.4 Document Structure

This document is structured as follows:

- **Chapter 2.** Some contextual scenarios and problems are presented, to motivate the use of solutions, such as fountain codes, for forward error correction. Furthermore, the evolution of fountain codes is described, culminating at the state-of-the-art Raptor codes.
- **Chapter 3.** A relatively in-depth description of how the RaptorQ code is specified, according to IETF's RFC 6330 [2], is given. The implementation of RaptorQ is described and some failure probability results are presented.
- **Chapter 4.** Explains how the RaptorQ standard can be broken through carefully choosing specific malicious faults. Furthermore, optimizations to the attack are discussed, and some possible solutions are presented to diminish the viability of the attack.
- **Chapter 5.** Summarizes the work and gives the overall conclusions.

Chapter 2

Context

“The White Rabbit put on his spectacles. ‘Where shall I begin, please your Majesty?’ she asked. ‘Begin at the beginning,’ the King said gravely, ‘and go on till you come to the end: then stop.”’

— Alice’s Adventures in Wonderland, Lewis Carroll.

2.1 Data Transmission

Analog media was replaced by its digital brethren to preserve quality, and add functionality and practicality. On the other hand, the explosion of the Internet use has led to an increase in high-speed computer networks (or vice-versa?) which make the digital content available to potentially anyone, anywhere, at any time. This is what fuels modern scientific and economic developments centered around the distribution of said content to a worldwide audience. The success of services like YouTube[1] or Spotify[2], is rooted in this marriage between digital content and the Internet.

Digital media has become an integral part of our lives. From listening to streamed audio, watching a video or satellite TV, or making a simple phone call, a large part of our professional and leisure lives are filled with digital media/information. Thus, it is fairly obvious that the reliable transport of the digital data to heterogeneous clients becomes a central and critical issue, for receivers can be anywhere and connected to networks with widely different qualities of service.

2.1.1 Transmission Control Protocol

The protocol used by any Internet transmission is the Internet Protocol (IP) [3]. The data to be transmitted is subdivided into packets. These packets have headers where information about their source and destination is stored, pretty much like a letter. Routers inspect the packet's header and forward it to another router closer to the destination, until the packet actually reaches its destiny. To do this, routers consult *routing tables* (which are regularly updated) through which they can determine the *shortest path* to reach the packet's destination.

However, as usual practice differs from theory, and the IP which, in theory should be sufficient for data delivery, is not. Routers get overwhelmed many times by incoming traffic, leading to dropped packets which will never reach their destination. To overcome this problem researchers proposed the Transmission Control Protocol (TCP) [4]. TCP is used "above" the IP and has withstood the test of time, as it remains the most widely used transmission protocol in the Internet, with many other popular protocols basing themselves on it (e.g., HTTP [5], SSH [6], SFTP [7]).

For every packet sent an acknowledgment is expected from the receiver. If the acknowledgment is not received after a prescribed period of time, the packet is considered lost and resent. The transmitter will also adjust the transmission rate in accordance with the loss rate.

[1]`www.youtube.com`
[2]`www.spotify.com`

In reality, TCP does not wait for acknowledgments of individual packets before sending the next one, but instead has at any time a number of packets in transit (window). The acknowledgment of a packet is only expected after all the previous packets have been acknowledged. When the sender receives an acknowledgment for a packet, without receiving an acknowledgment for a previous packet (using, for example, the selective acknowledgment mechanism), it detects the loss of the said packet. Consequently, the number of packets allowed to be in transit is reduced, which effectively reduces the rate at which the packets are sent to the receiver: this provides *rate control*. The reduction of the transmission rate has the objective of reducing traffic at the routers and to alleviate the network load[3].

2.1.2 User Datagram Protocol

The User Datagram Protocol (UDP) [8] was envisioned for shorter messages without so strict reliability requirements. It is simpler than the TCP and is also used above the IP. The packet has a header, also containing information about its origin and destination, and is routed through the network. There are no guarantees that it will arrive. Thus, it may be lost due to a router overflow or wireless transmission error. Each UDP packet is sent independently (i.e., there is no order) and may be sent in an arbitrarily high rate that can easily overload the network.

Even lacking TCP's higher reliability and rate control, UDP is useful in a number of use cases. For example, in applications where there is need for more responsiveness, such as with a video stream, since the effect of having the stream stopped waiting for a missed packet to be retransmitted is probably more harmful to the experience than missing a single packet amongst thousands.

Another use of UDP is that it can be employed effectively in conjunction with a broadcast/multicast enabled network, to transport content to a group in a scalable way. For example, broadcast file delivery applications often use UDP because the sent packets can be delivered concurrently to many receivers in a scalable way.

In these types of applications, the packet sending rate is fixed at the source according to the available capacity of the network and/or the application requirements. However, adding a reliability protocol on top of UDP can be quite valuable. This is one of the main uses for *forward error correction* (FEC) codes, namely fountain codes, specially if they add little to none overhead to the communication.

[3]There is an implicit assumption that losses have occurred due to routers being overwhelmed.

2.2 Example Transmission Patterns

2.2.1 Point-to-point Transmission

A *point-to-point* transmission is the simplest possible scenario. A *sender* transmits data to a *receiver*, as depicted in Figure 2.1.

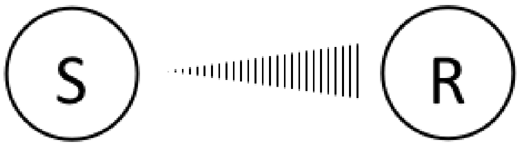


Figure 2.1: Point-to-point transmission scenario between sender S and receiver R.

In this case, if the distance between the two participants is not too large, TCP is the ideal protocol. However, for larger distances TCP is often inefficient: transmission is idle whilst the sender waits for acknowledgments, hence not fully availing the network's capacity. Additionally, if there is a packet loss, the transmission rate will slow down even more.

2.2.2 Point-to-multipoint Transmission

In a *point-to-multipoint* scenario, a single sender transmits to multiple receivers. A typical use case is video streaming between a server and many clients (see Figure 2.2). Unless the number of receivers is small, TCP has scalability issues in this scenario because the sender needs to keep track of the packet reception at all receivers (incurring into high processing overhead). Furthermore, since TCP is *connection oriented*, each receiver needs to receive a separate stream of data.

Therefore, the server load and the network load increases with the number of receivers, challenging the reliable transmission of data. This phenomenon makes it difficult to provide a scalable broadcast service on the Internet. However, in recent years such systems have started to be deployed with the help of HTTP caching server infrastructures.

UDP is often used in this type of settings, handling the scalability issue much better than TCP. However, due to the *best effort* nature of UDP, in a scenario with a considerable loss rate the degradation of experience (e.g., when watching a video stream or listening to streamed audio) may be intolerable. It would be interesting to have some mechanism that would appease this phenomenon, while still retaining UDP's efficiency.

2.2.3 Multipoint-to-point Transmission

A *multipoint-to-point* transmission setting happens when there are multiple senders transmitting (the same data) to a single receiver, as seen in Figure 2.3.

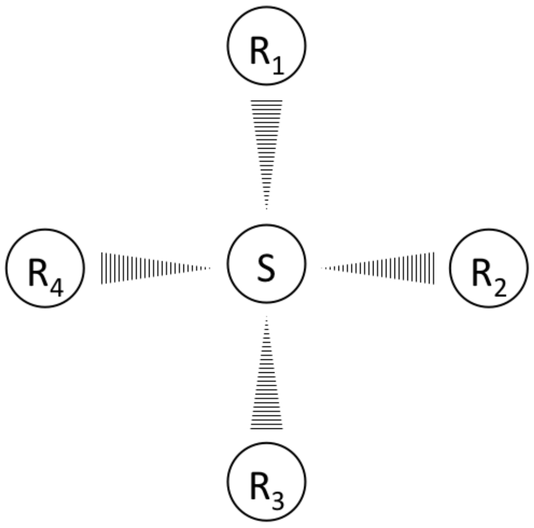


Figure 2.2: Point-to-multipoint transmission scenario between sender S and receivers R_1, R_2, R_3 and R_4.

Besides the problems discussed in the case of *point-to-point* transmission (see Section 2.2.1), using TCP (or UDP) in this scenario leads to a big network inefficiency: the senders have to be coordinated in order to send different parts of the data, otherwise duplicate packets will waste the network's resources.

It would be very interesting to have a mechanism of sending ubiquitous "generic" packets, which as a set would automatically dovetail into the original data. Hence, eliminating the need for sender coordination.

2.2.4 Multipoint-to-multipoint Transmission

Finally, the more complex transmission scenario is when a group of senders (each possessing a piece of data) are transmitting information to multiple receivers. We can see such a scenario represented in Figure 2.4.

An use case for such a scenario is a *peer-to-peer* network. In this case, all the previously discussed problems for the other transmission settings are also valid here. Moreover, the difficulties are gravely amplified when the participants are transient, that is, in a network with a high *churn rate* (which is usually the case for large peer-to-peer networks).

Once again, it would be interesting to have some mechanism that allowed for receivers to get ubiquitous "generic" packets that are independent of each other. This would allow for re-entering receivers to just resume the transmission, where they left off, even with a different sender.

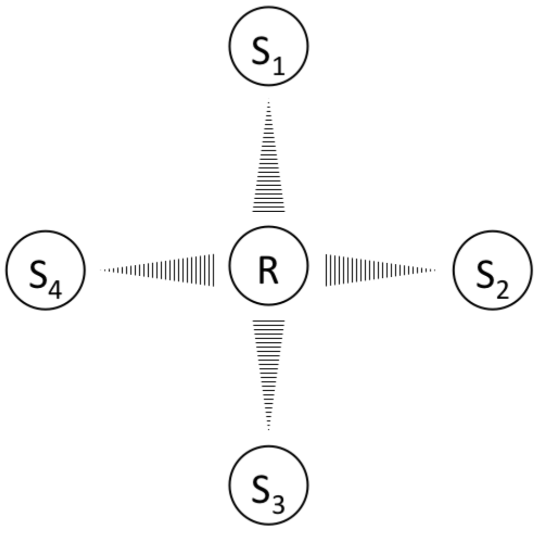


Figure 2.3: Multipoint-to-point transmission scenario between senders S_1, S_2, S_3 and S_4 to receiver R, where the same data is transmitted by all senders.

2.3 Fountain Codes

2.3.1 Preliminaries

Before getting into the details of fountain codes, a description of the *Binary Erasure Channel* (BEC) is due. Furthermore, some introductory terminology is presented to help the comprehension of the inner-works of the fountain codes.

Binary Erasure Channel

In information theory and telecommunications an *erasure channel* is a memoryless channel where symbols are either transmitted correctly or erased. Hence, the output alphabet (y) is the input alphabet (x) plus the erasure symbol, which is specified as 'e'. For an erasure probability ρ, the conditional probability of the channel is:

$$Pr(y|x) = \begin{cases} 1-\rho & \text{y = x;} \\ \rho & \text{y} = e; \\ 0 & \text{otherwise.} \end{cases}$$

This is a commonly-accepted model for packet transmission on the Internet, mainly because it models somewhat accurately the real-world scenarios: (1) some packets are simply lost during the transmission and never arrive at the receiver; (2) some other packets do arrive but are corrupted during the transmission, hence the receiver detects the flaw and discards them. It is easy to see how these two types of problems can be resumed to an *erasure*: the packets are either received correctly or an erasure occurred.

For the study of fountain codes, the binary erasure channel (BEC) is relevant. This channel is used frequently in information theory because it is one of the simplest channels

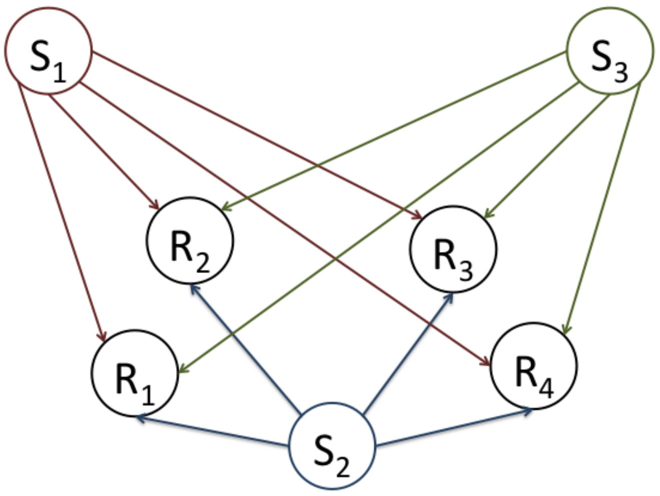


Figure 2.4: Multipoint-to-Multipoint transmission scenario between senders S_1, S_2 and S_3 to receivers R_1, R_2, R_3 and R_4.

to analyze. The BEC was introduced by Peter Elias of MIT in 1954 as a toy example. A BEC corresponds to an erasure channel model when the input can only take values 0 and 1. That being the case, the channel capacity is well-known to be $C = 1 - \rho$.

Let us suppose that there is an *oracle* capable of telling the source whenever a transmitted bit gets erased. There is nothing the source can do to avoid the erasures, but it can fix them when they happen. For example, the source could repeatedly transmit a bit until it gets through. Therefore, having an oracle allows to achieve a rate of $1 - \rho$ on average. Naturally, an oracle is not available normally and hence $1 - \rho$ is an upper bound.

Although fountain codes can be applied to general erasure channels, the analysis of the codes' properties focus almost exclusively on binary input symbols.

Terminology

Before proceeding we refer the reader to Figure 2.5, for a visual reference to the terminology that will be used, namely for blocks and symbols. The data that will be transmitted is divided into blocks, *source blocks*[4]. Usually, each block is encoded/decoded independently. *Symbols* are the fundamental data unit of the encoding and decoding processes, and even though the number of symbols in a block may vary, the size (in bytes) of each symbol is always the same. The term *source symbols* is used for the original data symbols, and *encoding symbols* for the symbols that result from the encoding process. Moreover, some codes apply a pre-code before encoding the data, and from this process results the *intermediate symbols*.

A code is called *systematic* if the encoding symbols correspond to the source symbols

[4]Some standards will divide each source block further into *sub-blocks* specially for larger sets of data. Sub-blocks are not represented in the figure for simplicity.

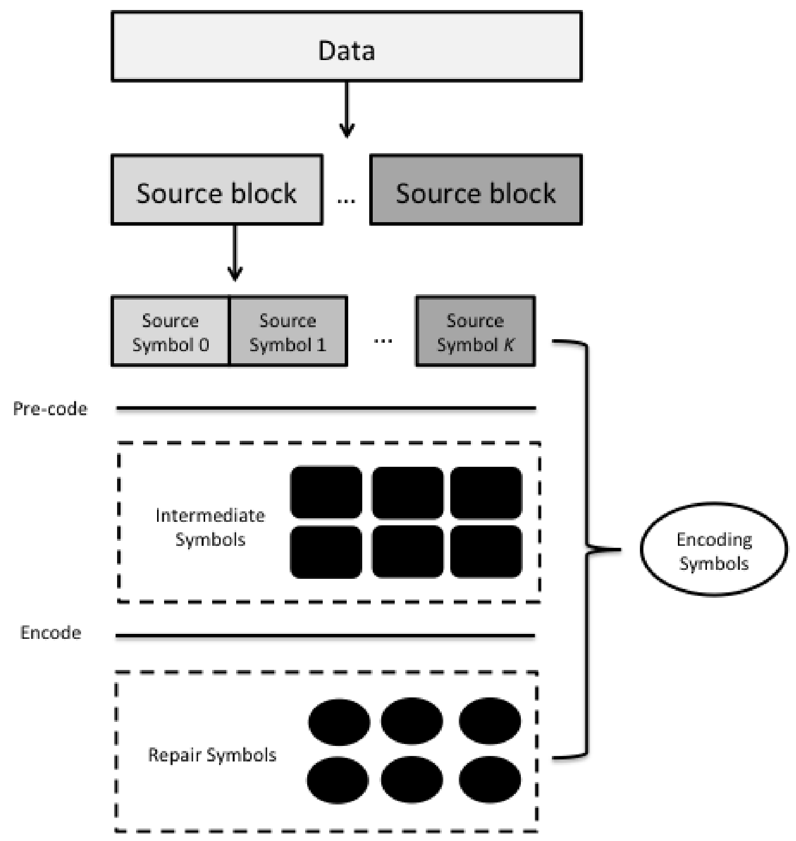


Figure 2.5: Block division and symbol generation for a systematic code.

together with the *repair symbols*. In this case, the *repair symbols* are "generic/universal" symbols that can repair (almost) any source symbol that is missing. The encoding symbols for *non-systematic* codes are only the generic repair symbols. Systematic codes are preferable to non-systematic codes because in the case when no failures occur the original information can be retrieved instantly.

The *overhead* used for decoding the received symbols is the number of *extra* encoding symbols (or repair symbols in the case of a systematic code) used in the decoding process. As an example, let us consider a scenario where the original source block was partitioned into 10 source symbols, from which 15 encoding symbols were generated. The receiver only received 12 encoding symbols. Instead of using only 10 encoding symbols, the 12 received symbols can be used, greatly increasing the probability of a successful decoding. In this case, the *overhead* was 2 symbols. The decoding *failure probability* $f(o)$ is the probability that the decoding fails with overhead o; we call a the set of pairs $\{(o,f(o)) : o = 0,1,...\}$ the *overhead-failure curve*.

2.3.2 The Digital Fountain Ideal

Fountain codes a.k.a. rateless erasure codes are a class of erasure codes with the property that a potentially limitless sequence of encoding symbols can be generated from a given set of source symbols (see Chapter 50 of [9]). They also have the property that the original source symbols can be recovered with high probability, from any subset of the encoding symbols of size equal to or only slightly larger than the number of source symbols.

They were devised as the ideal (theoretical) protocol for transmitting a file to many users, with different access times and channel fidelity. The name *fountain* or *rateless* refers to the fact that these codes do not exhibit a fixed code rate. The code rate (or information rate [10]) of a forward error correction code is the proportion of the data-stream that is useful (non-redundant). That is, if the code rate is k/n, for every k bits of useful information, the encoder generates totally n bits of data, of which $n - k$ are redundant. Usually the metaphor of a water fountain is used to describe the ideal concept behind fountain codes: when filling a bucket from a fountain, which particular drops fill the bucket is irrelevant, only the amount of water in the bucket matters. In an analogous fashion, the output packets of *fountain encoders* (a.k.a., *digital fountains*) must be universal like drops of water and hence be useful independently of time or the state of a user's channel. The particular set of received encoding symbols does not influence the successful recovery of the original data, only the number of received encoding symbols does.

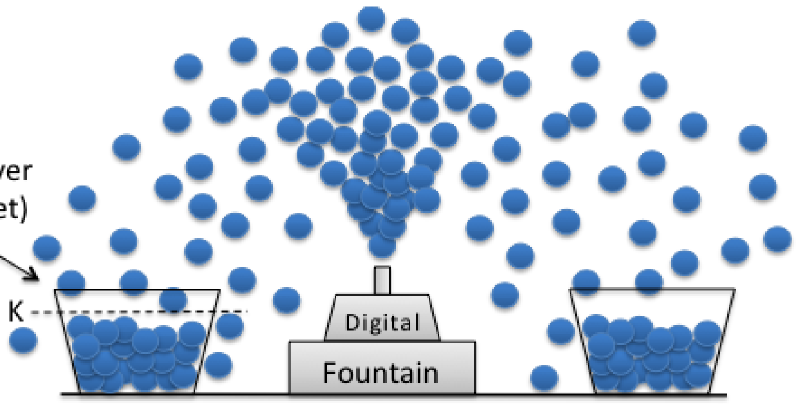


Figure 2.6: Illustration of a *digital fountain*.

In the case of a file that is split into K packets (or source symbols) and is encoded for a transmission in a BEC, an ideal digital fountain should have the following properties:

1. It can generate an endless supply of encoding packets with constant encoding cost per packet;
2. An user can reconstruct the file using any K packets with constant decoding cost per packet, meaning the decoding is linear in K;
3. The space needed to store any data during encoding and decoding is linear in K.

From these properties it is easy to verify that digital fountains are as reliable and efficient as TCP, but also universal and tolerant. They embody an effective solution to the transmission scenarios presented previously (see Section 2.2).

In the point-to-point scenario, the sender can generate encoding symbols from the data using a digital fountain, and place the encoding symbols into packets (*encoding packets*) that are transmitted via UDP (for example). For real-time applications, the packets can be sent at any rate that is below the rate at which the fountain encoder generates encoding symbols. Even though UDP does not offer any reliability property, the reliability of the transmission is ensured by the fountain property: as soon as the receiver collects K (plus a few extra) encoding symbols it can try to decode them, and recover the original data with high probability. However, the question of rate control remains, but in some cases it can be elegantly solved by exploiting the fountain property [11, 12].

In the case of point-to-multipoint transmission, the sender generates encoding symbols and places them into packets, which are transmitted, for example, via broadcast or multicast. The fundamental property of fountain codes guarantee that each receiver is capable of decoding the original data, receiving approximately the same amount of data that needs to be sent, independently of packet loss. Thus, one sender can efficiently and reliably deliver to a potentially limitless number of receivers (hence, being much more scalable than a TCP option, for example).

In the case of multipoint-to-point transmission, the various senders use fountain encoders to generate encoding symbols, and the receiver collects encoding symbols from the various senders. Through the properties of fountain codes, the mix of encoding symbols is irrelevant to the successful decoding of the original data. That is, there is no need for the senders to organize prior to transmission to determine which parts of the data each one will send. As soon as the receiver collects K (plus a few extra) encoding symbols, it should recover the original data. With the properties of fountain codes, we actually reduce the multipoint-to-point scenario to a embarrassingly parallel problem. That is, if the receiver needs to collect K symbols, and there are s senders, each sender only needs to (successfully) send K/s symbols.

The multipoint-to-multipoint transmission setting is solved in similar fashion, thus there is no need to elaborate any further.

Reed-Solomon (RS) codes [13] are the first example of fountain-like codes, because the data is divided into K source symbols and can be recovered from any K encoding symbols. However, RS codes require quadratic decoding time and are limited to a small block length. *Low-density parity-check* (LDPC) codes [14] come closer to the fountain code ideal, managing to reduce the decoding complexity by the use of the belief-propagation algorithm (which will be explained in Section 2.3.4) and interactive decoding techniques. However, early LDPC codes were restricted to fixed-degree regular graphs, causing significantly more than K encoding symbols to be needed to successfully decode the transmitted signal.

For the remainder of this chapter we will explore fountain codes that approximate the digital fountain ideal. These codes exploit the sparse graph structure that make LDPC codes effective, but allow the degrees of the nodes to take on a distribution. These codes require n encoding packets close to K (i.e., the required overhead is very low).

Construction Outline

In a very top-level manner, fountain codes are generally constructed based on a *probability distribution* $\mathfrak{D}$ [15] on the vector space $\mathbb{F}_2^K$ – for a vector $(x_1, ..., x_K)$ of K source symbols. The encoding process for generating the encoding symbols would be:

1. Sample $\mathfrak{D}$ to obtain a vector of binary values $(a_1, ..., a_K) \in \mathbb{F}_2^K$;
2. Calculate encoding symbol y_j with $y_j = \sum_i a_i x_i$.

The samplings of the fountain encoder are independent from encoding symbol to encoding symbol (step 1). This is extremely important as it induces an uniformity property on the symbols generated and ensures the fountain properties.

The average computational cost for generating an encoding symbol is simply the average weight[5] of the vector $(a_1, ..., a_k) \in \mathbb{F}_2^k$ when sampled from $\mathfrak{D}$, multiplied by the computational cost of adding two symbols together. Usually, the operation used for adding the symbols is the XOR, which is very efficient. Thus, it is important to keep the average weight as small as possible.

An important property of fountain codes is that it should be possible to decode the source symbols with little overhead with high probability.

Ideally, all encoding symbols are generated independently of one another. Furthermore, the probability of decoding failure should be independent of the mix of encoding symbols received, and only the number of received symbols should matter.

In practice, what is important is that the failure probability decreases as quickly as possible as a function of increasing overhead, i.e., the overhead-failure curve is *steep*. Equally important is that the decoder should be computationally efficient.

Random Binary Fountain

To explain the construction details of a *Random Binary Fountain* would be going out of the scope of this document. However, the random binary fountain is specially relevant when analyzing fountain codes as a reference point used for comparison. Thus, we will briefly expose its properties. A random binary fountain is a digital fountain where the distribution $\mathfrak{D}$ is the uniform distribution on $\mathbb{F}_2^K$. For a random binary fountain code

[5]Since these are vectors of binary values, the average weight will be the average number of 1's contained in the vectors.

operating on a source block with K source symbols, the overhead-failure curve is pointwise majorized by $\{(o, 2^{-o}) : o = 0, 1, ...\}$ with respect to the maximum-likelihood decoder. For example, at an overhead of $c - \log_2(K)$, the failure probability is $1/K^c$. In fact, it is possible to show that for not too small values of o, $f(o)$ is *roughly* 2^{-o} [16]. Therefore, a random binary fountain code has a quickly decreasing failure probability as a function of the overhead. Namely, the failure probability decreases by almost a factor of two for each increase of one in the overhead.

On the other hand, random binary fountain codes suffer from a large encoding and decoding computational complexity: on average, every encoding symbol will be the sum of around half the source symbols, requiring $K/2$ operations on average.

To sum up, the random binary fountain code achieves a good overhead-failure curve. However, both encoding and decoding are computationally complex. Ideally, one should look for a code with the same (or better) overhead-failure curve, but with much better encoding and decoding efficiency. For a more in-depth study of random digital fountains and the impact of the probability distribution $\mathfrak{D}$, we refer the reader to Luby [17], Harrelson et Al. [18] and Luby and Shokrollahi [16].

2.3.3 Tornado Codes

Tornado codes were first described in 1997 by M. Luby et al. [19], and were improved later on by the same authors in 2001 [20]. They are generally considered to be the first steps towards achieving the digital fountain ideal, discussed in Section 2.3.2. They do approach Shannon capacity [21] with linear decoding complexity (as idealized). However, they are *block codes* hence not *rateless*: violating the fountain property of generating an endless supply of encoding symbols.

Let us consider a typical point-to-multipoint scenario, where a single transmitter tries to transfer a file to a larger number of recipients through an erasure channel. The objective is to complete the file transfer with a minimum number of encoding symbols and low decoding complexity.

For K source symbols, *Reed-Solomon* (RS) codes [13] can achieve this with $K \times \log(K)$ encoding and quadratic decoding times. The reason for the longer decoding time is that in RS codes, every redundant symbol depends on all information symbols. By contrast, every redundant symbol depends only on a small number of information symbols in Tornado codes. Thus they achieve linear encoding and decoding complexity, with the cost that the user requires slightly more than K packets to successfully decode the transmitted symbols. Moreover, Tornado codes can achieve a rate just below the capacity $1 - \rho$ of the BEC. Thus, they are capacity-approaching codes.

Tornado codes are closely related to Gallager's LDPC codes [14], where codes are based on sparse bipartite graphs with a fixed degree d_λ for the source symbols and d_ρ for the encoding symbols. In fact, Tornado codes use a layered approach. All layers except

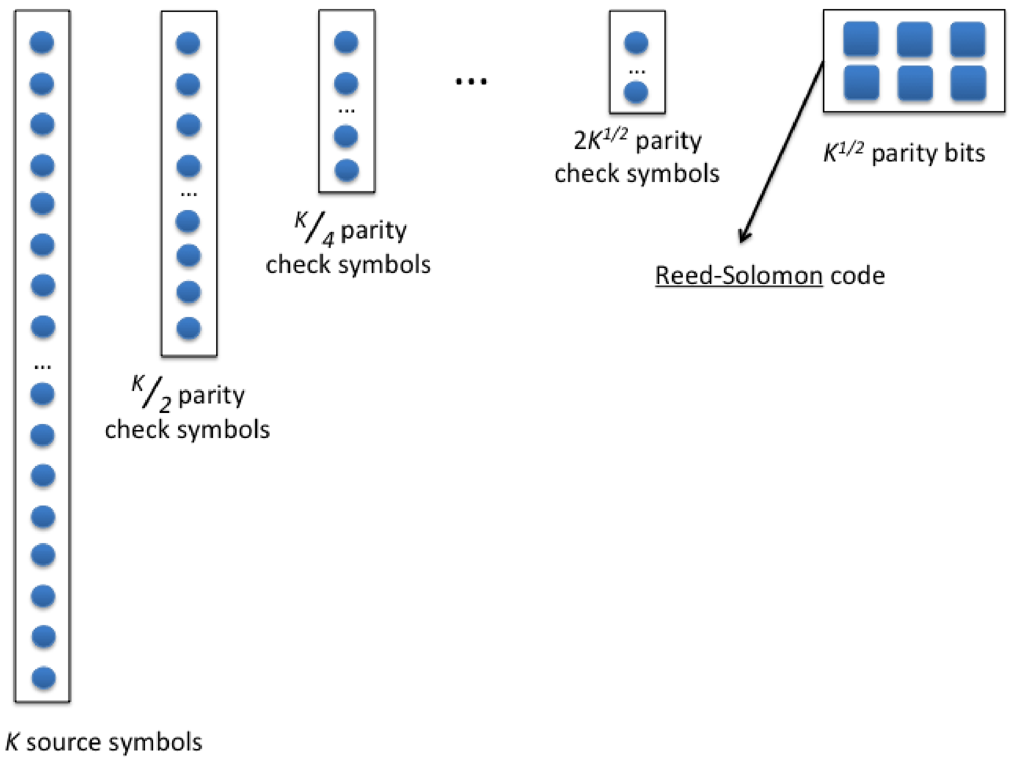
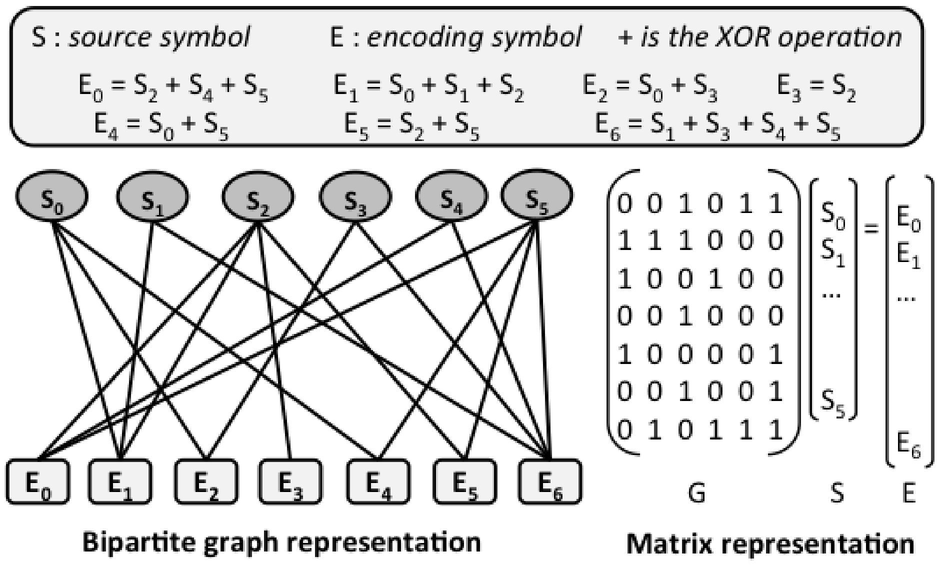


Figure 2.7: Example of the encoding process for a Tornado code. The K source symbols are inputted into a cascade of sparse bipartite graphs and a Reed-Solomon code.

the last use an LDPC error correction code, which is fast but has a chance of failure. The final layer uses a Reed–Solomon correction code, which is slower but is optimal in terms of failure recovery. Tornado codes dictates how many levels, how many recovery blocks in each level, and the distribution used to generate blocks for the non-final layers.

Unlike regular LDPC codes, Tornado codes use a cascade of irregular bipartite graphs. The main contribution is the design and analysis of optimal degree distributions for the bipartite graph such that the receiver is able to recover all missing bits by a simple erasure decoding algorithm. The innovation of Tornado code has also inspired work on irregular LDPC codes [22, 23, 24].

The idea is pretty straightforward, let us follow the process depicted in Figure 2.7. To protect the K source symbols from erasures, $\frac{K}{2}$ parity symbols are generated. To protect the $\frac{K}{2}$ parity symbols from erasures, another $\frac{K}{4}$ parity symbols are created. To further protect the $\frac{K}{4}$ parity symbols, $\frac{K}{8}$ are used, and so on and so forth. At a certain point, e.g., when the number of nodes reduces to $K^{\frac{1}{2}}$, recursion stops and a Reed-Solomon code is applied to protect the $K^{\frac{1}{2}}$ nodes. The decoding process start from the Reed-Solomon code and propagate to the left until all the lost source symbols are recovered. Even though the decoding of the Reed-Solomon code is of quadratic complexity, the overall decoding time is still linear in K.



S : *source symbol*
E : *encoding symbol*
+ *is the XOR operation*
$E_0 = S_2 + S_4 + S_5$
$E_1 = S_0 + S_1 + S_2$
$E_2 = S_0 + S_3$
$E_3 = S_2$
$E_4 = S_0 + S_5$
$E_5 = S_2 + S_5$
$E_6 = S_1 + S_3 + S_4 + S_5$
S_0 S_1 S_2 S_3 S_4 S_5
E_0 E_1 E_2 E_3 E_4 E_5 E_6
$\begin{pmatrix} 0&0&1&0&1&1\\1&1&1&0&0&0\\1&0&0&1&0&0\\0&0&1&0&0&0\\1&0&0&0&0&1\\0&0&1&0&0&1\\0&1&0&1&1&1 \end{pmatrix} \begin{bmatrix} S_0\\S_1\\ \dots \\ S_5 \end{bmatrix} = \begin{bmatrix} E_0\\E_1\\ \dots \\ E_6 \end{bmatrix}$
G
S
E
Bipartite graph representation
Matrix representation

Figure 2.8: LT code: Relations between source symbols (S) and encoding symbols (E), and their representation as a bipartite graph and in a matrix.

2.3.4 Luby Transform Codes

Luby Transform (LT) codes [17] are usually regarded as the first practical implementation of fountain codes for the BEC. They are rateless, thus, the encoder can generate as many encoding symbols as required to decode K source symbols.

The encoding algorithm is relatively simple, and is based on two random number generators. The first generator outputs the number of source symbols that should be XORed to produce a new encoding symbol and is called the *degree generator*. The second generator outputs random integers uniformly distributed between 0 and $K - 1$. This generator is called degree times in order to obtain the indexes of the source symbols that have to be XORed.

The decoding algorithm is very efficient, however, it may or may not succeed. LT codes are designed around this algorithm in such a way that the algorithm succeeds with high probability. This decoding algorithm has been rediscovered many times [14, 20, 25, 26, 27], and is known under the names of "belief-propagation decoder", "peeling decoder", "sum-product decoder" or yet "greedy decoder". It is similar to parity-check processes.

Belief-propagation is best described in terms of the "decoding graph" corresponding to the relationship between the source symbols and the encoding symbols. This is a bipartite graph between K source symbols and $N \geq K$ encoding symbols, as seen in Figure 2.8. The algorithm repeats the following until failure occurs in Step 1 or the decoder stops successfully in Step 4:

1. Find an encoding symbol E_i of degree 1; S_j is its unique neighbor among the source symbols. If there is no such degree 1 encoding symbol, the decoding fails.

2. Decode $S_j = E_i$.

3. Let $i_1, ..., i_l$ denote the indices of encoding symbols connected to source symbol S_j; set $Ei_m = Ei_m \oplus S_j$ for $m = 1, ..., l$, and remove source symbol S_j and all its edges from the graph ($\oplus$ is the XOR operation).

4. If there are source symbols that still need to be decoded, then go to Step 1.

Considering the example of Figure 2.8, encoding symbol E_3 is equal to source symbol S_2, while encoding symbol E_5 is the XOR of source symbols S_2 and S_5. Now imagine that all the encoding symbols were received. By applying the algorithm, in the first iteration it would be possible to recover S_2. In the second iteration, because S_2 has already been decoded, it is possible to decode S_5. In the third iteration, S_4 is decoded through E_0, and so on and so forth.

The encoding and decoding algorithms can also be represented as matrix operations (see Figure 2.8). The rows of matrix G specify the relations between the source symbols in S and the encoding symbols in E. Row i of G is defined using the two random number generators, where the number of 1's is the degree and the columns where they appear indicate the source symbols that are XORed to produce E_i. Therefore, one can create more encoding symbols simply by adding extra rows to G. The encoding algorithm corresponds to a matrix multiplication, $G \cdot S = E$, and similarly the decoding algorithm becomes a multiplication by the inverse of G, $S = G^{-1} \cdot E$. If it is impossible to invert G, then there is a *decoding failure*, as the source symbols cannot be recovered. To address this issue, further encoding symbols should be received, which are used to define a new G matrix that hopefully can be inverted.

The complexity of belief-propagation decoding is essentially same as the complexity of the encoding algorithm, in the sense that there is exactly one symbol operation performed for each edge in the bipartite graph between the source symbols and the encoding symbols during both encoding and belief-propagation decoding.

This is probably the main attraction of belief-propagation decoding, as it is typically decoding that is hard to make efficient. From a performance point of view, the computational complexity of decoding (and encoding) is linear in the average degree returned by the degree generator multiplied by the size of the source block (which accounts for the number of symbols and their size). Consequently, amongst the codes using belief-propagation decoding, what will distinguish better designed codes from lesser ones will be the probability density function of the chosen degree generator. Its definition represents a challenge to balance a small average number of XOR operations (for less complexity), with the need for a high probability of successful recovery of the source symbols. Namely, one would like to keep the number of encoding symbols N needed for decoding as close to K as possible[6].

Such a distribution was given by Luby [17], leading to the class of *Luby Transform codes*. The *Robust Soliton distribution* presented by Luby, offers an average degree of $O(\log(K))$. Hence, $O(\log(K))$ symbol operations are needed to generate one encoding symbol, and $O(K \times \log(K))$ symbol operations are required to decode all the symbols. In conclusion, K source symbols can be recovered from any $K + O(\sqrt{K} \times \log^2(K/\delta))$ encoding symbols with probability $1 - \delta$.

The performance of fountain codes can in general be measured by the inefficiency of the code describing the average amount of additional symbols required for successful decoding when compared with an ideal code. In the case of LT codes, one needs around 5% to 10% extra symbols, which is not negligible in practical terms. Furthermore, for large values of K the decoding complexity is still relatively high. This has stimulated the development of new fountain codes.

2.3.5 Raptor Codes

Raptor codes were introduced by Shokrollahi in 2006 [28], but had already been filed for patent in 2001 [29]. They provide improved system reliability while also offering a large degree of freedom in parameter choice. Raptor codes can be viewed as a concatenation of several codes, as shown in Figure 2.9. These revolve around the LT code [17], which plays an important role in Raptor codes' performance.

Raptor codes can be viewed from different angles. On the one hand, they might be viewed as a systematic block code, but on the other hand, the initial idea of fountain codes is also inherent. Looking at Figure 2.9, it can be seen that the *non-systematic* Raptor code is constructed not by encoding *source symbols* with the LT code, but *intermediate symbols* generated by some outer high-rate block code, i.e., the L *intermediate symbols* are themselves code symbols generated by some code with K' input *source symbols* (seen in Figure 2.9 as the "Pre-Coding" step). The most-inner code is a non-systematic LT code with L input symbols, which provides the fountain property of the Raptor code. The LT code, as explained in Section 2.3.4, is served by a tuple generator, whose tuples have the necessary parameters for the LT encoder[7]. Finally, a *systematic* realization of the code is achieved by applying some pre-processing to the K source symbols such that the K' input symbols to the non-systematic Raptor code are obtained.

Raptor codes have extremely fast encoding and decoding algorithms, i.e., a small constant average number of symbol operations per encoded symbol generated, and a similar small constant number of symbol operations per source symbol recovered. Thus, over-

[6]Note that a purely random function would not offer attractive encoding and decoding complexities, as we have discussed in Section 2.3.2.

[7]Here the tuple generator represents the random generators used by the LT code.

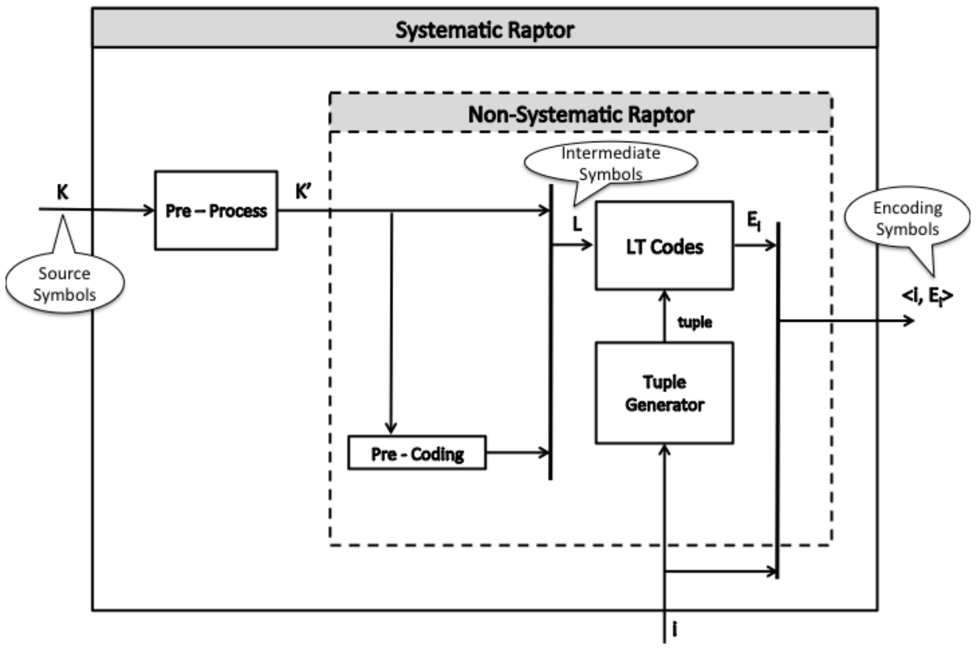


Figure 2.9: Conceptual diagram of building Raptor codes as a concatenation of other codes.

all Raptor codes achieve a near optimal encoding and decoding complexity (to within a constant factor).

It is difficult to design LT codes for which the average degree is constant: in this case there is, with high probability, a constant fraction of the source symbols that do not contribute to the values of any of the received encoding symbols. Independently of the algorithm used, these source symbols can never be recovered. The basic idea behind Raptor codes is to use a (usually high-rate)[8] code to pre-code the source symbols (creating the *intermediate symbols*). Next, a suitable LT code is applied to the intermediate symbols, to produce the encoding symbols. Once the LT decoder finishes its operation, a small fraction of the intermediate symbols will still be unrecovered. However, if the pre-code is chosen appropriately, then this set of symbols can be recovered using the erasure decoding algorithm for the pre-code.

When we apply the pre-code to the K' source symbols of the non-systematic Raptor, $L > K'$ intermediate symbols are generated. There are $L - K'$ constraints that define the relationship between the source symbols and the intermediate symbols. These constraints can be viewed as symbols, hereafter called *constraint symbols*.

Both the received encoding symbols and the constraint symbols are used for decoding

[8]The name *Raptor* code comes from "rapid Tornado". Tornado codes [19] are themselves a layered approach of other codes (*Low Density Parity Check* [14] and *Reed-Solomon* codes [13]), as briefly discussed in Section 2.3.3.

the intermediate symbols. The right interplay between the pre-code and the LT code used, is crucial for obtaining codes with good overhead-failure curves.

Systematic Raptor Codes. Are usually preferable to non-systematic Raptor codes, not only because in case when there is no failure, decoding is immediate, but also because there is wider variety of applications for systematic Raptor codes. The overhead-failure curve for systematic Raptor codes should be independent of the mix of received source symbols and repair symbols – i.e., only the total number of encoding symbols received determines decodability[9], as conceptualized by the digital fountain ideal.

One possible trivial construction of a systematic Raptor code is to simply use the encoding symbols generated from a non-systematic Raptor code as the repair symbols, and then just designate the source symbols to also be encoding symbols. This trivial construction works very poorly because the overhead-failure curve will depend strongly on the mix of received source and repair symbols. It is particularly bad when the majority of the encoding symbols received are repair symbols. Details are provided in [30].

An entirely different approach is thus needed to design systematic Raptor codes. Such an approach is outlined in [28, 31]. To dive further into this would be going out of the scope of this thesis, but the basic idea is that the "Pre-process" box (seen in Figure 2.9) is actually responsible for "decoding" (i.e., making the inverse operation of the "Non-Systematic" part) in such way that, when the K' *pre-processed* symbols are encoded, they result in the original K source symbols.

Inactivation Decoding. Is the algorithm used by Raptor codes for decoding. Using belief-propagation decoding can require a large overhead for small values of K to achieve a reasonably small failure probability. The inactivation decoding algorithm [32] combines the optimality of the Gaussian elimination, with the efficiency of the belief-propagation algorithm. When the belief-propagation would fail, sometimes it would still be mathematically possible to decode. The inactivation decoder makes use of these mathematical possibilities, while still employing the efficient belief-propagation decoding as much as possible. For this, it views the decoding process as a system of linear equations to be solved, and the key to the design of such linear system of equations is to ensure that the matrix is full rank with high probability (otherwise decoding will fail)[10]. Very concisely, when the belief-propagation algorithm is stalled because there is no encoding symbol with degree 1, one or more symbols are "inactivated" and considered decoded for the remainder of the belief-propagation decoding process. At the end, Gaussian elimination is used to recover the values of the *dynamically* inactive symbols, and these in turn determine the

[9]This is an important notion, however difficult to employ in practice. As we will see in Chapter 4, we will exploit the fact that this notion has not materialized in the current standards to perform our attack.

[10]Our attack will target this property precisely, as we will see in Chapter 4 we try to force the reduction of the decoding matrix's rank.

values of the other intermediate symbols. With the intermediate symbols decoded, one can trivially recover any missing source symbol.

Any Raptor code will outperform LT codes in terms of computational complexity, and more advanced Raptor codes have better overhead-failure curves than LT codes in practice. Shokrollahi [28], exemplifies one Raptor code construction that, given a constant $\epsilon > 0$, the average number of symbol operations per generated encoding symbol is $O(\log(1/\epsilon))$, the number of symbol operations to decode the source block is $O(K \times \log(1/\epsilon))$, and for overhead $\epsilon \times K$ the failure probability is $1/K^c$ for a constant $c > 1$ that is independent of ϵ.

LT codes require the decoding cost to be $O(\log K)$ in order for every source symbol to be recovered and decoding to be successful. Raptor codes, on the other hand, were developed as a way to reduce decoding cost to $O(1)$[11]. In fact, if designed properly, a Raptor code can achieve constant per-symbol encoding and decoding cost with overhead close to zero in a space proportional to K [28]. This has been shown to be the closest code to the universal digital fountain ideal.

Raptor codes have been used for years in broadcast networks [33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43], namely for content delivery through different channels, including satellite transmissions. They have been standardized in IETF's RFC 5053 [1] and RFC 6330 [2]. In addition, they have been widely adopted by the military for mission critical systems/operations, and for scenarios where communication may be intermittent and/or with high loss rates (e.g., after natural disasters). The Raptor code standardized in IETF's RFC 5053 [1], a.k.a. R10, was the first Raptor code adopted into a number of different standards. It exhibits an overhead-failure curve that essentially is that of a random binary fountain code. The most advanced Raptor code, RaptorQ as described in IETF's RFC 6330 [2], has an even better overhead-failure curve.

[11]By preprocessing the LT code with a standard erasure block code, e.g., a Tornado code.

Chapter 3

The RaptorQ FEC Code

"In theory, there is no difference between theory and practice, but in practice there is."[1]

[1]Written on the interior glass wall of the EPFL cafeteria in the Computer Science Building BC, just near *Place Alan Turing*. Wikipedia attributed to Johannes L. A. van de Snepscheut, to Yogi Berra, to Chuck Reid, to William T. Harbaugh, to Manfred Eigen, and to Karl Marx.

The RaptorQ code is the most advanced Raptor code and is described in IETF's RFC 6330 [2]. It is built upon the R10 code [1], improving it in several ways. RaptorQ supports larger source blocks with up to 56,403 source symbols, and can generate up to 16,777,216 encoding symbols. It also has a much better behavior under network failures (i.e., a steeper overhead-failure curve) and superior coding efficiency. To achieve this performance the RaptorQ code combines two major new ideas: the first is to resort to larger alphabets, and the second is to use a technique called "permanent inactivation" for decoding, which builds upon the "dynamic inactivation" [32] used in previous Raptor codes.

The chapter starts by giving a more practical view of how one can use the RaptorQ FEC scheme in communication. Later on it introduces the standard while consolidating the concepts presented in the previous chapters. We also evaluate the failure probability of our implementation and discuss the implementation's development.

3.1 Overview of Data Transmission with RaptorQ

RaptorQ is able to recover from the loss of packets that may occur anywhere between the sender and the receiver nodes. This covers for instance problems in routers that have to drop packets due to excessive load, or data corruptions that are detected using a checksum added to the packets (causing the receiver to discard the packet). Applications that may benefit from this capability are many and diverse, but file multicasting is a particularly good example. When a file is multicast, it is hard to address the different losses that are typically experienced by the various channels connecting the receivers using an *ack+retransmit* mechanism. In this case, since disparate packets arrive at each receiver, the sender would have to find out which packets are missing and next retransmit them, eventually more than one time, creating a high load (and delay) even with relatively small network loss probabilities[2]. This sort of problem is avoided with RaptorQ because data can be reconstructed from distinct subsets of the packets.

Figure 3.1 displays how data (i.e., a *source object*) according to RFC 6330 can be transmitted using RaptorQ. The data is first divided in blocks, called *source blocks*, that are processed independently by the RaptorQ encoder. Source blocks are then partitioned into K equal sized units named *source symbols*[3]. The number of source symbols across the various source blocks may vary (i.e., K may change), but the size of a source symbol

[2]Imagine a network with a loss probability of 1%, and an application that wants to send a 10MByte file, divided in 10K packets of 1KByte each, to 100 receivers. In the first transmission, every receiver will lose approximately 100 packets. Therefore, each of them will have to communicate with the sender to inform which packets are missing, so that later on a specific retransmission is done for every receiver. Furthermore, since several of the retransmitted packets will also be dropped, the process has to be repeated a number of times.

[3]For now, we will not consider the need to add padding in some cases.

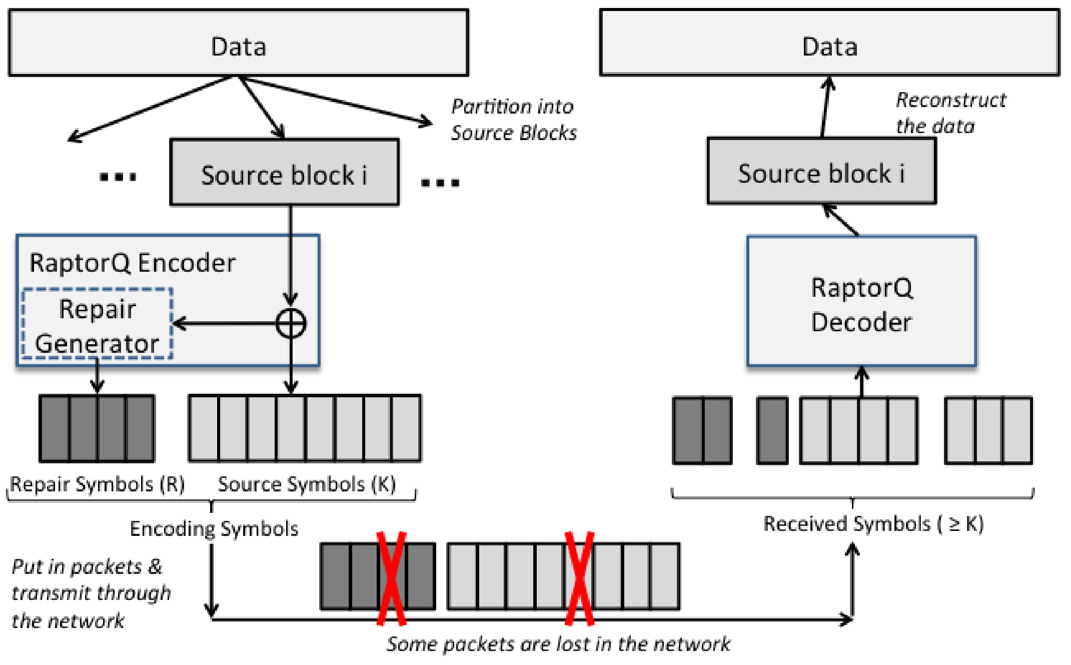


Figure 3.1: Overview of data transmission with the RaptorQ FEC: sender (left) and receiver (right).

is always T bytes. The value of T is normally selected in such a way that it corresponds to the payload size of a network packet (i.e., the MTU of the network minus the headers). This way, a discarded packet only affects a single symbol.

The RaptorQ encoder then receives the source symbols in order to generate a number of *repair symbols*. Since RaptorQ is a fountain code, as many repair symbols as needed can be created on the fly. Moreover, since the code is *systematic*, the *encoding symbols* that are transmitted through the network are constituted by the source symbols plus the repair symbols. Meaning that in the case when there is no packet loss, there is no decoding overhead.

During the transmission of the packets, some of them can suffer failures and be lost. The RaptorQ decoder then takes the received encoding symbols (any subset with a size equal or slightly larger than K) to recover the source block. The *code overhead* metric indicates the number of encoding symbols, beyond the number of source symbols, that is required for the decoding process (e.g., an overhead of 1 indicates that $(K + 1)$ encoding symbols are used). In general, the minimum value for the overhead is 0, as this means that recovery is possible with exactly the same amount of information as the original data. One of the particularly interesting characteristics of RaptorQ is that, independently of the value of K and for wide range of network loss rates, it can successfully decode with high probability with overheads as low as 0 to 2.

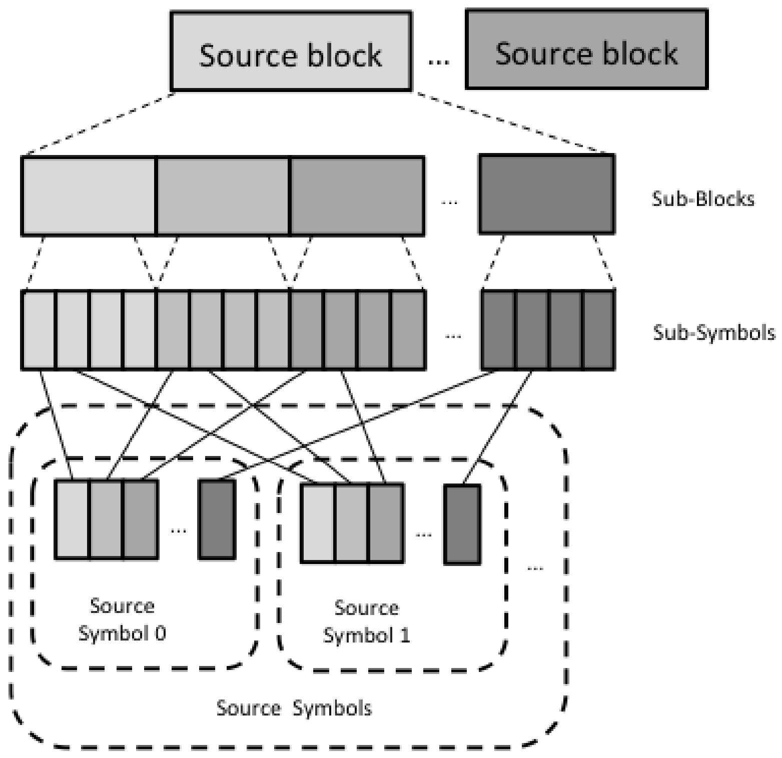


Figure 3.2: Overview of RaptorQ's data partitioning algorithm.

3.2 RaptorQ Construction

This section gives a top-level explanation on the design of the RaptorQ code, standardized in [2]. When using the RaptorQ code, the data to be encoded must be partitioned into *source blocks*. The partitioning algorithm is "optional", in the sense that it may be a linear one: break the total data into sequential *source blocks* of size $K \times T$. It may also be implementation dependent, but [2] specifies one. This algorithm is very important when using larger sets of data, as it introduces entropy and may also affect performance. The algorithm used in the standard is illustrated in Figure 3.2: (1) the data is partitioned into *source blocks*; (2) each source block is partitioned further into *sub-blocks*; (3) the sub-blocks are divided into *sub-symbols*; (4) each *source symbol* is constituted by one sub-symbol of each sub-block. This algorithm ensures perfect interleaving of the data across all sub-blocks of a source block, so that the amount of data received for each sub-block of a source block is guaranteed to be the same amount as for all other sub-blocks of the same source block - independent of packet loss patterns. Note that further dividing the data into sub-blocks is optional, and is more relevant when using larger sets of data. Figure 3.3 describes the encoding process that is applied to each source block. As we will see further in this section, the decoding process is actually very similar, obeying the same process scheme.

For the next sections, we will be describing in greater detail what each step in Figure

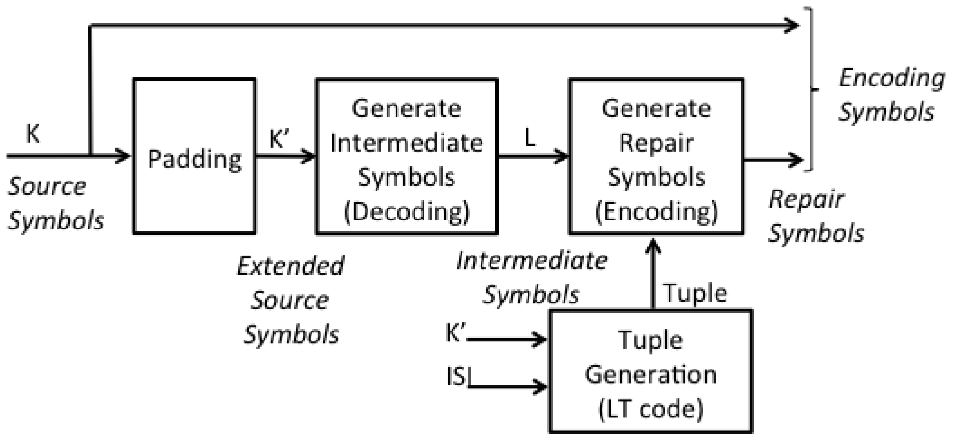


Figure 3.3: Overview of the RaptorQ encoding process.

3.3 entails, and how RaptorQ's encoding and decoding processes are built.

3.2.1 Padding

RaptorQ supports only a finite set of values for the number of symbols in a source block. Thus, sometimes there is the need for padding, from which results an *extended source block*. RaptorQ uses a precomputed table with these values (and other associated parameters, which are used by the encoding and decoding processes), let us call them K'. Hence, each source block is *augmented* with K' – K *padding symbols* of value 0. K' is the value in that table that is closest to K, but greater than or equal to K.

Using a predefined set of possible values for how many symbols a (extended) source block has, minimizes the amount of table information that needs to be stored at each endpoint, and effectively contributes to faster encoding and decoding. The original number of symbols per source block K, is assumed to be known at both ends of the communication. Thus, being the table also known at both endpoints, the padding symbols are not transmitted[4]. The recipient has all the necessary information to reconstruct the padding symbols during the decoding process. Hence, no network resources are wasted.

The *padding symbols* are regarded as regular *source symbols* by the encoding and decoding processes. Consequently, hereinafter we will make no further distinction between both of them.

3.2.2 Generating Intermediate Symbols

The second step when encoding a source block is to generate $L > K'$ intermediate symbols from the K' source symbols.

[4]Note that their value is fixed at 0, and therefore, they are also known at both sides of the transmission.

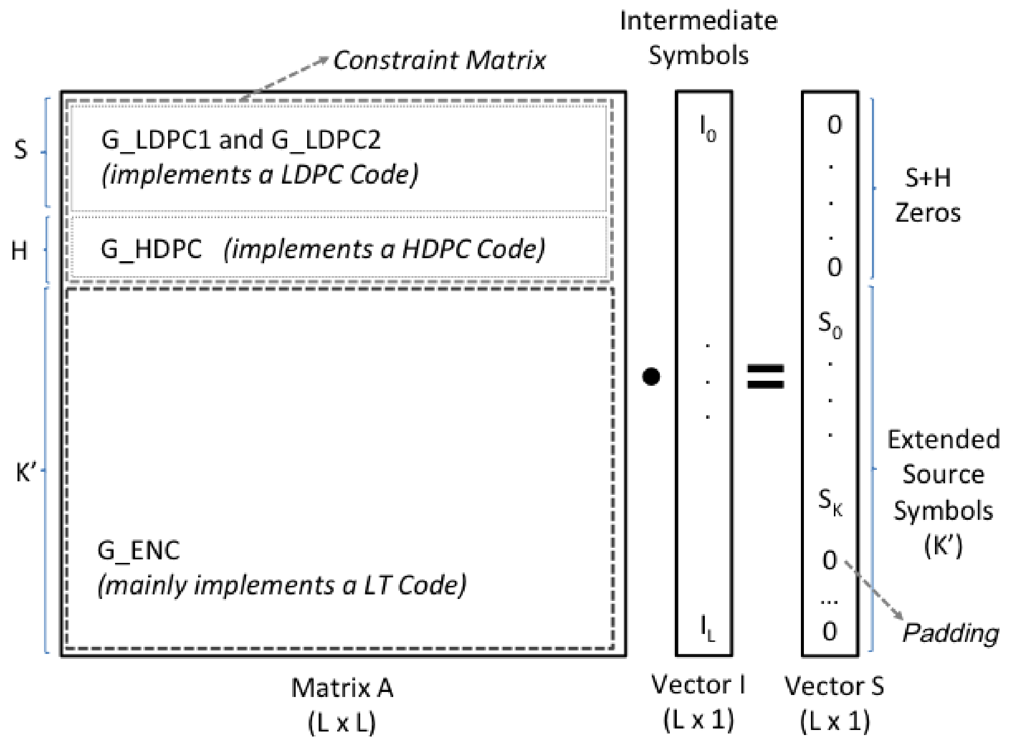


Figure 3.4: Computing the intermediate symbols during encoding.

Symbol Identification

The number of source symbols in a source block, K, needs to be known at the sender and the receiver. Based on its value, one can then compute K' since no padding symbols are actually transmitted. The *Encoding Symbol ID* (ESI) identifies the encoding symbols of a source block (as RaptorQ is systematic, the encoding symbols of a source block consist of the source symbols and the repair symbols associated with it). The ESIs for the *source symbols* are $0, 1, 2, ..., K - 1$, and the ESIs for the *repair symbols* are $K, K + 1, K + 2, ...$.

However, for purposes of encoding and decoding data, the value of K' is used (source symbols and padding symbols). Thus, the encoder and decoder use the *Internal Symbol ID* (ISI) to identify the symbols associated with the *extended source block*. Consequently, the *source symbols*'s ISIs are (once again) $0, 1, 2, ..., K - 1$, the ISIs for the *padding symbols* are $K, K + 1, K + 2, ..., K' - 1$, and, finally, the ISIs for the *repair symbols* are $K', K' + 1, K' + 2, ...$.

Calculating the Intermediate Symbols

The intermediate symbols are calculated by solving a system of linear equations. The process of building this system and the properties it holds, represents the secret to RaptorQ's incredible reliability (i.e., low probability of decoding failure). A representation of such a system is depicted in Figure 3.4.

As explained in Section 2.3.5, Raptor codes can be viewed as a coupling of various codes. The RaptorQ code is no different. Figure 3.4 shows that $Matrix\ A$ is divided into 3 parts. Each part represents one or more sub-matrices, but for simplicity we will not go into so much detail.

Each part of the $Matrix\ A$ actually represents one type of code used. The top part, consisting of the first S lines, corresponds to a LPDC code. The middle part has H lines, and corresponds to a *High-density Pairity Check* (HDPC) code used (where *finite fields* of higher dimension are used). Finally, the bottom part, consisting of the last K' lines, represents a LT code.

Constraints. The two top parts are used as *constraints* that establish *pre-coding relationships* amongst the intermediate symbols. Each of the first $S + H$ rows of $Matrix\ A$ represent a pre-coding relation, an equation[5]. The constraints are generated with the help of a pseudo-random number generator defined in the standard.

LT Code. The LT part, is responsible for actually pre-coding the source symbols into intermediate symbols[6]. Furthermore, as we described in Section 2.3.4, the LT encoding process relies on two random number generators. In the RaptorQ standard, the two random generators were carefully substituted by pseudo-random generators that keep the nice characteristic of ensuring effective decoding. These pseudo-random generators receive as seed the identifier (ISI) of the encoding symbol (among others), which is communicated in the header of the packet. Therefore, both the sender and the receiver can generate autonomously and deterministically the same ("random") values (and for that matter, also an adversary that knows the seed information). These generators are represented in Figure 3.3 as the box "Tuple Generation".

Non-binary alphabets. RaptorQ employs a HDPC code with values over the finite field $\mathbb{F}_{256}$. Using a code over $\mathbb{F}_{256}$ as part of the pre-coding is computationally efficient and contributes largely to a better overhead-failure curve. The rationale behind this is discussed in greater detail in Section 3.3.1 of [16]. RaptorQ is predictable in terms of its failure probability as a function of overhead, and the dependency of the failure probability on the loss rate is minimal, as there is a graceful degradation of the probability as the rate grows.

Vector S. $Vector\ S$ (seen at the right side of Figure 3.4) is very easy to construct: (1) the rows corresponding to the constraints (first $S + H$), have the value 0; (2) the last K'

[5]Note that these relationships are nothing but the set of indexes of intermediate symbols that must be summed to generate the source symbols. It is interesting to note that the whole encoding and decoding processes are defined by two types of relationships: *constraint relationships* among the intermediate symbols; and *LT-PI relationships* between the intermediate symbols and the encoding symbols.

[6]The matrix representation of the LT process, just as seen in Figure 2.8.

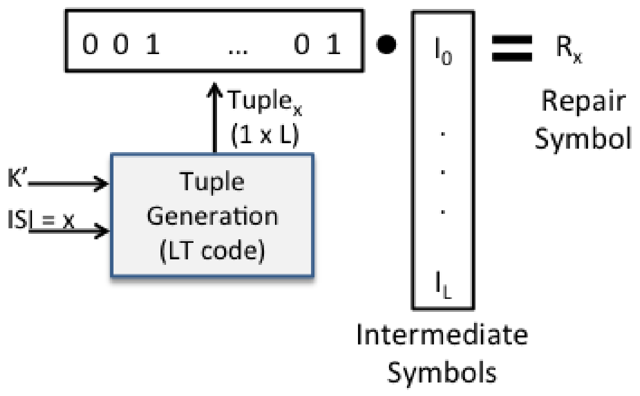


Figure 3.5: Computing the repair symbols during encoding.

rows, are the the source symbols (in the case of the padding symbols, as aforementioned, the value is 0), each symbol in a different row (in order).

Solving the system. With the system of linear equations built, it is necessary to solve it to calculate the intermediate symbols. Since $A \cdot I = S$, I can be obtained by inverting A: $I = A^{-1} \cdot S$. The system of equations is created using specific pre-coding relationships, which guarantees certain properties. These properties ensure that $Matrix\ A$ is always invertible.

It is interesting to note that this part of the encoding process actually corresponds to a decoding operation: the intermediate symbols are being *recovered* (or *decoded*), so that they can be used in the next step of the encoding process (see Figure 3.3), where they are *encoded* to produce the repair symbols.

3.2.3 Generating Repair Symbols

The third and final part of the encoding process depicted in Figure 3.3 corresponds to generating the encoding symbols, which consist of *repair symbols* and the original *source symbols*. The *source symbols* are ready from the start, so what remains is to generate the repair symbols.

Figure 3.5 displays how the repair symbols can be calculated: the first step is to get the indexes of the intermediate symbols that need to be added[7] to produce the repair symbol. The "Tuple Generator" receives as parameters K' and the repair symbol's ISI x. The tuple returned is then used to determine which intermediate symbols should be XORed to produce the repair symbol.

In congruence with our previous line of thought, we can see the generation of a repair

[7]Recall that the add operation actually corresponds to a XOR.

symbol as a multiplication between a matrix row and the intermediate symbols' vector. Looking at Figure 3.5, it is possible to see that we can get the set of intermediate symbols to be XORed, by feeding the "Tuple Generator" with the ISI of the repair symbol we want to generate. This set of indexes can be represented as a row (an equation) that when multiplied by the vector of intermediate symbols, will result in the repair symbol that one wants to generate. This process can be repeated for as many repair symbols as needed, only by changing the ISI fed to the "Tuple Generator".

It is relevant to mention that the tuple generated contains not only information about the LT code, but also relative to the *permanently inactive (PI) symbols*. Which we will explain in the next section when we talk about the decoding process.

Furthermore, just for completeness's sake, we should mention that we can also generate the source symbols with this same process (using their respective ISIs). However, in practice this is obviously unnecessary, since we already have the source symbols. They are used only to calculate the intermediate symbols, but it is interesting to note how everything fits in together.

To summarize the encoding procedure: in Figure 3.3, the extended source block first passes through a decoding process, this is solving the system of linear equations in order to calculate the resulting intermediate symbols. The system of linear equations is illustrated in Figure 3.4. The constraints needed to put it together come with the help of the "Tuple Generator". When the intermediate symbols have been computed, they are employed to create the repair symbols, again, using the "Tuple Generator". Finally, the set of the original source symbols together with the repair symbols, result in the *encoding symbols*.

3.2.4 The Decoding Process

The decoding process is actually the same process as encoding. The decoder is assumed to know the structure of the source block it is to decode (e.g., K, T, K'), as this information can be added to the headers of the packets. The decoder can then create the extended source block.

To decode the extended source block, let us assume that the receiver gets $N \geq K'$ encoding symbols for that source block. If all source symbols arrive at the receiver, then the decoding is complete. Otherwise, the construction of a system of linear equations, similar to the previous one, takes place. The system of equations is similar and not equal due to a couple of minor differences: (1) any equation corresponding to a missing source symbol is replaced by an equation corresponding to a repair symbol; (2) if additional repair symbols are received, they will also take part of the system of equations, ensuring a much greater probability of successful decoding.

Figure 3.6 provides an example decoding operation. In the example, there are 8 source symbols and 2 repair symbols, and one of each was lost during the transmission: source symbol S_i was lost and only the repair symbol R_x was received. As described in Section

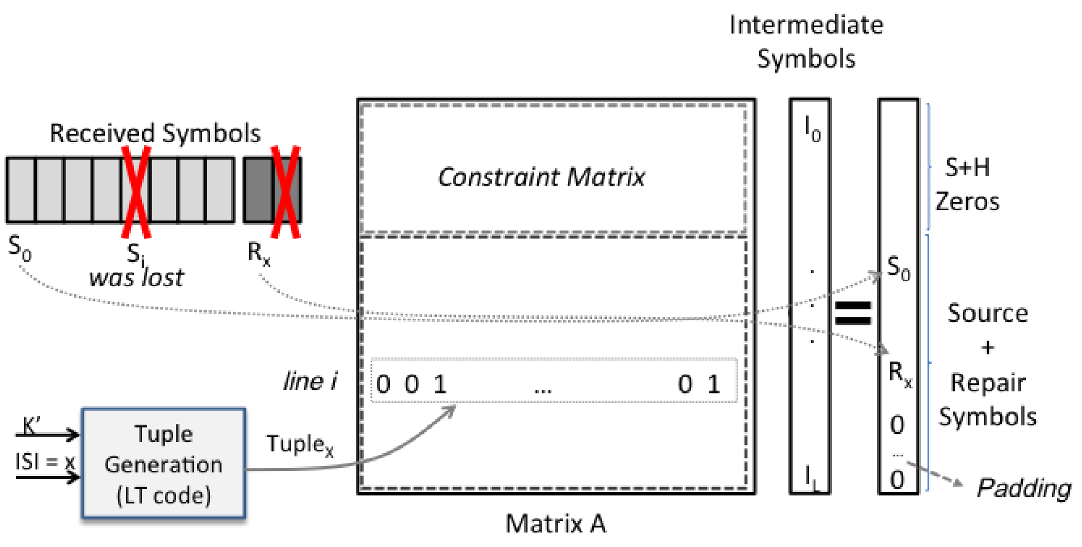


Figure 3.6: Computing the intermediate symbols during decoding.

3.2.2, a system of linear equations $A \cdot I = S$ (see Figure 3.4) needs to be built. However, we are missing one of the source symbols. Even though we are able to determine its corresponding row in $Matrix\ A$, since we do not know its value, we cannot complete $Vector\ S$. However, we did receive one repair symbol, R_x. Using its ISI, x, we can generate a tuple using the "Tuple Generator", which can then be used to compute the indexes of the intermediate symbols that should be XORed to generate R_x. We can then replace S_i's row in $Matrix\ A$ by R_x's row, and use R_x's value in $Vector\ S$ instead of S_i's. Let us call our new matrix and vector $A*$ and $S*$ respectively. We have now a complete system: $A * \cdot I = S*$. We can solve it by inverting $A*$, such that $I = A *^{-1} \cdot S*$. However, on contrast to the encoding process's original $Matrix\ A$, we have no guarantee that $A*$ is invertible. If that is not the case, we have a decoding failure. In spite of that, there is a very high probability that $A*$ will be invertible, as we will see in Section 3.3.1, when we look at the decoding failure probabilities.

To greatly improve the chances of $A*$ being invertible, it is possible to use one or more extra repair symbols. We could do that if we had received more repair symbols. We would then use their equations in $Matrix\ A*$ and their values in $Vector\ S*$ as *extra* rows. These extra rows will greatly increase the probability of $A*$ being invertible. Moreover, since there are more rows than columns, it is sure to be a linear dependency between the rows of $A*$. The system should have only L equations, however, that is no problem because after $A*$ is reduced to its *row echelon form*, only L equations will remain. Since there is a larger set of rows, it is less probable that one cannot find a set of L rows that are *linearly independent*. Hence, a higher probability of $A*_{L \times L}$, being invertible.

Upon successfully solving the system of linear equations, the result is once again the set of intermediate symbols. The intermediate symbols can then be used to recover any missing source symbol, in a similar fashion to generating the repair symbols (see Figure

3.5), namely by: (1) using the “Tuple Generator” (by feeding it the ISI of the missing source symbol) to compute the set of intermediate symbols to be XORed, and (2) XOR those intermediate symbols which will result in the missing *source symbol*. All source symbols can be recovered through this process.

Permanent Inactivation Decoding

In the beginning of this chapter, it was said that one of the major reasons for RaptorQ’s superiority over previous Raptor codes, was a new technique that built upon *inactivation decoding*, called *permanent inactivation*.

Permanent inactivation overcomes many of the shortcomings of “dynamic inactivation” or “on-the-fly inactivation”. For permanent inactivation, we designate a subset of the intermediate symbols as already inactive before decoding starts – *permanently inactive* (PI) symbols. The algorithm chosen for solving the system of linear equations, has a major effect on the computational efficiency of the decoding, thus it should be an algorithm that takes advantage of the properties ensured by the chosen codes during the encoding process. The permanent inactivation technique provides some benefits: the overhead-failure probability curve of the resulting code constructed using permanent inactivation[8] is similar to that of a random binary fountain code, whereas the constructed decoder matrix potentially only has a small number of dense columns (compared with a random binary fountain code where all of the decoder matrix columns are dense). Permanent inactivation becomes even more compelling when we combine it with *High Density Pairity Check* rows defined over $\mathbb{F}_q$ for $q > 2$ (e.g., $\mathbb{F}_{256}$), because with a very high probability the decoding matrix will be full rank, whilst maintaining the decoding matrix largely sparse, consisting almost entirely of symbols over $\mathbb{F}_2$, with only a small number of symbols that are over a large field $\mathbb{F}_q$.

Decoding Schedule. The process of decoding using permanent inactivation is rather interesting, and is explained in some detail in on RFC 6330 [2]. At the heart of the decoder is the process of forming a *decoding schedule*. The decoding schedule consists of the sequence of row operations and row and column reordering during the Gaussian elimination process, and it only depends on $A*$ (and not on $S*$). Thus, the decoding of $Vector\ I$ from $Vector\ S*$ can take place concurrently with the forming of the decoding schedule, or the decoding can take place afterwards based on the decoding schedule.

[8]Note that to use permanent inactivation, the encoding symbols are generated differently, namely by the “Tuple Generator”.

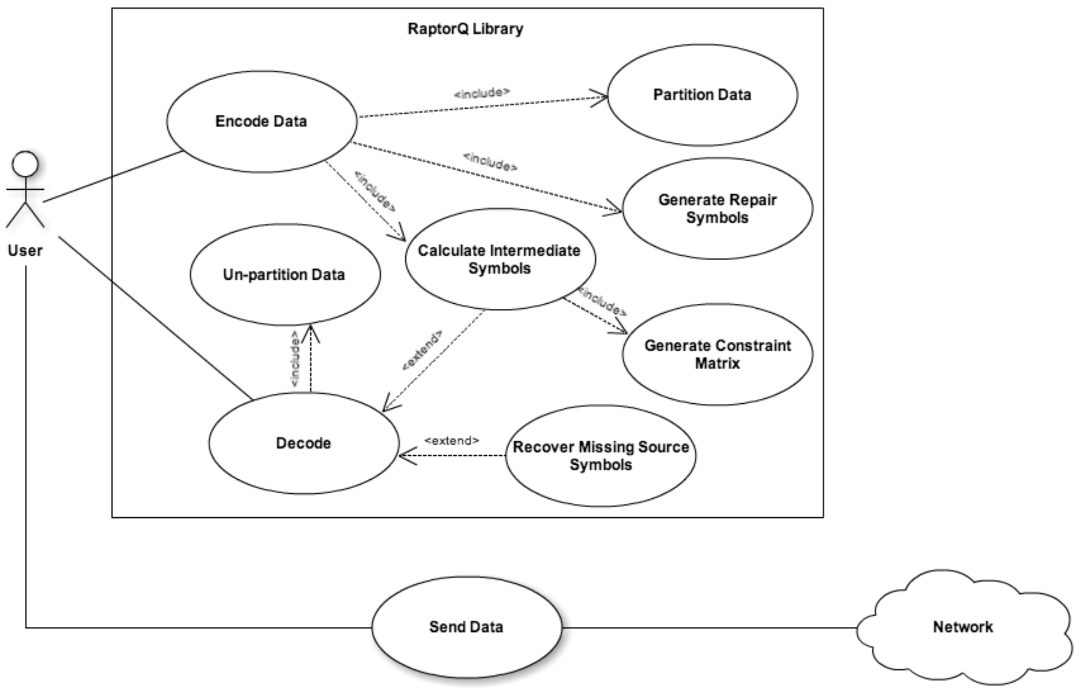


Figure 3.7: The main use cases for our library is encoding and decoding data.

3.3 Implementation

Since the code is relatively recent and the standard is complex, we are in the process of developing the first[9] public domain implementation of RaptorQ. The implementation of the library was made in Java SE 7[10].

Use Case Diagram. Figure 3.7 shows a diagram of the main use cases for using the developed RaptorQ library. Those are encoding and decoding data. The act of encoding data includes the action of partitioning such data into blocks, and calculating the intermediate symbols for generating the repair symbols. To calculate the intermediate symbols, generating the constraint matrix is necessary. If there are missing source symbols, the act of decoding the received encoding symbols requires calculating the intermediate symbols and recovering those missing source symbols. Unpartitioning the data is always required when decoding the set of received encoding symbols. Moreover, we can see that our library does not offer the necessary support for sending or receiving the encoded data, it is used only for encoding and decoding the data, the transport is up to the user.

[9]In our search, we found two very early implementations, far from complete: `http://code.google.com/p/libcatid/source/browse/trunk/src/codec/RaptorQ.cpp?r=1033` and `https://github.com/Meyermagic/RaptorQ-Python`. Both have not been updated in over a year.

[10]`http://www.oracle.com/technetwork/java/javase/overview/index.html`

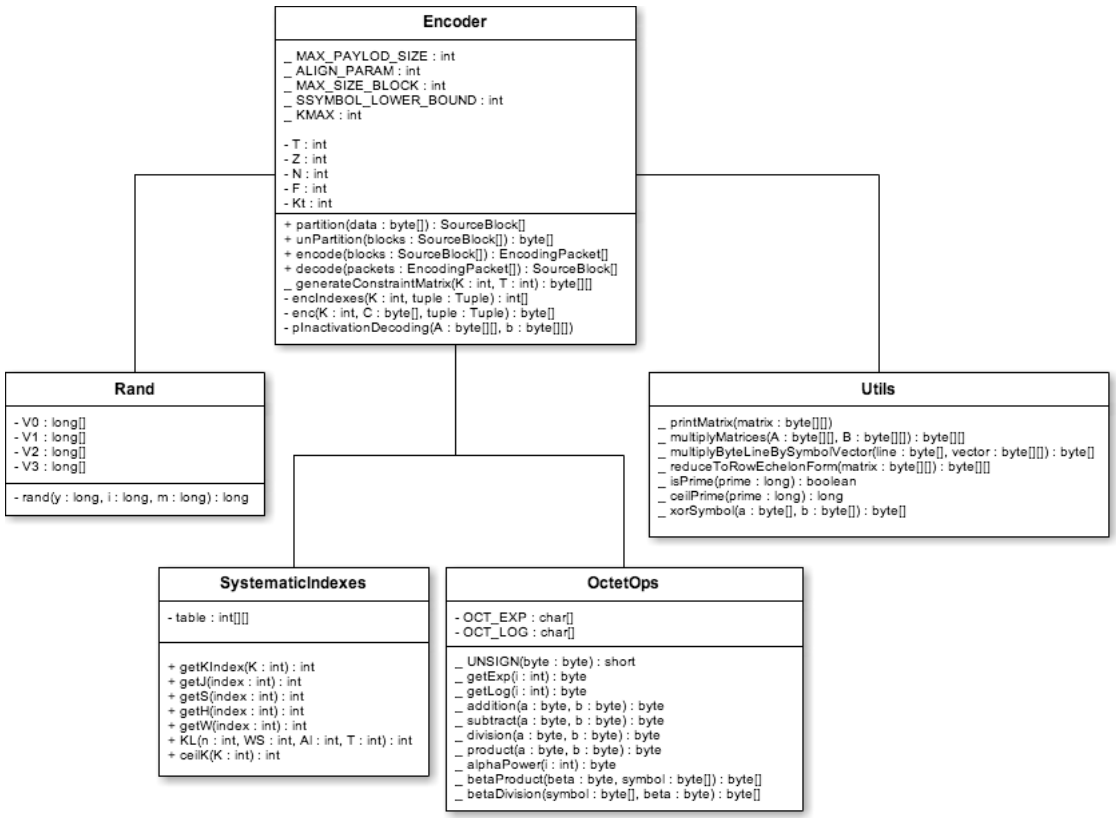

Encoder
_ MAX_PAYLOD_SIZE : int
_ ALIGN_PARAM : int
_ MAX_SIZE_BLOCK : int
_ SSYMBOL_LOWER_BOUND : int
_ KMAX : int
- T : int
- Z : int
- N : int
- F : int
- Kt : int
+ partition(data : byte[]) : SourceBlock[]
+ unPartition(blocks : SourceBlock[]) : byte[]
+ encode(blocks : SourceBlock[]) : EncodingPacket[]
+ decode(packets : EncodingPacket[]) : SourceBlock[]
_ generateConstraintMatrix(K : int, T : int) : byte[][]
- encIndexes(K : int, tuple : Tuple) : int[]
- enc(K : int, C : byte[], tuple : Tuple) : byte[]
- pInactivationDecoding(A : byte[][], b : byte[][])
Rand
- V0 : long[]
- V1 : long[]
- V2 : long[]
- V3 : long[]
- rand(y : long, i : long, m : long) : long
Utils
_ printMatrix(matrix : byte[][])
_ multiplyMatrices(A : byte[][], B : byte[][]) : byte[][]
_ multiplyByteLineBySymbolVector(line : byte[], vector : byte[][]) : byte[]
_ reduceToRowEchelonForm(matrix : byte[][]) : byte[][]
_ isPrime(prime : long) : boolean
_ ceilPrime(prime : long) : long
_ xorSymbol(a : byte[], b : byte[]) : byte[]
SystematicIndexes
- table : int[][]
+ getKIndex(K : int) : int
+ getJ(index : int) : int
+ getS(index : int) : int
+ getH(index : int) : int
+ getW(index : int) : int
+ KL(n : int, WS : int, Al : int, T : int) : int
+ ceilK(K : int) : int
OctetOps
- OCT_EXP : char[]
- OCT_LOG : char[]
_ UNSIGN(byte : byte) : short
_ getExp(i : int) : byte
_ getLog(i : int) : byte
_ addition(a : byte, b : byte) : byte
_ subtract(a : byte, b : byte) : byte
_ division(a : byte, b : byte) : byte
_ product(a : byte, b : byte) : byte
_ alphaPower(i : int) : byte
_ betaProduct(beta : byte, symbol : byte[]) : byte[]
_ betaDivision(symbol : byte[], beta : byte) : byte[]

Figure 3.8: Class diagram of the most relevant classes of the RaptorQ library.

Class Diagram. Figure 3.8 shows a class diagram of the principal classes that were implemented in the RaptorQ library. The most relevant class is the `Encoder` class, its instance will interface with the user. Its main methods are for partitioning, unpartitioning, encoding and decoding the data. Those are the methods that the user will most likely invoke. The `Encoder` class resorts to four "helper" classes: the `Rand` class is responsible for one of the pseudo-random generators; the `SystematicIndexes` class stores the table with the parameter information for each K', and provides the methods for lookups and auxiliary methods such as ceiling K; the class `OctetOps` offers methods for the arithmetic operations on octets (i.e., over finite fields); finally, the `Utils` class provides some utilitarian methods, such as operations on matrices.

Sequence Diagram - Encoding Process. Figure 3.9 is a top-level depiction of the encoding process: the user interacts with the `Encoder` class, first partitioning the data into blocks, and then proceeds to encode the blocks. The process of encoding the blocks consists of building the constraint matrix for the system of linear equations. The constraint matrix is composed by a few sub-matrices, namely the sub-matrix that represents the LT code, which stores the indexes of the intermediate symbols that must be XORed to generate the source symbols. The next step is to solve the system of linear equations, for that RaptorQ employs the technique of *permanent inactivation decoding*. The last step

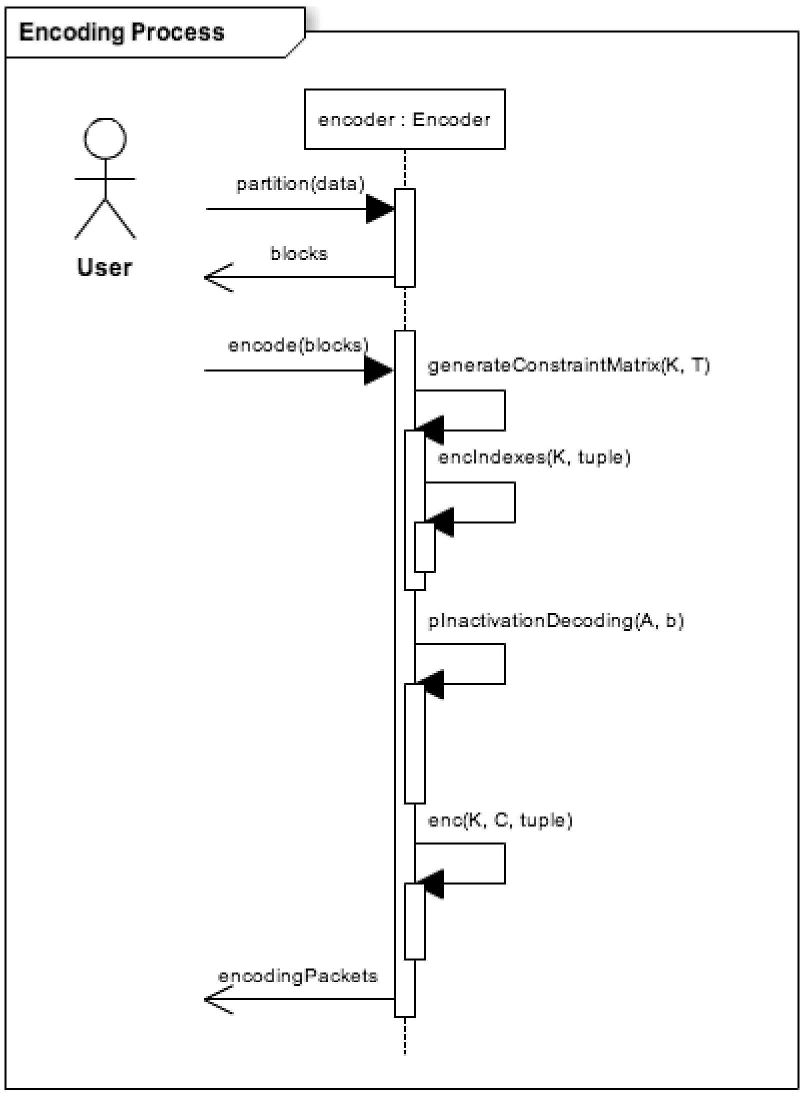


Figure 3.9: Sequence diagram describing the encoding process for RaptorQ.

of the encoding process is to generate the repair symbols: by encoding the intermediate symbols.

Sequence Diagram - Decoding Process. The decoding process is represented in Figure 3.10. The first step is to analyze the received encoding symbols, to see if any source symbols are missing, and if so, if enough repair symbols have been received. If all the source symbols are received, the decoding of that block is finished and the source block can be returned. If source symbols were lost during the transmission, a process very similar to the encoding process takes place. The constraint matrix is built, but the lines corresponding to the missing source symbols are replaced by lines for the received repair symbols. The next step is to solve the system of linear equations. If the system is inconsistent, the decoding fails and the source block is not recovered. Otherwise the intermediate symbols are calculated, and can then be used to recover the missing source symbols.

3.3.1 Evaluation

As previously mentioned, one of RaptorQ's greatest advantage is its steeper overhead-failure curve. Basically, it is extremely rare for the decoding process to fail, which is very important as this type of codes may be used in mission critical systems and scenarios. This section presents some results for the failure probability of our implementation of the RaptorQ standard, and compare it to the evaluation found in Appendix B.3 of [16]. This helps validate the results obtained in [16], but also ensures that our implementation is correct, since a minor difference from the standard could gravely affect the failure probability.

The methodology used was the following: for the values of K equal to 10, 26 and 101, we encoded random input data, and then forced a random loss of 10$\%$, 20$\%$, 50$\%$, 60$\%$ and 85$\%$ of the encoding symbols. Then, decoding was attempted with the received encoding symbols. Furthermore, we did experiments with different overheads. An overhead of 0 means that decoding is attempted after K encoding symbols are received (for an overhead of 1 and 2, this would mean $K + 1$ and $K + 2$ encoding symbols, respectively). Each test was repeated between 20 million and 30 million times, to get a reasonable level of confidence in the results. This is not a performance benchmark, and these results should be reproducible in any machine (but may take longer to calculate). However, for completeness's sake, the machine where the experiments were carried out is a Dell PowerEdge R410:

- Intel Xeon E5620 @ 2.40GHz
- 32GB of DDR3 RAM
- Ubuntu Server 64bit (kernel 2.6.32-21)

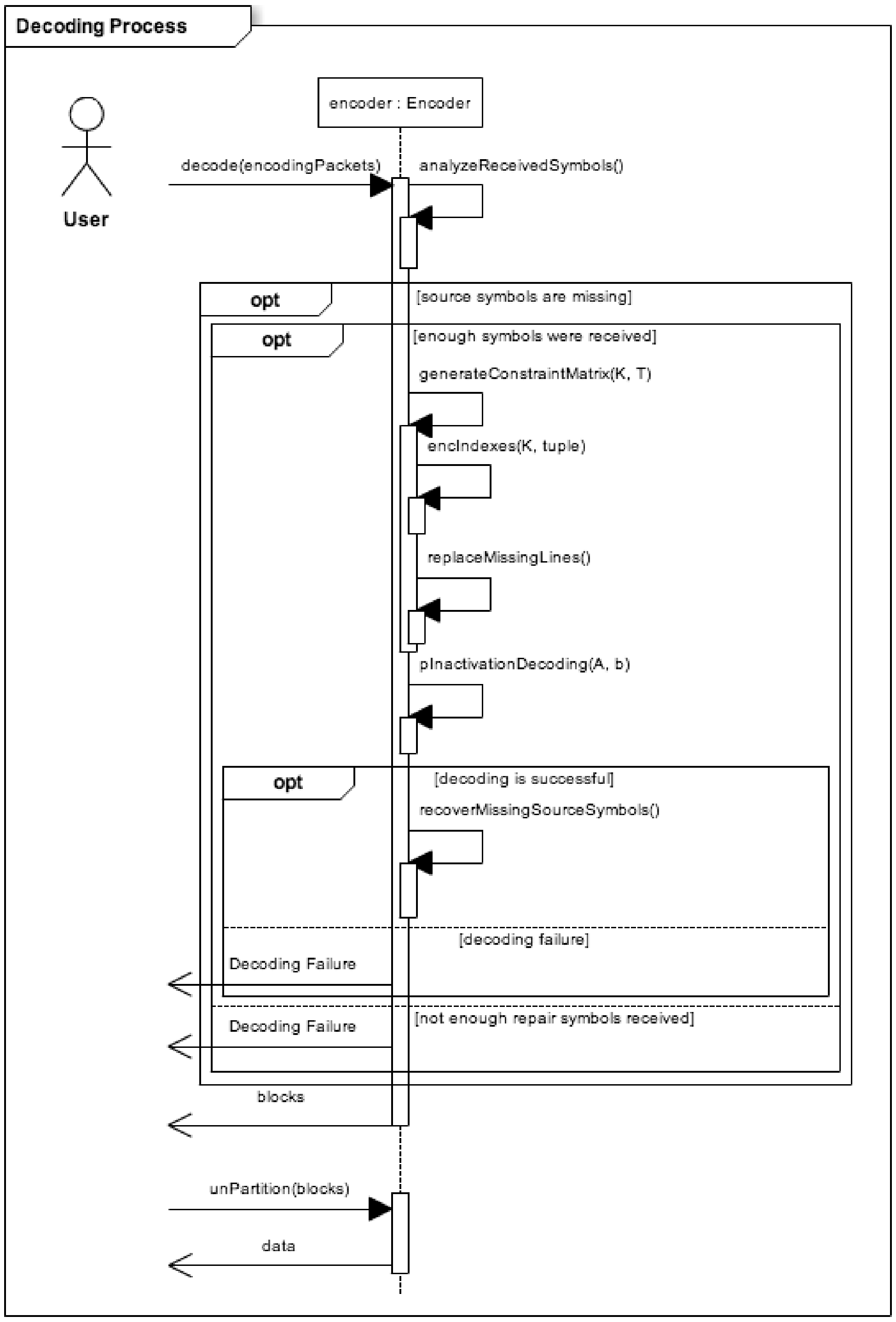


Figure 3.10: Sequence diagram describing the decoding process for RaptorQ.

	K								
	0 overhead [$\cdot 10^{-3}$]			1 overhead [$\cdot 10^{-5}$]			2 overhead [$\cdot 10^{-7}$]		
Loss	10	26	101	10	26	101	10	26	101
10%	0	5.4	5.7	0	0	3.8	0	0	2.5
20%	0	4.0	4.8	0	2.3	2.4	0	0	0.5
50%	0	3.9	4.9	0	1.6	2.5	0	0.9	1.2
60%	4.8	4.1	4.9	0	1.5	2.2	0	0	2.1
85%	0	12.7	4.7	0	0.8	2.4	0	0	1.3

Table 3.1: Decoding failure probability, for a code overhead between 0 and 2 symbols, a network loss rate between 10% and 85%, and K equal to 10, 26 and 101.

The results are displayed in Table 3.1. They confirm the reliability claimed by the RaptorQ standard, as the failure probability is very small in all experiments. Furthermore, in some tests, we never observed decoding failure. For $K = 10$, we only saw failed decodings for 60% loss with 0 overhead. The reason behind this phenomenon may become clearer when we discuss our attack, but it is associated with the periodic nature of the RaptorQ standard (which we will further explore in the next chapter). Additionally, we can see that for 2 overhead symbols, the probability must be in the lows 10^{-7} because repeating the tests up to 30 million times was not enough to get results with an acceptable level of confidence: for the cases when we actually got a decoding failure, it was once or twice in almost 30 million tests. These results fall in line with the ones presented in [16].

Figures 3.11, 3.12 and 3.13 are graphs for the decoding failure probability for 0, 1 and 2 overhead symbols, respectively. By isolating the results this way, it can be seen that, independently of the overhead used, higher values of K have higher failure probability. Looking at Appendix B.3 of [16], one can see that this behavior happens for values of K lower than 100. For values of K in the hundreds the probability of failure stabilizes, and in the thousands the probability not only is somewhat stable but is actually lower than in the hundreds. To make a more in-depth analysis of the behavior of the decoding failure probability more (higher) values of K should have been tested. However, this is not the objective of this work, and would be going out of its scope. The intention (and what should be retained from these results) is only to validate RaptorQ's very low decoding failure probabilities to better comprehend the impact that an attacker may or may not have on its robustness.

3.3.2 Implementation Obstacles

As reference for the implementation, IETF's RFC 6330 [2] was used, but sometimes the book "Raptor Codes" from Luby and Shokrollahi [16] helped in understanding the reasoning behind a few aspects of the construction of the code. By the nature of both documents, RFC 6330 is more objective, while the book has a more pedagogic approach:

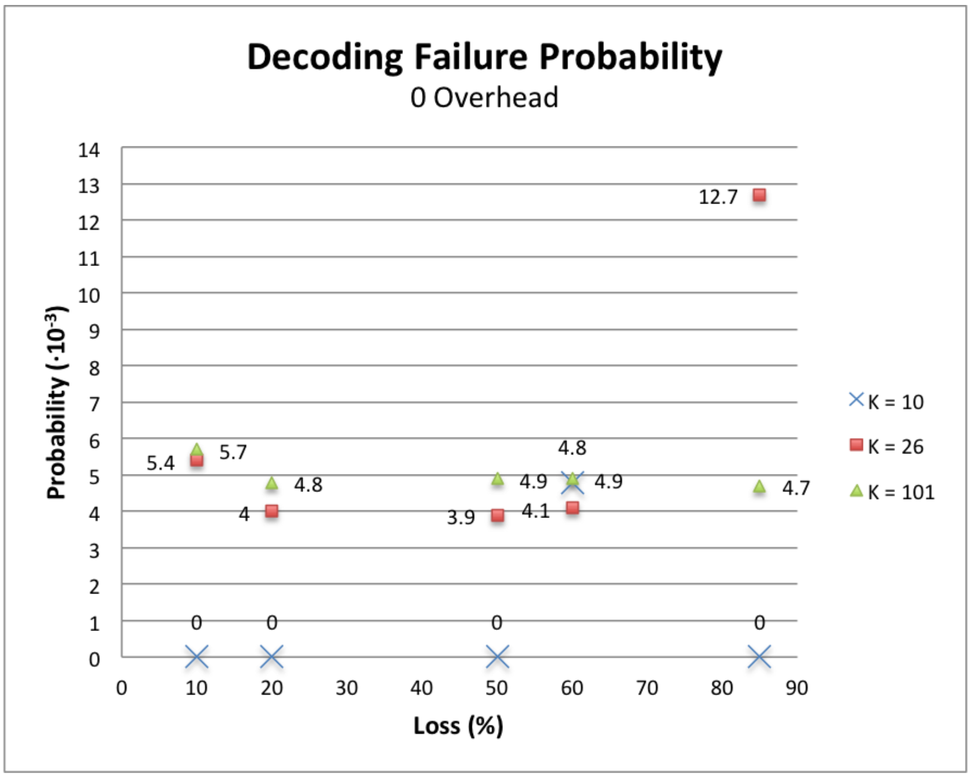


Figure 3.11: Graph of the decoding failure probability results for 0 overhead symbols.

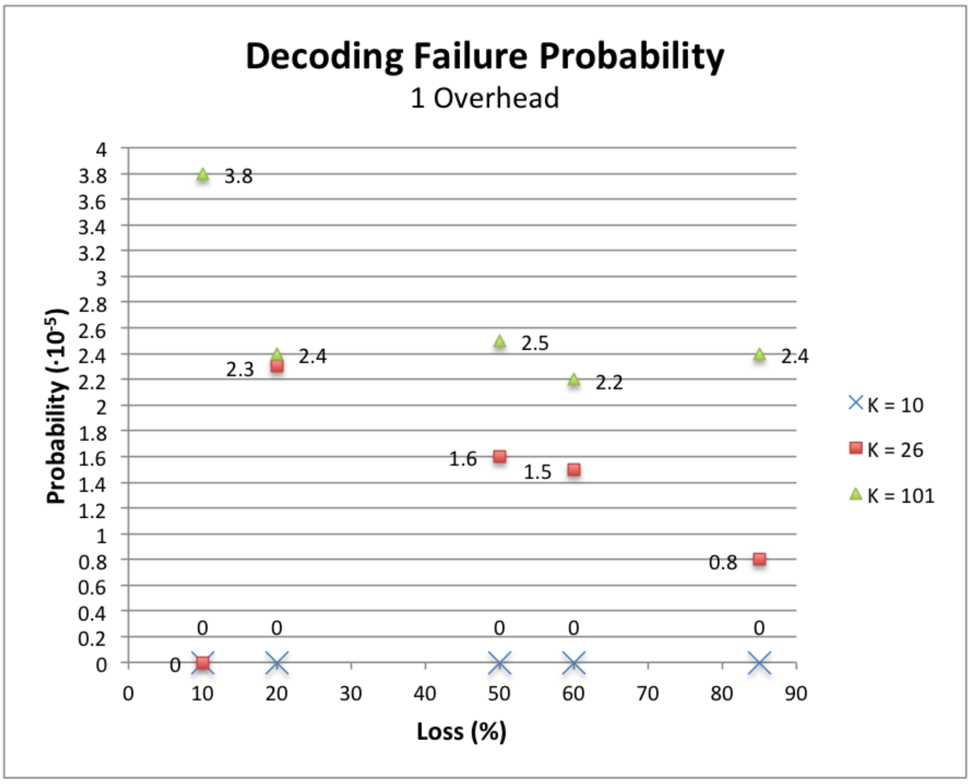


Figure 3.12: Graph of the decoding failure probability results for 1 overhead symbols.

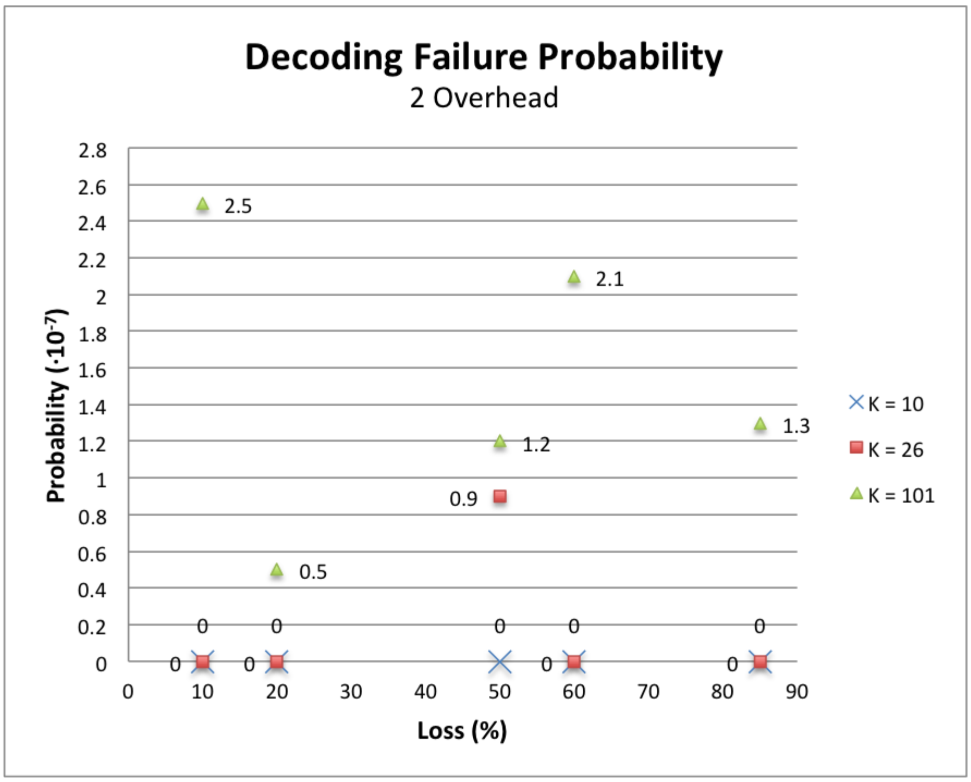


Figure 3.13: Graph of the decoding failure probability results for 2 overhead symbols.

the authors explain the reasoning behind certain options (resorting to demonstrations and examples), which eases the comprehension.

In some cases, IETF's RFC 6330 was not very clear about a few aspects, leaving space for some ambiguity and doubt. For instance, in our view, the construction of the sub-matrices G_{LPDC} 1 and 2 of $Matrix\ A$ for the encoding and decoding processes, is much easier to comprehend following the book than IETF's RFC 6330. In fact, during our research we actually found someone[11] who quit implementing RFC 6330, and turned back to IETF's RFC 5053 [1] (R10), because of this very issue. Regarding IETF's RFC 6330, the most common issue was that, due to the objective nature of the document, most of the times there was a lack of "connection" between the different parts of the specification. This is where the book "Raptor Codes" came in and helped us understanding the "big picture", to see how each piece of the specification fitted together.

Definitely the greatest obstacle we had to overcome was the lack of support. The latest version of IETF's RFC 6330 presently[12] is from August 2011, roughly 2 years old. These codes' success depends largely on their adoption by various standardization entities. This is a process that takes its time, so RaptorQ is a relatively new code. Consequently, it has

[11]http://stackoverflow.com/questions/6504759/raptorq-fec-implementation-obstacle
[12]http://tools.ietf.org/html/rfc6330

been mostly out of the public's eye. Qualcomm has a commercial solution[13] that uses the RaptorQ technology, however RaptorQ is far from widely known. As a consequence, it is very difficult to find any sort of support because the people that could offer some support are not in the public. When dealing with cutting edge technology and innovation, this kind of obstacle is a natural "occupational hazard". However, since this was by far the greatest challenge we faced during the development of the RaptorQ library, we find it to be noteworthy.

[13]`http://www.qualcomm.com/solutions/multimedia/media-delivery/raptor-technology`

Chapter 4

Breaking the RaptorQ Standard

"There is nothing like looking, if you want to find something. You certainly usually find something, if you look, but it is not always quite the something you were after."

— The Hobbit, J. R. R. Tolkien.

4.1 The Attack

Probably, one of the most interesting properties of FEC codes is the ability to use the same FEC packets/symbols to simultaneously repair different independent packet losses at multiple receivers. *Independent* packet losses must be emphasized, as recovery should be completely independent of loss patterns (e.g., a burst loss). The book *Raptor Codes* [16], written by two of the authors of IETF's RaptorQ RFC 6330 [2], includes the following text:

> ... we will assume that the set of of received encoded symbols is independent of the values of the encoded symbols in that set, an assumption that is often true in practice. These assumptions imply that for a given value of k, the probability of decoding failure is independent of the pattern of which encoded symbols are received and only depends on how many encoded symbols are received.

We believe that it is possible to break that assumption, since it was considered for *benign* environments.

Successful attack. First, let us define a successful attack. The objective of the code is to correct network erasures, which means is to recover the original source symbols that were not received, without the need for retransmission. A successful attack corresponds to the case when a malicious adversary can prevent, the recovery of the missing source symbols. Therefore, the receiver is unable to obtain one (or more) of the source symbols, and cannot fully recover the original data (that should have been transmitted).

Adversary. It is assumed an adversary with network control that can arbitrarily intercept and drop any network packet (e.g., with an infected router or a malicious proxy server).

4.1.1 Rationale

The attack is based on the construction of the RaptorQ code (see Section 3.2). More specifically, it exploits the system of linear equations used for the encoding and decoding processes, and the identification of the symbols (ISIs).

To successfully attack the code, it is necessary to cause the decoding process to fail. In practical terms, the attacker must hinder the calculation of the intermediate symbols. The reasoning behind this is simple: if the decoder calculates the intermediate symbols, then the decoding process, although not finished, is definitely successful – every source symbol can be recovered without the need for more packets to be transmitted.

Fortunately for the attacker, she only needs to prevent one of the source blocks from being recovered, since the encoding and decoding processes are independent for each source block. Therefore, by avoiding one source block from being recovered, it is enough to prevent the recovery of the whole original data.

One simple solution to forcefully cause a decoding failure would be to drop one of the source symbols and all of the repair symbols, assuming the use of a systematic Raptor code. In the case of an non-systematic Raptor code, one could also simply drop all packets. These would be obvious *Denial-of-Service* (DoS) attacks. They are inelegant, and can be trivially detected (e.g., with an intrusion detection system).

As discussed in Section 3.2.2, the intermediate symbols are calculated by solving a system of linear equations. Therefore, the attacker's objective should be to prevent the resolution of the system of equations. There are three possible outcomes from solving a system of linear equations:

1. The system is consistent and well determined, and thus has a single unique solution;
2. The system is consistent but underdetermined and has infinitely many solutions;
3. The system is inconsistent (a.k.a. overdetermined) and thus has no solution.

The first case represents a successful recovery of the intermediate symbols and, consequently, a successful decoding process. Hence, the second and third cases are the ones the attacker is interested in (because they represent a decoding failure). Usually, a system of linear equations is consistent but underdetermined when the number of equations is lower than the number of unknowns, and a system is inconsistent if it has more equations than unknowns.

In more practical terms, and since this system of linear equations corresponds to matrix operations, for a *coefficient matrix* $A_{m \times n}$ and an *augmented matrix* $Ab_{m \times (n+1)}$ we have:

1. `rank`(A) = `rank`(Ab) & `rank`(A) = n ⇒ consistent and determined;
2. `rank`(A) = `rank`(Ab) & `rank`(A) < n ⇒ consistent but underdetermined;
3. `rank`(A) ≠ `rank`(Ab) ⇒ inconsistent.

This implies that the attacker must change the rank of the system's matrix. It is out of her grasp to raise the rank of the matrix. However, she might be able to lower it. Since it is irrelevant for the success of the attack if the decoding process fails because the system is inconsistent or underdetermined, it is enough to lower the rank of the *coefficient matrix*.

Since the attacker has only network control, i.e., she does not control the machine where the decoding process is running, she must do this by selecting which packets may

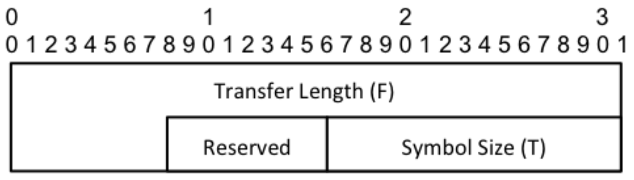


Figure 4.1: RaptorQ's Common FEC Object Transmission Information (OTI).

pass, or, by modifying them. The latter is not very attractive because not only it requires reverse engineering the messages (we would like to keep the attack implementation independent as much as possible) but also, it would not work if communication is encrypted or/and made through secure channels (e.g., IPsec [44]). So, how can we attack the RaptorQ standard, without having to understand or modify the messages content?

The answer to that question is on the way the standard identifies each symbol. IETF's RFC 6330, which describes the RaptorQ Raptor code, says that: the symbols' identifiers, ESI and ISI, are sequential and start at 0.

Since the attacker has network control, and the standard's recommendation is to send one[1] symbol per network packet, the attacker can count from the first packet (ESI and ISI of value 0), the packets that go by and their respective ESI. However, both the encoding and decoding processes take into account the value of the ISI, not ESI. Obviously, the padding should not be transmitted through the network, so the attacker would not be able to know the difference between the source symbols and repair symbols. This could hinder the attack.

However, RFC 6330 describes a *Common FEC Object Transmission Information* (OTI) format that can be seen in Figure 4.1. This OTI packet is used to transfer the necessary information from the encoder to the decoder, so it can calculate the necessary parameters for decoding (e.g., K and K'). By intercepting this packet, the attacker could obtain the necessary information (*Transfer Length* and *Symbol Size*) to determine K, thus being able to know the ISIs of each symbol passing through the network by only counting the packets.

If the implementation does not follow the standards and uses a different format, then some reverse engineering may be in order. If the implementation does not send an OTI packet at all and just "assumes" that the decoder knows the value of K, then it might be reasonable to assume that the attacker also knows the value of K. If it is not, then the attacker may try a technique similar to the one presented in Section 4.4, where the possibility of attacking over secure channels is discussed.

There are more practical considerations to have in mind when planning this attack

[1]IETF's RFC 6330 [2] "`RECOMMENDS`" (in allusion to the terminology introduced in IETF's RFC2119 [45]) that "each packet contains exactly one symbol". This is a common practice as this way a discarded packet only affects a single symbol.

because the encoder and decoder offer flexibility through some other parameters (e.g., the maximum size block that is decodable in working memory). The RFC does (for the most part) suggest default values for those parameters, as do other standards and technical specification texts.

How does the knowledge of the ISI help the attacker? Since all aspects of the code are standardized, as long as the target implementation follows the standard, the attacker may calculate the ISIs of the necessary combination of missing source symbols and received repair symbols to force the decoding to fail (as it would, very rarely, when facing accidental faults). Basically, the attacker continuously causes the accidental faults that would only rarely occur.

4.2 Proof of concept

In our process of breaking the RaptorQ standard, we started by confirming that our line of thought could be implemented in practice before investigating on how to make it efficient. Thus, this section describes a proof of concept solution and the results obtained from it.

The assumption is that the adversary has some sort of network control, which in turn means that she can decide what symbols arrive at the receiver. Thus, she can drop one of the source symbols and all the repair symbols that would replace it (in the system of linear equations), until she sees one that would render the system of linear equations inconsistent - i.e., a repair symbol whose pre-coding constraint (line in the *decoding matrix*) is *linearly dependent* of another equation in the system of linear equations. As a result, the adversary would have decreased the *decoding matrix's rank*, rendering the system of linear equations inconsistent. Hence, the decoding would fail.

Example. Let us look at Figure 4.2. Assuming a scenario such as the one depicted, with $K' = 10$ (10 source symbols) and 3 repair symbols, an example of a successful attack would be the following: the attacker *drops* the first ($ISI = 0$), fifth ($ISI = 4$) and ninth ($ISI = 9$) packets; and when the receiver replaces the lines corresponding to those symbols (in $Matrix\ A$) by the ones relative to the received repair symbols, she would have introduced a linear dependency between the lines of the $Matrix\ A$, lowering its rank and rendering the system of equations inconsistent.

It is very interesting to take notice that the attack is completely independent of the data being transmitted. The pre-coding constraint corresponding to a repair symbol is generated based only in K' and the symbol's ISI. Therefore, the attack is based fundamentally on how the standard identifies the symbols, which potentially allows the exploitation of communications using encrypted packets, such as when packets are transmitted over IPsec[44].

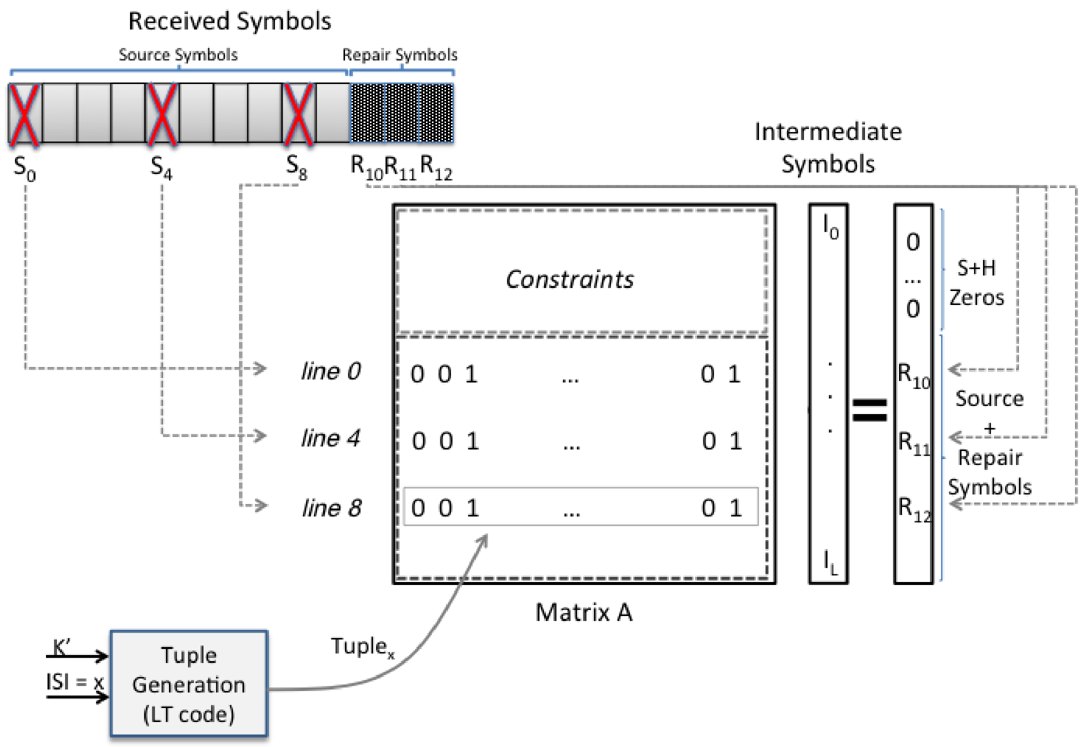


Figure 4.2: Example attack for K' = 10, 10 source symbols and 3 repair symbols.

4.2.1 Evaluation and Discussion

Since the attack drops all repair symbols but the ones that will cause a linear dependency among the equations, this may require many network packets to be eliminated. If the number of eliminated packets is high above the average packet loss for that particular network/system, the attack can be easily detected. Consequently, it would be interesting to investigate how many packets must be deleted, for different scenarios.

A scenario was considered where the sender application is streaming information to the receiver. In the experiment, 28 different values for K' were tested. For each test, the last source symbol[2] is deleted, and replaced with repair symbols until the *decoding matrix's rank* was decreased. In greater detail, the experiment is as follows: (1) the constraint matrix, $Matrix\ A$, is generated; (2) the last row of the matrix (which corresponds to the LT code for the last source symbol) is replaced by the LT code of the following repair symbols (i.e., if the last symbol is ISI = 9, it is replaced by the LT code for ISI = 10, 11, ...); (3) every time the row is replaced, the matrix is reduced to its row echelon form; (4) if there are rows constituted only by 0's, then there was a linear dependency amongst the rows (thus, at the time of decoding the system of linear equations would be inconsistent); if not, then (5) the original matrix is retrieved and the next repair symbol (its ISI) is tested.

The tests were run always with 0 overhead symbols. Furthermore, for each test, it was

[2]Which corresponds to the last equation in the system.

Tries / K	10	26	32	42	55	62	75	84	91	101	153	200	248	301	355	405	453	511	549	600	648	703	747	802	845	903	950	1002
1	43	115	266	2	127	117	430	390	212	63	179	70	42	66	119	187	207	488	10	36	192	213	339	10	189	302	663	1185
2	174	1173	484	195	154	168	481	399	237	1105	433	313	93	244	235	406	237	681	128	98	606	485	513	794	297	449	695	1788
3	224	1250	734	456	161	315	584	936	294	1321	528	375	312	576	244	557	537	705	345	331	639	898	1128	829	370	580	886	1804

Table 4.1: Number of encoding symbols that must be lost.

counted how many symbols needed to be lost to successfully attack up to three times. That is, looking at Table 4.1: for K' = 10, 1 source symbol (the 10^{th}) and 42 repair symbols were dropped in order to force a decoding failure; more 131 repair symbols (totaling 174 packets) were eliminated to force a second decoding failure; and finally, another 50 repair packets (total-ling 224 packets) were lost to attack the code for a third time.

Table 4.1 shows that the number of *encoding symbols* that had to be deleted for each K' vary a lot, from hundreds to just 2. This is because these are *independent events*. Sometimes the number of encoding symbols that must be dropped is very high, meaning that such an attack would be more conspicuous. But still, this demonstrates that the RaptorQ standard can be broken when facing malicious faults.

It should be noted that it would be scientifically relevant to also present results for overheads of 1 and 2 symbols. The reason why this was not done is simple: for many of those values, we could not find the set of encoding symbols that should be lost in order to force a decoding failure. Given the very low probabilities of decoding failure that were presented in Table 3.1, this is comprehensible. Note that only one of the source symbols was removed, allowing for only one repair symbol to take its place, and this source symbol is fixed – it is the last source symbol. Thus, this attack is very limited.

4.3 Refined Attack

The proof of concept confirms that our motivation was well founded. However, the results presented in Table 4.1 are still too high for many of the tested values of K', and do not contemplate the cases when overhead symbols are used in the decoding process. Thus, the attack should be refined to make it more viable.

Since the proof of concept attack only replaced the last source symbol, an obvious way to increase the chances of introducing a linear dependency in the set of equations is to replace the other source symbols. This would allow the discovery of the one that requires less encoding symbols to be lost. But why stop there? Why not try to increase the chances even further, by dropping more than one source symbol? One can even try replacing each combination of source symbols, with different combinations of repair symbols. This way, it is ensured that every possible case is considered. Hence, a scenario could be found where much less encoding symbols needed to be dropped. Naturally, given the brute force nature of this attack, it would result in a very high number of combinations, which could prevent results from being obtained in an useful time frame, due to the massive number of computations that would be needed.

An approximation to this idea would be an algorithm like the one described in Algorithm 1. The algorithm receives two parameters: (1) $upperLimit$ - the maximum number of repair packets the attacker is willing to drop; and (2) K - the number of symbols in an extended source block (a.k.a., the K'). The former is useful to determine when to terminate the algorithm, giving some parametrization to how much time and computation the attacker is willing to spend. Moreover, it can parametrize the "risk" of the attack i.e., if the attacker drops too many symbols, the attack may be easily detected (it is interesting to keep the number of dropped packets as low as possible, so the attack is stealthy). The latter tells us how many source symbols there are, and is also needed to construct the constraint matrix.

Let us look at Algorithm 1 in greater detail. In lines 2 to 4, the array `targetRepairs` is populated with the ISIs of the repair symbols that are available for this attack. In lines 5 to 7, the array `targetLines` is populated with the ISIs of the source symbols that can be targeted to be eliminated. In lines 8 to 23 is where the experimentation occurs. Starting at 1 target source symbol and incrementing until K, all the combinations of target source symbols are stored in the variable `combinationsOfLines` (line 9). Then, for every combination of target source symbols (lines 10 to 22), the combinations of available repair symbols are tested. The variable `combinationsOfISIs` stores all the combinations of available repair symbols for the number of target source symbols being tested at that moment (line 11). Finally, for each combination of target source symbols, the target source symbols are replaced by every combination of available repair symbols for that number of target source symbols (lines 12-21). The test is as follows: (1) the matrix rows corresponding to the repair symbols being tested are generated; (2) the constraint matrix is generated; (3) the matrix rows corresponding to the target source symbols are replaced by the rows corresponding to the repair symbols being tested; (4) the matrix is reduced to its row echelon form; (5) if the rank of the matrix is lower than L, then the attack tested was successful. If the algorithm finds an attack that does not imply dropping more than $upperLimit$ packets, by the time it finishes it will have printed all the attack vectors found for that value of K.

Algorithm 1 Breaking the RaptorQ code standard.

```
procedure ATTACK($upperLimit$, $K$)
    for $ISI \leftarrow 0, upperLimit + K$ do
        $targetRepairs[ISI] = ISI + K$
    end for
    for $symbol \leftarrow 0, K$ do
        $targetLines[symbol] = symbol$
    end for
    for $lines \leftarrow 1, K$ do
        $combinationsOfLines \leftarrow \binom{targetLines}{lines}$
        for all $setOfLines$ in $combinationsOfLines$ do
            $combinationOfISIs \leftarrow \binom{targetRepairs}{lines}$
            for all $setOfISIs$ in $combinationsOfISIs$ do
                (1) Calculate repair lines corresponding to the ISIs in $setOfISIs$;
                (2) Generate the constraint matrix;
                (3) Replace the lines in $setOfLines$ with the repair lines;
                (4) Perform Gaussian elimination to reduce to row echelon form.
                if $rank < L$ then
                    print($setOfLines$)
                    print($setOfISIs$)
                end if
            end for
        end for
    end for
end procedure
```

Note that all of this computation may be done before hand, in order to make the attack extremely fast (i.e., without introducing detectable lag into the communication) and drop the computational requirements of the infected machine to a bare minimum. All the infected machine needs to do is get the target ISIs from a source (e.g., a file) and drop the ISIth packets in the case of source symbols, and only let the ISIth packets pass in the case of repair symbols.

4.3.1 Results

Algorithm 1 was implemented (with some minor efficiency tweaks) and run for the same values of K tested in the proof of concept attack. For each value of K, the attack was experimented against 0, 1 and 2 overhead symbols, and the number of packets that had to be dropped was counted. If the number of dropped packets is high above the average

Overhead \ K	10	26	32	42	55	62	75
0	3	3	2	2	2	2	2
1	7	4	6	2	4	3	4
2	20	41	24	10	20	12	51

Overhead \ K	84	91	101	153	200	248	301
0	2	1	2	2	1	2	3
1	6	8	7	3	8	4	19
2	7	22	19		190		

Overhead \ K	355	405	453	511	549	600	648
0	2	2	1	1	1	1	1
1	24	8	31	36	38	190	2
2							

Overhead \ K	703	747	802	845	903	950	1002
0	1	1	1	1	2	1	101
1	178	8	143	11	18	6	82
2							

Table 4.2: Number of encoding symbols that must be lost.

packet loss for that particular network/system, the attack can be easily detected. Thus, since attackers normally want to be as stealth as possible, the practicality of the attack can be measured by how low is the number of packets dropped.

The results are presented in Table 4.2. As can be seen, it was possible to find combinations of missing source symbols and received repair symbols, without having to lose many packets. Note that in Section 3.3.1, the failure probability for the 0 overhead tests was in the order of 10^{-3}, for 1 overhead of 10^{-5}, and for 2 overhead symbols it was in the lows 10^{-7}.

We are still in the process of collecting the missing values to fully fill Table 4.2. The algorithm to compute the attack, on the one hand, ensures the best possible results, but on the other hand, is very time consuming due to the extremely large amount of combinations considered.

In spite of that, one can infer some conclusions from the results that have already been collected. This attack causes a decoding failure probability of 100% by requiring most of the times less than 1%[3] of the total number of packets to be eliminated. Just by carefully picking the source symbols to drop and the repair symbols to pass, the attacker can have a massive impact on the failure probability, completely destroying the robustness shown for accidental faults. In addition, she has to do this only for *one* source block. So, if she was attacking a communication that used the latest RaptorQ code, parametrized with $K = 648$ and 0 overhead symbols, she would only have to eliminate 1 symbols (0.15$\%$ of the total number of packets transmitted) of one of the source blocks, in order to hinder

[3]Considering an overhead of 0 repair symbols.

the communication. Keeping in mind that the probability of that happening by accident, would be in the order of 10^{-3} for each source block. If $K = 648$ and 1 symbol of overhead was used, she would have to eliminate only 2 symbols (0.31% of the total number of packets), to force a decoding failure, that, if it were to occur by chance, the probability would be in the order of 10^{-5}.

Attack 4.1 is the output of our experiment for $K = 10$ and 0 overhead symbols. It contains the information on the attack vector found, namely:

- The lines of constraint matrix that need to be replaced;
- The ISIs of the source symbols that must be eliminated;
- The ISIs of the repair symbols that must be used;
- The total number of encoding symbols lost;
- The rows corresponding to the repair symbols that must be used, which need to replace the target rows in the constraint matrix.

More attack vectors such as the one presented in Attack 4.1 can be found in Appendix A.

Attack 4.1: Attack vector for $K = 10$ and 0 overhead

```
- K: 10
- Overhead: 0
- Epsilon: 0.1

Target lines: [17, 21, 25]
Target  ISIs: [0, 4, 8]
Payload ISIs: [10, 11, 12]
Body count  : 3 (30.0%)


------ PAYLOAD LINES ------

[0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,0,0,0,0,0,0,1,1,0,0]
[0,0,1,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,1,0]
[0,1,1,0,1,0,1,1,1,1,0,1,0,1,1,0,1,0,0,0,0,1,1,0,0,0,0]

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

4.4 Attacking over secure channels

Raptor codes have been used for years in broadcast networks [33, 34, 35], standardized in IETF's RFC 5053 [1] and RFC 6330 [2]. In addition, they have been widely adopted by the military for mission critical systems/operations, and for scenarios where communication may be intermittent and/or with high loss rates (e.g., after natural disasters). Due to the criticality of the scenarios where these codes are used, it is not only relevant to study their resilience and dependability in plain-text channels, but also when communication is made over secure channels, such as IPsec [44]. This is important because in critical scenarios the codes might be used together with protection mechanisms.

The attack conceived in the previous sections is directed at the design of the code's standard, not the message's content. Namely, it exploits the sequentiality of the ISIs (that always begins at 0), which are then used as a seed (together with K') to the tuple generator that is employed to construct the system of linear equations. Therefore, without having to look inside the message's content, better yet, without even the need of messages being transmitted (precomputing), an attacker can foresee, for each value of K', which set of (ISIs of) encoding symbols would cause a failure in the decoding process.

When using encrypted messages, for example, in a secure channel, the attack is in theory just as viable. However, in practice there could be some difficulties: (1) the attacker needs to know the value K', because it is crucial to determine the attack vector that should be applied; (2) the packets may be unordered, so the attacker will not be able to know if a packet is the i^{th} packet. In what regards to the latter, for the remainder of this section FIFO channels are assumed.

In some deployment cases, it might be reasonable to assume that the attacker knows the value of K'. If that is the case, the attack can be executed as described in the previous sections, without further work needed by the attacker. It may also be reasonable to assume that the value of K' is one amongst a small set of values, and in this case the attacker needs to try the attack for the various possible values of K', until the attack is successful.

However, in the cases where the attacker has no idea which value of K' is being used, the attack may be more difficult to execute, and require more work from the attacker. A technique that may be applied is as follows: the encoding and decoding processes are independent for each source block; thus, it is reasonable to assume that, from the network perspective, there will be a noticeable lapse between the packets (i.e. encoding symbols) of one source block and the next source block. As long as the attacker is able to detect such a lapse between the network packets from on source block to another, she will be able to perform the attack. Let us deepen our reasoning for that by looking back at the same example presented previously in Figure 4.2.

In this scenario, the attacker would not be able to differentiate the repair symbols from the source symbols. However, as long as she was able to detect the time lapse between the encoding symbols of each source block, she could count the 13 encoding symbols. From there she can use the attack vector corresponding to $K' = 12$ (since 13 is not one of the available values of K' for RaptorQ); the attack would fail, and she would try the attack vector for $K' = 10$ (11 is also not a value of K' admissible in the RaptorQ standard), and the attack would succeed in only two tries. So, this sort of trial and error can yield positive results from the point of view of an attacker. Note that the *padding symbols* are not transmitted through the network, thus may slightly offset the values the attacker is testing, but not prevent him from successfully executing the attack.

Even though the use of secure channels may increase the difficulty of the attack, it is definitely still possible. Given a critical system that requires security and reliable communication to the point of using RaptorQ over secure channels, it is a matter of serious concern that it is even mildly possible for an attacker to hinder the communication injecting a small number of faults in such an inconspicuous way.

4.5 Discussion

The RaptorQ code was never proposed to be resilient against malicious faults, however, in our view, due to the critical situations where it is used, some changes should be considered when implementing the standards. The RFC for RaptorQ presents some security considerations, but these are mostly concerned with multicast delivery, namely: (1) Denial-of-Service attacks where an attacker corrupts packets which would be seen as legitimate by the receivers, causing them the computational cost of decoding, only to recover unusable data; and, (2) if an attacker forges or corrupts a session description (in multicast delivery) then receivers could be using incorrect protocols for decoding. Both of these concerns, can be solved with authentication, integrity and reverse path forwarding checks.

Note that none of those solutions, would actually be able to prevent our attack. That is because the attack is based on the standard's design flaws. Encrypting the messages may increase the difficulty of executing the attack, but in the end the design is still the same. Even if the implementation does not follow to the letter the RFCs (e.g., does not use the described functions), the target ISIs for elimination will change, but the attack is still viable as long as the implementation follows the base design described in the RFCs. This is why we were able to execute the attack without having to consider the messages' content, since we knew the implementation being used, we could calculate the target ISIs.

The attack will work on any Raptor code that suffers from the issues present in the RaptorQ standard, namely the sequential symbol identification (always starting at 0) paired with the pseudo-randomness of the LT codes[4]. Implementations should take that into consideration and employ appropriate mechanisms to circumvent this design flaws. For the remainder of this section, we will propose some solutions and discuss their pros and cons, and why and when they could be applied.

[4]There is probably nothing to be done about this because with pure randomness it would be impossible to recover the data.

4.5.1 Proposed Solutions

A very straight-forward way of solving the problem is for the receiver to request any missing symbol it needs, or, to request more repair symbols. Obviously, this is not a very attractive solution because it goes against the nature of fountain codes. Also, the attacker might still be able to drop those packets, if she knows the implementation well enough. Finally, this is not a solution at the standard's level, but a mechanism that is implementation dependent. Thus, we do not recommend this as a way to secure the RaptorQ code.

If communication is encrypted or made through a secure channel, it may be enough to rethink the order in which the encoding symbols are sent, and interleaving the source and repair symbols. Of course, this has to be done in an unpredictable pattern, otherwise an informed attacker could still counter it. Note that this only works if the communication is encrypted, otherwise the attacker will still be able to do the attack: by reverse engineering the message structure, and consulting the ESI of each symbol to see if it is a target or not.

Another, more elaborate solution, would be to smartly use a *cryptographically secure pseudo-random number generator* (CSPRNG), such as [46] or [47]. A CSPRNG is a *pseudo-random number generator* (PRNG) with properties that make it suitable for use in cryptography, namely: (1) there is polynomial-time algorithm that can predict the next bit with probability of success better than 50%; and (2) in the event that part or all of its state has been revealed (or guessed correctly), it should be impossible to reconstruct the stream of random numbers prior to the revelation.

A CSPRNG is capable of generating a sequence of numbers that approximates the properties of random numbers. As with any PRNG, the sequence is not truly random in that it is completely determined by a relatively small set of initial values, called the PRNG's state, which includes a truly random seed. If the encoder and the decoder were configured with the same pre-configured seed, they could use the CSPRNG to generate the ESIs (and ISIs) of the symbols in an unpredictable pattern. The attacker could see the ESI in the *encoding packet* where the symbol was, but, would not know if that was the i^{th} symbol. Whilst the decoder would still be able to know that, since it is also configured with the same seed as the encoder, and has access to the same CSPRNG. Using this technique secures against our attack, even when using unencrypted communication, as long as the attacker does not have nor guesses the seed. Furthermore, there could be a flag, configured at both ends, that specified if the original identification mechanism should be used, or if the CSPRNG should be used. Although using the standard identification renders the communication vulnerable to our attack, developers may give users this configuration option, for when the code should follow the standard (e.g., for when there is an interplay of implementations, that is, the decoder implementation is different from the encoder's, hence the need for following a mutual standard).

Chapter 5

Conclusion

“Back in the office, Socrates drew some water from the spring water dispenser and put on the evening’s tea specialty, rose hips, as he continued. ‘You have many habits that weaken you. The secret of change is to focus all your energy not on fighting the old, but on building the new.”’

— Way of the Peaceful Warrior, Dan Millman.

The main goal of this work was to study the effect a malicious attacker can have on the robustness of the RaptorQ code. In order to achieve that, a fully capable and compliant implementation of the RaptorQ standard[2] was developed. At the moment it is not public because there are still a few performance optimizations to be made prior to the release. Moreover, the implementation was used to study the resilience of the RaptorQ FEC code against accidental faults. This study helps assessing the impact of our attack.

In what regards to our attack, the work was started by first ensuring that a malicious attacker could actually have some ill effect on RaptorQ's robustness. On that purpose, an attacker with network control was assumed, who was capable of intercepting and dropping any packet between the sender and the receiver. The rationale behind our attack was described, and a proof of concept attack was established. The attack tries to introduce a dependency among the equations in the system of linear equations used to calculate the intermediate symbols. The process of calculating the intermediate symbols can be considered the core of RaptorQ's encoding and decoding processes.

The results from the proof of concept attack showed that by choosing which packets reached the receiver, an attacker can affect the probability of decoding failure. Thus, piercing RaptorQ's robustness. However, the proof of concept attack was far from fully exploiting the latent potential of the attack. The results from the proof of concept attack did not represent a viable attack. The total number of packets that had to be eliminated was for most cases analyzed very high. If the number of packets lost during the attack is well above the average packets loss during benign communication, the attack can be easily detected.

Subsequently, a new attack was idealized, much more complete than its predecessor, maximizing the usage of the attack surface available to an attacker. Analyzing the results from this refined attack, it proves to be a much more viable option. For 0 overhead symbols, the probability of failure when facing accidental faults is in the order of 1×10^{-3}. With our attack, the probability of failure is 100%, and for the refined attack, for a large part of the values analyzed the number of packets that must be "lost" is lower than 1% (for 0 overhead symbols). Such an attack is much harder to detect, and can be easily confused with sporadic network loss. Furthermore, the attack payloads can be precomputed for each value of K (they are independent of the content being transmitted), which significantly reduces computational requirements of the malicious machine from which the attack is executed (e.g., it can be a compromised router).

Although RaptorQ is fairly recent, many standards have already adopted older Raptor codes, namely R10 [1]. Since RaptorQ is the Raptor code with the most attractive properties, there is a tendency for standardization bodies to adopt RaptorQ into their own standards.

The attack described in this thesis is implementation independent, as it exploits the standard's own design. As a consequence, it can be used against any RaptorQ implementation. However, the same rationale could be employed to attack other Raptor codes. Namely, the R10 code also suffers from the same design flaws exploited in our attack against RaptorQ. Therefore, this thesis may have practical implications not only relating to the RaptorQ code, but also previous standards.

Finally, some solutions were proposed. The more complete solution uses a *cryptographically secure pseudo-random number generator* (CSPRNG), and renders the attack *impossible*[1] both in encrypted communication and clear-text. This solution could be adopted into the standard, but also, it can be easily integrated with any existing implementations.

[1]The attack is not really impossible, however, it becomes a *guessing game* (i.e., the probability of successfully attacking is the same as the probability of decoding failure for accidental faults).

Appendix A

Attack Vectors

In this appendix some of the attack vectors found through experiments are presented. Each attack vector contains the information needed to perform the attack (for those specific parameters): (1) the lines of the constraint matrix (and (2) the ISIs for their corresponding source symbols) that need to be replaced, by the lines corresponding to (3) the ISIs of the repair symbols that will act as the payload of the attack. Moreover, (4) the total number of encoding symbols lost, and (5) the lines corresponding to the payload repair symbols are also available.

Attack A.1: Attack vector for K = 10 and 0 overhead

```
 - K: 10
 - Overhead: 0
 - Epsilon: 0.1

 Target lines: [17, 21, 25]
 Target  ISIs: [0, 4, 8]
 Payload ISIs: [10, 11, 12]
 Body count   : 3 (30.0%)


------ PAYLOAD LINES ------

[0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,0,0,0,0,0,0,1,1,0,0]
[0,0,1,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,1,0]
[0,1,1,0,1,0,1,1,1,1,0,1,0,1,1,0,1,0,0,0,0,1,1,0,0,0,0]

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

Attack A.2: Attack vector for K = 10 and 1 overhead

```
 - K: 10
 - Overhead: 1
 - Epsilon: 0.1

 Target lines: [17, 21, 23, 26]
 Target  ISIs: [0, 4, 6, 9]
 Payload ISIs: [11, 12, 16, 17]
 Body count   : 7 (63.63636363636363%)


------ PAYLOAD LINES ------

[0,0,1,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,1,0]
[0,1,1,0,1,0,1,1,1,1,0,1,0,1,1,0,1,0,0,0,0,1,1,0,0,0,0]
[0,1,1,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0]
[0,0,0,0,0,0,0,0,1,0,1,0,1,0,0,0,0,0,0,1,0,0,0,1,0,0,0]

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

Attack A.3: Attack vector for K = 26 and 1 overhead

```
 - K: 26
 - Overhead: 1
 - Epsilon: 0.1

 Target lines: [41, 44, 45, 47]
 Target  ISIs: [20, 23, 24, 26]
 Payload ISIs: [27, 28, 29, 30]
 Body count   : 4 (14.814814814814813%)


------ PAYLOAD LINES ------

[1,1,1,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,
    1,1,1,0,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0]
[1,0,0,0,1,0,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,0,1,0,0,
    0,1,0,0,0,1,0,0,0,1,0,0,0,0,1,0,0,0,0]
[0,0,0,1,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1,1,0,
    0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,1,0,0]
[0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1]

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

Attack A.4: Attack vector for K = 32 and 0 overhead

```
 - K: 32
 - Overhead: 0
 - Epsilon: 0.1

 Target lines: [40]
 Target  ISIs: [19]
 Payload ISIs: [33]
 Body count   : 2 (6.25%)


------ PAYLOAD LINES ------

[0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,1,0,1]

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

Attack A.5: Attack vector for K = 32 and 1 overhead

```
 - K: 32
 - Overhead: 1
 - Epsilon: 0.1

 Target lines: [25, 28, 34, 53]
 Target  ISIs: [4, 7, 13, 32]
 Payload ISIs: [33, 34, 35, 37]
 Body count   : 5 (15.151515151515152%)


------ PAYLOAD LINES ------

[0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,1,0,1]
[0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,1,1,0]
[0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,1,0,0,0,0,0,0,0,1,0]
[0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,1,1,0,0,0,0,1,0]

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

Attack A.6: Attack vector for K = 42 and 0 overhead

```
 - K: 42
 - Overhead: 0
 - Epsilon: 0.1

 Target lines: [24]
 Target  ISIs: [3]
 Payload ISIs: [43]
 Body count   : 2 (4.761904761904762%)
```

```


------ PAYLOAD LINES ------

[0,0,0,1,0,0,1,0,0,0,0,0,0,0,1,0,0,1,0,0,1,0,0,0,0,0,0,0, ↙
    ↳ 1,0,0,1,0,0,1,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0, ↙
    ↳ 1,0,0,0,0,0,0,1,0]

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

Attack A.7: Attack vector for K = 91 and 0 overhead

```
 - K: 91
 - Overhead: 0
 - Epsilon: 0.1

 Target lines: [90]
 Target  ISIs: [63]
 Payload ISIs: [91]
 Body count  : 1 (1.098901098901099%)


------ PAYLOAD LINES ------

[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0, ↙
    ↳ 0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↳ 0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↳ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↳ 1,1,0,0,0,0,0,0,0,1,0,0]

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

Attack A.8: Attack vector for K = 101 and 0 overhead

```
 - K: 101
 - Overhead: 0
 - Epsilon: 0.001

 Target lines: [80]
 Target  ISIs: [53]
 Payload ISIs: [102]
 Body count  : 2 (1.9801980198019802%)


------ PAYLOAD LINES ------

```

```
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0, ↙
    ↳ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0, ↙
    ↳ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↳ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0, ↙
    ↳ 0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,1,0,0,0,0,0]

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

Attack A.9: Attack vector for K = 153 and 0 overhead

```
 - K: 153
 - Overhead: 0
 - Epsilon: 0.001

 Target lines: [38, 171]
 Target  ISIs: [5, 138]
 Payload ISIs: [153, 154]
 Body count   : 2 (1.3071895424836601%)


------ PAYLOAD LINES ------

[0,1,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,1, ↙
    ↳ 0,0,0,0,0,0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,1, ↙
    ↳ 0,0,0,0,0,0,0,1,0,0,0,0,1,0,0,1,0,0,0,0,1,0,0,0,0,0, ↙
    ↳ 0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,1,0,0,0,0,0, ↙
    ↳ 0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0, ↙
    ↳ 0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0, ↙
    ↳ 0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0, ↙
    ↳ 0,0]
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0, ↙
    ↳ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0, ↙
    ↳ 0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0, ↙
    ↳ 0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0, ↙
    ↳ 0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0, ↙
    ↳ 0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↳ 0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1, ↙
    ↳ 0,0]

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

Attack A.10: Attack vector for K = 153 and 1 overhead

```
 - K: 153
 - Overhead: 1
 - Epsilon: 0.001

 Target lines: [51, 184]
 Target  ISIs: [18, 151]
 Payload ISIs: [155, 156]
 Body count   : 3 (1.948051948051948%)


------ PAYLOAD LINES ------

[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↳ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0, ↙
    ↳ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0, ↙
    ↳ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↳ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↳ 0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↳ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0, ↙
    ↳ 0,0]
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↳ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↳ 0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↳ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0, ↙
    ↳ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↳ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↳ 0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1, ↙
    ↳ 0,0]

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

Attack A.11: Attack vector for K = 248 and 0 overhead

```
 - K: 248
 - Overhead: 0
 - Epsilon: 0.001

 Target lines: [138]
 Target  ISIs: [99]
 Payload ISIs: [249]
 Body count   : 2 (0.8064516129032258%)


------ PAYLOAD LINES ------

[0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0, ↙
    ↳ 0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0, ↙
    ↳ 0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0, ↙
    ↳ 0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0, ↙
    ↳ 0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0, ↙
    ↳ 0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↳ 0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0]

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

Attack A.12: Attack vector for K = 248 and 1 overhead

```
 - K:  248
 - Overhead: 1
 - Epsilon: 0.001

 Target lines: [157, 271]
 Target   ISIs: [118, 232]
 Payload ISIs: [249, 252]
 Body count   : 4 (1.6064257028112447%)


------ PAYLOAD LINES ------

[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,  ↙
    ↳ 0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0]
[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,1 ,0 ,0 ,0 ,0 ,0]

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

Attack A.13: Attack vector for K = 355 and 0 overhead

```
 - K: 355
 - Overhead: 0
 - Epsilon: 0.001

 Target lines: [91]
 Target  ISIs: [50]
 Payload ISIs: [356]
 Body count   : 2 (0.5633802816901409%)


------ PAYLOAD LINES ------

[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0]

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

Attack A.14: Attack vector for K = 355 and 1 overhead

```
 - K: 355
 - Overhead: 1
 - Epsilon: 0.001

 Target lines: [41, 302]
 Target  ISIs: [0, 261]
 Payload ISIs: [372, 379]
 Body count   : 24 (6.741573033707865%)


------ PAYLOAD LINES ------

```

```
[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,1 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0]
[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳  0 ,0 ,0 ,0]

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

Attack A.15: Attack vector for K = 453 and 0 overhead

```
 - K:  453
 -  Overhead:  0
 -  Epsilon:  0.001

 Target  lines:  [147]
 Target   ISIs:  [100]
 Payload  ISIs:  [453]
 Body  count    :  1  (0.22075055187637968%)


```

```
------ PAYLOAD LINES ------

[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0]

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

Attack A.16: Attack vector for K = 453 and 1 overhead

```
- K: 453
 - Overhead: 1
 - Epsilon: 0.001

 Target lines: [47, 165]
 Target  ISIs: [0, 118]
 Payload ISIs: [482, 484]
 Body count   : 31 (6.828193832599119%)


------ PAYLOAD LINES ------

[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 1 ,0 ,0 ,0]
[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,1]

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

Attack A.17: Attack vector for K = 511 and 0 overhead

```
 - K: 511
 - Overhead: 0
 - Epsilon: 0.001

 Target lines: [157]
 Target  ISIs: [110]
 Payload ISIs: [511]
 Body count   : 1 (0.19569471624266144%)


```

```
------ PAYLOAD LINES ------

[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1]

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

Attack A.18: Attack vector for K = 549 and 0 overhead

```
- K: 549
 - Overhead: 0
 - Epsilon: 0.001

 Target lines: [238]
 Target  ISIs: [187]
 Payload ISIs: [549]
 Body count   : 1 (0.18214936247723132%)


------ PAYLOAD LINES ------

[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0]

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

Attack A.19: Attack vector for K = 549 and 1 overhead

```
 - K:  549
 - Overhead:  1
 - Epsilon:  0.001

 Target lines:  [51,  165]
 Target  ISIs:  [0,  114]
 Payload ISIs:  [572,  587]
 Body count   :  38 (6.909090909090909%)


------ PAYLOAD LINES ------

[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0]
[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0]

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

Attack A.20: Attack vector for K = 600 and 0 overhead

```
 - K:  600
 -  Overhead:  0
 -  Epsilon:  0.001

 Target  lines:  [132]
 Target   ISIs:  [81]
 Payload  ISIs:  [600]
 Body  count    :  1  (0.16666666666666669%)

```

```

------ PAYLOAD LINES ------

[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0]

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

Attack A.21: Attack vector for K = 648 and 0 overhead

```
 - K:  648
 - Overhead:  0
 - Epsilon:  0.001

 Target  lines:  [319]
 Target   ISIs:  [266]
 Payload  ISIs:  [648]
 Body  count    :  1  (0.15432098765432098%)


------ PAYLOAD LINES ------

[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪  0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1]

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

Attack A.22: Attack vector for K = 648 and 1 overhead

```
 - K: 648
 - Overhead: 1
 - Epsilon: 0.001

 Target lines: [319]
 Target  ISIs: [266]
 Payload ISIs: [650]
 Body count   : 2 (0.3081664098613251 5%)


------ PAYLOAD LINES ------

[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0]

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

Attack A.23: Attack vector for K = 703 and 0 overhead

```
 - K: 703
 - Overhead: 0
 - Epsilon: 0.001

 Target lines: [270]
 Target  ISIs: [213]
 Payload ISIs: [703]
 Body count   : 1 (0.1422475106685633%)


------ PAYLOAD LINES ------

[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 , ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0, ↙
    ↪ 0,0,0,0]

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

Attack A.24: Attack vector for K = 747 and 0 overhead

```
 - K: 747
 - Overhead: 0
 - Epsilon: 0.001

 Target lines: [116]
 Target  ISIs: [59]
 Payload ISIs: [747]
 Body count   : 1 (0.13386880856760375%)


------ PAYLOAD LINES ------

[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,1 ,0 ,0]

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

Attack A.25: Attack vector for K = 747 and 1 overhead

```
 - K: 747
 - Overhead: 1
 - Epsilon: 0.001

 Target lines: [57, 275]
 Target  ISIs: [0, 218]
 Payload ISIs: [754, 755]
 Body count   : 8 (1.06951871657754%)



------ PAYLOAD LINES ------

[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,1 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0]
[0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0]

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

Attack A.26: Attack vector for K = 802 and 0 overhead

```
 - K: 802
 - Overhead: 0
 - Epsilon: 0.001

 Target lines: [120]
 Target  ISIs: [57]
 Payload ISIs: [802]
 Body count  : 1 (0.12468827930174563%)


------ PAYLOAD LINES ------

[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,1 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0]

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

Attack A.27: Attack vector for K = 845 and 0 overhead

```
 - K:  845
 - Overhead:  0
 - Epsilon:  0.001

 Target  lines:  [179]
 Target   ISIs:  [116]
 Payload  ISIs:  [845]
 Body  count    :  1  (0.1183431952662722%)


------  PAYLOAD  LINES  ------

[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0]

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

Attack A.28: Attack vector for K = 845 and 1 overhead

```
 - K: 845
 - Overhead: 1
 - Epsilon: 0.001

 Target lines: [526]
 Target  ISIs: [463]
 Payload ISIs: [856]
 Body count   : 11 (1.3002364066193852%)


------ PAYLOAD LINES ------

[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0]

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

Attack A.29: Attack vector for K = 903 and 0 overhead

```
 - K: 903
 - Overhead: 0
 - Epsilon: 0.001

 Target lines: [63, 373]
 Target  ISIs: [0, 310]
 Payload ISIs: [903, 904]
 Body count   : 2 (0.22148394241417496%)


------ PAYLOAD LINES ------

[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0]
[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0]

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

Attack A.30: Attack vector for K = 903 and 1 overhead

```
 - K: 903
 - Overhead: 1
 - Epsilon: 0.001

 Target lines: [63, 104]
 Target  ISIs: [0, 41]
 Payload ISIs: [909, 921]
 Body count   : 18 (1.991150442477876%)


------ PAYLOAD LINES ------

[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,1 ]
[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1, ↙
    ↪ 1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0, ↙
    ↪ 0,0]

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

Attack A.31: Attack vector for K = 950 and 0 overhead

```
 - K: 950
 - Overhead: 0
 - Epsilon: 0.001

 Target lines: [722]
 Target  ISIs: [653]
 Payload ISIs: [950]
 Body count   : 1 (0.10526315789473684%)


------ PAYLOAD LINES ------

[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,  ↙
    ↪ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↪ 0 ,0 ,0]

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

Attack A.32: Attack vector for K = 950 and 1 overhead

```
 - K: 950
 - Overhead: 1
 - Epsilon: 0.001

 Target lines: [838]
 Target  ISIs: [769]
 Payload ISIs: [956]
 Body count   : 6 (0.6309148264984227%)

```

```

------ PAYLOAD LINES ------

[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,  ↙
    ↳ 0 ,0 ,0]

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

Bibliography

[1] M. Luby et al. “Raptor Forward Error Correction Scheme for Object Delivery”. In: *NWG RFC 5053* (2007).

[2] M. Luby et al. “RaptorQ Forward Error Correction Scheme for Object Delivery”. In: *IETF RFC 6330* (2011).

[3] J. Postel. “Internet protocol”. In: *IETF RFC 791* (1981).

[4] J. Postel. “Transmission control protocol”. In: *IETF RFC 793* (1981).

[5] R. Fielding et al. “Hypertext transfer protocol–HTTP/1.1”. In: *NWG RFC 2616* (1999).

[6] T. Ylonen and C. Lonvick. “The secure shell (SSH) transport layer protocol”. In: *NWG RFC 4253* (2006).

[7] J. Galbraith and O. Saarenmaa. “SSH File Transfer Protocol”. In: *SecshWG Internet-Draft* (2006).

[8] J. Postel. “User datagram protocol”. In: *IETF RFC 768* (1980).

[9] D. MacKay. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003.

[10] W. Huffman and V. Pless. *Fundamentals of error correcting codes*. Cambridge University Press, 2003.

[11] M. Luby et al. “Wave and equation based rate control using multicast round trip time”. In: *ACM SIGCOMM Computer Communication Review* 32.4 (2002), pp. 191–204.

[12] M. Luby and V. Goyal. “Wave and Equation Based Rate Control (WEBRC) Building Block”. In: *NWG RFC 3738* (2004).

[13] B. Cipra. “The ubiquitous reed-solomon codes”. In: *SIAM News* 26.1 (1993).

[14] R. Gallager. “Low-density parity-check codes”. In: *IRE Transactions on Information Theory* 8.1 (1962), pp. 21–28.

[15] D. MacKay. “Fountain codes”. In: *IEEE Proceedings - Communications* 152.6 (2005), pp. 1062–1068.

[16] A. Shokrollahi and M. Luby. *Raptor codes*. Now Publishers Inc, 2011.

[17] M. Luby. “LT Codes”. In: *Proceedings 43rd Annual IEEE Symposium on Foundations of Computer Science* (2002), pp. 271–280.

[18] C. Harrelson, L. Ip, and W. Wang. “Limited randomness LT codes”. In: *Proceedings of the Annual Allerton Conference on Communication Control and Computing* 41.1 (2003), pp. 492–501.

[19] M. Luby et al. “Practical loss-resilient codes”. In: *Proceedings of the 29th Annual ACM Symposium on Theory of Computing* (1997), pp. 150–159.

[20] M. Luby et al. “Efficient erasure correcting codes”. In: *IEEE Transactions on Information Theory* 47.2 (2001), pp. 569–584.

[21] C. Shannon. “Communication in the presence of noise”. In: *Proceedings of the IRE* 37.1 (1949), pp. 10–21.

[22] M. Luby, M. Mitzenmacher, and A. Shokrollahi. “Analysis of random processes via and-or tree evaluation”. In: *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms* (1998), pp. 364–373.

[23] M. Luby et al. “Improved low-density parity-check codes using irregular graphs”. In: *IEEE Transactions on Information Theory* 47.2 (2001), pp. 585–598.

[24] T. Richardson, A. Shokrollahi, and R. Urbanke. “Design of capacity-approaching irregular low-density parity-check codes”. In: *IEEE Transactions on Information Theory* 47.2 (2001), pp. 619–637.

[25] B. LaMacchia and A. Odlyzko. “Solving large sparse linear systems over finite fields”. In: *Advances in Cryptology-CRYPTO’90* (1991), pp. 109–133.

[26] M. Luby et al. “Practical loss-resilient codes”. In: *Proceedings of the 29th Annual ACM Symposium on Theory of Computing* (1997), pp. 150–159.

[27] V. Zyablov and M. Pinsker. “Decoding complexity of low-density codes for transmission in a channel with erasures”. In: *Problemy Peredachi Informatsii* 10.1 (1974), pp. 15–28.

[28] A. Shokrollahi. “Raptor codes”. In: *IEEE Transactions on Information Theory* 52.6 (2006), pp. 2551–2567.

[29] A. Shokrollahi, S. Lassen, and M. Luby. “Multi-stage code generator and decoder for communication systems”. In: *US Patent 7,068,729* (2006).

[30] A. Shokrollahi. “Theory and applications of raptor codes”. In: *Proceedings of MathKnow* (2009), pp. 59–89.

[31] A. Shokrollahi and M. Luby. “Systematic encoding and decoding of chain reaction codes”. In: *US Patent 7,532,132* (2009).

[32] A. Shokrollahi, S. Lassen, and R. Karp. “Systems and processes for decoding chain reaction codes through inactivation”. In: *US Patent 6,856,263* (2005).

[33] 3GPP. “Technical Specification Group Services and System Aspects; Multimedia Broadcast/Multicast Service; Protocols and Codecs”. In: *ETSI TS 26.346 V6.1.0* (2005).

[34] Digital Fountain and Siemens. “Specification Text for Systematic Raptor Forward Error Correction”. In: *TSG System Aspects WG4 PSM ad hoc S4-AHP238* (2006).

[35] Digital Video Broadcasting (DVB). “IP Datacast over DVB-H: Content Delivery Protocols”. In: *ETSI TS 102 472 v1.2.1* (2006).

[36] Open Mobile Alliance. “File and Stream Distribution for Mobile Broadcast Services”. In: *Mobile Broadcast Services V1.0* (2009).

[37] Open Mobile Alliance. “Broadcast Distribution System Adaptation - IPDC over DVB-H.” In: *OMA-TS-BCAST_DVB_Adaptation-V1_0-20080226-C* (2008).

[38] Digital Video Broadcasting (DVB). “Transport of MPEG-2 TS Based DVB Services over IP Based Networks”. In: *ETSI TS 102 034 V1.4.1* (2009).

[39] Digital Video Broadcasting (DVB). “DVB Document A131”. In: *MPE-IFEC* (2008).

[40] Digital Video Broadcasting (DVB). “Interaction channel for satellite distribution systems”. In: *ETSI EN 301 790 V1.4.1* 301 (2005), p. 790.

[41] Digital Video Broadcasting (DVB). “Transport of MPEG 2 Transport Stream (TS) Based DVB Services over IP Based Networks”. In: *ETSI TS 102 034 v1.3.1* (2007).

[42] ATIS IIF. “IPTV ARCH Specification: Media Formats and Protocols”. In: *WT 18* (2009).

[43] Telecommunication Standardization Sector of ITU. “Series H: Audiovisual and Multimedia Systems: IPTV multimedia services and applications for IPTV – General aspects”. In: *Recommendation ITU-T H.701* (2009).

[44] R. Oppliger. “Security at the Internet layer”. In: *Computer* 31.9 (1998), pp. 43–47.

[45] S. Bradner. “Key words for use in RFCs to Indicate Requirement Levels”. In: *NWG RFC 2119* (1997).

[46] Federal Information Processing Standards. “Digital Signature Standard (DSS)”. In: *FIPS PUB 186-4* (2013).

[47] ANSI Standard. “X9. 31 Appendix A.2.4”. In: *Digital signatures using reversible public key cryptography for the financial services industry (rDSA)* (1998).

[48] M. Luby et al. “Raptor codes for reliable download delivery in wireless broadcast systems”. In: *Proceedings of the 3rd IEEE Consumer Communications and Networking Conference* 1 (2006), pp. 192–197.