Stopping a Rapid Tornado with a Puff

José Lopes and Nuno Neves
University of Lisbon, Faculty of Sciences, LASIGE
jlopes@lasige.di.fc.ul.pt, nuno@di.fc.ul.pt

***Abstract*—RaptorQ is the most advanced fountain code proposed so far. Its properties make it attractive for forward error correction (FEC), offering high reliability at low overheads (i.e., for a small amount of repair information) and efficient encoding and decoding operations. Since RaptorQ's emergence, it has already been standardized by the IETF, and there is the expectation that it will be adopted by several other standardization bodies, in areas related to digital media broadcast, cellular networks, and satellite communications. The paper describes a new attack on RaptorQ that breaks the near ideal FEC performance, by carefully choosing which packets are allowed to reach the receiver. Furthermore, the attack was extended to be performed over secure channels with IPsec/ESP. The paper also proposes a few solutions to protect the code from the attack, which could be easily integrated into the implementations.**

I. Introduction

Forward error correction (FEC) is a technique for the recovery of errors in data disseminated over unreliable or noisy communication channels [1]. The central idea is that the sender encodes the message in a redundant way by applying an error-correcting code, which allows the receiver to repair the errors. An *erasure code* is a FEC code with the capability to recuperate from losses in the communications. The data is divided into K source symbols, which are transformed in a larger number of N encoding symbols such that the original data can be retrieved from a subset of the encoding symbols. An immediate benefit of this approach is that the receiver gains the ability to amend the errors without needing a reverse channel to request the retransmission of data, at the cost of a fixed higher bandwidth forward channel. FEC is therefore applied in situations where retransmissions are costly, such as when broadcasting data to multiple destinations, or when communication links are one-way.

Fountain codes are a class of erasure codes with two attractive properties: an arbitrary number of encoding symbols can be produced on the fly, simplifying the adaptation to varying loss rates; and the data can be reconstructed with high probability from *any* subset of the encoding symbols (of size equal to or slightly larger than the number of source symbols) [1], [2]. A typical use case scenario for fountain codes appears when a single source multicasts a file to many destinations. In such a scenario, resorting to TCP channels would not be scalable because the sender needs to keep track of which packets arrive at each receiver. Multicasting with UDP would solve this limitation, but would lack the reliability offered by TCP. Coding the file with a fountain code and disseminating over UDP addresses both problems – each receiver would be able to recover the (different) erasures affecting its own channel, without the need for retransmissions.

Raptor (or *Rapid Tornado*) codes [3] are the closest solution to an ideal digital fountain code. They achieve constant per-symbol encoding/decoding cost with a near-zero overhead. Two versions of Raptor codes have been standardized by the Internet Engineering Task Force (IETF), called R10 [4] and RaptorQ [5]. R10 appeared first, and has been adopted in a number of different standards [6]–[16], covering areas related to the transmission of data over cellular networks, satellite communications, IPTV and digital video broadcasting. RaptorQ is the most recent Raptor code, and it provides greatly enhanced reliability as well as efficient encoding and decoding functions. RaptorQ has been standardized in RFC 6330 [5] and other standards will probably adopt it as well. Its applications are various, and include military operations and communication with intermittent transmissions and/or with high loss rates [17]. To the best of our knowledge, Qualcomm's closed and commercial solution[1] is the only one currently available to the public. However, this may change in the future as RaptorQ could obsolete R10 [18].

RaptorQ was designed for *benign* environments where erasures occur independently and randomly. In such an environment, the code's reliability is excellent.

> *"... RaptorQ codes offer close to optimal protection against arbitrary packet losses at a low computational complexity."* [19]

> *"If the decoder receives K'+2 encoding symbols <...> then on average the decoder will fail to recover the entire source block at most 1 out of 1,000,000 times."* [5]

This paper investigates to what extent a malicious adversary may affect RaptorQ's reliability. This is justified not only because of the anticipated utilization of the code in several communication standards, but also due to the large number of application areas that are envisioned. We show that it is possible to hinder the decoding process with little effort, thus preventing the reconstruction of the original data.

Some RFCs about FEC utilization in IP networks [20], and also the description of RaptorQ [5], already discuss a few security implications of the codes. These considerations address common attacks like the access to confidential data and the corruption of the communication flows (e.g., by modifying the content of the packets). As stated, FEC protection does not offer any kind of security. However, all previously considered attacks can be prevented with encryption, source authentication and integrity checks. IPsec/ESP [21] paired with checking the integrity of the decoded data is thus one of the recommended solutions that could secure the flows.

In this paper, we study a novel approach to attack RaptorQ, which is directed at the standard's own design and is indepen-

[1] http://www.qualcomm.com/solutions/multimedia/media-delivery/raptorq

dent of the data being propagated. The idea is to create erasures in the network that (1) will allow the decoding operation to be performed but (2) with an unsuccessful outcome (i.e, with a decoding failure). Since RaptorQ is highly robust, if this activity is carried out in a random way then the probability of a successful attack would be extremely small. Thus, to ensure that decoding always fails, we had to exploit some of the operations defined in the RFC, such as the deterministic identification of encoding symbols (i.e., the packets) paired with the pseudo-randomness of the inner (LT) code. This way we could predict the format of the decoding matrix (a system of linear equations) and force the injection of linear dependencies that stopped the recovery of packet losses.

Under regular conditions, in order to execute the attack, an adversary needs to have access to the flow of packets between the source and the destination(s) and the capability to drop/forward packets accordingly to a pre-computed erasure pattern. Given a sequence of packets produced by a transmitter, the erasure pattern defines which packets may go through and which must be eliminated. Therefore, the attack can be executed with minimal packet delay, ideally at line-speed. We will explain how a TFTP application using RaptorQ in its communication can be precluded from reconstructing the whole file at a server. Note that TFTP was chosen just for the demonstration of the attack's practicality. The attack mechanisms hold for the wide range of uses of RaptorQ.

In some cases, when one can guess the configuration parameters of RaptorQ by looking at the flow of packets (e.g., the time intervals between them), the attack can be extended to encrypted channels. Later in the paper we will analyze what are the necessary conditions to carry out such attacks and also discuss prevention options. In order to explore the feasibility of the attack on encrypted RaptorQ transmissions, we explain how it could be successfully carried out in the TFTP application running on top of IPsec/ESP.

Surprisingly, for many of the configurations of RaptorQ, we could discover erasure patterns where only a few packets have to be deleted to stop effective decoding (as shown later in Table II). This means that the attack can be made stealthy as receivers experience a loss rate with a magnitude within the range of typical networks. The main observable difference is that these losses happen to have a much higher impact – the decoder is unsuccessful. In other words, RaptorQ is stopped from fulfilling its mission of providing protection against erasures with very high probability, even though it is allowed to execute under the expected conditions.

The remainder of the paper is organized as follows: Section II gives some background information. The RaptorQ standard is explained in more detail in Section III. Section IV describes our attack on IETF's RFC 6330 [5] and the impact it can have on the code's reliability. Next, two experiments are performed on how to disrupt the execution of TFTP application, first with clear channels and later on with IPsec/ESP. Finally, in Section VII, we briefly discuss how the attack could be extended to the R10 code, and explain potential solutions that could be employed to protect the RaptorQ standard.

II. Background

A. Related Work

Some investigation has been performed on malicious packet dropping and on its detection. For instance, Zang et al. observed that selectively eliminating a few packets in a TCP connection may severely damage performance [22], and a statistical module was proposed to detect this sort of attacks. Bradley et al. describe a solution for the discovery and reaction to routers that drop or misroute packets [23]. A distributed probing technique to detect and mitigate malicious packet dropping in wireless ad hoc networks was discussed in [24]. Our work differs from previous research because it targets the FEC code that is used to protect from erasures. In many cases just one or two packets need to be dropped to prevent the recovery of the original data, making it extremely hard to detect only from a network perspective.

We presented a preliminary version of this work in [25]. This version contributes a refined attack, a significantly extended discussion and countermeasure suggestions.

B. Fountain Codes

Fountain codes (*a.k.a. rateless erasure codes*) [1], [2] are capable of reconstructing the original data even if some of the transmitted packets (carrying the encoding symbols) are lost in the network. They have the property that a potentially limitless sequence of encoding symbols can be generated from the original data (i.e., the source symbols). They also have the characteristic that the original data can be rebuilt with high probability from any subset of the encoding symbols of size larger than or equal to the number of source symbols. The name *rateless* refers to the fact that these codes do not exhibit a fixed code rate. The code rate (or information rate [26]) of a FEC 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.

Reed-Solomon codes [27] can be seen as a first example of fountain-like codes because data is partitioned into K source symbols and recovery can occur with any K encoding symbols. These codes require however a quadratic decoding time and are limited to small block lengths. *Low-density parity-check* (LDPC) codes [28] come closer to the fountain code ideal. Nevertheless, early LDPC codes were restricted to fixed-degree regular graphs, requiring significantly more than K encoding symbols to correctly decode the propagated signal. *Tornado* codes [29], although they do approach Shannon capacity [30] with linear decoding complexity, are block codes and hence not rateless. *Luby Transform* (LT) codes [31] are considered the first practical *rateless* codes for the Binary Erasure Channel (BEC). The encoding and decoding algorithms of LT codes are conceptually simple – they are similar to a parity-check process.

Raptor codes [3], [32] were developed as a way to reduce the decoding cost to $O(1)$ by preprocessing the LT code with a standard erasure block code (e.g., Tornado code). When designed properly, a Raptor code can achieve constant per-symbol encoding/decoding cost with an overhead close to zero. At the time of this writing, this class of codes has been shown to be the closest to the ideal universal digital fountain.

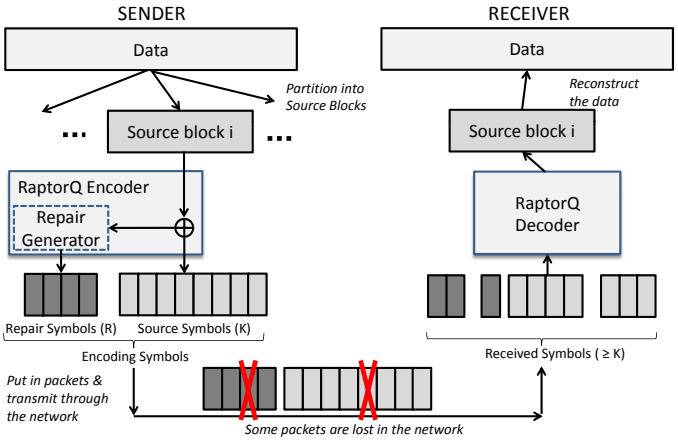

Fig. 1. Overview of data dissemination with the RaptorQ FEC.

C. Data dissemination with a RaptorQ FEC

RaptorQ is able to recover from the loss of packets that may occur anywhere between the sender and the receiver nodes. This covers problems in routers that have to drop packets due to excessive load or data corruptions that are detected using a checksum (causing the packet to be discarded). This sort of capability can be helpful in file multicasting, as it happens that erasures are experienced in diverse ways in the various links connecting to the receivers. If an "*acknowledge or retrasmit*" solution is employed, then the source needs to determine which packets are missing at each destination and must resend them. Furthermore, retransmissions may have to be repeated several times, creating unnecessary delays even with minor network loss probabilities.[2] An approach based on FEC codes solves these problems because the original file can be reconstructed from distinct subsets of the packets. Therefore, it is only necessary to ensure that a certain number of packets is delivered.

Figure 1 displays how data (i.e., a *source object*) is disseminated with the RaptorQ FEC [5]. The data is first divided into blocks, called *source blocks*, that are processed independently by the 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 their size is always T bytes. The value of T should be selected in such a way that it corresponds to the payload size of a packet (i.e., the MTU of the network minus the headers). This way, an erasure only affects a single symbol.

The encoder receives the source symbols to generate a group of *repair symbols*. Since RaptorQ is a fountain code, as many repair symbols as needed can be computed on the fly. Moreover, since the code is *systematic*, the *encoding symbols* are composed by the source symbols plus the repair symbols. Therefore, source symbols can be placed directly in the packets, as no processing needs to be performed by the encoder. The repair symbols are forwarded to the network as they are produced. The benefit is that in the case where there are no packets lost, decoding can be skipped and the data can be immediately delivered to the application.

The decoder takes the collected encoding symbols (any subset with a size equal or slightly larger than K) to rebuild the source block. Each repair symbol compensates for a missing source symbol. If more repair symbols arrive, they can also take part in the decoding process, greatly contributing to the probability of a successful decoding. These extra repair symbols are the *overhead symbols*. The *code overhead* is the minimum number of overhead symbols required to start the decoding process. If N encoding symbols are used, then the code overhead is given by $o = N - K$. The code overhead can be agreed upon between the sender and receiver, the minimum value is $o = 0$. Previous fountain codes in general demanded reasonable levels of overhead to ensure a successful decoding. RaptorQ however, can successfully decode with a high probability with overheads as low as 0 to 2 (see Section III-D).

III. Overview of the RaptorQ standard

This section gives a top-level explanation of the operation of RaptorQ [5], focusing in more detail on the aspects that are relevant to the understanding of the attack.

A. Partitioning the object data

RaptorQ needs the input object to be partitioned into source blocks. The actual way of doing this is left to the application, which has some freedom to define how the data should be divided. Still, the code expects a few configuration parameters to be provided, including the total size of the input object, the symbol size T and the number of source blocks. Additionally, the RFC also includes an example partitioning algorithm (Section 4.3 of [5]). The algorithm makes the symbol size equal to the maximum payload of a packet, ensuring that a recommendation is followed where each packet contains exactly one symbol. It then distributes the data through a number of source blocks, trying to maximize their size while taking into consideration the memory limitations of the receivers. After that, the source block is further divided into K source symbols. The standard also considers a second level of partitioning where a source block is broken down in sub-blocks.[4] Since this operation is optional and has no impact in our work, we will disregard it in the remainder of the paper.

As Section IV will explain, our attack will create very precise erasures based on the fact that a single symbol is placed in a packet. As is recommended in the specification [5]

B. Encoding process

Figure 2 displays the main steps of the encoding process that is applied to each source block. As we will see, the decoding tasks are quite similar. First, the source block may need to be padded to K'. Second, the pre-code is applied to the source symbols to create the *intermediate symbols*. Finally,

[2]Imagine a network with a loss probability of 1%, and a client that wants to send a 10MByte file, partitioned into 10K packets of 1KByte each, to 100 receivers. In the first multicast, every receiver will lose approximately 100 packets. Therefore, each of them will have to inform the sender about which packets are missing, so that later on a specific retransmission is done for every receiver. Since several of the resent 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.

[4]This is used in the situation where a receiver has a very small main memory space available, which could result in less than desired FEC protection.

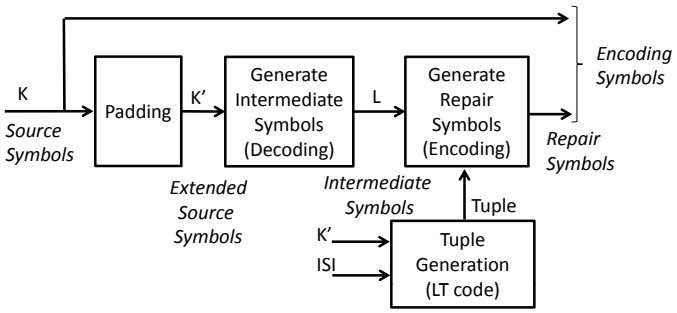

Fig. 2. Overview of the RaptorQ encoding process.

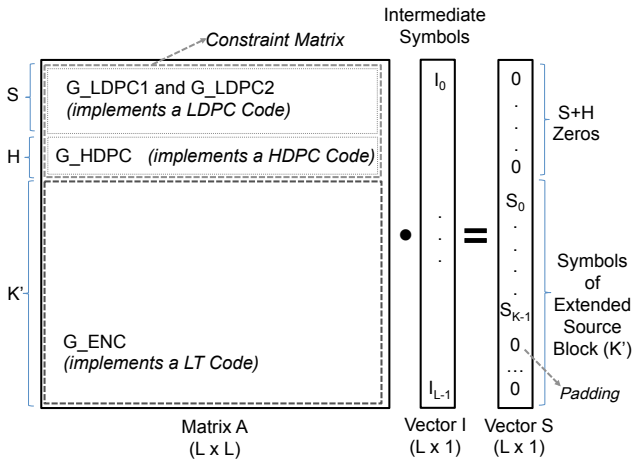

Fig. 3. Computing the intermediate symbols during encoding.

as many repair symbols as requested are generated. In the following we will explain in greater detail these steps.

1) Adding padding: RaptorQ supports only a finite set of values for the number of symbols in a source block. Thus, sometimes padding needs to be added, creating an *extended source block*. RaptorQ resorts to a precomputed table with these values and other associated parameters, which are used by the encoding and decoding processes (Table 2 in Section 5.6 of [5]). The extended source block has K' symbols, where the first K are the original symbols and the remaining $K' - K$ are *padding symbols* filled with zeros. K' is the value in the table that is closest to K, but greater than or equal to. Using a predefined set of possible values for K' minimizes the amount of table information that needs to be stored at each endpoint and effectively contributes to faster encoding and decoding. Since K can be calculated from the input configuration parameters at both peers, the padding symbols do not need to be transmitted.

2) Identifying the symbols: Each source block is identified with a non-negative integer called the *Source Block Number* (SBN). The encoding symbols produced from a source block are identified with an *Encoding Symbol ID* (ESI). As RaptorQ is a systematic code, the encoding symbols consist of the source symbols plus the repair symbols associated with them. The ESIs for the source symbols are $0, 1, 2, ..., K-1$ and the ESIs for the repair symbols are $K, K+1, K+2, ...$ Thereby, each encoding symbol is uniquely designated by a pair (SBN, ESI), and this information must be conveyed in the packets to the receiver(s).

However, for purposes of encoding and decoding data, K' source and padding symbols are utilized. Thus, the encoder and decoder employ the *Internal Symbol ID* (ISI) to identify the symbols belonging to an extended source block. The source symbols' ISIs are (once again) $0, 1, 2, ..., K-1$, the padding symbols have ISIs $K, K+1, K+2, ..., K'-1$, and, finally, the ISIs of the repair symbols are $K', K'+1, K'+2, ...$

As we will see in Section IV, the attack will exploit the fact that the symbol identification is well-defined and deterministic.

3) Computing the intermediate symbols: The second step of encoding is to calculate $L > K'$ intermediate symbols out of the extended source block. This is achieved by building and solving a system of linear equations, whose unknowns are the intermediate symbols. Figure 3 shows that the constraint matrix A is divided into three parts, each associated to a different kind of code. The top part, consisting of the first S equations, corresponds to a Low Density Parity Check (LDPC) code. The middle part has H equations and represents a High Density Parity Check (HDPC) code. Finally, the bottom part, consists of K' equations that apply a Luby Transform (LT) code.

Vector S: The vector with the constant terms has a well defined form (see right side of the figure): (i) the first $S + H$ rows correspond to pre-coding constraints and have value 0; and (ii) the last K' rows are the source symbols plus, if they exist, the padding symbols.

LDPC and HDPC Codes: The two top parts of matrix A are used as *constraints* that establish pre-coding relationships amongst the intermediate symbols. Each of the first $S + H$ rows of the matrix defines an equation that relates a subset of intermediate symbols, in such a way that their sum is equal to zero. These constraints are specified in some detail in the RFC, and sometimes they have a simple format (e.g., three 1's per column in G_LDPC_1) while in others they require exponentiations and pseudo-random numbers (e.g., in the G_HDPC).

Both the LDPC and LT parts of the matrix have equations with coefficients in the finite field $\mathbb{F}_2$ (i.e., 0 or 1). The HDPC code defines equations with values belonging to the finite field $\mathbb{F}_{256}$. Using a code over $\mathbb{F}_{256}$ in the constraints greatly contributes to a better overhead-failure performance of RaptorQ (the rationale is discussed in detail in Section 3.3.1 of [32]). The drawback is that some of the mathematical operations can become more complicated to implement (e.g., multiplication). The standard addresses this difficulty by providing a way to perform these operations with the help of a few table lookups.

LT Code: The LT part is responsible for actually pre-coding the source symbols into the intermediate symbols. The LT codes encoding procedure relies on two random number generators. In the RaptorQ standard, these generators are hidden inside a "Tuple Generator" function that carefully substitutes them with pseudo-random generators, which have the useful characteristic of ensuring effective decoding. "Tuple Generator" is defined in the RFC with the help of the `Enc` and `Tuple` procedures (Sections 5.3.5.3 and 5.3.5.4 of [5]). The function receives as input the identifier of a symbol (ISI) and the total number of symbols in the extended source block (K'). As

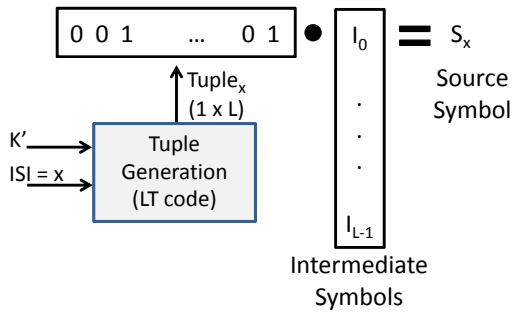

Fig. 4. Computing the repair symbols during encoding.

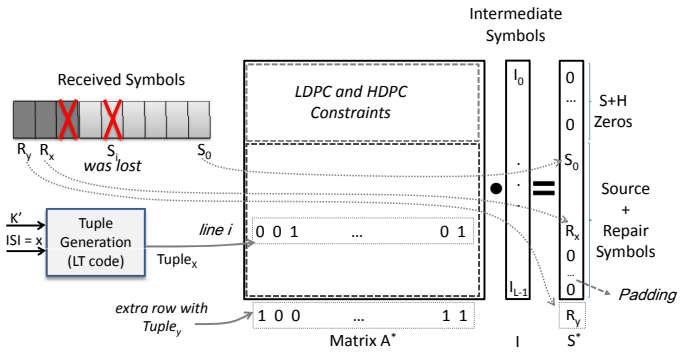

Fig. 5. Computing the intermediate symbols during decoding.

result, it produces a tuple with L entries with values of either 0 or 1. Therefore, the tuple corresponds to a vector with one dimension (or an equation) that can be added to matrix A. For example, when the "Tuple Generator" is called with ISI equal to 0, the returned tuple is placed in row 0 of the LT part of matrix A.

In Section IV, we will explain that by predicting the outcome of the "Tuple Generator", the adversary can insert specific equations in the matrix.

Resolving the system: The system of linear equations $A \cdot I = S$ has to be solved so that the intermediate symbols can be obtained. In other words, it is necessary to find the inverse matrix of A to get $I = A^{-1} \cdot S$. In the encoding process, the inverse matrix can always be found because the equations are constructed in such a way that A is full rank.

Since this task is the most time consuming of the whole algorithm, it is recommended that an optimized method is utilized. A potential candidate is based on a *permanent inactivation* technique mentioned later in the section. The reader should notice that this part of the process actually corresponds to a decoding operation – the intermediate symbols are being *recovered* from well known information (the source symbols), so that they can be used to create the repair symbols.

4) Producing the repair symbols: The final step of encoding is depicted in Figure 4 and corresponds to the calculation of the *repair symbols*. This step is accomplished through a very efficient procedure that applies a LT code to the intermediate symbols. The "Tuple Generator" is again called, but now with the identifier of the target repair symbol ($ISI = x$). The returned tuple has L entries with values of either 0 or 1. Next, the repair symbol is obtained by multiplying the tuple by the intermediate symbols vector. This multiplication corresponds in practice to an XOR of some of the intermediate symbols as: (i) the entries of the tuple with value 1 choose the symbols that are utilized in the calculation, and (ii) the addition of two symbols is implemented with an XOR (in the finite field $\mathbb{F}_{256}$ the add operation is an octet-by-octet XOR).

An alternative way of viewing this step is to recall the system of linear equations of Figure 3. The tuple corresponds to an extra equation that is placed at the bottom of matrix A, while the repair symbol is an additional entry at the end of vector S. Therefore, after resolving the pre-coding constraints, the matrix A together with the I vector can be utilized to generate all encoding symbols (both the source[5] and the repair). Moreover, as many repair symbols as desired can be produced on the fly only by changing the ISI provided to the "Tuple Generator" and by appending additional equations.

To summarize the encoding procedure: the extended source block is constructed; then, a system of linear equations containing the pre-coding constraints is solved to get the intermediate symbols; the repair symbols are calculated using a "Tuple Generator" that selects a subgroup of the intermediate symbols to be XORed together. Finally, the set of the original source symbols together with the repair symbols become the *encoding symbols*.

C. The decoding process

The decoding process is actually very similar to encoding. The decoder is assumed to know the structure of the source block (e.g., K, T), as this configuration information must be conveyed by the transmitter. Therefore, the decoder knows how to create the extended source block, incorporating the appropriate padding symbols. To decode the extended source block, let us assume that the receiver gets $N \geq K$ encoding symbols for that block. If all source symbols arrive, then decoding is immediately finished. Otherwise, the construction of a system of linear equations takes place, similar to the previous one (Figure 3). There are differences in the systems of equations mainly due to a couple of minor issues: (i) any equation of a missing source symbol is replaced by an equation corresponding to a repair symbol (in the LT part); (ii) if additional repair symbols are received, they may also take part of the system of equations, ensuring a much greater probability of successful decoding (these are the overhead symbols).

Figure 5 provides an example decoding operation. There are six source symbols and three repair symbols, and one of each was lost in the network: source symbol S_i was dropped and repair symbols R_x and R_y were received. As described in Section III-B3, a system of linear equations $A \cdot I = S$ needs to be built in order to calculate the intermediate symbols. However, one of the source symbols is missing. Even though we are able to determine its associated equation in line i of the LT part of matrix A, since we do not know its value, we cannot complete vector S.

[5]Of course, in practice this is obviously unnecessary, since the source symbols are already available.

However, two repair symbols arrived, R_x and R_y. Using the ISI of the first, x, we can generate a tuple using the "Tuple Generator", which can then be used to replace S_i's row in matrix A. The new matrix has the R_x's tuple in the i row and uses R_x's value in vector S. Let us call the new matrix and vector A^* and S^* respectively. Additionally, since R_y is available, the "Tuple Generator" can be employed to create the tuple y, which is inserted as an extra equation of the matrix.

We have now a complete system: $A^* \cdot I = S^*$. It can be solved by inverting A^*, such that $I = A^{*-1} \cdot S^*$. However, in contrast to the encoding process with matrix A, it can happen that A^* is not invertible. In fact, if there is no A^{*-1}, we have a *decoding failure*. In spite of that, there is an extremely high probability that the system of linear equations can be resolved, as we will see in Section III-D, when decoding failure probabilities are analyzed.

The extra rows of A^* greatly increase the probability that the matrix is invertible (due to repair symbols like R_y). Since there are more rows than columns, with certainty there will be a linear dependency among the equations of A^*. However, this creates no difficulties because after A^* is reduced to its *row echelon form*, only L equations should remain. Since there is a larger number of rows, it is less probable that one cannot find a set of L rows that are linearly independent. Hence, the higher probability of A^* 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 an equivalent fashion to generating the repair symbols (see Figure 4), namely by: (i) calling the "Tuple Generator" with the ISI of the target source symbol, to compute the tuple that selects the intermediate symbols to be XORed, and (ii) XOR those intermediate symbols which will result in the missing source symbol.

To solve the system of linear equations, instead of using regular Gaussian elimination techniques such as LU decomposition, an algorithm that benefits from the specific structure of the system of linear equations is recommended, namely one that explores the fact that matrix A^* is sparse.

1) Permanent inactivation decoding: Permanent inactivation is a technique that overcomes many of the shortcomings of the dynamic inactivation of R10. It is used to solve the system of linear equations of RaptorQ in an efficient way, contributing for a significantly better performance (Section 5.4 of [5]). In permanent inactivation, a subset of the intermediate symbols is designated as inactive before decoding starts.[6] It is executed in five phases, which transform matrix A^* into successively simpler formats, allowing in the end an easy calculation of the unknowns (or the discovery that there was a decoding failure).

In Section IV, our attack will focus on making matrix A^* non-invertible, thus forcing a decoding failure.

D. Evaluation of overhead / failure probability

One of RaptorQ's greatest advantages is its steep overhead-failure curve. In regular circumstances, it is highly unusual for the decoding process to fail, which is important as this type of codes may be employed in mission critical scenarios. RaptorQ was shown to have a failure to decode probability that closely matches the performance of an ideal fountain code [33], approximating the model of (1):

$$Pf_{RQ}(N,K) = \begin{cases} 1 & \text{if } N < K \\ 0.01 \times 0.01^{N-K} & \text{if } N \geq K \end{cases} \quad (1)$$

As expected, the decoder cannot recover the original data if it gets less encoding symbols N than the number of source symbols K. However, when more encoding symbols arrive, the probability of failure declines very rapidly, and with only two additional symbols decoding is successful almost every time (i.e., with an overhead of $o = N - K = 2$, failures occur once in a million times). This is generally more than enough for the vast majority of applications using UDP.

As part of our investigation, we also wanted to analyze (and confirm) the performance of RaptorQ in the field, under a straightforward communication scenario. Since no public domain distribution currently exists for the standard, we created our own implementation[7]. RaptorQ execution was evaluated in the following way: input data was generated for the values of K equal to 10, 26 and 101; next, the data was encoded and enough encoding symbols were generated to tolerate random network losses of 10%, 20%, 50%, 60% and 85%; after this, decoding was attempted with the remaining encoding symbols. The experiments were carried out for the most commonly recommended overheads (i.e., $o = N - K$, with $o = 0...2$), and each test was repeated between 20 million and 30 million times to get a reasonable level of confidence in the results.

TABLE I. DECODING FAILURE PROBABILITY FOR A CODE OVERHEAD OF 0, 1 AND 2 SYMBOLS AND A NETWORK LOSS RATE RANGING BETWEEN 10% AND 85%.

	Number of Source Symbols (K)								
	Overhead 0 [$\cdot 10^{-3}$]			Overhead 1 [$\cdot 10^{-5}$]			Overhead 2 [$\cdot 10^{-7}$]		
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 I displays the observed failure probabilities. These results fall in line with the evaluation found in Appendix B.3 of [32] and they confirm the reliability claimed by the RaptorQ standard. The failure probability is very small in all experiments, in the order of 10^{-3} for $N = K$ and decreasing rapidly to 10^{-7} for two overhead symbols. In some tests, we never observed a decoding failure, while in several others the failures were only seen once or twice. For instance, in our tests for $K = 10$ we only observed failures with 60% network loss.

IV. BREAKING THE RAPTORQ STANDARD

One of the most interesting properties of FEC codes is the ability to use the same repair symbols to recover from different

[6]For example, the "Tuple Generator" sets a few entries in the tuple to 1, which correspond to the *permanently inactive* (PI) symbols.

[7]The source code is available under an Open Source license at: http://www.lasige.di.fc.ul.pt/openrq/

independent packet losses at various receivers. *Independent packet losses* must be emphasized, as the erasure patterns are mainly the result of a random process that causes very diverse packet drops (from single events to large bursts). As stated in the book *Raptor Codes* [32], written by two of the authors of RaptorQ's RFC [5], the code was built under the assumption:

> *"... 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."*

In a *benign* environment this sort of reasoning is completely acceptable, as faults are of an accidental origin. However, in a malicious setting, an adversary can attempt to break the "independence" assumption to prevent the code from recovering the original data even if "enough" packets arrive.

A. Successful attack

A successful attack occurs when the adversary is able to foil the correct recovery of one or more source blocks at the receiver. In this case, the recipient cannot obtain the full transmitted data, which corresponds in many practical scenarios to no information being retrieved. For example, if some authentication/integrity check needs to be performed before the content is utilized, missing a small portion of the data normally precludes the validation of the rest (e.g., if the check is based on a signature over a hash of a file).

In order to compromise the FEC operation, the adversary may try to corrupt the stream of packets, either by changing their content or by inserting erroneous packets. The standard of RaptorQ already discusses this sort of malicious action (Section 6 of [5]), and provides a solution based on source authentication and integrity checking. The effect is that the modified/inserted packets are simply eliminated at the destination, in fact transforming them into erasures that are trivially addressed by the code. Another form of disruption would be to change the information of configuration (e.g., the symbol size) forwarded by the sender to the receiver. Once again, the standard recommends the use of source authentication to take care of this issue.

Our attack is different because it only performs erasures, which typically should be well tolerated by the code. Furthermore, we strive to be stealthy in the sense that enough encoding symbols are allowed to reach their destination so that decoding can be attempted. Consequently, this rules out trivial attacks where a source symbol and all repair symbols are erased.

B. Adversary model

The adversary is capable of listening to the information being exchanged among the peers, and it is assumed that she is not one of such peers. Therefore, she is either on the path of the packets or she can have them be rerouted through a link under her control. Additionally, the adversary can drop specific packets in the network. In order to have this sort of capability, the adversary could control one of the routers near the sender. Alternatively, if she has physical access to the link, certain frames can be read and corrupted to force their removal. As we will see, in an initial version of the attack, the adversary uses the headers of the packets to decide on which ones should be eliminated.

The adversary necessitates an extra capability in the extended attack where the FEC operation is disturbed with encrypted channels (e.g., with IPsec). She needs to observe the flow of packets before they suffer reordering or losses in the network. This means that the attack has to be executed in a place in the network where the sequence of packets arrives without being altered.

Of course, sometimes the attack may have to be carried out in less than optimal conditions (i.e., there are certain periods of time where our assumptions do not hold). When this occurs, the adversary may erase the wrong packets, and enough information will arrive at the receiver to allow a source block to be retrieved. However, when the network starts to behave as expected, the adversary becomes effective again at causing decoding failures. Consequently, in this case, the attack is probabilistic.

C. Rationale for the attack

The transmitter node normally operates by dividing the original data in a set of source blocks, which are then processed individually (see Figure 1). The source block is further partitioned into K source symbols that are sent together with a number R of repair symbols. The value of R is normally calculated by the sender based on the locally perceived loss rate of the network. For example, for a loss rate of l and a desired decoding overhead of o, the minimum number of repair symbols is (2):

$$R \geq \left\lceil \frac{lK + o}{1 - l} \right\rceil \tag{2}$$

With this level of redundancy, the receiver can get enough encoding symbols ($K + o$) that with a very high probability enable the source blocks to be rebuilt and eventually the whole data. Of course, in some settings, this solution for reliable communication could be complemented with additional mechanisms to address the rare cases when decoding does not succeed. For example, a retransmission protocol could be used afterwards or extra repair symbols could be generated and transmitted. However, since these mechanisms are not considered in the RaptorQ standard and are application specific, we will disregard them and focus on how to disrupt the common part of the execution.

To defeat the FEC operation it is sufficient to erase one of the source symbols and then force the decoding process to fail in the associated source block. The remaining packets can go through and no extra drops are necessary. Therefore, the efforts can be concentrated in breaking the recovery of a single source block, as this is enough to ensure a successful attack. One however should keep in mind that, since decoding is similar in every source block, it is easy to replicate the malicious actions with the remaining ones (or a subset of them).

One simple solution to forcefully cause a decoding failure would be to eliminate a few source symbols and all repair

symbols. This however would be an obvious Denial-of-Service (DoS) attack, which is inelegant and could be trivially discovered. To increase the difficulty of detection, the attack should be performed in such a way that erasures are somewhat similar to the losses that are usually observed in a network, but with a much higher impact – these erasures happen to halt source block recovery.

As explained in Section III-C, once the RaptorQ decoder computes the intermediate symbols, then the decoding process is definitely successful – every source symbol can be obtained without the need for more packets to be transmitted. Consequently, it is necessary to hinder the calculation of the intermediate symbols. Since these symbols are calculated as the result of solving a system of linear equations, the adversary's objective should be to impede the completion of this task. There are two possible outcomes associated with the failure of finding a solution for a system of linear equations:

1) *no solution* if the system is inconsistent (a.k.a. overdetermined);
2) *many solutions* if the system is consistent but underdetermined.

The first case does not occur with RaptorQ because of the way the equations and repair symbols are computed at the sender. The other case is observed when the number of linearly independent equations is smaller than the number of unknowns. Given that the system of linear equations corresponds to a *coefficient matrix* $A^*_{M \times L}$ and an *augmented matrix* $Ab^*_{M \times (L+1)}$, where $M \geq L$, then for the second case to hold the following condition must be true: ($\texttt{rank}(A^*)$ = $\texttt{rank}(Ab^*)$) & ($\texttt{rank}(A^*) <$ L). This implies that to induce a decoding failure, the adversary must force the rank of the matrix A^* to be inferior to L, i.e., the number of intermediate symbols.

D. Forcing a decoding failure

The encoding symbols are accumulated by the recipient until a set of size $N \geq K$ arrives. Then, the decoder initiates the reconstruction of the source block. Each encoding symbol defines a linear combination (or a constraint) among the intermediate symbols. The linear combination is specified only by the ISI of the encoding symbol and the number of symbols in the extended source block, K'. Accordingly, it corresponds to a linear equation among the intermediate symbols, i.e., a line of matrix A^*.

Matrix A^* has exactly L linear equations if $N = K$, part of them stipulated by the pre-coding relations among the intermediate symbols ($S + H$) and the remaining K' due to the encoding symbols and the padding ($L = S + H + K'$; $K' = K + Pad$, where Pad is the number of padding symbols). If there are more encoding symbols than K, then the matrix will have more linear equations than unknowns, which is helpful to increase the probability that a solution can be found.

The adversary has some sort of network control, which lets her selectively eliminate packets and decide on which packets (i.e., encoding symbols) arrive at the receiver. For example, she can drop one of the source symbols and a number of repair symbols that would replace it. Then, a specific repair symbol would be allowed to go through, to be included in the set of encoding symbols kept in the receiver. The consequence of this action is that when decoding is applied, the system of linear equations of matrix A^* would be modified in one of the equations.

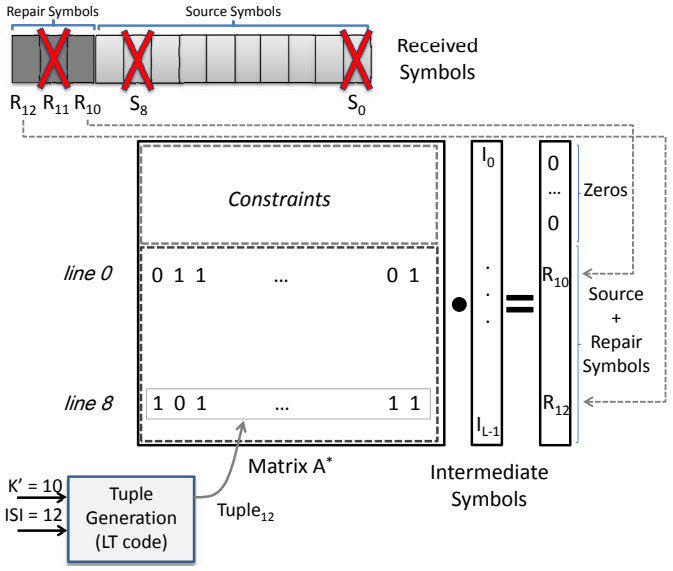

Fig. 6. Example attack for $K' = 10$, with 10 source symbols and 3 repair symbols.

This reasoning shows that the adversary may to some extent define the system of linear equations that is used by the decoder. Therefore, at least in theory, she could make that system of linear equations underdetermined, by letting encoding symbols arrive at the receiver whose constraints (lines in matrix A^*) are *linearly dependent* of the other equations. As a result, the adversary would have decreased the rank of matrix A^* below L, preventing the intermediate symbols from being recovered.

It is interesting to take notice that the attack is completely independent of the data being transmitted. The constraints depend fundamentally on how the standard identifies the encoding symbols, which is something that is completely deterministic. The first encoding symbol has ISI 0, the second has ISI 1, and so on. This means that the selection of which packets should be erased can be pre-computed before the attack, which permits its execution in real-time (and ideally at line speed). Moreover, it suggests that exploitation can potentially happen even if communications are encrypted.

Example: Assume a scenario like the one depicted in Figure 6, where the sender estimates 17% for the average loss rate of the network and there are $K = 10$ source symbols per source block. In this case, three repair symbols are transmitted to tolerate the erasures that might take place (overhead $o = 0$). The adversary removes the first ($ISI = 0$), ninth ($ISI = 8$) and twelfth ($ISI = 12$) packets. When the receiver builds the system of linear equations of matrix A^*, the lines corresponding to the source symbols are replaced by the ones relative to the received repair symbols. However, the resulting matrix A^* would have a linear dependency among the lines, lowering its rank and rendering the system of equations underdetermined.

Algorithm: The attack removes all source and repair symbols but the ones that will cause a linear dependency among the equations of matrix A^*. Potentially, for each value K, there are several erasure patterns that cause this same type of problem, but some will require many more network packets to be eliminated. If this number is high above the average packet loss for a particular network, then the attack becomes harder to put in practice.[8] Consequently, we investigated algorithms that could not only discover the necessary erasure patterns, but that could also minimize the network losses.

An obvious way to increase the chances of introducing a linear dependency in the system of equations is to methodically eliminate each of the source symbols and find a repair symbol that produces a linear dependency. This would support 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, and hence, a scenario could be found where the attack is much more efficient.

Naturally, given the brute force nature of the approach, it could produce a very high number of combinations to be explored. This could prevent results from being obtained in a useful time frame, due to the massive number of computations that had to be performed. One however should notice that the search is quite simple to parallelize, and a large number of machines can be enrolled to run the algorithm concurrently. For example, each machine could execute the algorithm for a distinct value of K. Moreover, once a combination is found, it can be used forever – as long as the standard stays unchanged.

Algorithm 1 describes the approach that we are currently using to discover the erasure patterns. The algorithm receives three input arguments: (1) K is the number of symbols in the source block; (2) o is the decoding overhead that should be simulated; and (3) $upperLimit$ is the maximum number of symbols the attacker is willing to drop. $UpperLimit$ parameterizes the "risk" of the attack, i.e., if too many symbols are dropped then the attack becomes less stealthy. Additionally, it is also associated with the time and computational effort that should be invested to find a solution.

The algorithm starts by setting up the various constants that configure the execution of the code by calling `getParameters` (line 2). This function basically does a table lookup (Section 5.6 of [5]) and a few calculations (Section 5.3.3.3 [5]), and returns the number of symbols in the extended source block K' and the dimensions of the decoding matrix and its submatrices (i.e., S, H and L; recall Figure 3). Next, it initializes the variable $dropS$, which holds the minimum number of symbols that has to be dropped to carry out the attack.

The array `targetEq` is populated with the ISIs of the potential *target encoding symbols* for elimination (lines 4-7). It includes all source symbols, and eventually the repair symbols that would create the overhead equations (the padding symbols are excluded). The array `targetRepair` is initialized with the ISIs of the *target repair symbols* that are available for this attack, i.e., to substitute the deleted encoding symbols (lines 8-9). The decoding matrix is generated assuming that no symbol was lost and that o overhead extra equations are to be employed (lines 10-14; Section 5.3 [5]).

Next, the algorithm initiates the search for erasure patterns that create a decoding failure. Each pattern is defined by two sets containing: (a) the ISIs of target encoding symbols that have to be dropped; and (b) the ISIs of the target repair symbols that should be allowed to go through (the rest have also to be eliminated).

The loop is executed for sets of target encoding symbols with sizes of 1 to $K + o$. For each size, the various combinations of target encoding symbols are found and stored in

Algorithm 1 Find erasure pattern for RaptorQ.

```
1:  procedure FINDATTACK(K, o, upperLimit)
2:      (K', S, H, L) ← getParameters(K)
3:      dropS ← upperLimit
4:      for s ← 0, K − 1 do
5:          targetEq[s] ← S + H + s
6:      for s ← K', K' + o − 1 do
7:          targetEq[s + K − K'] ← S + H + s
8:      for r ← 0, upperLimit − 1 do
9:          targetRepair[r] ← K' + o + r
10:     A ← generateDecodingMatrix(K')
11:     if o ≠ 0 then
12:         for i ← L, L + o − 1 do
13:             tuple ← createTuple(K', i − L + K')
14:             A[i] ← createEq(K', tuple)
15:     for n ← 1, K + o do
16:         while dropS > n do
17:             combinationsEq ← C(targetEq, n)
18:             combinationsRep ← C(targetRepair, n)
19:             for all setEq in combinationsEq do
20:                 if setEq has no source symbol then
21:                     continue
22:                 for all setR in combinationsRep do
23:                     if max(setR) − K' − o + 1 ≥ dropS then
24:                         continue
25:                     i ← 0
26:                     for all r in setR do
27:                         tuple ← createTuple(K', r)
28:                         rEq[i] ← createEq(K', tuple)
29:                         i ← i + 1
30:                     newA ← A
31:                     i ← 0
32:                     for all s in setEq do
33:                         newA[s] ← rEq[i]
34:                         i ← i + 1
35:                     reduceToRowEchelonForm(newA)
36:                     if getRank(newA) < L then
37:                         dropS ← max(setR) − K' − o + 1
38:                         wFile(K, o, dropS, setEq, setR)
```

[8]The adversary could always delete packets to make the sender adjust the average loss rate to higher value, and as a consequence, increase the number of repair symbols that are transmitted. At that point, the attack would become viable again.

TABLE II. NUMBER OF ENCODING SYMBOLS THAT MUST BE DROPPED AND INDUCED LOSS RATE (IN %) OF THE BEST ERASURE PATTERN FOR DIFFERENT NUMBERS OF SOURCE SYMBOLS (K) AND OVERHEADS (OVER).

	Number of Source Symbols (K)													
Over	**10**	**26**	**32**	**42**	**55**	**62**	**75**	**84**	**91**	**101**	**153**	**200**	**248**	**301**
0	3 (23.1)	3 (10.3)	2 (5.9)	2 (4.6)	2 (3.5)	2 (3.1)	2 (2.6)	2 (2.3)	1 (1.1)	2 (1.9)	2 (1.3)	1 (0.5)	2 (0.8)	3 (1)
1	7 (38.9)	4 (12.9)	5 (13.2)	2 (4.4)	4 (6.7)	3 (4.6)	4 (5)	6 (6.6)	8 (8)	7 (6.4)	3 (1.9)	8 (3.8)	4 (1.6)	2 (0.7)
2	12 (50)	9 (24.3)	7 (17.1)	4 (8.3)	5 (8.1)	5 (7.3)	5 (6.1)	7 (7.5)	4 (4.1)	9 (8)	4 (2.5)	6 (4.7)	11 (4.2)	15 (4.7)
	355	**405**	**453**	**511**	**549**	**600**	**648**	**703**	**747**	**802**	**845**	**903**	**950**	**1002**
0	2 (0.6)	2 (0.5)	1 (0.2)	1 (0.2)	1 (0.2)	1 (0.2)	1 (0.2)	1 (0.1)	1 (0.1)	1 (0.1)	1 (0.1)	2 (0.2)	1 (0.1)	1 (0.1)
1	2 (0.6)	8 (1.9)	2 (0.4)	7 (1.4)	2 (0.4)	4 (0.7)	2 (0.3)	3 (0.4)	8 (1.1)	6 (0.7)	3 (0.4)	2 (0.2)	6 (0.6)	4 (0.4)
2	10 (2.7)	6 (1.5)	14 (3)	50 (8.9)	5 (0.9)	10 (1.6)		7 (1)			57 (6.3)			

variable `combinationsEq` (line 17). Similarly, the sets with the combinations of target repair symbols are also computed. Then, every combination of target encoding symbols $setEq$ is tried with all combinations of target repair symbols $setR$. $SetEq$ has however to include at least one source symbol, since otherwise the recovery of the source block is immediate at the receiver (lines 20-21).

The test to find out if a decoding failure arises with the replacement of $setEq$ with $setReq$ is done in the following way: (1) the repair equations are created, based on the ISIs in $setR$ (lines 25-29); (2) decoding matrix is modified, so that the target encoding symbols equations are substituted by the repair equations (lines 30-34); (3) finally, the matrix is reduced to its row echelon form, so that its rank can be determined. If the rank is lower than L then the attack was successful. The value of *dropS* is updated and the erasure pattern is stored in a file (lines 36-38).

The algorithm includes two optimizations worth mentioning. The exhaustive search part of the algorithm is only executed if the new erasure being targeted is more efficient than the best pattern discovered until that moment (line 16). Additionally, the set of target repair symbols is analyzed to find out if it would require a total number of dropped symbols larger than *dropS* (lines 23-24, where `max` returns the largest ISI in the set). With both optimizations in place, it is possible to greatly reduce the number of tests, especially as better erasure patterns are discovered. Both cases prevent unnecessary executions of `reduceToRowEchelonForm`, which is by far the most computationally complex part of our algorithm.

Note that RaptorQ processes each source block in a deterministic way. Therefore, all of this computation can be performed beforehand, to generate the erasure patterns for every combination of $(K, overhead)$. Therefore, the attack can be made fast, without introducing a detectable lag into the communication. That is, we can run the algorithm once for the target values of K, and then perform the attack as many times as desired by doing simple lookups. The computational requirements of the attack machine can be reduced to a bare minimum, as all it needs to do is get the ISIs of the erasure pattern from some source (e.g., a file) and drop/forward the related packets.

E. Erasure patterns

Algorithm 1 was implemented and tested with 28 example numbers of symbols per source block (K), ranging between 10 and 1002. Other examples could have been tried as the algorithm works for all values of K considered in the standard. For each K, the attack was tried with zero, one and two overhead symbols. For each successful attack, we collected the erasure pattern and counted the number of packets that had to be dropped. As the count is associated with the level of stealth, it can be seen as a measure for the practicality of the attack. The algorithm has been running non-stop for a little over six months in a machine with two Intel Xeon E5520 at 2.27GHz, each with 4 cores and 2 threads per core, and 32GB RAM. Every other week we find novel erasure patterns that either are capable of causing a decoding failure in a new pair $(K, overhead)$ or improve the attack efficiency of previously discovered patterns. The results collected so far are displayed in Table II. Each entry in the table presents the number of encoding symbols that have to be erased and, in parentheses, the induced loss rate percentage ($l = D/(K+o+D)$, where D is the number of erasures). Note that in some cases no erasure pattern has been found yet. This is due to the brute force nature of the approach, which typically requires many combinations to be tested for higher values of K and overhead.

The table shows that for most cases we were able to discover an erasure pattern that could prevent the decoding of a source block. The values for the number of dropped symbols varied a lot, from a few hundreds to just one, which is expected as these are *independent events*. For overheads of 0 extra symbols, the attack is successful in all evaluated K and in most scenarios requires only a single packet to be eliminated. Therefore, just by carefully picking the source symbol to be deleted and letting a repair symbol pass, the adversary can have a massive impact on the failure probability, completely destroying the robustness shown for accidental faults.

For non-zero overheads, the attack is still viable in most of the scenarios. At least one erasure pattern was found for all tests with overhead 1, and we are still experimenting with some values of K for overhead of 2. In some cases, the attack is extremely efficient. For instance, with $K = 648$ and 1 symbol of overhead, the adversary would have to eliminate only 2 symbols (0.31% of the total number of packets in a source block), to force a decoding failure, that, if it were to occur by chance, the probability would be in the order of 10^{-5}. In other cases, the number of packets that has to be eliminated is reasonably high (e.g., 50 for $K = 511$ with 2 overhead symbols). However, if one considers the whole transmission of a reasonably sized file, which involves many source blocks, then the overall loss rate is still acceptable – recall that the adversary only needs to prevent the recovery of a single source block.

Risk Assessment: Note that the discussed erasure patterns were optimized to drop as few encoding symbols as necessary. In most scenarios that would be ideal to preserve the stealthiness of the attack. However, in a few cases this may not necessarily be true. Some networks may lose packets in bursts and loss rate may also vary by orders of magnitude.

Let us consider a scenario where the network loss is high. The sender will compensate for this problem by transmitting more repair symbols. Using one of our erasure patterns would imply dropping very few packets for the attack, and then eliminating several repair symbols (all the ones after the last encoding symbol that should go through). Erasing many packets, namely consecutive packets, may hinder stealthiness.

However, the attack offers more flexibility than that. Instead of going for the most "efficient" erasure pattern, the adversary could use a supposedly worse erasure pattern, which incurs in more packet erasures, but suits better the current network behavior. In some cases an erasure pattern that needs to drop 20 packets may be stealthier than one that only requires 3 packets to be deleted.

Note that there are natural limits to the stealthiness of the attack. If the expectation of decoding errors is very low to begin with, then even a single decoding error may raise suspicion, and two in close succession should trigger an inquiry into the reasons. In practice, however, it may well be the case that a number of decoding errors will be attributed to implementation bugs or other inexplicable behavior such "bad network weather" before an attack is seriously considered.

V. Attacking a TFTP application

This section describes how a Trivial File Transfer Protocol (TFTP) application protected with the RaptorQ FEC was attacked. TFTP is a simple protocol for file transfer, which is specified in IETF's RFC 1350 [34]. TFTP communication is made over an unreliable datagram channel (UDP) and its functionality is basically confined to the reading and writing of files between a client and a server.

a) Application: We modified the TFTP client and server included in the Apache Commons Net module of the Apache Commons project[9], which implements the client side of many Internet protocols, including TFTP. The full distribution also provides a TFTP server.

The operation of the client and server had to be updated in order to take advantage of RaptorQ. Let us look at the use case when a client wants to write a file (a file read works similarly, but the client and server roles are exchanged). The data being transmitted must first be encoded. More specifically, the *write request* (`WRQ`) was extended to also transmit the FEC configuration parameters. After receiving an acknowledgment for the `WRQ`, the file to be uploaded is partitioned into source blocks that are then encoded. The send buffer (`_sendBuffer`) is filled with serialized encoding symbols, and for each `TFTPDataPacket` sent thereafter, its payload is an encoding symbol. On the server side, upon receiving the `WRQ` packet, the FEC parameters are initialized and the server awaits for the data. Instead of writing the data to a file immediately, as in the original implementation, the server stores the received encoding symbols in memory. When enough symbols arrive, the decoding takes place and only then the data for that block is stored in the file.

[9]http://commons.apache.org/

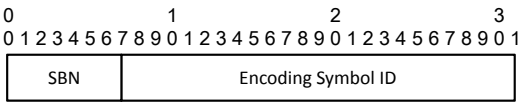

Fig. 7. RaptorQ's FEC Payload ID Format.

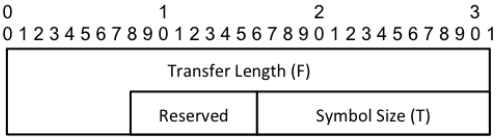

Fig. 8. RaptorQ's Common FEC Object Transmission Information (OTI).

According to RFC 1350 [34], if a packet is lost in the network, the intended recipient will timeout and may retransmit his last packet (which can be data or an acknowledgment). The `WRQ` and `DATA` packets are acknowledged by an `ACK` (or `ERROR`) packet. A `DATA` packet includes an acknowledgment for the latest `ACK` packet. We "relaxed" this communication constraint, so the server would use RaptorQ to tolerate missing `DATA` packets, and the client would not retransmit the previous `DATA` packet if the corresponding `ACK` did not arrive. Naturally, to do that we had to remove most of the timeouts and retransmission mechanisms used, namely in the `handleWrite` and `handleRead` functions at the TFTP server, and the `sendFile` and `receiveFile` functions at the client.

b) Collecting Information: In order to perform the attack, the adversary needs to get some information about the packets being transmitted to decide if they should be eliminated or not. This information includes the identifiers of the encoding symbols being sent and also configuration parameters of the flow, such as the number of symbols per source block. Fortunately, this information is also required by the receiver for correct decoding, and therefore it must be forwarded by the source.

RaptorQ RFC 6330 [5] defines a 4-octet header called the *FEC Payload ID* that must be added to each packet containing an encoding symbol (illustrated in Figure 7). The header has one octet to identify the source block (the SBN) and the remaining 3 octets represent the ESI of the encoding symbol. Since the adversary has access to the network, and the structure of the packets is well defined, she can easily reverse engineer the packets and see the ESIs of each encoding symbol. Therefore, even if the network reorders the packets, this does not have an impact on the attack.

The adversary also has to discover the number of symbols per source block to find out the erasure pattern that should be applied. Furthermore, since the padding is not sent to the network, she would not be able to separate the source symbols from the repair symbols, which could hinder the attack. RFC 6330 [5] describes a *Common FEC Object Transmission Information* (OTI) that is used to transfer information to the receiver, so it can calculate the necessary parameters for decoding (e.g., K and K') (see Figure 8). By intercepting the

packet that contains this information, the adversary can obtain the necessary knowledge (*Transfer Length* and *Symbol Size*) to compute K.

c) The proof of concept attack: The client and server machines were in different VLANs, and there was a third machine acting as a router, connecting both VLANs. In this third machine, we ran a packet sniffing tool that automatically executed the attack. The actual network was a gigabit Ethernet, and the three separate machines had two quad-core 2.27 GHz Intel Xeon E5520 with 32 GB of RAM memory.

We developed a packet sniffing tool that is capable of inspecting and dropping specific packets with the help of the `libnetfilter_queue` library. The netfilter project[10] is responsible for the packet filtering framework inside the Linux 2.4.x kernel and later series. The `libnetfilter_queue` provides an API that allows our tool to retrieve packets from the kernel network stack. The packets are intercepted above the link layer, but before they are handed over to the network layer. Therefore, the tool gets the IP packets, and if necessary, it reassembles the IP fragments. Then, it applies a filter to determine what to do with the packet.

Since the header structure is well-defined and our implementation respects it, it was straightforward to reverse engineer the messages (the same can be said in regards to UDP headers). All UDP packets were analyzed, looking for the OTI information so that K could be computed. After the OTI packet was captured, we listened for UDP packets and matched the first 4 octets with the FEC Payload ID. For each matching packet, the filter looked up in a table containing the erasure patterns, to find out if that symbol should or should not be reinjected in the network stack. In more detail, the filter takes the ESI present in the FEC Payload ID, and from K is able to know K' and calculate the symbol's ISI (ISI = ESI + $K' - K$). If the ISI for a source symbol is present in the erasure pattern for K then it would be dropped. However, if it is for a repair symbol it would be forwarded. Otherwise, all source symbols are forwarded and all repair symbols are dropped.

If more repair symbols are allowed to pass than the ones exactly needed, the receiver, in this case the TFTP server, may be able to recover the data. The reason is that it might add extra rows to matrix A^* for these repair symbols, beyond the ones for the normal overhead. The solution for this is for the adversary to drop all other repair symbols apart from the ones that would cause the decoding failure.

Since the table with the erasure patterns was created prior to the attack (using Algorithm 1), the computational requirements were minimal and the introduced latency in the communication was negligible: the router machine had only to perform simple table lookups and decide if the packet should be forwarded or not. As a result, the attack induced the TFTP server into considering that enough encoding symbols had already been received to recover a block of the file. However, when decoding was attempted, it always failed to retrieve that source block. The attack was experimented with the first 10 values of K from Table II with overheads 0, 1, and 2. In our environment, all attacks were successful.

[10]http://www.netfilter.org/

d) Extending the proof of concept attack: If other packet losses occur besides the ones resulting from the attack, the receiver will not get enough symbols to initiate the decoding process. The sender could transmit more repair symbols to compensate for those erasures. However, the linear dependency is removed when the equations for the missing source symbols are replaced with the equations for those repair symbols – these repair symbols would nullify the attack. A more "attack-friendly" solution would be for the adversary to block all repair symbols being sent (apart from those included in the erasure pattern), but randomly retransmit the encoding symbols that should reach the receiver. Since UDP is being used, packet duplication is expected and dealt with at the destination. This way, the adversary can make the communication more reliable, ensuring the arrival of all necessary encoding symbols.

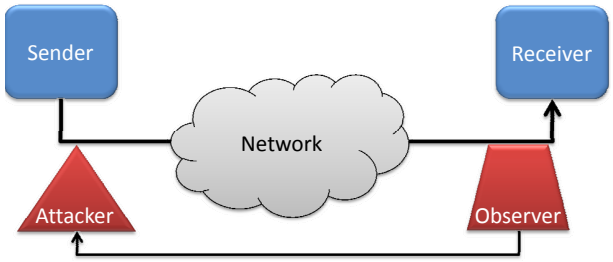

Fig. 9. Attack using two infected machines: one to attack, and the other to observe which packets are reaching the receiver.

A more elegant way to achieve the same result, but also requiring more craftsmanship from the adversary, is to compromise another machine nearer the destination. This machine would communicate with the machine performing the attack via a (reliable) side channel, indicating which encoding symbols are reaching the receiver. We illustrate this in Figure 9. With the help from the observer machine, the adversary could retransmit the missing source symbols more precisely.

VI. Attacking over Secure Channels

In addition to being used for broadcast networks and media delivery services, Raptor codes have been adopted for military operations and other mission critical systems. Due to the criticality of the scenarios, it is relevant to study the code's reliability also when communication is made over secure channels. This is important because in critical scenarios RaptorQ might be used together with complementary protection mechanisms, as is suggested in IETF's RFC 6363 [20]. An attack could be, for example, against the privacy of the communication (e.g., to access non-free content) or to corrupt the packets (forcing the receiver to decode the data only to find it is garbage). Through out Section 9 of RFC 6363 [20] one of the most proclaimed solutions is the use of IPsec/ESP at the network layer.

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 fact that ISIs are associated sequentially to the symbols (always beginning at 0 in each source block), 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, an adversary 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 adversary needs to know the value K because it is crucial to determine the erasure pattern that should be applied; (2) the packets may be reordered or lost by the network, so the adversary will not be able to know if a certain packet corresponds to a specific ISI. In what regards to the latter, this may be easy to address, as long as the attack machine is near the source. Normally, packets are transmitted in the order of their ISIs to the network, and therefore the adversary only needs to count the packets.

Moreover, a minor hindrance is the fact that the value of K may vary for different source blocks. However, if the implementation follows the standard's partitioning algorithm, *if* K changes, it should be only once and by of a factor of 1 (e.g., the first source blocks would have K source symbols and the remainder would have $K + 1$).

a) Collecting Information: Since messages are encrypted, the adversary cannot see the content of the OTI as in the clear-text attack. In some deployment cases, it might be reasonable to assume that the adversary knows the value of K because it could be a fairly static parameter. If that is the case, the attack can be executed by counting and dropping packets accordingly to the erasure pattern. It may also be reasonable to assume that K is one amongst a small group of pre-selected values. In this situation, the adversary only needs to try the attack for the various possible values in the group, until one of them is successful. If the adversary has no idea on the value of K, then she will not be able to perform the attack because she will not know which erasure pattern should be injected. Following, we present a reasoning that might be used to guess K in some scenarios.

Since the encoding and decoding processes are independent for each source block, there is a risk that applications of RaptorQ encoders perform encoding on demand on a per-source block basis. In such a case, lapses between the packets of consecutive source blocks may become noticeable (as illustrated in Figure 10). An adversary can exploit this information leak to make educated guesses about the value of K. Knowing K in turn re-enables the attack.

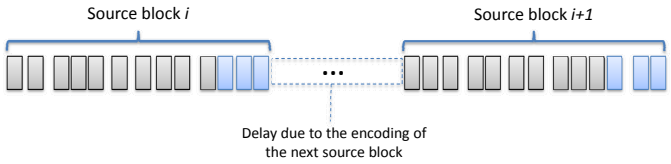

Fig. 10. Time lapse between packets of different source blocks.

Consider the scenario of Figure 10, where $K = 10$ source symbols and three repair symbols are transmitted through the secure channel per source block. The adversary 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 try the attack vector corresponding to $K = 13$; the attack would fail, but she could experiment with the attack vectors for other values of K, until one of them is successful. So, this sort of trial and error can yield positive results from the point of view of an adversary. Recall that it is sufficient to prevent one of the source blocks from being recovered to preclude the full transmission of the file from client to the server.

It is important that application developers keep this threat in mind when deploying RaptorQ codes in their implementations. A buffered sending mechanism might already suffice to smooth the timing patterns so that the attack is prevented. As far as the authors are concerned, a risk of this kind is worth being mentioned in the RFCs to make sure implementors are well aware of it and may take the necessary precautions.

b) The attack: The setup of Figure 11 was used in the experiment, based on same hardware as the previous attack. The communication between the TFTP client ($Node$ 1) and the server ($Node$ 2) was made with UDP over a IPsec [35] channel using AH+ESP. In this environment, we observed that the time lapse between the last packet of a source block and the first packet of the next source block was on average at least two orders of magnitude greater than the average interval between two consecutive packets of the same source block. Specifically, the lapse between source blocks was in the order of milliseconds, while the interval separating two packets was around tens of microseconds.

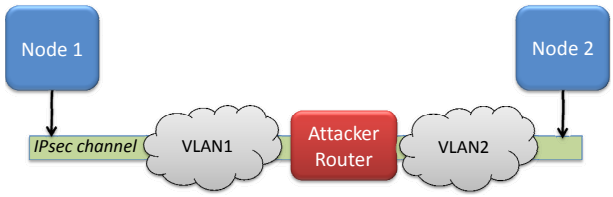

Fig. 11. Setup used to attack over secure channels.

Note that the approach used by our packet sniffer fits perfectly the needs for the attack over IPsec. With the correct filter we can make the router machine behave just like an infected machine would and drop packets at the network level, never needing the data from the protocols higher in the network stack. The filter was parameterized with the expected upper limit for the time lapse between packets of the same source block. It counts each packet going by, and when there is a difference greater than that upper limit, it is assumed that there is a "source block change". Then, the attack can begin. The filter drops packets accordingly to the erasure pattern being injected, which is based on the number of packets counted. Every time a new source block is detected, the filter tests the next lower value of K, with the expectation that eventually the right value is used.

In our case, since we had access to all the machines, we could see in $Node$ 2 when the decoding for a source block was unsuccessful. One of the scenarios we tested had a reasonably large file, with twelve source blocks each with $K = 453$ encoding symbols. The decoding was carried out with zero overhead, but one repair symbol was sent (in order to tolerate one missing encoding symbol). Recalling our results in Table II, for this case there was the need to drop only one of the source symbols, namely the 101^{st}. Furthermore, since our network environment was fairly small and controlled, the packets arrived in order to the attack router, which simplified the attack. In bigger and more complex networks, this might

not be the case, which will require several tries until a source block is not recovered.

VII. Discussion

RaptorQ was not designed to be resilient against malicious faults. However, due to the criticality of the environments where it may be deployed, it is advisable to consider the possibility of attacks. The RaptorQ RFC already includes a few security considerations, namely related to: (1) denial-of-service attacks where an adversary corrupts/inserts packets that would be seen as legitimate by the receivers, causing the computational cost of decoding, only to recover unusable data; and, (2) if an adversary forges a session description (in a multicast delivery), then the receivers would employ incorrect parameters for decoding. Both of these concerns can be solved with authentication, integrity and reverse path forwarding checks.

Nevertheless, none of these solutions would actually be able to prevent our attack. The attack explores intrinsic characteristics of the code and the highly deterministic operation imposed by the standard. Encrypting the messages may protect the FEC operation, but in the end it can still be disrupted. Even if the code implementation does not follow to the letter the RFC (e.g., substitutes one the described functions), the target ISIs for elimination would change, but the code could still be compromised as long as the same base design was followed.

A. Extend the attack to the R10 code

The rationale for 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.[11] As a consequence, the R10 code [4] should also be vulnerable. Moreover, it should be easier to defeat the execution, as the R10 overhead / failure probability curves are less robust than the ones of RaptorQ [18]. The impact can be significant as R10 appears in at least eleven standards (see pages 43 to 45 of [32]), covering areas related to satellite communications, IPTV and digital video broadcasting, mobile and Internet multicast services.

In order to address the attack, the implementations should take that into consideration appropriate mechanisms to circumvent the identified limitations. In the remainder of this section, we will propose and discuss a few solutions.

B. Removing the vulnerability

A very straight-forward way of solving the problem is for the receiver to request from the sender (or from some other alternative source) any missing symbols it needs, or, to ask for more repair symbols. Obviously, this approach should only be used sparingly, as it goes against the nature of fountain codes and an adversary may still be able to drop the extra packets (at the cost of losing her stealthiness). Nevertheless, it was included in the 3GPP MBMS standard to tolerate accidental decoding failures of R10 [6].

[11] With regard to the last point, there is probably nothing to be done about this because with pure randomness it would be impossible to recover the data.

Another solution would be to use encrypted communication and transmit the encoding symbols in a random order. An attacker would not be able capture the information on the OTI, and even if she managed to get the value of K, she would not be able to apply the erasure pattern effectively (only with a pure lucky guess). Clearly, this approach would only be successful with encrypted channels, which could limit its applicability.

A more elaborate solution would resort to a *cryptographically secure pseudo-random number generator* (CSPRNG) [36]. A CSPRNG produces deterministic pseudo-random numbers from a high entropy seed, with properties that make it appropriate for secure systems. During the setup of the communication, the encoder and decoders would privately agree on an initial seed for the CSPRNG (e.g., the source could generate the seed and distribute it through a secure channel to the recipients). Then, the source would (1) associate to the encoding symbols pseudo-random ESIs obtained with the CSPRNG (e.g., ISI = 0 is ESI = 5512; ISI = 1 is ESI = 12567; etc), and it (2) would reorder how the encoding symbols are transmitted, interleaving in an unpredictable way the source and repair symbols. Since the destination is able to generate the same pseudo-random numbers, it can create the mapping between the ISIs and the ESIs. Therefore, the decoder would know how to translate the ESIs of the arriving symbols to the ISIs, and setup the decoding matrix with the correct equations. Naturally, this solution only works as long as the adversary does not compromise a legitimate peer, from whom she could learn the actual parameters.

Of course, the effectiveness of this technique increases with the number of source symbols per source block, as there are more combinations for the ordering of the packets. Nevertheless, it should be sufficient to protect the code under our threat model, even when communications are in the clear. In order to compromise this solution, the adversary would have either to discover the seed of the CSPRNG or be able to reverse engineer the ISI-ESI mapping from the ESIs and the data included in the packets (something not trivial to achieve, especially with encrypted channels).

A more secure solution, but requiring more changes to the standard, is to employ a CSPRNG to generate the ISIs that are assigned to the source and repair symbols (e.g., ISI = 5436 is ESI = 0; ISI = 912 is ESI = 1; etc). Since the equations used in the encoding/decoding matrix are created in accordance with the ISI, this would make the system of linear equations to be in part pseudo-random. As before, decoding is possible because the receivers can rebuild the ISI-ESI mapping. The main advantage of this solution is that it would be much harder to discover the erasure patterns prior to the attack and reverse engineer matrix A from the content of the packets. Moreover, it would allow source symbols within a source block to be identified with consecutive values, respecting the guidelines suggested in [20].

VIII. Conclusions

The main goal of this paper is to study the effect a malicious adversary can have on the robustness of the RaptorQ standard. It shows that in many configurations it is possible to

break the forward error correction capabilities of RaptorQ with a small number of selected erasures. Consequently, the code is stopped from fulfilling its mission of providing protection against arbitrary packet losses with high probability.

The attack requires the ability to listen and drop packets, and it can be performed at line speed even if the communication flows are secured with IPsec. It exploits the way the standard associates identifiers to the (source and repair) symbols, to predict the linear equations included in the encoding/decoding matrices. As long as the adversary can discover the coefficients of the equations, she can force the injection of linear dependencies into the system of equations, and thus preclude the recovery of packet losses.

As explained, the attack is general and takes advantage of the RaptorQ standard's own design. Therefore, it should be possible to extend the attack to any Raptor code specification that follows a similar organization, such as the R10 code [4]. This indicates that our attack may also have practical implications in other previously standardized codes.

Some solutions were proposed to address the identified limitations. The most interesting approach uses a CSPRNG, and renders the attack *impractical*[12] in our threat model, both in encrypted channels and clear-text. A solution based on this technique could be adopted into the standard, but also, it could be easily integrated with any existing implementations.

ACKNOWLEDGMENTS

We would like to thanks the reviewers for their comments, and in particular Volker Roth for his suggestions that helped to improve the paper. We would also like to thank the support by the EC through projects FP7-607109 (SEGRID) and FP7-257475 (MASSIF), and by the FCT through the SITAN project (PTDC/EIA-EIA/113729/2009) and the LaSIGE Strategic Project (PEst-OE/EEI/UI0408/2014).

REFERENCES

[1] D. MacKay, *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2005.

[2] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege, "A Digital Fountain Approach to Reliable Distribution of Bulk Data," in *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1998, pp. 56–67.

[3] A. Shokrollahi, "Raptor Codes," *IEEE Transactions on Information Theory*, vol. 52, no. 6, pp. 2551–2567, June 2006.

[4] M. Luby, A. Shokrollahi, M. Watson, and T. Stockhammer, "Raptor Forward Error Correction Scheme for Object Delivery," IETF, RFC 5053, October 2007.

[5] M. Luby, A. Shokrollahi, M. Watson, T. Stockhammer, and L. Minder, "RaptorQ Forward Error Correction Scheme for Object Delivery," IETF, RFC 6330, August 2011.

[6] 3GPP, "Multimedia Broadcast/Multicast Service (MBMS): Protocols and codecs (Release 11)," 3rd Generation Partnership Project, TS 26.346, September 2013.

[7] Digital Fountain and Siemens, "Specification Text for Systematic Raptor Forward Error Correction," *TSG System Aspects WG4 PSM ad hoc S4-AHP238*, April 2006.

[8] Digital Video Broadcasting (DVB), "IP Datacast over DVB-H: Content Delivery Protocols," *ETSI TS 102 472 v1.2.1*, March 2006.

[9] Open Mobile Alliance, "File and Stream Distribution for Mobile Broadcast Services," *Mobile Broadcast Services V1.0*, February 2009.

[10] ——, "Broadcast Distribution System Adaptation - IPDC over DVB-H," *OMA-TS-BCAST_DVB_Adaptation-V1_0-20080226-C*, 2008.

[11] Digital Video Broadcasting (DVB), "Transport of MPEG-2 TS Based DVB Services over IP Based Networks," *ETSI TS 102 034 V1.4.1*, 2009.

[12] ——, "DVB Document A131," *MPE-IFEC*, November 2008.

[13] ——, "Interaction Channel for Satellite Distribution Systems," *ETSI EN 301 790 V1.4.1*, vol. 301, p. 790, 2005.

[14] DVB, "Transport of MPEG-2 TS Based DVB Services over IP Based Networks," Digital Video Broadcasting, DVB Document A86, June 2012.

[15] ATIS IIF, "IPTV ARCH Specification: Media Formats and Protocols," *WT 18*, 2009.

[16] Telecommunication Standardization Sector of ITU, "Series H: Audiovisual and Multimedia Systems: IPTV multimedia services and applications for IPTV – General aspects," *Recommendation ITU-T H.701*, March 2009.

[17] Qualcomm, "RaptorQ Forward Error Correction Technology Overview and Use Cases," Qualcomm Technologies, Qualcomm Research Presentation, October 2012.

[18] C. Bouras, N. Kanakis, V. Kokkinos, and A. Papazois, "Embracing RaptorQ FEC in 3GPP Multicast Services," *Wireless Networks*, vol. 19, no. 5, pp. 1023–1035, July 2013.

[19] M. Watson, T. Stockhammer, and M. Luby, "Raptor Forward Error Correction (FEC) Schemes for FECFRAME," IETF, RFC 6681, August 2012.

[20] M. Watson, A. Begen, and V. Roca, "Forward Error Correction (FEC) Framework," IETF, RFC 6363, October 2011.

[21] S. Kent, "IP Encapsulating Security Payload (ESP)," IETF, RFC 4303, December 2005.

[22] X. Zhang, S. F. Wu, Z. Fu, and T.-L. Wu, "Malicious Packet Dropping: How it Might Impact the TCP Performance and How we can Detect it," in *Proceedings of the International Conference on Network Protocols*, 2000, pp. 263–272.

[23] K. Bradley, S. Cheung, N. Puketza, B. Mukherjee, and R. Olsson, "Detecting Disruptive Routers: A Distributed Network Monitoring Approach," *IEEE Network*, vol. 12, no. 5, pp. 50–60, 1998.

[24] M. Just, E. Kranakis, and T. Wan, "Resisting Malicious Packet Dropping in Wireless Ad Hoc Networks," in *Ad-Hoc, Mobile, and Wireless Networks*. Springer, 2003, pp. 151–163.

[25] J. Lopes and N. Neves, "Robustness of the RaptorQ FEC Code Under Malicious Attacks," in *Inforum*, 2013.

[26] W. Huffman and V. Pless, *Fundamentals of Error Correcting Codes*. Cambridge University Press, 2003.

[27] B. Cipra, "The Ubiquitous Reed-Solomon Codes," *SIAM News*, vol. 26, no. 1, 1993.

[28] R. Gallager, "Low-Density Parity-Check Codes," *IRE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, 1962.

[29] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann, "Practical Loss-Resilient Codes," in *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*. ACM, 1997, pp. 150–159.

[30] C. Shannon, "Communication in the Presence of Noise," in *Proceedings of the IRE*, vol. 37, no. 1. IEEE, 1949, pp. 10–21.

[31] M. Luby, "LT Codes," in *Proceedings 43rd Annual IEEE Symposium on Foundations of Computer Science*. IEEE, 2002, pp. 271–280.

[32] A. Shokrollahi and M. Luby, *Raptor Codes*. Now Publishers Inc, 2011.

[33] Qualcomm Incorporated, "Rationale for MBMS AL-FEC Enhancement," *TSG-SA4#64 Meeting, TDoc S4-110449*, April 2011.

[34] K. Sollins, "The TFTP Protocol (Revision 2)," IETF, RFC 1350, July 1992.

[35] R. Oppliger, "Security at the Internet layer," *Computer*, vol. 31, no. 9, pp. 43–47, 1998.

[36] E. Barker and J. Kelsey, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators," National Institute of Standards and Technology, NIST Special Publication 800-90A, January 2012.

[12] The attack is not really impossible to execute, 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.