Robustness of the RaptorQ FEC Code Under Malicious Attacks

José Lopes and Nuno Ferreira Neves

Faculdade de Ciências da Universidade de Lisboa, 1749-016 Lisboa, Portugal
jlopes@lasige.di.fc.ul.pt
nuno@di.fc.ul.pt

Abstract. The RaptorQ code is the most advanced standardised fountain code. Its properties make it very attractive for forward error correction, offering great reliability even at low overheads (i.e., for a small amount of repair information), but also admirable encoding and decoding performance. Since the code is relatively recent and the standard is complex, we are in the process of developing the first[1] public domain implementation of RaptorQ (still a prototype). In the paper, we start by verifying the robustness of our implementation and comparing the results to the ones available in the literature. Then, we discuss an attack to the RaptorQ standard that can increase the probability of decoding failure to 100%, which could be carried out by any malicious adversary with network access. Such an attack puts the standard's robustness at stake, and should be carefully considered when deploying RaptorQ code in mission critical networks/systems.

1 Introduction

In coding theory, *fountain codes* (a.k.a. *rateless erasure codes*) are a class of erasure codes with the property that a potentially limitless sequence of encoding symbols can be generated from a given set of source symbols. They also have the property that the original source symbols can ideally be recovered from any subset of the encoding symbols of size equal to or only slightly larger than the number of source symbols. The name *fountain* or *rateless* refers to the fact that these codes do not exhibit a fixed code rate. The output packets of *fountain encoders* must be universal and hence be useful independently of the time or the state of a user's channel.

A file, which is to be split into k packets (or source symbols) and is to be encoded for a *Binary Erasure Channel* (BEC), could be transmitted with an ideal digital fountain code ensuring the properties:

1. It can generate an endless supply of encoding packets with constant encoding cost per packet;

[1] In our search, we found two very early implementations, far from complete: [12] and [13]. Both have not been updated for the past year.

2. A user can reconstruct the file using any k packets with constant decoding cost per packet, meaning the decoding is linear in k;
3. The space needed to store any data during encoding and decoding is linear in k.

Probably, one of the most interesting properties of FEC codes is the ability to use the same FEC packets/symbols to simultaneously repair different independent packet losses at multiple receivers. *Independent* packet losses must be emphasized, as recovery should be completely independent of loss patterns (e.g., a burst loss). The book *Raptor Codes* [10], written by two of the same authors of RaptorQ described in IETF's RFC 6330 [2], includes the following text:

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

We believe that we are able to break that assumption, since it was considered for *benign* environments. As we will verify in Section 2.1, RaptorQ [2] offers very high reliability under accidental faults scenarios. Our objective is to study the effect that an adversary carrying out attacks in the network (e.g., corrupting or dropping specific packets) can have on RaptorQ's standard resilience. In particular, we would like to determine if it is possible to hinder the decoding process, thus preventing recovering the original message, and the cost of executing such attack.

Since there is no public domain implementation of RaptorQ, we had to program it ourselves. The current implementation still needs to be optimized, but it is fully compliant with RFC 6330 [2]. Our evaluation results and attacks will rely on our implementation.

This document is structured as follows: next section provides some context on the evolution of fountain codes; Section 2 briefly explains the design of the RaptorQ code and its properties, and presents the results for the probability of decoding failure we got with our implementation; Section 3 presents the rationale behind our attack, and how it was implemented; finally, Section 4 offers the conclusions and considers some future work.

1.1 Context

Reed-Solomon (RS) codes [3] are the first example of fountain-like codes because a message of k symbols can be recovered from any k encoding symbols. However, RS codes require quadratic decoding time and are limited to small block length n. *Low-density parity-check* (LDPC) codes [4] come closer to the fountain code ideal. However, early LDPC codes are restricted to fixed-degree regular graphs, causing significantly more than k encoding symbols to be needed to successfully

decode the transmitted signal. Finally, *Tornado* codes [5], although they do approach Shannon capacity [6] with linear decoding complexity, they are block codes and hence not rateless.

LT codes [7] are considered to be the first practical *rateless* codes for the BEC. The encoding and decoding algorithms of LT codes are simple; they are similar to parity-check processes. LT codes are known to be efficient: k information symbols can be recovered from any $k + O(\sqrt{k}\ln^2(k/\delta))$ encoding symbols with probability $1 - \delta$ using $O(k\ln(k/\delta))$ operations. However, their bit error rates cannot be decreased below some lower bound, meaning they suffer from an error floor.

Raptor codes [8] (*rapid Tornado*) were developed as a way to reduce 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 (optimal) per-symbol encoding/decoding cost with an overhead close to zero. At the time of this writing, this has been shown to be the closest code to the ideal universal digital fountain. A great source on Raptor codes is the book *Raptor Codes*[10] by A. Shokrollahi and M. Luby, as there is very little other literature about the inner works and construction of such codes. In Section 2.1 we compare our results to the ones found in [10].

The basic idea behind Raptor codes is to use a (normally high-rate) code to pre-code the source symbols, generating *intermediate symbols*. These are then encoded using a LT encoder to produce *encoding symbols*. Figure 1 illustrates how Raptor codes can be built, as a concatenation of several codes; for a more in-depth view on this, we refer the reader to [11].

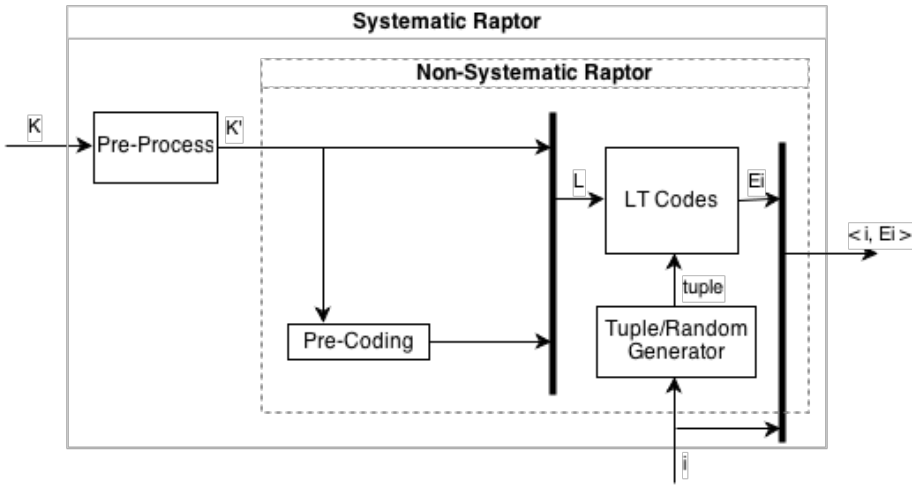

Fig. 1. Conceptual diagram of building Raptor codes as a concatenation of other codes.

Furthermore, we refer the reader to Figure 3, for a more visual reference to the terminology used, namely for blocks and symbols. *Symbols* are the fundamental data unit of the encoding and decoding process, and even though the number of symbols in a block may vary, the size (in bytes) of each symbol is always the same. We use the term *source symbols* for the original data symbols; *encoding symbols* for the symbols that result from the encoding process; and *intermediate symbols*, in the case of a Raptor code, for the symbols that come from applying the pre-code.

A code is called *systematic* if the encoding symbols are constituted by the source symbols and the *repair symbols*. In this case, the *repair symbols* are "universal" symbols that can repair (almost) any source symbol that is missing. Systematic Raptor codes are preferable to non-systematic Raptor codes, because in case no failures occur the original information can be retrieved instantly.

If we follow the flow of data in Figure 1 we can see that, if we are using a systematic Raptor code, the K original source symbols pass through some pre-processing to generate K' new symbols (this pre-processing is necessary as we want the original source symbols to be present in the final encoding symbols). The next steps correspond to a regular (*non-systematic*) Raptor code: (1) pre-coding generates L intermediate symbols, which are then (2) encoded using a LT encoder. Note that the pre-coding itself may be a concatenation/merge of different codes.

The R10 code (Raptor 10), standardised in IETF's RFC 5053 [9], was the first Raptor code accepted into a number of standards. It is a systematic Raptor code. The R10 code overhead-failure curve is designed so that it drops off quickly to achieve a failure probability of around 10^{-6} with an overhead of a few symbols, for the entire range of supported number of source symbols per source block.

The *RaptorQ* code [2] is the most advanced Raptor code. It is built upon the R10 code, improving it in many ways. RaptorQ supports larger source blocks (up to 56,403 source symbols) and up to 16,777,216 encoded symbols. It also has a much steeper overhead-failure curve, which results in positive practical consequences. In general, the RaptorQ code provides superior reliability and better coding efficiency when compared with the R10 code. To achieve this performance, the RaptorQ code combines two major new ideas: the first is the use of symbols over larger alphabets, and the second is the use of a technique called "permanent inactivation".

2 RaptorQ

This section gives a top-level explanation on the design of the RaptorQ code, standardised in [2]. Figure 2 provides an overview of the encoding process; it is interesting to note that the process for encoding and decoding is similar, at least when seen from a top-level point of view.

When using the RaptorQ code, the data to be encoded must be partitioned into *source blocks*. The partitioning algorithm is "optional", in the sense that it may be a linear one: break the total data into sequential *source blocks* of

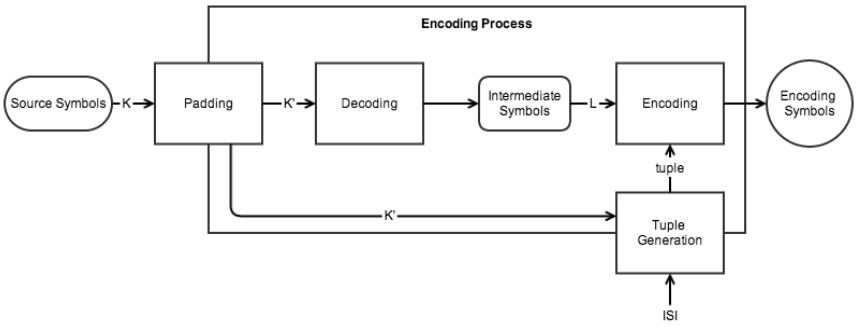

Fig. 2. Overview of the encoding/decoding process for a source block.

size K. It may also be implementation dependent, but [2] standardises one, so our implementation follows it. This algorithm is very important when using larger sets of data, as it introduces entropy and may also affect performance. The algorithm used in the standard also ensures perfect interleaving of the data across all sub-blocks of a source block, so that the amount of data received for each sub-block of a source block is guaranteed to be the same amount as for all other sub-blocks of the same source block - independent of packet loss patterns.

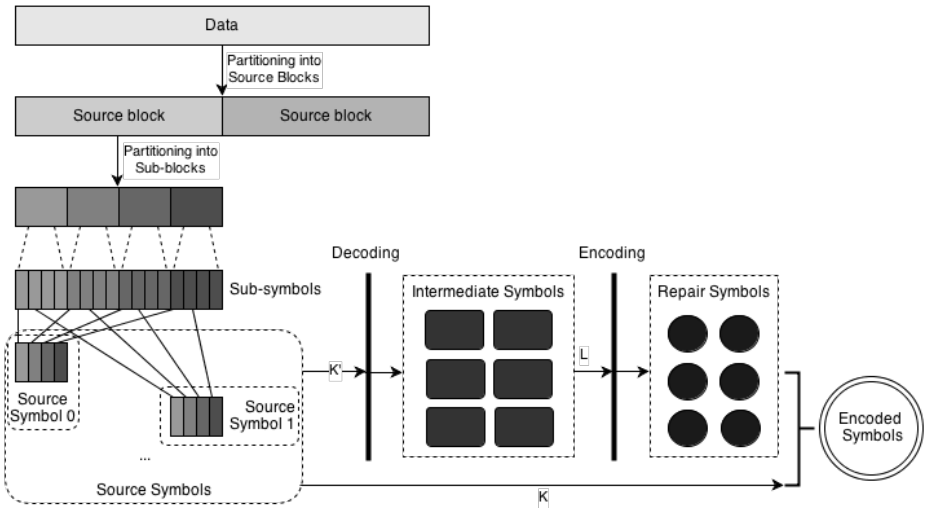

Fig. 3. Block division and symbol generation.

For a given source block of K source symbols, the source block is *augmented* with $K' - K$ padding symbols (depicted in Figure 2). The value of K' is defined

in a precomputed table for each K. This minimizes the amount of table information that needs to be stored at each endpoint, and actually contributes to faster encoding and decoding. The *padding symbols* are treated as regular *source symbols* by the encoder and the decoder.

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

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

Encoding process. The first step of encoding a *source block* is to augment it, creating the *extended source block* as aforementioned. The second is to generate $L > K'$ intermediate symbols from the K' source symbols (also depicted in Figure 2). This corresponds to the *precoding* characteristic of *Raptor* codes, introduced in Section 1.1.

Through *pre-coding relationships*, that must hold within the L intermediate symbols, we are able to calculate them (this depends on the *pre-code* chosen to be used, [2] standardises a pre-coding algorithm). That is, we are able create a set of equations that *multiplied* by the intermediate symbols will result in 0. Each of these equations represent a constraint. Adding to that a set of equations based on the value of K', that, multiplied by the intermediate symbols, should be equal to the *source symbols*, we have a system of linear equations to solve. We use the "Encoder" seen in Figure 2 to generate this last set of constraints. The solution to that system is the set of *intermediate symbols*. For a better illustration of how this system of linear equations is constructed, we point the reader to Figure 4.

Once the intermediate symbols have been generated, repair symbols can be produced. We can generate more pre-coding constraints[2][3] through the ISI of the repair symbol, using the "Tuple Generator" and the "Encoder" seen in Figure

[2] Note that these relationships are nothing but the set of indexes of intermediate symbols, that must be summed to generate the repair symbols, or, in the decoding process, to recover a missing source symbol.

[3] It is interesting to note that the whole encoding and decoding processes are defined by two types of relationships: *constraint relationships* among the intermediate symbols; and *LT-PI relationships* between the intermediate symbols and the encoding symbols.

2, that, when multiplied by the *intermediate symbols* will result in new *repair symbols*. Note that we can generate the source symbols the same way, however, because RaptorQ is a systematic code, the source symbols will be the same after such process. It is also worth mentioning that an efficient algorithm for solving this system of equations is described in [2], as mentioned in Section 1.1, it is called *permanent inactivation*.

Let us come full circle: in Figure 2, we can see that the extended source block first passes through the decoding process (*permanent inactivation*), this is solving the system of linear equations in order to calculate the resulting intermediate symbols. The system of linear equations is illustrated in Figure 4. The constraints needed to put it together, come from the the "Encoder" and "Tuple Generator" seen in Figure 2. When the intermediate symbols have been calculated, we can use them to create new repair symbols, again, using the "Encoder" and "Tuple Generator" seen in Figure 2. Finally, the set of the original source symbols (RaptorQ is a systematic Raptor code) together with the repair symbols, result in the *encoding symbols*.

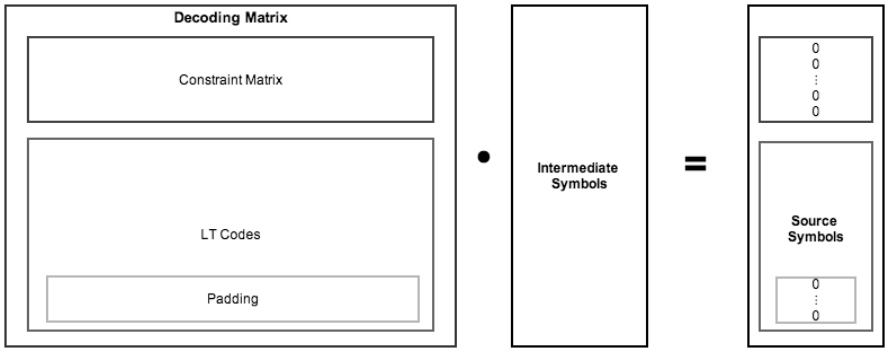

Fig. 4. The decoding process of RaptorQ, illustrated in a system of linear equations.

Decoding process. As mentioned before, the decoding is actually the same process as encoding. The decoder is assumed to know the structure of the source block it is to decode (e.g., K, T, K'); the decoder can then create the extended source block. To decode the extended source block, let us assume that the receiver receives $N \geq K'$ encoding symbols, for that source block. If all source symbols were received (which can be easily seen through their ESI), decoding of that source block is complete; otherwise, the construction of a system of linear equations, similar to the previous one takes place. We say similar and not equal, due to a couple of minor diferences: (1) any missing equation for a source symbol is replaced with an equation of a repair symbol; (2) if additional repair symbols

were received, they will also take part of the system of equations, ensuring a much greater probability of successful decoding.

Upon successfully solving the system of linear equations, the result is once again the set of intermediate symbols. Which can be easily used to recover the missing original source symbols for that block, namely by: (1) generating the correspondent pre-coding relation (through their ISI, using the "Encoder" and "Tuple Generator"), and (2) multiplying that relation by the intermediate symbols, resulting in the missing *source symbol*; all source symbols can be recovered through this process.

2.1 Evaluation

A major advantage of RaptorQ is its steep overhead-failure curve, which makes the standard more robust and appropriate to be used as a FEC code for mission critical communication and data storage. It achieves this by using *non-binary alphabets*, that is, finite fields, namely $\mathbb{F}_2{}^8$. Using a code over $\mathbb{F}_2{}^8$ as part of the precoding, although it may look like a random code, it is computationally efficient and contributes largely to a better overhead-failure curve. The rationale behind this is discussed in greater detail in Section 3.3.1 of [10]. RaptorQ is predictable in terms of its failure probability as a function of overhead, and the dependency of the failure probability on the loss rate is minimal, as there is a graceful degradation of the probability when the rate grows.

We present some results for the failure probability of our implementation of the RaptorQ standard and compare it to the evaluation found in Appendix B.3 of [10]. The methodology was the following: for the values of K equal to 10, 26 and 101, we encoded random input data, and then forced the random loss of 10%, 20%, 50%, 60% and 85% of the encoded symbols. Then, decoding was attempted with the arriving encoded symbols. Furthermore, we did experiments with different overheads[4]. An overhead of 0 means that decoding is attempted after K encoded symbols are received (for an overhead of 1 and 2, this would mean $K+1$ and $K+2$ encoded symbols, respectively). Each test was repeated between 10 million and 20 million times, as was done in [10], to get a reasonable level of confidence in the results.

	Loss				
K	10%	20%	50%	60%	85%
10	0	0	0	$4{,}78 \cdot 10^{-3}$	0
26	$5{,}38 \cdot 10^{-3}$	$4{,}02 \cdot 10^{-3}$	$3{,}93 \cdot 10^{-3}$	$4{,}09 \cdot 10^{-3}$	$1{,}27 \cdot 10^{-2}$
101	$5{,}67 \cdot 10^{-3}$	$4{,}77 \cdot 10^{-3}$	$4{,}89 \cdot 10^{-3}$	$4{,}91 \cdot 10^{-3}$	$4{,}74 \cdot 10^{-3}$

Table 1. Failure probability with 0 overhead.

[4] By *overhead* we mean extra *repair symbols*, besides the ones necessary to fill in the gap of the missing *source symbols*.

	Loss				
K	10%	20%	50%	60%	85%
10	0	0	0	0	0
26	0	$2{,}18 \cdot 10^{-05}$	$1{,}59 \cdot 10^{-05}$	$1{,}56 \cdot 10^{-05}$	$1{,}50 \cdot 10^{-5}$
101	$3{,}82 \cdot 10^{-05}$	$2{,}48 \cdot 10^{-05}$	$2{,}47 \cdot 10^{-05}$	$2{,}22 \cdot 10^{-05}$	$2{,}53 \cdot 10^{-05}$

Table 2. Failure probability with 1 overhead.

	Loss				
K	10%	20%	50%	60%	85%
10	0	0	0	0	0
26	0	0	$4{,}17 \cdot 10^{-7}$	0	0
101	$2{,}09 \cdot 10^{-07}$	0	$2{,}10 \cdot 10^{-07}$	$3{,}13 \cdot 10^{-07}$	$1{,}06 \cdot 10^{-07}$

Table 3. Failure probability with 2 overhead.

The results are displayed in Tables 1, 2 and 3. They confirm the reliability claimed by the RaptorQ standard, as the failure probability is extremely small in all experiments. Furthermore, in some tests, we never observed decoding failure. For $K = 10$, we only saw failed decodings for 60% loss with 0 overhead. We are not sure what causes this case to fail while others succeed, but it is associated with the periodic nature of the RaptorQ standard[5].

These results fall in line with [10], and to extend further this evaluation would be going out of the scope of the article.

3 Attack

This section intends to find out the effect a malicious adversary with network access can have on the RaptorQ's standard reliability. We consider an attack on the RaptorQ standard to be successful, if it can prevent the complete recovery of a source block. We assume that the attacker knows the value of K', as the decoder should also know it. For simplicity, the attacker is assumed to be capable of dropping specific packets while in transit between the sender to the receiver (alternatively, the adversary could corrupt the packet, which would force its deletion when integrity checks are made at the receiver).

In order to thwart the decoding of a source block, it suffices to avoid a single source symbol of that block from being recovered. However, as long as the decoder can calculate the *intermediate symbols*, it can recover any symbol that is needed. Therefore, the attack must keep the decoder from being able to recover the intermediate symbols. To inhibit the calculation these symbols, one must look back at the system of linear equations depicted in Figure 4. The objective is to stop that system of linear equations from being solved. However, since the attacker only has network control, he has no control over the construction of the system of linear equations.

[5] In future work, this can be exploited to further improve our attack.

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

It is very interesting to take notice that the attack is completely independent of the encoding symbols' data. The pre-coding constraint corresponding to a repair symbol is generated based only in K' and the symbol's ISI. K' is, by our assumption, known to the attacker, and the ISI is, by standard, sequential. Therefore, the attack is based fundamentally on how the standard identifies and transmits the packets, allowing the exploitation of communications using encrypted packets, such as when packets are protected with IPsec[1].

Another noteworthy point is that the attacker may have pre-computed tables, holding, for each K', which encoding symbols should be dropped. This would allow a reduction in the computational requirements for the attack to a bare minimum.

3.1 Evaluation and Discussion

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

Unfortunately, the calculation of the exact number would be dependent on the implementation that is employed - i.e., different implementations may use distinct *transport* protocols, with various packet sizes, resulting in more or less encoding symbols per packet. Hence, our approach will not be evaluating how many network packets must be dropped in order for an attack to be successful, but to measure how many encoding symbols[6] must be removed. Note that RFC 6330 [2] does recommend sending one symbol per packet.

We contemplate a scenario where the sender application is streaming information to the receiver. If the receiver fails to recover the information, the sender application will send more repair symbols, and another trial of decoding is performed. We tested 28 different values for K'. For each test, we deleted the last source symbol, and replaced it with repair symbols, until we could decrease the *decoding matrix's rank*. We executed the test with the receiver application trying 3 times to decode, always with 0 overhead.

As we can see in Table 4, the number of *encoding symbols* that were lost for each K' vary a lot (as expected), from hundreds to just 2. We can also

[6] These includes the original source symbols that also need to be dropped.

# Tries	10	26	32	42	55	62	75	84	91	101	153	200	248	301
1	43	115	266	2	127	117	430	390	212	63	179	70	42	66
2	174	1173	484	195	154	168	481	399	237	1105	433	313	93	244
3	224	1250	734	456	161	315	584	936	294	1321	528	375	312	576

# Tries	355	405	453	511	549	600	648	703	747	802	845	903	950	1002
1	119	187	207	488	10	36	192	213	339	10	189	302	663	1185
2	235	406	237	681	128	98	606	485	513	794	297	449	695	1788
3	244	557	537	705	345	331	639	898	1128	829	370	580	886	1804

Table 4. Number of encoding symbols that must be lost.

observe that the number of packets that need to be lost, between tries, also varies considerably: for example, for $K = 55$, after the second try, we only had to kill an extra 7 symbols to successfuly attack again; but for $K = 101$ we see a distance of 1042 symbols that needed to be dropped between the first and second tries.

Sometimes the number of encoding symbols that must be dropped is very high, meaning that such attack would be more conspicuous. But still, this is a *proof of concept* that the RaptorQ standard can be broken when facing malicious faults. We are currently working in a way to refine this attack, and make it more efficient, and hopefully, harder to detect.

4 Conclusion

The usage of fountain codes for forward error correction is increasing, with more standards adopting it. The RaptorQ code is, to the public's eyes, the vanguard of fountain codes, thus, it plays an important role in this field.

We have implemented the first public domain implementation of the RaptorQ standard, and attested its fidelity when facing accidental network faults. However, we have also described an attack that an adversary with network control could do, to guarantee decoding failure at the receiver, thus preventing data reconstruction. Hence, our motivation was not unfounded, and there is work to be done, if we want to deploy these codes in mission critical network/systems, where there is a (possibility of) malicious environment and faults.

We have also evaluated the number of encoding symbols that must be dropped in order for our attack to be successful, for some values of K'. However, most of the time those values are too high, and indicate that our attack may be easily detected, therefore, impractical if the attacker wants to remain hidden. Nonetheless, we still have achieved the goals we set, and we have a proof of concept that RaptorQ standard can be broken in a malicious environment; but there is more work to be done.

We intend to continue our research in this area. Our future work starts by maintaining and optimising our implementation. As far as RaptorQ's robustness when deployed in a malicious environment goes, we also intend to investigate further, and evolve the attack we have hereby described. There are many ways

we may be able to refine our attack, and try to reduce the necessity of dropping so many encoding symbols. For example, we have been trying to decrease the *decoding matrix's rank* by replacing rows - i.e., *row rank*, however, column rank and the row rank are always equal, so, in some cases it might be faster to decrease the rank by replacing columns. How an attacker with only network control could do that, remains to be seen.

Other interesting future work would be to try to, depending on the loss patterns of the network we are working with, mask our malicious faults, to make it look like regular packet loss. That is, instead of completely dropping the packets, we could try to damage it to fit the erasure patterns of the network - i.e., making it look like accidental faults.

Acknowledgments. This work was partially supported by the EC through project FP7-257475 (MASSIF), by the FCT through the Multiannual program (LaSIGE) and project PTDC/EIA-EIA/113729/2009 (SITAN).

References

1. R. Oppliger, *Security at the Internet layer*, IEEE Computer, 31(9):43–47, September 1998.
2. M. Luby, A. Shokrollahi, M. Watson, T. Stockhammer and L. Minder, *RaptorQ Forward Error Correction Scheme for Object Delivery (RFC 6330)*, IETF Request For Comments, August 2011.
3. B. Cipra, *The Ubiquitous Reed-Solomon Codes*, SIAM News, Volume 26-1, January 1993.
4. R. Gallager, *Low Density Parity Check Codes*, Monograph, M.I.T. Press, 1963.
5. M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman and V. Stemann, *Practical Loss-Resilient Codes*, Proceedings of the twenty-ninth annual ACM symposium on Theory of computing, pages: 150–159, 1997.
6. C. Shannon, *Communication in the presence of noise*, Proc. Institute of Radio Engineers vol. 37 (1): 10–21, January 1949.
7. M. Luby, *LT Codes*, The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002.
8. A. Shokrollahi, *Raptor codes*, IEEE Transactions on Information Theory: 2251-2567, 2006.
9. M. Luby, A. Shokrollahi, M. Watson and T. Stockhammer, *Raptor Forward Error Correction Scheme for Object Delivery (RFC 5053)*, IETF Request For Comments, October 2007.
10. A. Shokrollahi and M. Luby, *Raptor Codes*, Now, 2011.
11. M. Luby, M. Watson, T. Gasiba, T. Stockhammer and W. Xu *Raptor Codes for Reliable Download Delivery in Wireless Broadcast Systems*, IEEE CCNC, Las Vegas, 2006.
12. http://code.google.com/p/libcatid/source/browse/trunk/src/codec/RaptorQ.cpp?r=1033
13. https://github.com/Meyermagic/RaptorQ-Python