Adding Integrity Verification Capabilities to the LDPC-Staircase Erasure Correction Codes

Mathieu Cunche, Vincent Roca

Adding Integrity Verification Capabilities to the LDPC-Staircase Erasure Correction Codes

Mathieu CUNCHE Vincent ROCA
INRIA Rhône-Alpes, Planète research team, France, {firstname.name}@inria.fr

***Abstract*— File distribution is becoming a key technology, in particular in large scale content broadcasting systems like DVB-H/SH. They largely rely on Application Level FEC codes (AL-FEC) in order to recover from transmission erasures. We believe that sooner or later, content integrity and source authentication security services will be required in these systems. In order to save the client terminal resources, which can be a handheld autonomous device, we have designed a hybrid system that merges the AL-FEC decoding and content integrity/source authentication services. More precisely our system can detect a random object corruption triggered by a deliberate attack with a probability close to 100% almost for free in terms of computation overhead.**

I. Introduction

a) File distribution and AL-FEC: File/object distribution is becoming a key technology, in particular in large scale content broadcasting systems like DVB-H/SH. They largely rely on Application Level Forward Erasure Correction codes (AL-FEC), not only to recover from transmission erasures but also to improve the content broadcasting scheme itself (e.g. the FLUTE/ALC protocol stack of DVB-H/SH). More specifically, AL-FEC codes work over a *packet erasure channel*, where packets either arrive without any error or are lost. Packet erasures can result from transmission errors (that exceed the error correcting capabilities of the physical layer codes), or congestion problems within an IP router, or simply because the receiver is a mobile device that is currently disconnected. If the patented Raptor AL-FEC codes[7] are well suited to broadcasting systems (they are part of the 3GPP and DVB standards), LDPC codes form an interesting alternative.

b) The LDPC-Staircase AL-FEC Codes: The LDPC-staircase codes [8] (also called double-diagonal or repeat accumulate codes) are particularly interesting for file delivery systems. Thanks to their parity check matrix structure these codes feature a high encoding/decoding speed, which means they can easily encode in a single pass objects that are composed of a huge number of source symbols (typically several 10,000s). This is a great advantage when compared to small block codes like Reed-Solomon codes[11]. Besides these codes have been standardized at IETF as RFC 5170 [12] and a high performance on-the-shelf GNU/LGPL codec is available [10]. For all these reasons they are used in this work. Note that in the following we consider that symbols (AL-FEC coding point of view) are equivalent to packets (network point of view) since (usually) a symbol is carried in a single packet. Symbols can therefore be several hundreds of bytes long.

Several decoding techniques are possible. Iterative decoding is a usual, high performance technique [12]: given a set of linear equations, if one of them has only one remaining unknown variable, then the value of this variable is that of the constant term. So, this variable is replaced by its value in all the remaining equations and we reiterate. The value of several variables can therefore be found recursively. Applied to LDPC AL-FEC codes, the parity check matrix defines a set of linear equations whose variables are the source symbols and parity symbols. Receiving or decoding a symbol is equivalent to finding the value of a variable. When the decoding succeeds with this algorithm, all the source and parity symbols are received or rebuilt. This paper does not detail LDPC-staircase and interested readers can refer to [11][12].

c) Adding Content Integrity/Source Authentication Services: Within closed networks (e.g. DVB-H infrastructure), launching a DoS attack or injecting spurious traffic requires an expensive equipment, which limits the risks. But the situation is opposite in case of open networks, like the Internet or Wifi hotspots. Here the content integrity and source authentication services are often required to enable a receiver to check that what he received is actually the content that has been sent by the authorized sender.

These integrity/source verifications can be made either on a per-packet or per-object basis. This work focuses on the latter case. The traditional solution consists in signing a hash of the object with an asymmetric cryptographic function. In that case, with big objects, the computation time of the signature is low compared to the hash calculation over the object, especially with modern strengthened hash functions. This is the *reference solution* against which we will compare our scheme.

d) Goals of this Work: Our work explores an alternative solution that consists in adding object verification capabilities to an existing FEC scheme *while minimizing the computation and transmission overheads*. The resulting system, called *VeriFEC*, must be able to:

- detect the vast majority of corrupted objects with a reduced cost (i.e. enable a lightweight pre-check),
- detect all the corrupted objects with a cost close to standard integrity check (i.e. during full check),
- and keep exactly the same erasure recovery capabilities as the original AL-FEC scheme.

The corruption can be either intentional (i.e. mounted by an attacker) or not (e.g. caused by transmission errors that have not been detected/corrected by the physical layer FEC codes/CRC). In this work we first consider the case of *random corruptions*, and then we consider *intelligent attacks*.

II. Problem Analysis and Observations

This section introduces the attack model and the corruption propagation phenomenon that is the core of our proposal. Then it discusses the potential use of this phenomenon, both from the attacker and the receiver points of view.

A. The Attack Model

Let us consider an unsecured transmission channel. We first assume the attacker can corrupt an unlimited number of symbols randomly chosen (which includes the cases of errors not detected by the lower layers and attackers with limited capabilities). In a second step (section V) we will consider the case of intelligent attacks mounted by powerful attackers.

A first goal for the attacker can be to *corrupt the object without the receiver's noticing*. This corruption is anyway detected by the use of cryptographic hash over the whole object, and the detection probability is only limited by the robustness of the hash function itself against malicious attacks. Another goal for the attacker can be to mount a *Denial of Service (DoS) attack*, either by sending a large number of fake objects that will be received in addition to the legitimate objects or simply by corrupting as many objects as possible. This attack is trivial to launch. The challenge for the receiver is to *quickly identify corrupted objects and get rid of them with the lowest possible computational overhead.*

This work essentially focuses on the second type of attack, where the attacker tries to consume the receiver resources.

B. The Corruption Propagation Phenomenon

1) The Phenomenon: In order to recover from erasures, the iterative decoder rebuilds the missing symbols thanks to the received ones. Let us consider the following equation (one of the constraint equations defined by the LDPC parity check matrix): $S_0 \oplus S_1 \oplus S_2 \oplus S_3 = 0$.

We assume that the values s_1, s_2, s_3 of these symbols have been received, but not s_0. Then: $s_0 = s_1 \oplus s_2 \oplus s_3$. If S_3 has been corrupted in $s'_3 = s_3 \oplus \varepsilon$, whereas the other symbols have been correctly received. Then S_0 is decoded as:

$$s_1 \oplus s_2 \oplus s'_3 = s_1 \oplus s_2 \oplus s_3 \oplus \varepsilon = s_0 \oplus \varepsilon = s'_0$$

S_0 has inherited the corruption of S_3. Therefore, if a corrupted symbol is used during decoding, the decoded symbol inherits from the corruption. Furthermore, each newly decoded symbol can be used to decode other symbols recursively, and a corruption avalanche can take place. We call this the *corruption propagation phenomenon*.

2) Codeword Interpretation: This phenomenon can also be seen from a "codeword" point of view. Let us remind that, in the context of AL-FEC codes, a codeword is the vector formed by all the bits of a certain position in the set of source and parity symbols. The output of FEC decoding is always a codeword and this decoded codeword, w, satisfies the condition $Hw = 0$, where H is the LDPC parity check matrix. In our case, symbols are several hundreds of bits long, say s, which means that the set of source and parity symbols form s codewords, each of them satisfying the above relation. Said differently, LDPC decoding at the symbol level consists of s parallel LDPC decoding at the bit level (codeword), all of them sharing the same erasure pattern.

Let assume that the transmitted codeword w has been corrupted. The output of the decoder is necessarily another codeword, w' ($w \neq w'$). Since we are dealing with linear codes, the difference of two codewords is also a codeword, and in particular $e = w - w'$. Therefore a successful corruption can be seen as the addition of a codeword e (called the *corrupting codeword*) to the transmitted codeword w.

3) Experimental Approach: In order to quantify this phenomenon, we carried out experiments using the on-the-shelf LDPC-staircase C++ reference codec [10]. We chose an object composed of $20,000$ symbols[1], and used a coding rate $R = k/n = 2/3$ (i.e. $n - k = 10,000$ parity symbols are added). Symbols are transmitted in a random order in order (1) to carry out experiments without considering the channel loss model, and (2) to be sure that decoding operations will take place[2]. The attacker randomly chooses some symbols and corrupts them. We then count the number of corrupted *source* symbols after decoding[3]. The test is repeated 2,000 times for each value and we plot the min/average/max/90% confidence intervals.

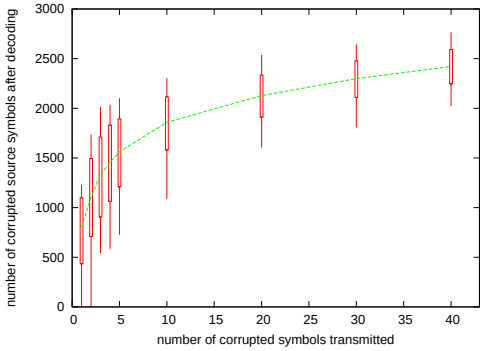

Fig. 1. Number of corrupted source symbols after decoding (average/min/max/90 % confidence interval) W.R.T. the number of corrupted symbols received.

Figure 1 shows that even a single corruption triggers on average more than 700 corrupted symbols after decoding (i.e., 3.5% of the object). However some experiments exhibit few symbol corruptions, which means that the symbols corrupted by the attacker have been used to rebuild only a small number of symbols. In some tests there is no corrupted symbol at all after decoding, which means that the symbols corrupted by the attacker were symbols that have not been used during decoding (e.g., because this symbol has already been rebuilt).

C. First Conclusions

1) For the Attacker: For the attacker, a massive corruption of the object can be achieved with only a limited attack over

[1] The size of each symbol has no impact and is not specified.

[2] Note that if the transmission order is not sufficiently random, a receiver can easily randomize the order in which the received symbols are given to the decoder, without requiring additional buffering since the target use-case (i.e. FLUTE/ALC) requires large buffering capabilities anyway.

[3] Note that we do not take into account the number of corrupted *parity* symbols after decoding since the ultimate goal of the attacker is to corrupt the object, not the temporary parity symbols.

the transmitted packets (a single symbol is often sufficient), which can be the attacker target. However, the attacker will usually have difficulties to create a limited targeted corruption. This is not totally impossible though, but it remains exceptional. This latter aspect will be detailed in section V.

2) For the Receiver: The receiver can regard the important corruption propagation phenomenon as either as a problem (e.g., a corrupted video will exhibit many glitches) or as an advantage (detecting a corruption is easier). In our case, VeriFEC heavily relies on this phenomenon.

III. Our Solution: VeriFEC

A. Principles

The idea is to take advantage of the propagation phenomenon by using an integrity verification of the decoded object in two steps. First a low cost preliminary check detects the vast majority of corrupted objects, and if the preliminary check does not detect anything, a complementary check is used to obtain a 100% detection probability[4].

The preliminary check consists in verifying only a subset of the source symbols after AL-FEC decoding (figure 2). Thanks to the corruption propagation phenomenon, we know that most random attacks (even on a single symbol, the worst case) will trigger many corruptions in the decoded object. Since we only check a subset of the object, the preliminary verification cannot reach a 100% detection probability, but we will show in section IV that in practice the vast majority of attacks are detected. The second check consists in verifying the remaining source symbols. Therefore, an object that successfully passed the two checks is certified 100% sure.

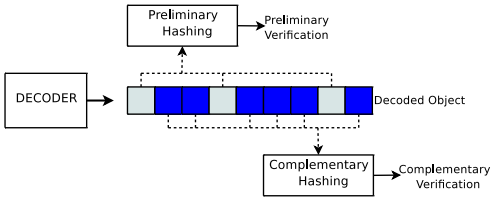

Fig. 2. VeriFEC preliminary versus complementary integrity verification.

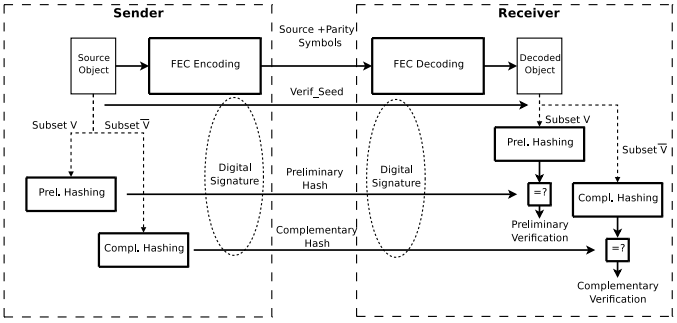

Fig. 3. VeriFEC global view.

In this paper we only consider a single receiver. However VeriFEC does not include any mechanism that would limit its field of application, and since there is no information sent by the receiver(s) to the sender, VeriFEC is massively scalable.

[4]This probability is in fact only limited by that of the hash function.

B. Details

1) Sender Side: The sender performs FEC encoding and sends the source and parity symbols as usual. In parallel he randomly selects a subset V of N_{verif} source symbols, by using a pseudo-random number generator and a seed, *prel_verif_seed*. Let $\overline{V}$ denote the complementary subset, equal to the source symbols that are not in V. Then he computes the hash over V, called *prel_hash*, and the hash over $\overline{V}$, called *compl_hash*. The {*prel_verif_seed; prel_hash; compl_hash*} triple is then sent to the receiver. Since the security of this triple is crucial, the sender digitally signs it [9] so that the receiver can check its integrity and authenticate the sender (we assume the receiver knows the sender's public key, e.g. thanks to a PKI). The signed triple can be sent in-band (using the unsecured channel) or out-of-band (e.g. in a web page). Sometimes a secure channel exists over which the triple can be transmitted. This is not expected to be the usual solution since it does not scale.

2) Receiver Side: The receiver proceeds to a standard AL-FEC decoding of the object, using the received symbols. In parallel the receiver retrieves the {*prel_verif_seed; prel_hash; compl_hash*} triple and checks the digital signature. Thanks to this check, the sender is also authenticated. The receiver then proceeds to a two step object integrity verification: thanks to the received *prel_verif_seed*, the receiver selects the same subset V of source symbols, computes the hash of this subset and compares it to the received *prel_hash*. If the two hashes differ, the receiver has detected for sure a corruption. Otherwise the receiver cannot conclude yet. Then he compares the hash of the complementary subset $\overline{V}$ to the received *compl_hash*. If the two hashes differ, the receiver knows for sure the object has been corrupted, otherwise he knows for sure the object is not corrupted.

IV. Performance Evaluation with Random Attacks

We have designed a VeriFEC class that derives from the underlying LDPCFecSession class of the LDPC-staircase C++ open source codec version 2.0 [10]. We use OpenSSL version 0.9.8c for the cryptographic primitives. More precisely digital signatures use RSA-1024 and the message digest is one of MD5 (banned from secure systems), RIPEMD-160 [4], SHA-1, and SHA-256.

We carried out experiments meant to appreciate the VeriFEC preliminary verification performances in terms of corruption detection capabilities and processing overhead. The same configuration as that of section II-B.3 is used: the object is composed of $k = 20,000$ symbols (except in section IV-C), and the coding rate is equal to $R = 2/3$ (except in section IV-C). We assume that the attacker does not want to be detected by the preliminary check and therefore corrupts a *single* symbol. We also assume that the attacker chooses the corrupted symbol randomly (intelligent attacks will be addressed in section V).

A. Dependency W.R.T. the Verification Ratio

We first study the number N_{verif} of source symbols that must be verified (i.e. the number of symbols in V) in order

to reach the desired corruption detection probability with the preliminary check. The higher the N_{verif} value, the higher the corruption detection probability (a full detection is achieved when N_{verif} equals k). However we also want to keep the processing overhead of the preliminary check to a minimum, and from this point of view N_{verif} should be as small as possible. In order to find an appropriate value, we carried out experiments where, for each verification ratio value (i.e. N_{verif}/k ratio), we calculate the percentage corruptions detected over $50,000$ tests.

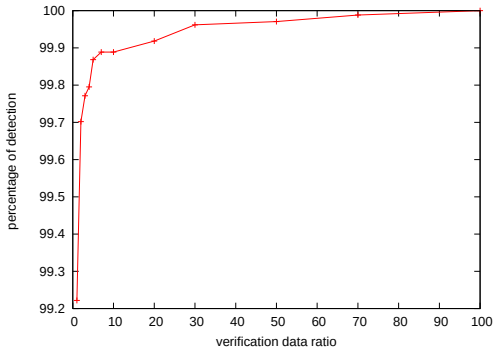

Fig. 4. Corruption detection probability of the preliminary check as a function of the verification ratio, N_{verif}/k.

As expected figure 4 shows that the detection probability increases with N_{verif}. But we also see that checking only 1% of the decoded object already enables to detect 99.22% of the attacks. We believe that *verifying 5% of the symbols to achieve a corruption detection probability of 99.86% is a good balance between detection and computation overhead*. This ratio will be used for the rest of the paper.

B. Computing Overhead Gains

Since the verification ratio is now set to 5%, we can study the computing gains made possible by the preliminary check as well as the global (two step) VeriFEC overhead with respect to the reference solution (i.e. standard FEC codec and signed hash over the entire object). To that purpose, we have measured the various times at a receiver with different hash systems, and we have calculated the average values over 200 runs. The experiments are carried out on a Dual-Core Intel Xeon 5120 processor, 1.86 GHz/4 GB RAM/Linux host. The symbol size is set to 1024 bytes, which means that we are dealing with 20 MB objects, adding 10 MB of parity data.

1) Preliminary Check Only: The first scenario corresponds to the case where the object is corrupted and the preliminary check detects this corruption (this is the most probable case).

Table I compares the cost of the standard solution to the cost of VeriFEC with the preliminary verification only, by showing their processing times and corresponding bitrates. We see that the relative gains are very significant, especially with modern, strengthened message digest algorithms, that incur a significant processing load. With SHA-256, the relative gains for FEC decoding/hash verification made possible by VeriFEC amounts to $59.2\triangle\%$ (even with SHA-1, this gain is significant, $31.7\triangle\%$). If we focus only on the verification process, we observe that VeriFEC reduces the overhead by $94,8\triangle\%$, which is in line with the theoretical 95% improvement (since we only check 5% of the symbols).

	MD5	*RIPEMD-160*	*SHA-1*	*SHA-256*
Receiver: bitrate				
FEC+signed hash (s)	651 Mb/s	473 Mb/s	586 Mb/s	337 Mb/s
VeriFEC (s)	865 Mb/s	861 Mb/s	858 Mb/s	828 Mb/s
Relative gain ($\triangle\%$)	**24.7**$\triangle\%$	**45.1**$\triangle\%$	**31.7**$\triangle\%$	**59.2**$\triangle\%$
Receiver: verification time only				
Signed hash verif. (s)	0.0657 s	0.1646 s	0.0937 s	0.3032 s
VeriFEC verif. (s)	0.0037 s	0.0086 s	0.0051 s	0.0157 s
Relative gain ($\triangle\%$)	**94.4**$\triangle\%$	**94.8**$\triangle\%$	**94.6**$\triangle\%$	**94.8**$\triangle\%$

TABLE I
BITRATE AND PROCESSING TIMES OF VERIFEC'S PRELIMINARY CHECK VERSUS THE STANDARD FEC+SIGNED HASH SCHEME.

2) Complete verification: The second scenario is when the preliminary verification has not detected any corruption, meaning that either the object is not corrupted or that the preliminary check failed to spot the corruption. The cost of the standard solution is compared to the cost of VeriFEC when both preliminary and complementary verifications are done.

	Standard (FEC+hash)	*VeriFEC (prel+compl hash)*	*overhead*
SENDER FEC + hash creation time	0.2946 s	0.2975 s	0.98 $\triangle\%$
RECEIVER FEC + verification time	0.3880 s	0.3911 s	0.80 $\triangle\%$
RECEIVER verif. time only	0.1724 s	0.1753 s	1.68 $\triangle\%$

TABLE II
TOTAL PROCESSING TIME OF VERIFEC VERSUS THE STANDARD FEC+SIGNED HASH SCHEME.

We can expect a little computation overhead because the data chunks given to the message digest function during the two verifications are not necessarily contiguous. We measured it, using the RIPEMD-160 hash function. Table II shows that this overhead remains small, $1.68\triangle\%$ (if we only consider the hash verification time).

3) Computing Overhead Gains W.R.T. the Object Corruption Ratio: We now appreciate the benefits of VeriFEC as a function of the object corruption ratio (i.e. the ratio of objects corrupted by an attacker). The computation cost is fixed in case of a standard "FEC plus complete hash" solution. On the opposite, this verification cost varies a lot with VeriFEC. If very few objects are corrupted, the (costly) complementary check is almost always performed. On the opposite, if a large number of objects are corrupted, then most corruptions are identified by the (cheap) preliminary verification, thereby saving processing time.

Let us introduce some notations:

- T_{Verif}: average time spent to verify the object,
- T_{Pre_Verif}: preliminary verification time,
- T_{Compl_Verif}: complementary hash verification time,
- $P_{Object\_Corruption}$: object corruption ratio,
- $P_{Pre\_Verif\_Detection}$: preliminary verification corruption detection probability.

With the VeriFEC system, the average verification time as a function of the object corruption ratio is given by:

$$T_{Verif} = T_{Preliminary\_Verif} + T_{Compl\_Verif} \times (1 - P_{Object\_Corruption} \times P_{Partial\_Hash\_Detection})$$

We use $P_{Pre\_Verif\_Detection} = 0.9986$ (section IV-A). We have experimentally measured the other parameters and, with RIPEMD-160, we found that on average: $T_{Preliminary\_Verif} = 0.0091s$ and $T_{Compl\_Verif} = 0.1662s$.

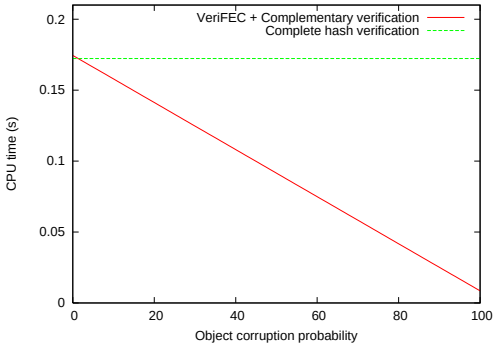

Fig. 5. Verification time as a function of the object corruption ratio.

Figure 5 shows the two curves for each solutions, not including the FEC decoding time (identical in both cases). We see that if there is no corruption, our system adds a little overhead. This overhead becomes null when the corruption ratio is 1.3%. Then, the higher the corruption ratio, the more effective our system is.

C. Dependency W.R.T. the Object Size and FEC Coding Rate

We now analyze the influence of both the object size (in terms of the number source symbols, regardless of the symbol size which has no influence) and the FEC coding rate on the corruption detection probability. Since these two parameters were fixed in the previous experiments, we now want to make sure that the VeriFEC efficiency remains good for different object sizes and coding rates.

Concerning the object size, experiments reported in [3] show that the detection probability quickly increases with the object size. With objects containing 4000 symbols, the corruption detection probability of the preliminary verification already amounts to 98.75%. So the VeriFEC system matches well the operational conditions of the underlying LDPC-staircase codes since these *large block* AL-FEC codes are known to perform well when the number of symbols exceeds a few thousands [11].

Concerning the coding rate, experiments reported in [3] show that the detection probability of the preliminary verification remains fairly stable (between 99.53% to 99.90%), even when the coding rate largely varies, between 0.33 and 0.91. Note that using coding rates below 0.33 is not recommended with LDPC-staircase [11].

We can therefore conclude that the object size and coding rate parameters do not impact the VeriFEC efficiency.

V. On Intelligent Attacks

In this section, we consider the case of an intelligent and powerful attacker. As the main benefit of VeriFEC is the high detection probability of the preliminary verification, we will only consider attacks that significantly reduce it, i.e. that lead to a non detection probability higher than $1,4.10^{-3}$ (section IV-A). Note that in any case, all attacks will finally be detected after the complementary check.

A. Preventing Simple Intelligent Attacks by Extending the V Subset of Verified Symbols

Let us first assume that the LDPC code is known by the attacker (the $\{k, n, seed\}$ triple is transmitted in clear text by default and fully defines the LDPC code [12]). In that case verifying only a subset of the source symbols during the preliminary check is no longer sufficient. Indeed an intelligent attacker can choose a corrupting codeword with only one '1' in the source bits. To find it, the attacker just needs to FEC encode the source bit vector (since he knows the code), and retrieve the associated parity bits (of course, there are many '1' in the parity bits in that case). Then the attacker adds this codeword to the received symbols and forwards the resulting symbols to the receiver[5]. The detection probability is then equal to N_{verif}/k, i.e. the verification ratio.

One counter measure is to *chose the verified subset V over all the source and parity symbols*. However, *the complementary subset $\overline{V}$ remains the same* and only encompasses source symbols. A consequence is that the receiver needs to rebuild the repair symbols of V. In fact, the iterative algorithm already rebuilds a large majority of the parity symbols, if not all, so this overhead can be neglected.

Another counter measure is to hide the LDPC code. This technique will be fully described in section V-B.3.

B. Preventing Low Weight Codeword Attacks

We now describe another attack using so called "Low Weight Codewords" (LWC) and we introduce counter measures.

1) The Need for Low Weight Codewords: For convenience, and without loss of generality, let us focus on one of the s codewords (we assume the attacker has received all the n symbols, and therefore knows the corresponding s codewords). Let $SS()$ be the function that selects the subset of N_{verif} bits, at the positions selected for the V subset, in the codeword. A corruption of w is not detected if the attacker creates a codeword $w' \neq w$ such that $SS(w') = SS(w)$[6]. Knowing the verified subset and finding a codeword having null bits in the verified subset is therefore sufficient to launch a successful attack for the preliminary verification.

A trivial counter measure is to hide the verified subset from the attacker. This can be done by one of the following techniques: sending the *prel_verif_seed* on a secure channel,

[5] Of course the attacker has s possible ways to add the corruption codeword to the original symbols, s being the symbol size in bits. This is not an issue.

[6] We assume that hash function is collision-resistant, i.e. the probability of having two different objects whose hash collide can be neglected.

or sending it encrypted, or sending it at the end of the transmission along with a secure way for the receiver to check that packets have not been excessively delayed while in transit (indeed, if the verified subset is revealed once the symbols have been received, it is too late to perform an attack). Hiding the *prel_verif_seed* is therefore an easy task.

If the *prel_verif_seed* is hidden, it is still possible for the attacker to hope that the verified subset V will not intersect with the non null bits of the corrupting codeword, e. The associated success probability depends on the Hamming weight of e, $H_w(e)$, and the size of the verified subset N_{verif}. The Non Detection Probability (NDP) is therefore:

$$NDP = \begin{cases} \prod_{i=0}^{N_{verif}-1} \frac{(n - H_w(e) - i)}{n - i} & \text{if } H_w(e) \leq n - N_{verif} \\ 0 & \text{if } H_w(e) > n - N_{verif} \end{cases}$$

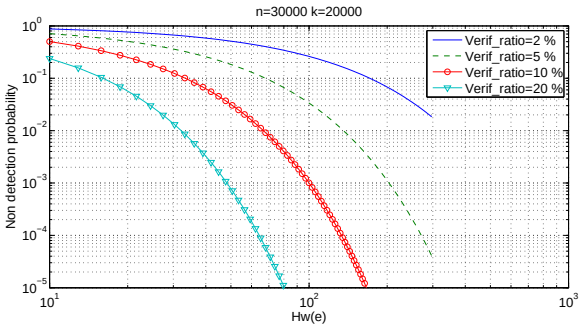

Fig. 6. Non Detection Probability (NDP) of the preliminary check as a function of the Hamming weight of the corrupting codeword for various verification ratios (n=30,000, k=20,000).

Figure 6 shows that the non detection probability falls quickly as the Hamming weight of the corrupting codeword increases. For a verification ratio of 5%, codewords of Hamming weight larger than 200 have a non-detection probability lower than 10^{-3}, i.e. lower than the intrinsic non detection probability of the VerifFEC preliminary check. Thus only the codewords of weight lower that 200 are a threat. This leads us to the problem of finding "Low Weight Codewords" (LWC).

2) Finding Low Weight Codewords: The attacker needs to use LWC, that are known to exist with LDPC codes. The attacker has two possibilities:

- he takes advantage of the s binary codewords extracted from the transmitted packets;
- or he tries to find LWC from the code definition itself, assuming he knows the LDPC code or is capable of inferring this LDPC codes thanks to the received packets.

Let us consider the first possibility. Here the probability that one of these s codewords be a LWC, or that a linear combination of these s codewords be a LWC, must be considered. Let N_w be the number of codewords of weight w in the code $\mathcal{C}$. This number can be approximated by $N_w \simeq \frac{C_n^w}{2^{n-k}}$. Assuming that the s transmitted codewords are linearly independent, they span a space T of dimension s and from this set of codewords, 2^s different codewords can be produced. Let $N_w^{\leq}$ be the number of codeword of weight lower or equal to w. We can give an upper bound on this quantity:

$$N_w^{\leq} \simeq \sum_{i=1}^{w} \frac{C_n^i}{2^{n-k}} \leq \frac{wC_n^w}{2^{n-k}} \leq \frac{w}{2^{n-k}} \frac{n^w}{w!} \leq \frac{wn^w}{2^{n-k}w^w}$$

The probability that a codeword of weight lower than w belong to this ensemble T is:

$$P_{s,w} = \frac{Card(T)}{Card(\mathcal{C})} * N_w^{\leq} = \frac{2^s}{2^k} * N_w^{\leq} \leq \frac{wn^w}{2^{n-s}w^w}$$

This probability goes to zero when n goes to infinity. In our case w (resp. s) is two (resp. one) order of magnitude smaller than n, so the probability that a LWC be transmitted is very small, and we can ignore them.

Let us consider now the second possibility. Finding a LWC of a known LDPC code can be achieved with an exhaustive search, or with less naive algorithms [1]. The complexity of such algorithms can be an obstacle for attackers with bounded computational capabilities. However, in order to obtain an unconditional security, we assume in the remaining of this work that the attacker can find a LWC if he knows the code. This leads us to the problem of hiding the LDPC code.

3) Hiding the LDPC Code: Let us now focus on the problem of hiding the code to the attacker. Changing the code for each transmission is trivial with LDPC-staircase codes, since these codes are are generated on the fly, using a PRNG and a 32-bit seed that can be easily changed at each transmission [12]. As for the *prel_verif_seed* (section V-B.1), the seed used for the generation of the code can be easily hidden.

However an attacker can also use *code recognition techniques* [13] to guess the code. The number of codewords required for the recognition of LDPC codes in a noisy environment has been studied in [2] (in our case we assume that the intercepted codewords do not contain any error). The problem of recognizing an LDPC code is equivalent to finding its parity check matrix. With LDPC-Staircase codes, the parity check matrix is $H = (H_1|H_2)$, where H_1 is a matrix with regular row and column degrees and H_2 is an $(n-k)\times(n-k)$ staircase matrix. Let ε be the set of such matrices. Let N_1 be the column degree and $t = N_1 \frac{R}{1-R}$ be the row degree of H_1, where R is the coding rate (these degrees are the result of the [12] specifications for these codes). H_1 defines a regular bipartite graph with k left nodes of degree $N1$ and $n-k$ right nodes of degree t. From [2](8) we have:

$$log_2(card(\varepsilon)) \sim \frac{N_1(t-1)}{t} log_2(n)$$

The necessary number of intercepted codewords for recovering the code is of the order $log_2(n)$. Let assume that s is such that $s < \frac{N_1(t-1)}{t} log_2(n) - C$, where C is a constant. The number of potential codes (i.e. the choices) is then of the order 2^C and therefore the probability of picking the good code (i.e. launching a successful attack) is 2^{-C}. By choosing $C = 10$, we make this non detection probability equal to $2^{-10} = 0.00098$, i.e. a little bit smaller than the VeriFEC preliminary verification non detection probability (section V).

Let us consider the same experimental conditions as in section II-B.3. So $k = 20,000$ and $R = 2/3$, and it follows that $n = 30,000$ and $t = 6$ ($N_1 = 3$ is the default with LDPC staircase codes using iterative decoding). Therefore $\frac{N_1(t-1)}{t} log_2(n) - C = 27.18$, which means that it is sufficient that $s < 27$ bits.

To conclude we can say that in practice, when $k = 20,000$ and $R = 2/3$, *using symbols that are 3 bytes long, hiding the LDPC code (i.e. the associated seed), and hiding the* prel_verif_seed *prevent an attacker from attacking the preliminary check of VeriFEC.*

VI. Related works

In [6], the authors introduce a scheme that corrects errors and verifies symbols with a "very high" probability when the errors are random. Then they extend the work to address the more complex problem of intelligent attacks by means of code scrambling. If we consider only the first contribution, the solution relies on the use of a specific decoding algorithm for a so-called qSC channel (q-ary Symmetric Channel). This solution completely differs from ours, that keeps the same iterative decoding algorithm, over the same erasure channel, but checks a subset of the source symbols after decoding. The goals are different too, since VeriFEC does not try to correct nor locate corruptions.

In [5] the authors present a system that allows to verify on the fly the symbols before decoding. This verification is done thanks to a homomorphic collision-resistant hash function. An advantage of this solution is that only correct symbols are used by the decoder. So the decoded object is guaranteed not to be corrupted if decoding succeeds. But this solution requires the use of addition over $\mathbb{Z}_q$ which are much more expensive than the Exclusive-OR operations used by LDPC-staircase codes. According to the authors, the system adds around 500% processing time overhead. This totally contradicts our goals of keeping the overhead as low as possible. Additionally, there is also a significant transmission overhead since a hash must be transmitted for each source symbol, whereas VeriFEC only requires the transmission of {*prel_verif_seed; prel_hash;compl_hash*}.

VII. Conclusions

In this work we have shown that corruption detection capabilities and source authentication can be efficiently added to the LDPC-staircase large block AL-FEC codes. The proposed scheme, VeriFEC, checks the integrity of the decoded object in two steps: the first step detects the vast majority of the corruption with a very low computational cost, while the second step finishes the verification to reach a 100% guaranty.

Thanks to comprehensive experiments, we found that *VeriFEC detects 99.86% of the most difficult random attacks (where a single symbol is corrupted) for less than* 6% *of the computation overhead required for a complete signed hash of the object*, without any penalty in terms of erasure recovery capabilities, the primary goal of AL-FEC codes. If the random attack is less subtle (e.g. if several symbols are corrupted) then the detection probability of the preliminary verification significantly increases to reach almost 100%. The case of intelligent attacks aiming to reduce the detection probability of the preliminary check has been addressed. We demonstrated that low weight codewords attacks can be prevented by reducing the symbol size and by hiding a small number of key parameters.

Globally, thanks to its low computation overhead, VeriFEC can be of great help to mitigate random or intelligent denial of service attacks. Additionally, if the threats only include random attacks and if a high integrity probability is sufficient, using the preliminary verification of VeriFEC only is meaningful. However this is a particular case, not the general case.

This scheme can be generalized to other LDPC codes, on condition these codes can be hidden instead of being totally defined by the $\{n; k\}$ tuple. It can also be used with Reed Solomon codes, but as the decoding speed of these codes is low compared to the integrity verification speed, the relative gain will be smaller.

Finally, in future works we will study techniques to hide the code to the potential attackers, for instance by adding known noise to the transmitted symbols. One goal is to relax the current constraint on the symbol size (s, see section V-B.3).

References

[1] A. Canteaut and F. Chabaud. A new algorithm for finding minimum-weight words in a linear code: application to mceliece's cryptosystem and to narrow-sense bch codes of length 511. *Information Theory, IEEE Transactions on*, 44(1):367–378, Jan 1998.

[2] M. Cluzeau and J.-P. Tillich. On the code reverse engineering problem. *Information Theory, 2008. ISIT 2008. IEEE International Symposium on*, pages 634–638, July 2008.

[3] M. Cunche and V. Roca. Adding integrity verification capabilities to the ldpc-staircase erasure correction codes. Research Report 6125, INRIA, February 2007.

[4] H. Dobbertin, A. Bosselaers, and B. Preneel. RIPEMD-160: A strengthened version of RIPEMD. *Fast Software Encryption, LNCS 1039, D. Gollmann, Ed., Springer-Verlag*, 1996.

[5] M. Krohn, M. Freedman, and D. Eres. In-the-fly verification of rateless erasure codes for efficient content distribution. In *IEEE Symposium on Security and Privacy*, May 2004.

[6] M. Luby and M. Mitzenmacher. Verification based decoding for packet based low-density parity check codes. *IEEE Trans. on Information Theory*, 50(1), January 2005.

[7] M. Luby, A. Shokrollahi, M. Watson, and T. Stockhammer. *Raptor Forward Error Correction Scheme for Object Delivery*, October 2007. IETF RMT Working Group, Request for Comments, RFC 5053.

[8] D. MacKay. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, ISBN: 0521642981, 2003.

[9] R. L. Rivest, A. Shamir, and L. M. Adelman. A method for obtaining digital signatures and public-key cryptosystems. Technical Report MIT/LCS/TM-82, 1977.

[10] V. Roca and al. *Planète-BCAST: Tools for large scale content distribution*. URL: http://planete-bcast.inrialpes.fr/.

[11] V. Roca and C. Neumann. Design, evaluation and comparison of four large block fec codecs, ldpc, ldgm, ldgm staircase and ldgm triangle, plus a reed-solomon small block fec codec. Research Report 5225, INRIA, June 2004.

[12] V. Roca, C. Neumann, and D. Furodet. *Low Density Parity Check (LDPC) Forward Error Correction*, June 2008. IETF RMT Working Group, Request for Comments, RFC 5170.

[13] Antoine Valembois. Detection and recognition of a binary linear code. *Discrete Applied Mathematics*, 111(1-2):199–218, 2001.