Falcon Codes: Fast, Authenticated LT Codes

Ari Juels
Cornell Tech
New York City NY, USA
juels@cornell.edu

James Kelley
Brown University
Providence RI, USA
jakelley@cs.brown.edu

Roberto Tamassia
Brown University
Providence RI, USA
rt@cs.brown.edu

Nikos Triandopoulos
RSA Laboratories
Cambridge MA, USA
nikolaos.triandopoulos@rsa.com

***Abstract*—In this paper, we introduce *Falcon codes*, a class of *authenticated error correcting codes* that are based on *LT codes* [23] and achieve the following properties, for the first time simultaneously: (1) with high probability, they can correct *adversarial* symbol corruptions in the encoding of a message, and (2) they allow for very efficient encoding and decoding times, even *linear* in the message length. We study Falcon codes in a new adversarial model for *rateless* codes over computational channels, and define a new security notion for corruption-tolerant encoding in this model. We then present three such LT-based coding schemes that achieve resilience to adversarial corruptions via judicious use of simple cryptographic tools while maintaining very fast encoding/decoding times. One variant Falcon code works well with small messages (100s of KB to 10s of MB) but two alternative scalable versions are designed to handle much larger inputs (e.g., messages that are several GBs in size). Our schemes are provably secure against computational adversaries in the standard model. We analyze our new schemes and show that Falcon codes are both asymptotically and practically efficient.**

I. Introduction

Error correcting codes find numerous applications in computing by increasing reliability in data storage and data transmission over unreliable channels. As such they comprise today a particularly useful tool in distributed and network security with more and more secure protocol designs employing some form of data encoding. Among a rich set of existing codes, due to their simplicity and strong error correcting capacity (i.e., information-theoretic rather than probabilistic), Reed-Solomon codes, or RS codes, are used very widely by applications. In an RS code, the input message is broken up into fixed-sized pieces and the pieces are regarded as coefficients of a polynomial, which is then repeatedly evaluated on different points to produce the output symbols. Although their asymptotic efficiency can vary depending on the implementation, RS codes typically involve encoding costs that are quadratic in the message size k,[1] thus in reality they tend to be costly.

Several alternative codes have been proposed to overcome the quadratic encoding/decoding overheads of RS codes. For instance, using layered encoding, Tornado codes [4] achieve encoding/decoding speeds that are 100 to 10000 times faster than RS codes. At the fastest end of the range lie LT codes [23], which achieve $O(k \log k)$ encoding/decoding speed and are very practical. Based on LT codes, Raptor codes [41] and Online codes [28] are the first (rateless) code to achieve *linear* coding times. Each of these LT-based codes is also a *fountain code*: a rateless code that can generate a practically unlimited number of output code symbols. But this great efficiency comes at a qualitative drawback: fountain codes have been designed and analyzed over a *random* channel rather than an adversarial one, essentially being capable to tolerate only data *erasures* and no (or very limited) data corruption. And even current standardized implementations of these codes are easy to attack by adopting malicious (non-random) corruption strategies.

Indeed, LT codes employ a random sparse bipartite graph to map message symbols, in one partition, into encoded symbols, in the other partition, via simple XORing (see Figure 1). By design, this graph provides enough symbol coverage so that a belief-propagation decoding algorithm can recover the input symbols (with a small, sender-determined probability of failure) despite random erasures of encoded symbols. But this algorithm will also readily *propagate (and amplify) any error* in the message encoding into the recovered message! Even worse, an adversary could exploit the structure of this graph to (covertly) increase the likelihood of a decoding failure. For instance, one can selectively erase nodes of high degree so that with high-probability not all input symbols are sufficiently covered by the remaining symbols.[2] Actually, this specific targeted-erasure attack raises a practical threat that has not been adequately addressed in the literature (cf. Section VII). Unfortunately, this was also neglected by existing RFCs that describe Raptor/RaptorQ codes for object delivery over the Internet: to increase practicality, the encoding graph is either completely deterministic or easily predictable by anyone,[3] thus trivially enabling such targeted-erasure attacks!

In view of this inconvenient trade-off between practicality and security, in this paper we consider the following natural question: is it possible to have codes that are simultaneously strongly tolerant to adversarial errors and very efficient in practice? Alternatively: what are the counterparts of RS codes within the class of fountain codes? Or, is it possible to devise extensions of Raptor codes that withstand malicious corruptions while maintaining their high efficiency?

This work shows that it is possible to achieve both coding efficiency and strong tolerance of malicious errors for computationally bounded adversaries. Specifically, we introduce

[1] RS codes with input size k and output size $n = O(k)$ are practically of quadratic in n complexity: even if in specific configurations they can achieve asymptotically $O(n \log n)$ encoding time, in most practical cases encoding with polynomial evaluation in $O(kn)$ time is faster.

[2] Decoders based on solving the implicit system of linear equations relating the symbols suffer from the same problems.

[3] For instance, each encoded symbol contains a list of the indices of its covering input symbols or a seed for a PRG to generate these indices, trivially allowing an adversary to selectively corrupt the symbols for maximum impact. Encrypting symbols and the associated index lists is not sufficient as the RFCs contain explicit tables of random numbers to be used in the encoding process.

Falcon codes, a class of *authenticated error correcting codes*[4] that are based on LT codes. Falcon codes can tolerate malicious symbol corruptions but also maintain very good performance, even *linear* encoding/decoding time. This can be viewed as a best-of-two-worlds quality, because existing authenticated codes (e.g., [18], [27], [29]) are typically Reed-Solomon based, thus lacking efficiency, and existing linear-time coding schemes (e.g., [28], [41]) are fountain codes that withstand only random erasures.

Moreover, Falcon codes can be extended in straightforward ways to meet the performance qualities of *any* other fountain code that employs an LT code. Specifically, our coding schemes can meet the performance optimality of Raptor or Online codes, while strictly improving their error correcting properties. Our experimental evaluation of Falcon codes relies exactly on such an implementation that is based on a Raptor code, perhaps the fastest linear-time fountain code at present. In this view, Falcon codes provide a general design framework for devising authenticated error correcting codes, which overall renders them a useful general-purpose security tool.

Contributions. Importantly, as part of this work, we have developed the first adversarial (corruption) model for analyzing the security of fountain codes against computationally-bounded adversaries. (Previously existing adversarial models were only applicable to fix-rate codes, such as RS codes.) We first introduce *private LT-coding schemes*, which model the abstraction of authenticated rateless codes that combine the structure of LT codes along with secret-key cryptography, and we then define a security game in which a stateful and adaptive adversary inflicts corruptions over message encodings that aim at causing decoding errors or failures (where the decoder does not recover any message). As we explain, we impose only minimal restrictions on the adversary: symbol corruptions and erasures can be arbitrary, but inducing trivial decoding failures by destroying (almost) all symbols is disallowed. Finally, we define security for private LT-coding schemes as the inability of the adversary to inflict corruptions that are (non-negligibly) more powerful than corruptions caused by a random erasure channel.

We also provide three constructions of *authenticated LT codes* that achieve this new security notion in our adversarial model while preserving the efficiency of normal LT codes. Our core scheme, shown schematically in Figure 3, leverages an extremely simple combination of a strong pseudorandom number generator (to randomize and protect the encoding graph), a semantically secure cipher (to encrypt encoded symbols and hide the encoding graph), and an unforgeable MAC (to authenticate symbols). To achieve better scalability, we extend this scheme to two more elaborate and highly optimized constructions: one fixed-rate code that applies our core secure encoding in blocks of input symbols, and its rateless extension that can produce unlimited encoded symbols. We provide a detailed comparison of our schemes in Table I.

Finally, we perform an extensive experimental evaluation of our constructions showing that, indeed, they achieve practical efficiency with low overhead. In particular, our schemes can achieve encoding and decoding speeds up to 300MB/s, several times faster than the standard Reed-Solomon encoding coupled with encryption and a MAC. Moreover, the overhead from our cryptographic additions to LT codes results in a slowdown of no more than 60% with typical slowdown close to just 25%. In Table I, we compare our constructions with a standard RS code, as well as the naïve "secure" LT code that only encrypts and MACs the output symbols, according to the following criteria: rateless property, asymptotic efficiency of encoding and decoding (FalconR's performance depends on the number of blocks, as we discuss in Section IV-C), the need for a strong PRG (FalconS can safely employ a weak, but fast, PRG for increased efficiency), and the achieved security related to adversarial data corruption. All of our constructions can withstand this worst-case behavior.

Although applying Falcon codes to reliable data transmission and storage is out of the scope of this work, we believe that many such applications are feasible. In particular, our schemes can be used as drop-in replacements for any error correcting code used against a computationally-bounded adversary and provide immediate efficiency gains. For instance, RS codes are often used to provide fault-tolerance in secure storage; Falcon codes can provide almost the same guarantees as RS codes while providing much higher efficiency. They also readily find application in proof-of-retrievability (PoRs) systems, such as HAIL [2], that provide auditing checks on the retrievability and recoverability of outsourced data via careful data encoding against data-corrupting attackers.

Organization. Section II provides background information on relevant coding theory and cryptography. Section III details our security formulation in a new adversarial setting applied to rateless codes. Section IV details our Falcon code constructions, including an authenticated LT code and two extensions that achieve scalability and generality. Section V provides a security analysis of Falcon codes. Section VI details on a comprehensive experimental evaluation of our new codes. We overview related work in the overlap of coding and security in Section VII and conclude with Section VIII.

II. Preliminaries

In what follows, we let λ denote the security parameter, and PPT refer to probabilistic polynomial-time algorithms. We also let $[\mathsf{Alg}]$ denote the set of all possible outputs of a PPT algorithm Alg, running on input parameters π, and $\tau \leftarrow \mathsf{Alg}(\pi)$ the particular output derived by a specific random execution of $\mathsf{Alg}(\pi)$. Analogously, we let $x \xleftarrow{R} S$ and $x \leftarrow D$, respectively, denote the process of sampling x from the set S uniformly at random or according to distribution D. Finally, we let $\circ$ denote string concatenation and $|S|$ denote the cardinality of a set S.

A. Coding theory

An *error correcting code* (ECC) is a message encoding scheme that can tolerate some corruption of the encoded data and still allow recovery of the message from the (possibly corrupted) codeword. Codes that are designed to recover only from partial data loss, but not data corruption, are called *erasure codes*. We first present the definition of *fixed rate* codes (also called *block* codes) which are the most common type of ECCs. Later, we will define *rateless* codes.

Defined over a fixed, finite set of symbols Σ (called the *alphabet*) and parameterized by integers k and $n \geq k$, a

[4]By this term, here, we informally refer to error correcting codes that employ cryptography to withstand adversarial symbol corruptions, typically (but not exclusively) by authenticating the integrity of the encoded symbols.

TABLE I. COMPARISON OF THREE FALCON CODE VARIANTS WITH REED-SOLOMON AND LT CODES. HERE, k IS MESSAGE LENGTH, b IS NUMBER OF BLOCKS, AND n IS THE CODEWORD LENGTH. Falcon IS THE BASIC SCHEME AND FalconS AND FalconR ARE THE SCALABLE VARIANTS. FOR FalconS AND FalconR, k INDICATES THE NUMBER OF SYMBOLS PER BLOCK.

Construction	Rateless	Efficiency	Strong PRG	Weak PRG	Adversarial Attacks
Falcon	yes	$O(k \log k)$	yes	no	yes
FalconS	no	$O(bk(\log k + \log b))$	yes	yes	yes
FalconR	yes	$O(bk \log k)$, $O(b \log b + bk \log \log b)$	yes	no	yes
Reed-Solomon	no	$O(nk)$	n/a	n/a	yes
LT code + encrypt & MAC	yes	$O(k \log k)$	no	yes	no

fixed-rate ECC specifies mappings between elements of the set of *messages* Σ^k and the set of *codewords* Σ^n. (Examples alphabets include $\Sigma = \{0,1\}^l$, the set of all l-bit strings, or a finite field $\mathbb{F}$.) For $m \in \Sigma^k$ and $c \in \Sigma^n$, the components m_i and c_i are called *message symbols* and *code symbols*, respectively. Any given message $m \in \Sigma^k$ is *encoded* to a corresponding *valid* codeword $c \in \Sigma^n$, and any *invalid* codeword $\hat{c}$, derived as a bounded distortion of c, can be *uniquely* decoded back to the original message m. If the first k symbols of a valid codeword corresponds to original k message symbols, then the code is said to be *systematic*. Parameters k and n are called the *message length* and *block length* of the code, respectively. The ratio $R = k/n$ is the *rate* of the code, capturing the amount of information transmitted per codeword and typically controlling the recovery strength of the code as follows. The *Hamming distance* between two n-symbol words x and y is the number of symbols in which they differ, defined as $\Delta(x,y) = |\{i \mid 1 \leq i \leq n, x_i \neq y_i\}|$. The *(minimum) distance* of an ECC is d, if for all valid codewords $c, c' \in \Sigma^n$ such that $c \neq c'$, we have $\Delta(c,c') \geq d$ (capturing the minimum number of changes needed to transform one valid codeword into another). Then, any code with minimum distance d allows for decoding invalid codewords distorted by up to $\lfloor d/2 \rfloor$ errors back to a unique message;[5] typically, the smaller the R is, the larger the d.

Definition 1: An *error correcting code* C over an alphabet Σ with rate R and minimum distance d, is a pair of maps $(\mathsf{Encode}, \mathsf{Decode})$, where $\mathsf{Encode} : \Sigma^k \to \Sigma^n$ and $\mathsf{Decode} : \Sigma^n \to \Sigma^k$, such that $k = Rn$, and for all $m \in \Sigma^k$ and for all $c \in \Sigma^n$ with $\Delta(c, \mathsf{Encode}(m)) \leq \lfloor d/2 \rfloor$, $\mathsf{Decode}(c) = m$.

Rateless ECCs. This class of error correcting codes, called *rateless* or *fountain* codes, employs no fixed block length n. Instead, rateless codes can generate a limitless stream of encoded symbols (i.e., a continuous "fountain"), allowing recovery of the original message (with high probability) from any subset of these symbols that is sufficiently large. Typically, for message length k, $(1+\varepsilon)k$ correct encoded symbols are needed to decode the message with probability at least $1-\delta$. Here, ε is called the *overhead* of the code and δ refers to the *decoding failure probability* of the code, the latter determining the range of possible values that the former may have; in particular, smaller values of δ require larger values of ε and vice versa. We denote this relationship by $F(\delta, \varepsilon)$.

Definition 2: A *(k, δ, ε)-rateless error correcting code* C over an alphabet Σ with decoding failure probability δ and overhead ε so that $F(\delta, \varepsilon)$, is a pair of maps $(\mathsf{Encode}, \mathsf{Decode})$, where Encode maps from Σ^k to Σ^∞ (infinite sequences of elements in Σ) and for any $m \in \Sigma^k$ and any finite subsequence s of $\mathsf{Encode}(m)$ of length at least $(1+\varepsilon)k$, $\mathsf{Decode}(s) = m$ with probability at least $1-\delta$.

[5]Not considered in this work is list-decoding, which allows mapping any invalid codeword, distorted with errors *beyond* this half-the-distance bound, back to a list of messages that always contains the correct original message.

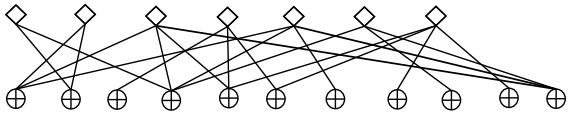

Fig. 1. LT-encoding: Each code symbol (bottom) is the XOR of $O(\log k)$ randomly selected message symbols (top).

Examples of rateless ECCs include LT codes [23], along with their derived extensions Raptor codes [41] and Online codes [28]. For LT codes, the main focus in this work, the encoding/decoding mappings take an extra input parameter: the *degree distribution* $\mathcal{D}$. Here, $\mathcal{D}$ is used to construct a sparse bipartite graph with input (message) symbols in one partition and code symbols in the other, also called *input* and *parity nodes*—see Figure 1. The degree of each parity node is selected according to $\mathcal{D}$ and its corresponding neighbors (message symbols) are selected uniformly at random. The code symbol corresponding to a given parity node is simply the XOR of the message symbols of the neighboring input nodes. The distribution used must be carefully chosen to achieve the desired success probability of $1-\delta$ for a given message length k, hence $\mathcal{D}$ is parameterized by k and δ, and thus denoted by $\mathcal{D}_{k,\delta}$. The value of δ and, now also, distribution $\mathcal{D}_{k,\delta}$ determine the possible values of ε. For LT codes, we denote this relationship by $F(\delta, \mathcal{D}_{k,\delta}, \varepsilon)$. For simplicity, we will omit the subscripts k and δ, leaving the dependence implicit, and will denote the degree distribution as $\mathcal{D}$.

Often, $\mathcal{D}$ instantiates to the *robust soliton distribution*, described in the original paper [23], which ensures that the average node degree is $O(\log k)$. This implies that: (1) using a balls-in-bins analysis, with high probability, every input symbol is covered by at least one of the $(1+\varepsilon)k$ code symbols, and thus can be decoded; and (2) encoding and decoding take $O(k \log k)$ time. Decoding uses a standard belief-propagation algorithm, thus correcting only symbol *erasures*, but not symbol *errors*. The decoding method of [26] augments belief propagation to also correct *random* errors with erroneous values distributed uniformly in Σ, but *not* adversarial errors.

Raptor (*rap*id *tor*nado) codes [41], a main application in our work, improve the performance of LT codes by employing *precoding* of the input message $m \in \Sigma^k$ before LT-encoding as follows. First, a linear-time erasure code (e.g., a low-density parity check code) is applied to m to get a group of *intermediate symbols*. Then, an LT code is used to produce each output symbol, again as the XOR of a random subset of the intermediate symbols, where these subsets, drawn via a variant of the robust soliton distribution (see [41]), are of *constant size*; this process is repeated until enough output symbols

are produced. Overall, Raptor codes have encoding/decoding time that is linear in k, and achieve high data rates with low overhead.[6] However, based on LT codes, they are essentially erasure codes and do not tolerate symbol corruption well. Analysis of Raptor codes over noisy channels [32] is also restricted to random (but not necessarily uniform) errors rather than adversarial ones.

B. Cryptographic tools

We overview the cryptographic primitives we employ to enhance LT codes to withstand adversarial errors—namely, message authentication codes (MACs), symmetric ciphers (SCs), and pseudorandom generators (PRGs), with which basic familiarity is assumed.

Keyed by secret $k \leftarrow \mathsf{Gen}_1(1^\lambda)$, a MAC produces a tag $t = \mathsf{Mac}(k, m)$ for message m, which can be used to verify the integrity of m by checking $\mathsf{VerMac}(k, m, t) = 1$. We require that a MAC is *existentially unforgeable* so that an adversary $\mathcal{A}$ cannot forge the tag for *any* message, even for one of its own choosing. Here, $\mathcal{A}$ is allowed to query a MAC oracle to get any number of example message-tag pairs before it outputs a target message-tag pair (m^*, τ^*) that does not belong in the queried pairs; then, with all but negligible probability in λ it holds that $\mathsf{VerMac}(k, m^*, t^*) \neq 1$.

Keyed by secret $k \leftarrow \mathsf{Gen}_2(1^\lambda)$, a symmetric cipher SC is an encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$ so that $\mathsf{Dec}(k, \mathsf{Enc}(k, m)) = m$ for any message m (in the appropriate message space). We require that a SC is *semantically secure* so that a ciphertext $c = \mathsf{Enc}(k, m)$ "hides" all the information about a given message m. Here, for any adversary $\mathcal{A}$ computing any function f on the message m given ciphertext c (and any auxiliary input), $f(m)$ can still be computed *without* the ciphertext, with all but negligible probability in λ. Intuitively, knowing the ciphertext c leaks no additional information about m.

Finally, given a short random seed $s \xleftarrow{R} \{0,1\}^\lambda$, a pseudorandom generator (PRG) serves as an efficient source of randomness by producing a long sequence of random-looking bits. A PRG is secure if its output is indistinguishable, with all but negligible probability in λ and with respect to any polynomial-time distinguisher, from a string of truly random bits. Equivalently, any algorithm taking random bits as input behaves only negligibly different when given pseudorandom bits instead.

III. Security Model

Our main goal is to extend LT codes to endure *adversarial* corruptions inflicted by a computationally-bounded adversary. Here, we present a new definitional framework for *private LT-coding schemes*, a new class of rateless codes that are based on LT codes and employ the use of secret-key[7] cryptography to resist polynomial-time adversarial errors. We introduce a corresponding new security notion we call *computationally secure rateless coding*. Our security model is general enough to also capture security for block codes.

Motivating scenarios. There are several adversarial settings which motivate our security model. First and foremost, we want to ensure that, with all but negligible probability, whenever Decode outputs a message, it is the same message that was encoded. That is, we wish to avoid corruption of the decoded message. Since total corruption or deletion of a message is catastrophic—and presumably users would stop using such an unreliable channel—we consider attacks that do not destroy or maul the entire message, which is standard when analyzing ECCs. For example, a client may encode their data and distribute it among several cloud providers, a subset of which are compromised by the adversary $\mathcal{A}$, limiting corruption to those servers.

An alternative scenario is where $\mathcal{A}$ is malicious network router that sees some, but not all of an encoded message. $\mathcal{A}$ can attack an LT-encoded message by erasing only high-degree symbols (i.e., those combining many input symbols) and leave low-degree symbols untouched. The decoder would then receive mostly low-degree symbols and be much more likely (i.e., with probability greater than δ) to fail and output nothing. This is essentially a stealthy DoS, or at least a quality-of-service attack, and is a violation of the upper-bound on the decoding failure probability. The receiver could wait for more good symbols from the non-malicious routers, but this would break the LT code's guarantee of needing only $(1+\varepsilon)k$ uncorrupted symbols to decode with probability greater than $1-\delta$. We seek to be secure even from this *targeted erasure* attack, an attack which was previously identified in [19], but was left as an open problem.

A. Private LT-coding schemes

Prior work has considered computationally bounded adversaries *only* against block codes. In particular, in his seminal paper [22], Lipton first modeled a *computationally bounded adversarial channel* that can corrupt at most a constant fraction ρ of the encoded symbols. Later, Lysyanskaya *et al.* [27] studied an (α, β)*-network (or channel)* that can arbitrarily corrupt (alter or delete) the n code symbols in an encoded message, and also insert new symbols into the codeword (even multiple versions of selected symbols), subject to two restrictions: (1) at least αn symbols survive corruption and (2) at most βn total symbols are received. Subsequent work by Micali *et al.* [29] provided a more involved security game that includes several rounds of encoding, bounded symbol corruptions, and decoding between a sender, an adversarial channel, and a receiver.

By explicitly bounding the fraction of all symbols that can be corrupted, however, these security models cannot capture corruptions against rateless codes. Since fountain codes can produce an unbounded number of symbols, the rate of corruption introduced by the adversary can continually grow and, indeed, can become *arbitrarily close to* 1, which would directly conflict with any bounded corruption rates. A more accurate modeling of errors over a rateless code is to *lower bound* the amount of *non-corruption* rather than upper-bound the amount of corruption, but as an absolute number (typically defined by the message length), not as a fraction of the encoded symbols. That is, we wish to ensure that there is some *minimum number*

[6] Since only a linear number of symbols are output, Raptor codes only strive to recover a *constant fraction* of the intermediate symbols via LT-decoding. Any gaps in these symbols symbols are recovered by decoding the precode.

[7] Our schemes use a PRG, a MAC, and a semantically secure cipher; the latter two can be replaced with public-key equivalents, while the PRG is inherently a secret-key scheme. However, coupling our schemes with a public-key key-agreement protocol (over a noiseless channel) would break their dependence on secret-key cryptography. But, for simplicity of analysis and presentation, we only utilize secret-key schemes.

of "good" encoded symbols that remain intact, and allow the remainder to be bad. In a block code, an upper-bound on badness implies a lower-bound on goodness (and vice versa), but this symmetry breaks in rateless codes.

We first provide the definition of *private LT-coding schemes*. This extends the definition of private coding schemes given in [29], via changes in the parameters and function specifications, to suit the class of rateless codes that are based on LT codes (instead of block codes). This class generically captures fountain codes that can produce an unlimited number of encoded symbols using the LT-coding technique over a set of "input" symbols (not necessarily the message symbols) using an appropriate degree distribution $\mathcal{D}$ (thus encompassing LT codes, Raptor codes and Online codes). Recall that for a degree distribution $\mathcal{D}$ for a message length k, we want that, given $(1+\varepsilon)k$ code symbols, decoding succeeds with probability $1-\delta$. We denote this relationship with $F(\delta, \mathcal{D}, \varepsilon)$. As in [22], we assume that the sender and receiver have a shared secret key sk. We also use nonces to prevent replays.

Definition 3: A $(k, \delta, \mathcal{D}, \varepsilon)$*-private LT-coding scheme* over an alphabet Σ with message length k, decoding failure probability δ, overhead ε and degree distribution $\mathcal{D}$ such that $F(\delta, \mathcal{D}, \varepsilon)$, and key space $\mathcal{K}$, is a triple of PPT algorithms $(\mathsf{Gen}, \mathsf{Encode}, \mathsf{Decode})$, where:

- Gen: on input security parameter 1^λ, outputs a random *secret key* $sk \in \mathcal{K}$;
- Encode: on input (1) secret key sk, (2) nonce ℓ, (3) decoding failure probability δ, (4) degree distribution $\mathcal{D}$, (5) overhead ε, (6) and the message $m \in \Sigma^k$, outputs an infinite sequence $\{c_i\}_{i=1}^{\infty}$, with $c_i \in \Sigma$, referred to as a *codeword* or an *encoding* of m;
- Decode: on input (1) secret key sk, (2) nonce ℓ, (3) decoding failure probability δ, (4) degree distribution $\mathcal{D}$, (5) overhead ε, (6) and a string $c \in \Sigma^*$, where $|c| \geq (1+\varepsilon)k$, outputs a string $m' \in \Sigma^k$ with probability at least $1-\delta$, or fails (with probability at most δ) and outputs $\perp$.

We require that for all $m \in \Sigma^k$, $\mathsf{Decode}(sk, \ell, \delta, \mathcal{D}, \varepsilon, c) = m$ with probability at least $1-\delta$, where c is a finite subsequence of $\mathsf{Encode}(sk, \ell, \delta, \mathcal{D}, \varepsilon, m)$ of length at least $(1+\varepsilon)k$.

The above definition is general enough to also express two types of "crypto-enabled" block codes. First, any ECC with fixed rate ρ can be captured by having a null distribution $\mathcal{D}$, if necessary, and adjusting δ and ε according to the code (e.g., for Reed-Solomon codes $\delta = \varepsilon = 0$, while for Tornado codes $\delta, \varepsilon > 0$). More importantly, we can refine Definition 3 to get *block* versions of LT-coding schemes that simply produce codewords of *fixed size* (above the decoding threshold). A $(k, \delta, \mathcal{D}, \varepsilon)$*-private block LT-coding scheme* with block-size n is defined as a $(k, \delta, \mathcal{D}, \varepsilon)$-private LT-coding scheme where Encode, given an additional input parameter n, produces codewords c with $|c| = n \geq (1+\varepsilon)k$. In what follows, let $\mathcal{LTS} = (\mathsf{Gen}, \mathsf{Encode}, \mathsf{Decode}, \pi)$, denote a $(k, \delta, \mathcal{D}, \varepsilon)$-private LT-coding scheme with $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$, its n-symbol refinement by $\mathcal{LTS}_n = (\mathsf{Gen}, \mathsf{Encode}, \mathsf{Decode}, \pi, n)$, and any (rateless or block) LT-coding scheme by $\mathcal{LTS}_*$.

Also, note that the probabilistic decoding requirement expressed by relation $F(\delta, \mathcal{D}, \varepsilon)$ imposes a minimum expansion factor $(1+\varepsilon)$ on message encoding. However, due to its dependence on the distribution $\mathcal{D}$, the failure bound δ holds when there are "enough" code symbols produced in an absolute sense. In practice, this further restricts message length k to be sufficiently large.[8]

B. Security game

We next present a general security model against computationally bounded adversaries that is applicable to private LT-coding schemes (and their fixed-size block variants). Our goal is to capture tolerance against adversarial symbol corruptions for LT-coding schemes, thus defining a much stronger security notion over the existing one that considers only random symbol corruptions.

We thus consider a transmission channel that is *fully* controlled by a computationally bounded adversary $\mathcal{A}$. That is, as new code symbols are produced by a private LT-coding scheme $\mathcal{LTS}$, $\mathcal{A}$ can maliciously corrupt any new or past such symbols. Moreover, the adversary is allowed to *adaptively* interact with the channel. That is, $\mathcal{A}$ is allowed to examine *any symbols* of its choice of the encoding (produced by $\mathcal{LTS}$) of *any message* of its choice and, additionally, to examine the decoded message (produced by $\mathcal{LTS}$) on *any corrupted set of symbols of its choice*. Finally, we consider a *stateful* channel. That is, $\mathcal{A}$ can remember any past selected message, its encoding, symbol corruptions and the corresponding recovered message, and depend current actions on the full such past history. Do note, however, that $\mathcal{A}$ is never given the bipartite graph underlying any of the LT-encodings.

In this powerful adversarial setting, we aim to ensure that any PPT $\mathcal{A}$ is only negligibly more powerful than the *random erasure channel* (the default operational setting for the LT codes). A random erasure channel chooses whether or not to erase a given symbol with some fixed probability $p \in (0,1)$, independent of its choice for the other symbols. Here, we further restrict p so that $1-p$ is a non-negligible function in the security parameter λ, thus allowing a non-negligible expected fraction of symbols to survive erasure (i.e., we assume it is feasible to communicate over the channel). We call such values of p *feasible* and denote this channel as REC_p. For a block code with message length k and block length n, we assume that $p \in (0, 1-k/n)$.

We define security in terms of a game $\mathsf{ChannelExp}_{\mathcal{A}, \mathcal{LTS}_*}(\pi)$, shown in Figure 2, that the adversary $\mathcal{A}$ seeks to win. $\mathcal{A}$ wins by either: (1) causing a *decoding failure*, where Decode outputs $\perp$; or (2) by causing a *decoding error*, where Decode outputs a message different from the one originally encoded. There are three participants in the game: the encoder Encode and decoder Decode of a given LT-coding scheme $\mathcal{LTS}_*$ with parameters $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$, and the adversary $\mathcal{A}$. The game consists of a "learning" phase and an "attack" phase. In the learning phase, there is a sequence of at most a polynomial number of rounds where, in the i-th round: (1) $\mathcal{A}$ selects a message m_i to be encoded by Encode; (2) Encode initializes itself with m_i; (3) $\mathcal{A}$ queries different symbols from Encode by having oracle access to symbols of the encoding of m_i; in particular, $\mathcal{A}$ interacts with oracle $\mathcal{O}_{m_i}$

[8] For example, the degree distribution in [41] for Raptor codes works well when k is in the tens of thousands or greater. Designing good degree distributions for smaller k (e.g., in the hundreds or thousands) remains an open problem.

which given as input an index j, it returns the j-th encoded symbol produced by Encode (for block codes, if $j > n$ then $\perp \leftarrow \mathcal{O}_{m_i}(j)$); (4) $\mathcal{A}$ outputs a (corrupted) codeword c_i consisting of N_i symbols in $\Sigma \cup \{\perp\}$, where a symbol $\sigma_i = \perp$ if it has been erased; (5) c_i is given to Decode for decoding; (6) Decode outputs a message r_i that is returned to $\mathcal{A}$. For simplicity, we assume that $\mathcal{A}$ outputs at least $(1+\varepsilon)k$ symbols, good or bad, at each step (otherwise Decode trivially fails and outputs $\perp$). For a memoryless channel, $\mathcal{A}$ makes no learning queries.

Eventually, $\mathcal{A}$ decides to enter the attack phase against scheme $\mathcal{LTS}_*$. The game proceeds with a special final round: $\mathcal{A}$ selects attack message m_a, queries encoded symbols of m_a using the oracle $\mathcal{O}_{m_a}(\cdot)$, and tries to cause decoding failure or a decoding error by computing a corrupted codeword c_a that decodes into message r_a. If Decode failed to decode (i.e., $r_a = \perp$) or decoded to the wrong message (i.e., $r_a \neq m_a$), then $\mathcal{A}$ wins; else, $\mathcal{A}$ loses. However, we require that, in the case of decoding failure, $\mathcal{A}$ must output at least $(1+\varepsilon)k$ unerased and uncorrupted symbols to win. Otherwise, $\mathcal{A}$ can trivially cause Decode to fail simply outputting only corrupted symbols, or by erasing almost all symbols.[9] For a decoding error, we have no such restriction. Let Q_m denote the set of symbols queried by $\mathcal{A}$ from $\mathcal{O}_m$. Abusing notation, let $c \cap Q_m$ denote the subset of symbols in codeword c (which was output by $\mathcal{A}$) that are in Q_m. Thus, for decoding failure, we require that $|c_a \cap Q_{m_a}| \geq (1+\varepsilon)k$.

$\mathsf{ChannelExp}_{\mathcal{A},\mathcal{LTS}_*}(\pi)$:

1) $\psi \leftarrow \perp$ ▷ Initial state of $\mathcal{A}$
2) $m_{\mathcal{R}} \leftarrow \perp$ ▷ Storage for decoded messages
3) $i = 1$ ▷ Number of queries
4) $s \leftarrow \mathsf{Gen}(1^\lambda)$ ▷ Secret-key generation
5) **while** $\mathcal{A}$ has a new query **do** ▷ Learning phase
   a) $\mathcal{A}(\pi, \psi, m_{\mathcal{R}}) \rightarrow (\psi', m_i)$ ▷ Message selection
   b) Generate a fresh nonce ℓ
   c) Set $\pi' = (1^\lambda, s, \ell, \delta, \mathcal{D}, \varepsilon)$
   d) Initialize oracle $\mathcal{O}_{m_i}$ to provide access to output symbols of $\mathsf{Encode}(\pi', m_i)$
   e) $\mathcal{A}(\pi, \psi', m_i)^{\mathcal{O}_{m_i}(\cdot)} \rightarrow (\psi'', c_i)$ ▷ Codeword corruption $c_i = (\sigma_{1,i}, \ldots, \sigma_{N_i,i})$ of size N_i
   f) $\mathsf{Decode}(\pi', c_i) \rightarrow r_i$ ▷ Message recovery
   g) Set $m_{\mathcal{R}} \leftarrow r_i$, $\psi \leftarrow \psi''$, $i \leftarrow i+1$, and continue
6) **end while**
7) $\mathcal{A}(\pi, \psi, m_{\mathcal{R}}) \rightarrow (\psi', m_a)$ ▷ Attack phase
8) Generate a fresh nonce ℓ
9) Set $\pi' = (1^\lambda, s, \ell, \delta, \mathcal{D}, \varepsilon)$
10) Initialize $\mathcal{O}_{m_a}$ to access symbols of $\mathsf{Encode}(\pi', m_a)$
11) $\mathcal{A}(\pi, \psi', m_a)^{\mathcal{O}_{m_a}(\cdot)} \rightarrow (\psi'', c_a)$
12) $\mathsf{Decode}(\pi', c_a) \rightarrow r_a$
13) If $r_a \neq m_a$ and $r_a \neq \perp$ then output 1
14) Else if $r_a = \perp$ and $|c_a \cap Q_{m_a}| \geq (1+\varepsilon)k$ then output 1
15) Else output 0

Fig. 2. Security game for private LT-coding schemes.

As a technical note, we disallow the adversary to query for symbols that are arbitrarily far along in the stream produced by $\mathcal{LTS}_*$ on any input message, thus avoiding the situation where $\mathcal{A}$ queries symbols that are infeasible for Encode to produce sequentially—this could give $\mathcal{A}$ an unrealistic amount of power.[10] In the game above, we call a query $\mathcal{O}_m(i)$ for the i-th symbol (τ, p)-*feasible* if,

$$P(\mathsf{Encode} \text{ outputs at least } i \text{ symbols over } REC_p) \geq \tau\ ,$$

and, further, we call τ-*admissible* any $\mathcal{A}$ making only τ-feasible queries for a given p. If $\mathcal{A}$ is τ-admissible for all feasible p and, additionally, τ is also non-negligible in λ, then we call $\mathcal{A}$ *admissible*. In what follows, we consider only admissible adversaries. Note that this restriction is related to the earlier restriction that we only consider feasible values of p for REC_p (i.e., $1-p$ is non-negligible). If p is allowed to be negligibly close to 1, then Encode will output an *exponential* number of symbols with high-probability. By restricting ourselves to feasible p, we ensure that Encode outputs a super-polynomial number of symbols over REC_p with only negligible probability; i.e., $P(\mathsf{Encode}$ outputs at least i symbols over $REC_p)$ is always negligible for super-polynomial values of i.

Computationally-secure LT-coding schemes. In normal operation over the random erasure channel, the probability that Decode fails is bounded by δ. We want to ensure that the probability that $\mathcal{A}$ wins the above game (either by causing a decoding failure or a decoding error) is at most negligibly greater than δ.

We do this by defining *computationally secure (rateless or block) LT-coding schemes*. Intuitively, we wish to ensure that the adversary $\mathcal{A}$ is only negligibly more likely to cause a decoding error or failure than an adversary who attacks the codeword with random erasures.[11] Thus, we first define a *random adversary* $\mathcal{R}$ interacting in a restricted manner with $\mathcal{LTS}_*$ in the above game. $\mathcal{R}$ takes as input the same tuple of parameters π and a probability p, where $1-p$ is non-negligible. Then $\mathcal{R}$ proceeds as follows: (1) it directly chooses an attack message $m_a \in \Sigma^k$ and outputs $(\perp, m_a)$, so that oracle $\mathcal{O}_{m_a}$ is initialized; (2) it queries $\mathcal{O}_{m_a}$ sequentially for encoded symbols; (3) for each retrieved symbol, it erases the symbol with probability p; otherwise it adds the symbol to a list (if the code has block-size n, then $\mathcal{R}$ erases at most pn symbols); (4) when $\mathcal{R}$ has more than $(1+\varepsilon)k$ symbols (or at least $(1-p)n$ symbols) in the list, it outputs the list and exits. Note that $\mathcal{R}$'s output is distributed identically to REC_p.

Let $\mathcal{LTS}_*$ be a private (rateless or block) LT-coding scheme, and let $\mathsf{ExpAdv}_{\mathcal{A},\mathcal{LTS}_*}(\pi)$ be the experiment $\mathsf{ChannelExp}_{\mathcal{A},\mathcal{LTS}_*}(\pi)$ as defined above, where $\mathcal{A}$ is a PPT admissible adversary. Also, let $\mathsf{ExpRand}_{\mathcal{LTS}_*}(\pi, p)$ be $\mathsf{ChannelExp}_{\mathcal{R}_p,\mathcal{LTS}_*}(\pi)$ where $\mathcal{R}_p = \mathcal{R}(\pi, p)$ and p is the erasure probability. Let $\mathsf{Adv}_{\mathcal{A},\mathcal{LTS}_*}(\pi, p) = |P[\mathsf{ExpAdv}_{\mathcal{A},\mathcal{LTS}_*}(\pi) = 1] - P[\mathsf{ExpRand}_{\mathcal{LTS}_*}(\pi, p) = 1]|$. We call a private LT-coding scheme *computationally secure* if we have that for all PPT admissible $\mathcal{A}$ and for all feasible $p \in (0,1)$, $\mathsf{Adv}_{\mathcal{A},\mathcal{LTS}_*}(\pi, p)$ is negligible in λ.

Definition 4: We say that a private (rateless or block) LT-coding scheme $\mathcal{LTS}_*$ is *computationally secure* (or just

[9] Note, forged symbols are superfluous when trying to cause decoding failure since if a forgery causes a decoding failure, then the corresponding *uncorrupted* symbol would have caused the same failure.

[10] If $\mathcal{LTS}_*$ employs a PRG with finite state, $\mathcal{A}$ can simply query symbols at multiples of the period of the PRG, thus getting identically encoded symbols (as they use the same randomness) which renders decoding impossible.

[11] We define security relative to a random channel rather than in absolute terms since the LT code itself reduces a random channel to a noiseless channel. That is, the cryptographic enhancements reduce the adversary to a random channel which is further reduced by the LT code to a noiseless channel.

secure) if, for all PPT admissible adversaries $\mathcal{A}$ where $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$, and for all feasible $p \in (0,1)$ (or $p \in (0, 1 - k/n)$ for block variants), we have that $\mathsf{Adv}_{\mathcal{A},\mathcal{LTS}_*}(\pi, p)$ is negligible in λ.

IV. Core Falcon Codes

We now present our core technical constructions of three private LT-coding schemes. Based on LT codes and designed with a variety of efficiency goals, our new schemes define a broad *class* of *rateless (fountain) codes* as well as some corresponding *block-code refinements* which we generically term as *Falcon codes*. By construction, Falcon codes enjoy a "best-of-both-worlds" property: they are designed to maintain the asymptotic performance of LT codes, thus allowing great degrees of efficiency, including fast encoding/decoding; and at the same time, they are crypto-enhanced to achieve strong error-correction capabilities. Our three LT-coding schemes described below comprise of: a *main scheme* Falcon that is rateless and particularly simple, a *scalable scheme* FalconS that is block refinement of our main scheme achieving better scalability, and a *randomized scheme* FalconR that is a rateless refinement of our scalable scheme. We refer to these as *core Falcon codes*, which in essence provide a new design framework for secure LT-based codes tolerant to malicious errors. We describe all encoders, and the corresponding decoders are found in Appendix B.

A. Main LT-coding Scheme

As explained in Section II, LT codes [23] are a family of erasure codes of high efficiency, both theoretically ($O(k \log k)$ encoding/decoding time for message length k) and practically. The encoder works by selecting (through the robust soliton distribution) $O(\log k)$ message symbols on average to combine to form a code symbol. The analysis in [23] was originally over the binary erasure channel, though it easily generalizes to larger symbols.

While there has been some work on extending LT codes to withstand errors (e.g., [26], [32]), the studied channels have particular noise characteristics, such as additive white Gaussian or uniformly distributed noise, and adversarial errors are not considered. For Raptor codes, which are derived from LT codes, Internet standard RFC5053 [24] and its amendment RFC6330 [25], suggest using a simple checksum (e.g., CRC32) to detect any random corruptions of the encoded symbols. While this may be sufficient for small, random errors, it will crumble quickly when faced with a malicious attacks.

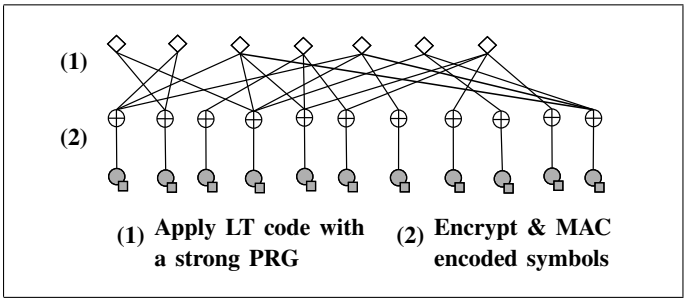

Fig. 3. An example encoding via authenticated LT codes.

Authenticated LT Codes. Our main scheme Falcon solves the challenge above by combining three cryptographic ingredients that are applied in a very simple and rather intuitive manner during LT-encoding (cf. Figure 3): a cryptographically strong PRG, a semantically secure cipher, and an existentially unforgeable MAC. (A nonce is also used to prevent replays.) First, the key change we perform in Encode compared to standard LT-encoding is to select each parity node's degree and its neighbors using the secure PRG. Then, after LT-coding, we encrypt all of the encoded (parity) symbols and compute a MAC tag for each encrypted symbol and the nonce (alternatively, we could use authenticated encryption with the nonce as additional authenticated data).[12] See also Figure 4. Intuitively, the encryption, MACs, and secure PRG work together to maintain the "goodput" of the channel by ensuring that each (uncorrupted) received symbol is just as "helpful" for decoding as when sent over a random erasure channel.

Note, in Figure 4, we use the subroutine LT-Encode which takes as input the seed s for the PRG, message size k, decoding failure probability δ, degree distribution D, overhead ε, the message m, and an index i, and then outputs the i-th code symbol. Note that we also use a key-derivation function (KDF) f to generate a seed for the PRG using the master seed s and the nonce ℓ. Any any secure KDF can be used as long as the output is sufficiently long to seed the PRG. If we have message size k, decoding failure probability δ, degree distribution $\mathcal{D}$, and overhead ε, then we denote this as $(k, \delta, \mathcal{D}, \varepsilon)$-Falcon.

This simple design provides a secure LT code (see Section V). First observe that the use of MACs ensures that any corruption in an encoded symbol is detected so that the symbol may be discarded. This step implements the standard known technique for reducing errors to erasures, that of authenticating the encoded symbols; we thus, informally, refer to our main scheme and its variants as an *authenticated LT code*.

Input: 1^λ, keys k_{enc}, k_{mac}, master seed s, nonce ℓ, message size k, decoding failure prob. δ, degree distribution $\mathcal{D}$, overhead ε, message m
Output: Authenticated codeword c

1) Set $s' \leftarrow f(s, \ell)$ ▷ f is a KDF; use s' to seed the PRG
2) Set $i = 0$ and $\pi = (1^\lambda, s', k, \delta, \mathcal{D}, \varepsilon, m)$
3) Initialize LT-Encode(π)
4) **for** as long as required **do**
5) Set $\sigma \leftarrow$ LT-Encode(π, i) ▷ i-th LT code symbol σ
6) Set $e_i \leftarrow \mathsf{Enc}(k_{enc}, \sigma \circ i)$ ▷ Encrypt and MAC
7) Set $\tau_i \leftarrow \mathsf{Mac}(k_{mac}, e_i \circ \ell)$
8) Output $c_i \leftarrow e_i \circ \ell \circ \tau_i$
9) $i \leftarrow i + 1$

Fig. 4. Encoder of main LT-coding scheme Falcon.

But although a necessary condition, symbol verification is not sufficient for achieving security: input and parity symbols are interrelated through the underlying bipartite graph, so corruption of certain parity symbols may seriously disrupt recovery of certain input symbols. An adversary can partially infer this graph structure by looking at symbol *contents* (since code symbols are simply the XOR of a random subset of message symbols) and target specific symbols for erasure, seeking to maximally disrupt decoding.[13] Symbol encryption

[12] We also include each symbol's index i in the stream of symbols to prevent reorder and deletion attacks. When transmitting over a FIFO *erasure* channel (where the receiver learns which symbols were erased in transit), then we can omit a symbol's index from the encoding.

[13] E.g., for a message containing a single 1 bit (the rest being 0's), $\mathcal{A}$ can easily discern which parity symbols include the non-zero message symbol.

ensures that the content of each symbol (and any information about the graph structure contained therein) is hidden from the adversary. Similarly, the strong PRG ensures that the adversary cannot exploit any biases or weaknesses in the PRG (e.g., output that is initially biased, cf. RC4) to aid in guessing what the digraph structure may be. Overall, as the structure of the graph is unpredictable, an adversary will, intuitively, be unable to do better than random corruptions and erasures.

Batching. Note that secure MACing requires input lengths to be at least as large as the security parameter, thus Falcon operates on message alphabets where each symbol is at least λ bits in size, i.e., for typical applications at least 10 or 16 bytes. Authenticating multiple symbols with a single MAC (as a single "batch") is a reasonable implementation in many circumstances—for instance, when grouping multiple symbols together to be sent in a single network packet. However, this batching results in corruption amplification during decoding (a single symbol corruption can cause many other possibly valid symbols to be discarded). But, in channels where errors are of low rate or are *bursty*, authenticating a batch of symbols would lead to throughput gains since the cryptographic overhead is reduced. Batching also removes the minimum-size restriction for the symbols as long as the total batch size is at least λ. Also note that batching increases the decoding overhead ε since $(1+\varepsilon)k$ is not necessarily divisible by the batch size. With a batch of b symbols, the decoder must receive $\lceil (1+\varepsilon)k/b \rceil b \leq (1+\varepsilon)k + b - 1$ symbols, giving an effective overhead of $\varepsilon' \leq \varepsilon + \frac{b-1}{k}$. Since the presence or absence of batching does not affect our security analyses (with the exception of the scalable, block code, detailed next), for simplicity, we assume that batches have size 1.

Finally, we note that appending MACs to parity symbols increases the space overhead per symbol. For a block code, this decreases the rate by a factor of $m/(s+m)$ where m is the MAC size and s is the symbol size. Similarly, for a rateless code, the number of raw bits transmitted increases by a factor of $1+m/s$. For instance, if $s = m$ (the minimum symbol size), then size of an output symbol is doubled. However, if $m \ll s$, then this overhead is marginal (e.g., for $s = 1024$ bytes and $m = 16$ bytes, the overhead is a factor $\approx 1.5\%$). Batching l symbols per MAC decreases the overhead to $m/(ls+m)$.

B. Scalable (Block) LT-coding Scheme

Although simple and efficient, the authenticated LT code presented above suffers from scalability problems in certain cases. In particular, if the input file and the internal state of the encoder do not fit into main memory, then the operating system will need to continually swap pages in and out. Then, on average, $O(\log k)$ random disk reads will be needed to produce a single parity symbol, as any such symbol depends on randomly selected message symbols. Magnetic disks have random read latencies around 5–10 milliseconds while SSDs have read latencies on the order of 10s of microseconds. But both of these are orders of magnitude larger than the 10s of nanoseconds require to read from RAM. Hence, the large amount of IO required when paging in and out will drastically slowdown the encoder and decoder (see Figure 13 in Section VI).

We can mitigate this limitation by adopting a simple divide-and-conquer strategy on our main scheme Falcon, towards the design of a new *scalable* scheme FalconS: we divide the input into blocks (of symbols) and then encode each block independently using Falcon. This immediately increases data locality during message encoding, which provides better scalability (as only a portion of the input must reside in memory at any given time) and further allows for easy parallelization (see Figures 9, 10, and 14 in Section VI). This method, though, introduces a new security concern: depending on parameterization, an adversary may put all of its effort in corrupting parity symbols coming from a single (or a few selected) block(s), thus significantly increasing its advantage in causing a decoding failure, even if all block encoders have produced symbols beyond the LT-decoding threshold.

To defend against this attack type, we adopt a technique due to Lipton [22]. We apply a *random permutation* π over all produced parity symbols across all blocks. This ensures that any corruptions performed by an adversary are distributed uniformly both *among* all blocks and *within* each block which, in turn, allows Falcon to use a weaker and faster PRG when producing an individual block encoding, since any corruption on this block cannot be targeted, but only random. Applying π over all parity symbols, though, necessitates a *fixed upper bound* on their total number, thus making FalconS a *block* code—this can be considered a useful byproduct of our new scheme. But employing (rateless) Falcon within (block) FalconS, as above, requires careful encoding parameterization. Given an adversarial *corruption rate* γ—with which an adversary corrupts a γ-fraction of all symbols, or γ-fraction of batches when using batching—and by applying a random permutation π to all of the code symbols from all of the blocks, we have that the number of symbol corruptions per block are *binomially distributed*. Therefore, we add extra redundancy in each block encoding produced by Falcon, expressed by *tolerance rate* τ, to absorb any variance in per-block errors—otherwise, block (and also total) decoding fails.[14] An analysis bounding the value of τ as well as a proof of the efficiency of our scheme are in Appendix A.

Figure 5 details the above encoding using permutation π *explicitly*—we call this variant of our scalable (block) scheme FalconSe—but it is possible for the permutation to be applied implicitly, which we detail next. As before, we use a secure KDF f to derive a session seed for G using the master seed s and nonce ℓ. The algorithm as shown does not simply use Falcon as a black-box subroutine; rather, the Falcon encoding algorithm directly integrated. This is done to include the block index i with each message symbol which is used to ensure that any re-ordering of symbols by the adversarial channel can be mitigated.[15]

However, there is a drawback to using an explicit permutation π: new parity symbols must be buffered until they are all generated (and then can be permuted!).[16] If resources are constrained, this extra buffering can demand too much

[14] These losses can be mitigated by applying an additional level of erasure coding to the input before breaking it up into blocks, but any erasure code can only tolerate a certain number of block losses. In any case, we must bound the probability that a block receives "too many" corrupted symbols.

[15] This is necessary even if the channel is FIFO, since we also allow the adversary to delete symbols.

[16] At most, symbols can be output piece-meal by pre-computing π and placing a generated symbol in its final place. But the output could be blocked for some time waiting for the appropriate symbol to be produced.

Input: 1^λ, keys k_{enc}, k_{mac}, master seed s, nonce ℓ, symbols per block k, degree distribution $\mathcal{D}$, block decoding failure prob. δ, corruption rate γ, block-corruption tolerance τ, number of blocks b, message m
Output: Authenticated codeword c

1) Set $s' \leftarrow f(s, \ell)$ ▷ f is a KDF
2) Seed G with s' ▷ G is a strong (but slow) PRG
3) Set $M = \frac{1}{1-(1+\tau)\gamma}(1+\varepsilon)k$ ▷ Parameterization
4) Generate permutation Π of $b * M$ elements
5) Partition m into blocks $B_1, \ldots, B_b$
6) **for** $1 \le i \le b$ **do**
7) Generate a seed s_i using G
8) Set $\pi = (1^\lambda, s_i, k, \delta, \mathcal{D}, \varepsilon, M)$
9) Use weak PRG H for LT-Encode
10) Set $(\sigma_{i,1}, \ldots, \sigma_{i,M}) = \mathsf{LT\text{-}Encode}(\pi, B_i)$
11) **for** $1 \le j \le M$ **do**
12) Set $e_{i,j} \leftarrow \mathsf{Enc}(k_{enc}, \sigma_{i,j} \circ i \circ j)$
13) Set $\tau_{i,j} \leftarrow \mathsf{Mac}(k_{mac}, e_{i,j} \circ \ell)$
14) Set $c_{i,j} \leftarrow e_{i,j} \circ \ell \circ \tau_{i,j}$
15) Map each $c_{i,j}$ to position $\Pi(i * M + j)$ to get $c' = (c'_1, \ldots, c'_{bM})$
16) Output $c' = (c'_1, \ldots, c'_{bM})$

Fig. 5. Encoder of scalable LT-coding scheme FalconSe.

memory, induce swapping, and greatly reduce performance. In such a situation an *implicit* permutation would be better (see Figure 13 in Section VI). In this variant, π is generated *first* and then parity symbols are produced (by Falcon from the appropriate block encoder) *in the order of their final position* (i.e., after π would have been applied). This allows a Falcon encoder to output parity symbols in a streaming manner (but out-of-order). We call this variant of our scalable scheme FalconSi. For FalconSi or FalconSe, if we have k message symbols per-block, block decoding failure probability δ, degree distribution $\mathcal{D}$, overhead ε, an adversarial corruption rate of γ, corruption-tolerance parameter τ, and b blocks in the encoding, we denote this as $(k, \delta, \mathcal{D}, \varepsilon, \gamma, \tau, b)$-FalconS, where FalconS can be either FalconSe or FalconSi.

To minimize the overhead of managing the implicit permutation, it is desirable that the PRG state in each encoder—used to generate the bipartite graph—can be easily and quickly reset to produce a desired segment of pseudorandomness (like in the Salsa20 stream cipher [1]). That is, the PRG state would be reset to produce the pseudorandom bits that would have been output if the symbol generation was done in order.[17] If this feature is not present, all needed pseudorandom bits can be pre-computed (namely, computing the degree and neighbors of a parity node). Fortunately, the space needed to store the pre-computed bits will often be much less than what would be required to store the output symbols. For example, say we have $k = 2^{16}$ input symbols and that the average node degree is $\log k = 16$. Then, a given node will only require, on average, $2 * 16 + 2 = 34$ bytes to store the degree and neighbor indices (we assume that the degree can fit in 2 bytes).

Decoding Failure Probability. Suppose we want to achieve an overall decoding failure probability of δ for this scheme. Decoding fails when any of the individual blocks fail to decode, and this could be from either too many corrupted symbols in a block, so that there are fewer than $(1+\varepsilon)k$ good symbols, or we were unlucky and the code symbols did not cover all input symbols. The probability of the first case can be made negligible via the corruption tolerance parameter τ; the second case is intrinsic to the schemes themselves. Note that in this case, the failures are *independent* and so are the successes. Hence, the probability of decoding success is $(1-\delta')^b = 1 - \delta$ and solving for δ', we have $\delta' = 1 - \sqrt[b]{1-\delta}$. For example, with $b = 100$ and $\delta = 0.05$, we have $\delta' = 1 - \sqrt[100]{1-0.05} \approx 0.000512$.[18] Note that the smaller value of δ' implies that the decoding overhead for each block ε' is increased.

C. Randomized Scalable LT-coding Scheme

Our block-oriented code FalconS does indeed achieve better scalability than Falcon. (As we show in Section VI, it can, for instance, provide over a 50% speed-up for an input file 32MB in size.) However, FalconS codes are inherently no longer rateless codes: any permutation that is explicitly/implicitly applied to all symbols across all blocks precludes the possibility of rateless encoding.[19]

We now present an alternative approach that is block-oriented, thus still scalable, yet allows for rateless encoding. As before, the idea is to break the input up into blocks, and then apply a Falcon code to each block. However, the key idea now is to produce symbols for the output encoding by applying Falcon codes *in parallel* to the blocks, giving the randomized, scalable, and rateless scheme FalconR. In particular, for each block an independent instance of Falcon is initialized with a unique random seed generated by a strong PRG. Then, another strong PRG, keyed with another random seed, is used to iteratively select a block *at random* whose encoder will simply output its next symbol; this random selection is repeated (at least) until *each* block encoding has reached the required decoding threshold. See Figure 6 below. As before, we use a KDF f to derive a seed for G using the master seed s and nonce ℓ. As with FalconS, we integrate Falcon directly into the FalconR encoder, allowing us to include with each symbol the index of its block, mitigating symbol deletion and reordering attacks. If we have b blocks and encode each using a $(k, \delta, \mathcal{D}, \varepsilon)$-Falcon code, then we denote this $(k, \delta, \mathcal{D}, \varepsilon, b)$-FalconR.

We emphasize the subtle difference between codes FalconSi and FalconR: in a FalconSi code, the encoder for each block produces its symbols in a random order *induced by* π, but in FalconR code these are produced in the (correct) order *induced by the block encoder itself.*

FalconR codes retain the rateless property of the original Falcon codes: new symbols can be produced by continuing to select blocks at random and outputting symbols. The security of FalconR reduces to the security of Falcon codes: the random selection of encoders ensures that adversarial corruptions are randomly and uniformly distributed among the blocks preventing too many corruptions from landing in any one block. Moreover, the secure encoder used on each block ensures that, for any corruptions that occur in that block, the adversary can do no better than a random channel. Note that while FalconS

[17] This would, of course, require the pre-computation of all node degrees; otherwise, it is impossible to know "where" a given node's randomness lies.

[18] If we apply a $[n, b]$ erasure code across the blocks, then the probability of decoding success (i.e., that at most $n - b$ blocks fail to decode) is $\sum_{i=0}^{n-b} \binom{n}{i}(1-\delta')^{n-i}(\delta')^i$, which can be solved for δ' via numerical methods.

[19] To create the permutation, the number of code symbols must be known.

can use a weak PRG when encoding an individual block, here each instance of Falcon must use a strong PRG.

Input: 1^λ, keys k_{enc}, k_{mac}, master seed s, nonce ℓ, symbols per block k, block decoding fail. prob. δ, degree distr. $\mathcal{D}$, overhead per block ε, number of blocks b, message m
Output: authenticated codeword c

1) Set $s' \leftarrow f(s, \ell)$ ▷ f is a KDF
2) Seed PRG G with s'
3) Generate b seeds $s_1, \ldots, s_b$ using G
4) Divide m into b blocks $m_1, \ldots, m_b$
5) Set $\pi_i = (s_i, k, \delta, \mathcal{D}, \varepsilon)$ for $1 \leq i \leq b$
6) Init. encoders $\mathsf{LT\text{-}Encode}_1(\pi_1, m_1), \ldots, \mathsf{LT\text{-}Encode}_b(\pi_b, m_b)$
7) **for** as long as required **do**
8) Sample a block index i from G
9) Let σ be the the j-th, symbol of $\mathsf{LT\text{-}Encode}_i$
10) Set $e \leftarrow \mathsf{Enc}(k_{enc}, \sigma \circ i \circ j)$, and $\tau \leftarrow \mathsf{Mac}(k_{mac}, e \circ \ell)$
11) Output $e \circ \ell \circ \tau$

Fig. 6. Encoder of randomized LT-coding scheme FalconR.

Efficiency. The asymptotic efficiency of our randomized scheme is not immediately obvious. Enough encoded symbols must be produced for each block, namely at least m code symbols, for some $m = \Omega(k)$, where k is the number of input symbols per block. This problem is a generalization of the standard balls-in-bins problem which asks how many balls must be thrown at random into b bins to ensure that each bin has at least one ball. Here, what is relevant is how many balls must be thrown into b bins to get at least m balls in each bin. In [30], it is shown that, as b goes to infinity, the expected number of balls thrown is $b[\log b + (m-1)\log\log b + C_m + o(1)]$, for some constant C_m. That is, the expected number is $b\log b + b(m-1)\log\log b + O(b)$. Thus, on average, we gain (at least) an additional $\log\log b$ factor in the time to encode and decode the file.[20] The tolerance parameter τ of FalconS is no longer needed here; each block uses the Falcon encoder and FalconR is rateless, so more symbols can always be produced.

If, however, we have a small b (e.g., $b = 10$), then since k must be sufficiently large (and hence, so must m), by the law of large numbers the average number of balls thrown is $O(bm)$. The asymptotic behavior of FalconR as b grows is important since we would like this scheme to scale up to very large files (e.g., 10s of gigabytes or more). Suppose the degree distribution requires 12000 code symbols to recover the input with high probability, and that symbols are at least 10 bytes in size, then we get an implicit lower-bound on block size of about 128KB (less if symbols in a block are batched together). As a 1GB file will contain approximately 8000 such blocks, both the case where there are many blocks and the case where there are few must be considered.

V. Security Analyses

Our various constructions of Falcon codes seek to reduce an adversarial corrupting channel to a *random erasure channel* (REC) (i.e., by using MACs to detect corruption). Through such reductions, our LT-coding schemes inherit many of the properties of the original LT codes (e.g., the overhead ε and failure probability δ). We next provide proof outlines of reductions to REC for our core Falcon code schemes.[21] Full proofs are omitted for lack of space.

The adversary $\mathcal{A}$ can win the game ChannelExp by causing a decoding failure (Decode outputs $\perp$) or a decoding error (Decode outputs an incorrect message). Since these are mutually exclusive events, they are considered separately in the following lemmas. Lemma 1 states that the ability of $\mathcal{A}$ to cause a decoding error in authenticated LT code is directly reducible to the security of the message authentication code. Lemma 2 states that the probability of causing decoding failure is only negligibly different from a random channel. For simplicity, we here use memoryless channels: the proofs extend in a straightforward way to stateful channels by simply utilizing the nonces. Recall that, as stated in Section III, we only consider admissible adversaries $\mathcal{A}$ that are compared to random erasure channels with a feasible erasure probability p.

Lemma 1: Let $M = (\mathsf{Gen}_1, \mathsf{Mac}, \mathsf{VerMac})$ be an existentially-unforgeable MAC used to authenticate the symbols of a $(k, \delta, \mathcal{D}, \varepsilon)$-Falcon code $\mathcal{LTS}$, where $F(\delta, \mathcal{D}, \varepsilon)$ holds, and let λ be the security parameter. For all sufficiently large k and for all PPT $\mathcal{A}$, the probability that $\mathcal{A}$ wins $\mathsf{ChannelExp}_{\mathcal{A},\mathcal{LTS}}$ via a decoding error is negligible in λ.

Intuitively, for $\mathcal{A}$ to force Decode to make a decoding error, Decode must accept at least one corrupt code symbol, which can only happen if the MAC for the symbol has been forged. But, we assume that the MAC is existentially unforgeable, hence $\mathcal{A}$ could not have forged any symbol, except with negligible probability.

Lemma 2: Let $\mathcal{LTS}$ be a $(k, \delta, \mathcal{D}, \varepsilon)$-Falcon code where the relation $F(\delta, \mathcal{D}, \varepsilon)$ holds, $\Pi = (\mathsf{Gen}_2, \mathsf{Enc}, \mathsf{Dec})$ be a semantically secure symmetric cipher used by $\mathcal{LTS}$, and let G be a secure PRG used by $\mathcal{LTS}$. Then, for all sufficiently large k and for all PPT admissible $\mathcal{A}$, the probability that $\mathcal{A}$ wins $\mathsf{ChannelExp}_{\mathcal{A},\mathcal{LTS}}$ via a decoding failure is negligibly different from δ.

Here, we reduce security against decoding failure to the security of the PRG and the semantic security of the cipher. Roughly, the proof proceeds as follows. Since a semantically secure cipher is used to encrypt encoded symbols, the codeword c leaks no "information" about the underlying encoding of m to $\mathcal{A}$. This means that there exists an $\mathcal{A}'$ that *without* access to c, that is, *independent* of c, but given some information about m, outputs a set of indices of symbols to be erased, such that Decode fails to decode with probability $\approx \delta$ (i.e., negligibly different from δ). Since the symbols to be erased are chosen independently of the encoding c, the remaining symbols form a random graph over the input symbols and, by definition, we have reduced $\mathcal{A}$ to REC. The following theorem summarizes the security of our scheme. The result follows directly from the two lemmas above.

Theorem 1: Let $\mathcal{LTS}$ be a $(k, \delta, \mathcal{D}, \varepsilon)$-Falcon code utilizing an existentially-unforgeable MAC scheme M, a semantically secure cipher scheme Π, and a secure PRG G. Then, for all sufficiently large k, for all PPT $\mathcal{A}$, and for all feasible p, where $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$, we have that $\mathsf{Adv}_{A,\mathcal{LTS}}(\pi, p)$ is negligible in λ.

[20] If we use batching, then the same analysis applies if we have the "transmission unit" be a batch instead of a single symbol.

[21] For simplicity and clarity, we use asymptotic security statements, rather than exact security statements that quantify an adversary's $\mathcal{A}$ resources.

Note that the above theorem and lemmas apply to *non-systematic* Falcon codes. With a systematic code, $\mathcal{A}$ knows that the first k code symbols are equal to the input symbols. $\mathcal{A}$ can then make corruptions that are decidedly *non-random* against these symbols. That is, $\mathcal{A}$ has some *a priori* knowledge of the underlying encoding graph, and can adjust its strategy accordingly. We leave as an open problem the creation of a secure, systematic Falcon code.

For our scalable FalconS codes (both with an explicit permutation and without), the security of the scheme follows from the results of Lipton's work on scrambled codes in [22]. In particular, the random permutation of the symbols ensures that the erasures and errors are uniformly distributed among the blocks and within each block as well, which is the definition of a random channel. Note that in this model, the adversary $\mathcal{R}_\gamma$ erases up to a γ-fraction of symbols (or a γ-fraction of batches, if batching is used) rather than erasing with probability γ. Recall that each block has $N = \frac{1}{1-(1+\tau)\gamma}(1+\varepsilon)k$ symbols.

Theorem 2: Let $\mathcal{LTS}_{bN}$ be a $(k, \delta, \mathcal{D}, \varepsilon, \gamma, \tau, b)$-FalconS code that uses an existentially-unforgeable MAC scheme M, a semantically secure symmetric cipher Π, a secure PRG G to generate the permutation, divides the input into b blocks, and generates N symbols per block. Then, for all sufficiently large k and for all PPT $\mathcal{A}$ that corrupt up to a γ-fraction of symbols, letting $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$, we have $\mathsf{Adv}_{\mathcal{LTS}_{bN},\mathcal{A}}(\pi, \gamma)$ is negligible in λ.

Finally, for our randomized Falcon codes, we utilize the main Falcon code as a subroutine and then use a strong PRG to select the block to produce the next symbol. The security of this scheme reduces to that of the PRG and the main Falcon code to get the following theorem. This result, however, does not follow directly from Lipton's work as above, since the symbols are output *in order from each block*. Rather, the PRG ensures that, when receiving symbols, with high-probability, after receiving $O(b[\log b + (m-1)\log\log b])$ uncorrupted symbols we can decode successfully. Using the main Falcon code ensures that the adversarial corruptions by $\mathcal{A}$ using *a priori* knowledge of the scheme—$\mathcal{A}$ knows that the first symbols in the stream are the first symbols encoded by the individual blocks—does not give $\mathcal{A}$ a significant advantage.

Theorem 3: Let $\mathcal{LTS}_R$ be a $(k, \delta, \mathcal{D}, \varepsilon, b)$-FalconR code that uses an existentially-unforgeable MAC scheme M, a secure PRG G, a semantically-secure symmetric cipher Π, a secure $(k, \delta, \mathcal{D}, \varepsilon)$-Falcon code $\mathcal{LTS}$, and divides the input into b blocks. Then, for all sufficiently large k and for all PPT admissible $\mathcal{A}$, and for all feasible p, where $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$, we have that $\mathsf{Adv}_{\mathcal{LTS},\mathcal{A}}(\pi, p)$ is negligible in λ.

VI. Experiments

In this section we detail several experiments performed that demonstrate the practicality of our constructions. The experiments were run on two machines: one with "abundant" resources and the other with more limited CPU power and RAM. We use these two machines to measure the raw speed and efficiency of our schemes without constraints and to show that our scalable schemes do achieve better performance, especially when RAM becomes scarce.

The majority of the experiments were performed on the powerful machine with two 2.6GHz AMD Opteron 6282SE with 16 cores each and 64GB RAM running 64-bit Debian Linux with the 3.2.0 kernel and gcc version 4.7.2. The "resource-constrained" benchmarks were run on a 2.6GHz, Intel Core i5 processor with 4GB of RAM running 64-bit Arch Linux with the 3.14.19 kernel and gcc version 4.9.1. All implementations are a mix of C and C++ and are single threaded unless otherwise specified—the parallel implementation (detailed below) utilized pthreads—and were compiled with the '-O2' optimization flag. All schemes use the Salsa20 stream cipher (see [1]) as the secure PRG, SFMT as the fast/insecure PRG (see [37]), and PBKDF2 for key derivation [15]. The Jerasure v2.0 library [13] was used for the Reed-Solomon erasure code.

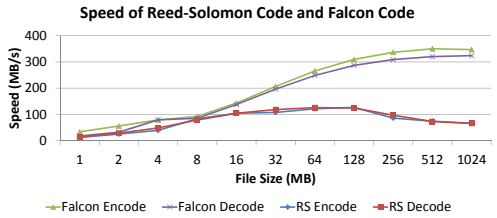

Fig. 7. The encoding throughput of Falcon compared to RS codes.

We use authenticated encryption for both confidentiality and authentication: specifically, we use AES in Galois/Counter Mode (GCM) provided in the OpenSSL library (version 1.0.1e and 1.0.1i on the Opteron and Core i5 machines, respectively). An alternative implementation would be to use AES in counter mode and pair it with a fast MAC, such as VMAC [20]. In some rough tests on the Core i5, we found the AES-VMAC combination to be the fastest for symbols more than 2KB in size and AES-GCM to be faster for smaller symbols.[22] The largest symbol size we consider is 4KB, with almost all tests run on symbols 1KB or smaller. Thus, for simplicity, we use AES-GCM in all tests. We did not use batching and, hence, encrypted and MACed each symbol individually.

The algorithms we benchmark are Falcon Raptor codes: we combine our authenticated LT code with a precoding step (where an erasure code is applied to the input data) to give us a secure Raptor code. Raptor codes are among the most efficient erasure-correcting codes available and we show below that our authenticated raptor codes themselves achieve high efficiency. Our implementations of both Falcon codes and Raptor codes are based on the `libwireless` code written by Jonathan Perry [36]. We used an LDPC-Triangle code as the precoding step using the implementation from [43]. Unless noted otherwise, all encoding and decoding was performed with 1KB symbols and adding 25% redundancy and the numbers given are an average of 10 trials.

Main Scheme Falcon**.** First, we compare our Falcon scheme to that of Reed-Solomon codes (RS codes), which is the de facto standard for encoding a file to withstand adversarial corruption. In Figure 7, we see the encoding and decoding speeds for Falcon versus an RS code on various file sizes when run on the Core i5. For Falcon, the number of symbols was held constant at ≈ 10000 for all files with symbol sizes ranging from 16 bytes to 128KB. The RS encoder utilized a systematic erasure encoding with $k = 204$ and $n = 255$

[22]For example, on a 16KB input, AES-GCM achieved 2.4 cycles-per-byte (cpb) while AES-VMAC achieved 1.7cpb; for a 256 byte input, AES-GCM was 7.6cpb and AES-VMAC was 14.0. All rates include key setup.

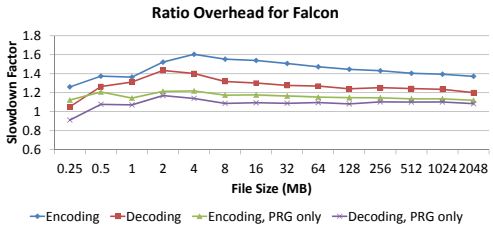

Fig. 8. Slowdown of Falcon versus an insecure Raptor code (no crypto).

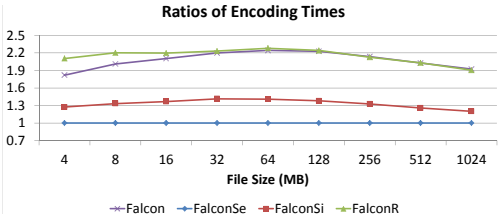

Fig. 9. Ratios of the encoding times of various file sizes by all schemes. Files were encoded with 64 blocks and 128 byte symbols.

(for 20% redundancy) over $GF(2^8)$ utilizing striping.[23] This allows us to quickly detect any tampering with the symbols and discard any that are corrupted and correct up to $n-k$ errors, the theoretical maximum.[24] As is clear from the graph, our scheme can achieve high throughput, reaching over 300MB/s for both encoding and decoding, and is several times faster than the RS encoder. Note that for inputs larger than 16MB, the throughput of Falcon can saturate a 1Gb network link.

In Figure 8, we see a comparison of Falcon against an insecure Raptor code where no cryptography was used (i.e., no encryption, MAC, or secure PRG). The numbers shown are the average of 50 trials. The simple scheme is generally between a factor of 1.25 and a factor of 1.6 slower than the (completely) insecure scheme, and is usually less than 1.5 times slower. Note that the overhead from the use of the secure PRG results in a slowdown of approximately 10-15% (shown in the lower two lines). For larger files, the percent overhead from the cryptography declines as the LT and precode encoding takes a larger percentage of the total encoding time. The "increase" in speed for decoding a 256KB file when using the secure PRG instead of an insecure one is due to environmental noise and the very short encoding times (a few hundred microseconds).

Scalability: Abundant Resources. In Figure 9, we can see that the FalconSe and FalconSi and schemes are, indeed, more scalable than Falcon. Note that FalconR performs worse than Falcon on smaller inputs and almost identically on larger inputs. (The difference in performance of these two is evident in Figure 13.) Falcon is slower than FalconSe and FalconSi primarily because it utilizes a secure PRG in the LT-coding and it has worse locality of reference: when performing the LT-coding, it combines symbols from across the entire file rather than just a segment of it. FalconSe is more efficient than FalconSi due to better locality of reference both for

[23] Striping is a technique where, instead of performing field operations over large symbols, the file is divided into symbols over a much smaller field (while using the same k and n) and encoded in small "batches" or "stripes." The small symbols are then grouped together to produce the large, output symbols. This can increase both encoding and decoding speeds.

[24] It is possible to use list-decoding for RS codes instead of MACs to correct up to $n-k$ errors, but list decoding algorithms, even the best ones, are many times slower than simply computing a MAC.

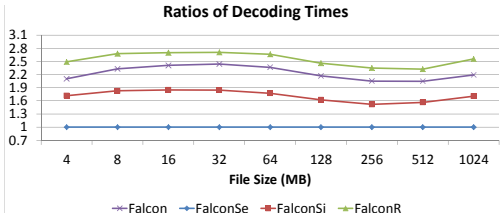

Fig. 10. Ratios of the decoding times of various file sizes by all schemes. Files were encoded with 64 blocks and 128 byte symbols.

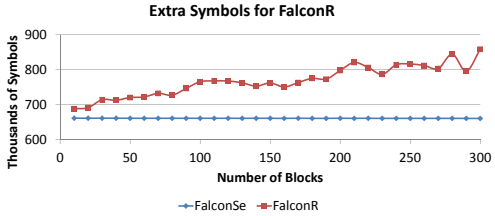

Fig. 11. Number of symbols generated by FalconSe and FalconR versus the number of blocks.

code and data since FalconSi continually switches between the encoders for each block. FalconSe encodes one block at a time and thus keeps less data resident at any given time, better utilizing caches. FalconR is the least efficient for three reasons: (1) it undermines locality by switching between encoders, (2) it generates more symbols than all other schemes (cf. Figure 11), and (3) it uses a secure PRG in the LT-coding, in contrast to the FalconS schemes. The benefit of using FalconR is shown in Figure 13 where it performs well when RAM is limited.

FalconR requires that we generate more symbols than FalconSe to ensure that we have enough to properly decode each block. FalconSe, in contrast, generates exactly the number of symbols required. In Figure 11 we can see the growth in the number of symbols needed by FalconR as compared to FalconSe. The input was a 2GB file and both schemes used 4KB symbols. The FalconR encoder works by having each block require a minimum number of symbols m to be produced and then generating symbols until each block has at least m symbols. When adding 25% redundancy, the encoder simply requires $(1.25)m$ symbols for each block. As mentioned in Section IV-C, [30] proves that for a small number of blocks, the expected number of symbols output is $O(bm)$. This is evident in in the figure since the number of symbols produced is close to optimal for $b < 40$. However, for larger b, the expected number produced is $O(b[\log b + (m-1)\log\log b])$, visible in the growing number of symbols generated as b increases.

Figure 12 compares the speed of FalconR to FalconSe on the same input as Figure 11. As the number of blocks increases, the size of each block decreases, allowing each individual block to be encoded faster. This is evident in the increasing speed of encoding and decoding for FalconSe. FalconR does not display this behavior: rather, its speed decreases since it must generate more symbols as the number of blocks increases as seen in Figure 11.

Scalability: Strained Resources. The benchmarks with strained resources were run on a 2.6GHz, Intel Core i5 processor with 4GB of RAM running Arch Linux. The tests compare each of the schemes on files ranging from 256MB to 3GB in increments of 256MB. Only the encoding speed was tested, not including I/O.

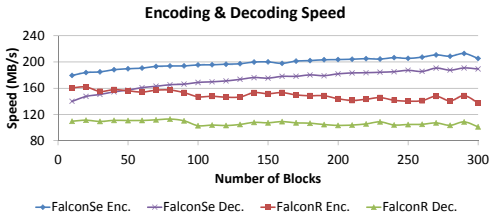

Fig. 12. Encoding/decoding speeds of FalconSe and FalconR.

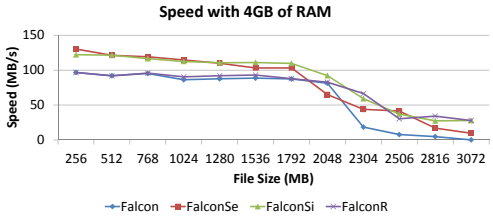

Fig. 13. Throughput of the different schemes in a more limited environment. Scalable schemes used 64 blocks.

In Figure 13, Falcon does reasonably well compared to the other schemes until the data and the overhead from the encoder take up too much RAM. As soon as Falcon needs to use external memory the throughput plummets and declines to almost zero. Comparing to Falcon, we can see that each of the scalable schemes are, indeed, more scalable and maintain modest speeds even for the largest inputs. Though, the throughput on the largest files is noticeably degraded compared to the smaller files, but this is expected. Note that FalconSe starts off with the highest speed, but then it slows down as the input size increases and becomes the slowest of the scalable schemes. This is due to the fact that FalconSe performs an explicit permutation of the output symbols, necessitating that all output symbols are buffered until they are permuted. FalconR and FalconSi, on the other hand, can simply output a symbol as soon as it is generated, keeping their memory utilization lower (and hence, induce less swapping). This smaller memory footprint gives them speed that is up to 3x faster than FalconSe for large files. It is notable that FalconR initially performs similar to Falcon (i.e., slower than the other schemes), but as the file size increases it scales well and matches the performance of FalconSi.

Parallelism. Each of the schemes described allows for some level of parallelism and can take advantage of multiple processing cores on a CPU. Falcon allows for parallelism at the symbol level: once the degree and neighbor set of a given symbol has been computed, it can be encoded independently of the other symbols. Each of the scalable schemes are trivially parallelizable at the block level; however, this does not readily lend itself to large performance gains on multi-core systems.

Figure 14 shows the results of running a multi-threaded version of FalconSe on the abundant-resource machine, using a 2GB file as input with various numbers of blocks and threads. In each case, the performance drops sharply after 4 threads, then declines slowly until another decline starting at 20 threads. This degradation in performance is due to the fact that encoding large files is a data-intensive operation. That is, the encoding operation is *memory-bound* rather than CPU bound. The precipitous drop at 8 threads is likely due to the on-CPU L1 and L2 caches becoming overwhelmed with the demand and contention, and the threads then needing to query the L3 cache and RAM for their data. Similarly, the drop at 20 threads

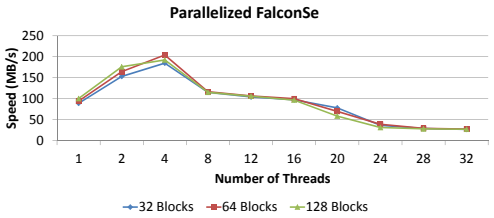

Fig. 14. Throughput of a parallelized version of FalconSe.

is likely due to L3 cache saturation and many references going all the way to RAM to be satisfied.

This phenomenon is due to the fact that the encode operation exhibits high spatial locality but poor temporal locality. Each symbol is used in its entirety (e.g., all 1KB of it), but once an input symbol has been XORed into an output symbol, that input symbol is typically not used again for some time. The spatial locality is exploited by the caches who can prefetch much of a symbol (especially for large symbols) and have the data cache-resident when the CPU needs it. If, however, there is too much demand on the cache, then the prefetched cachelines could be evicted for the sake of storing data that is needed immediately. In such a situation, the cache becomes useless. Thus, while the schemes are algorithmically simple to parallelize, the actual performance can suffer drastically when using many threads on current hardware.

VII. Previous Work

Computationally-bounded Channels. In this work, we prove our constructions secure against a computationally-bounded adversary, an approach which was first proposed in [22] (and further developed in [11]). In this, Lipton shows, using "code scrambling," that any code that has success probability q over the binary symmetric channel (BSC) succeeds with probability q over *any* computationally bounded channel (his construction was independently discovered in [26]). In [21], Langberg defines the notion of a private code: a code that takes a secret key as a parameter. He proves that any code over BSC, with error probability p, can be turned into a private code over an adversarial channel that corrupts at most pn symbols. This is done, as in [22], via random permutation over the code symbols, but Langberg makes no cryptographic assumptions.

In [29], the authors describe a game for a stateful adversarial channel that is computationally-bounded and they provide a construction that is provably secure. Their scheme is a combination of digital signatures and list-decoding: a message is signed and encoded with a list-decodable code, with the signature used to disambiguate the list-decoding (essentially the same scheme as found in the earlier work [27]). In [42], Smith protects against adversarial errors while using few random bits and standard complexity assumptions. The basic construction is the same as [22], except a random t-wise independent permutation is applied to an encoded message. In [12], the authors present codes for computationally bounded log-space channels, assuming the existence of one-way functions.

Codes & Cryptography. There are several examples of schemes that combine cryptography and ECCs together. One of the first was [18] where Krawczyk computes the hashes of erasure encoded symbols and then uses an ECC to encode each hash (an application of distributed fingerprints [17]). An example where cryptography is used in the heart of an ECC

is found in [34] (further developed in [35]) where the authors present a new rateless ECC called a *spinal code* that uses a hash function applied to segments of the input to create the "spines" used to produce code symbols. Cryptography has also been used to construct efficient locally-decodable codes such as [6] and [31] (the former also allows for local updates of the encoded message).

Proofs-of-Retrievability. A proof-of-retrievability (PoR), first defined and explored in [14], is scheme that make heavy use of cryptography and ECCs to secure remote storage. Two efficient PoRs are presented in [39] that are based on novel homomorphic authenticators. The work of [2] combines RS codes and universal hashing to produce secure, homomorphic MACs over the data, resulting in an efficient scheme secure against a *mobile adversary*. PoRs are further explored in [3] where a they describe rigorous theoretical framework for designing PoRs and provide improved variants of [14] and [39]. In [38], the authors present a PoR that utilizes Raptor codes in the audit protocol, allowing for efficient encoding, decoding, and auditing. In [40], the authors use a hierarchical log structure, where each level is erasure encoded, combined with Merkle trees and MACs to construct an efficient, dynamic PoR.

Multicast Authentication. In [27], the authors combine digital signatures with list-decoding to achieve error correction beyond the unique decoding radius as part of an authenticated multicast scheme. Specifically, they sign the message and then apply a list-decodable code; the received codeword is list-decoded and the signature is used to determine which list element is the correct decoded message. This combination of signatures and list-decoding was independently discovered in [29]. In [33], the authors also combine ECCs, cryptographic hashes, and signatures for secure multicast over adversarial erasure channels. In [44], the authors build on [27], and use LT codes as part of a multicast authentication scheme. They apply an LT code to a set of packets and use a combination of hashing, signatures, and list-decoding to detect packet corruption. However, they only consider data corruption and do not take into account the targeted erasure attack described earlier. In [16], the authors present *distillation codes*, which are a combination of an erasure code, a one-way accumulator, and a signature scheme. The input is signed and encoded, then the code symbols become the leaves of a Merkle tree. Each symbol is transmitted with its neighbors on the path to the root of the Merkle tree. When decoding, the Merkle tree detects corrupted symbols and the final message is verified via the signature.

Secure Storage & Non-malleable Codes In [5], the authors utilize LT codes, bilinear maps, and homomorphic MACs for a secure cloud storage scheme that allows for asymptotically efficient encoding and decoding, as well as repair of the data if any servers are lost. They note the targeted erasure attack on LT codes, but their solution was to check that all subsets of size k out of n code symbols can be successfully decoded and re-encoding the whole file if not. Related to secure storage are primitives known as *non-malleable codes*, first defined and constructed in [8]. Non-malleable codes seek to encode a message m such that, if it is tampered with, it will decode to a message *unrelated* to m. They were designed to protect against hardware tampering so that any manipulation results in a random internal state (rather than an adversarially chosen one). The work was further developed in [7] and [9].

VIII. Conclusion

We introduced a new security model for analyzing fountain codes over computationally-bounded adversarial channels, and presented Falcon codes, a class of (block or rateless) authenticated ECCs that are based on the widely used LT codes. Falcon codes are provably secure in our model while maintaining both practical and theoretical efficiency (including linear-time coding for Falcon Raptor codes). Their efficiency lends themselves a useful general-purpose security tool for many practical applications such as secure and reliable data transmission over a noisy (and possibly malicious) channel and secure data storage.

Acknowledgments

Research supported in part by the National Science Foundation under grants CNS–1012060 and CNS–1228485. We would like to thank the anonymous reviewers for their many insightful comments.

References

[1] D. Bernstein, "The Salsa20 family of stream ciphers," in *New Stream Cipher Designs*, ser. LNCS 4986, 2008, pp. 84–97.

[2] K. Bowers, A. Juels, and A. Oprea, "HAIL: A High-Availability and Integrity Layer for Cloud Storage," in *CCS*, 2009, pp. 187–198.

[3] ——, "Proofs of Retrievability: Theory and Implementation," in *CCSW*, 2009, pp. 43–54.

[4] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege, "A Digital Fountain Approach to Reliable Distribution of Bulk Data," *SIGCOMM*, vol. 28, no. 4, pp. 56–67, October 1998.

[5] N. Cao, S. Yu, Z. Yang, W. Lou, and Y. Hou, "LT Codes-based Secure and Reliable Cloud Storage Service," in *INFOCOM*, March 2012, pp. 693–701.

[6] N. Chandran, B. Kanukurthi, and R. Ostrovsky, "Locally Updatable and Locally Decodable Codes," IACR ePrint, Report 2013/520, August 2013.

[7] M. Cheraghchi and V. Guruswami, "Non-Malleable Coding Against Bit-wise and Split-State Tampering," *CoRR*, vol. abs/1309.1151, 2013.

[8] S. Dziembowski, K. Pietrzak, and D. Wichs, "Non-Malleable Codes," in *ICS*, January 2010, pp. 434–452.

[9] S. Faust, P. Mukherjee, D. Venturi, and D. Wichs, "Efficient Non-Malleable Codes and Key-Derivation for Poly-Size Tampering Circuits," IACR ePrint, Report 2013/702, 2013.

[10] R. A. Fisher and F. Yates, *Statistical tables for biological, agricultural and medical research*. Edinburgh: Oliver and Boyd, 1963.

[11] P. Gopalan, R. J. Lipton, and Y. Ding, "Error correction against computationally bounded adversaries," April 2004, unpublished manuscript.

[12] V. Guruswami and A. Smith, "Codes for Computationally Simple Channels: Explicit Constructions with Optimal Rate," in *FOCS*, 2010, pp. 723–732.

[13] J. S. Plank and K. M. Greenan, "Jerasure: A Library in C Facilitating Erasure Coding for Storage Applications – Version 2.0," U. of Tenn., Tech. Rep. UT-EECS-14-721, 2014.

[14] A. Juels and B. Kaliski, Jr., "PoRs: Proofs of Retrievability for Large Files," in *CCS*, 2007, pp. 584–597.

[15] B. Kaliski, "PKCS #5: Password-Based Cryptography Specification Version 2.0," RFC 2898, September 2000.

[16] C. Karlof, N. Sastry, Y. Li, A. Perrig, and J. Tygar, "Distillation Codes and Applications to DoS Resistant Multicast Authentication," in *NDSS*, 2004, pp. 37–56.

[17] H. Krawczyk, "Distributed Fingerprints and Secure Information Dispersal," in *PODC*, 1993, pp. 207–218.

[18] ——, "Secret Sharing Made Short," in *CRYPTO*, ser. LNCS 773, 1994, pp. 136–146.

[19] M. N. Krohn, M. J. Freedman, and D. Mazières, "On-the-Fly Verification of Rateless Erasure Codes for Efficient Content Distribution," in *IEEE Sym. on Sec. and Priv.*, 2004, pp. 226–240.

[20] T. Krovetz and W. Dai, "VMAC: Message Authentication Code using Universal Hashing," IETF Draft: draft-krovetz-vmac-01, 2007.

[21] M. Langberg, "Private Codes or Succinct Random Codes That Are (Almost) Perfect," in *FOCS*, 2004, pp. 325–334.

[22] R. Lipton, "A new approach to information theory," in *STACS*, ser. LNCS 775, 1994, pp. 699–708.

[23] M. Luby, "LT Codes," in *FOCS*, 2002, pp. 271–281.

[24] M. Luby, A. Shokrollahi, M. Watson, and T. Stockhammer, "Raptor Forward Error Correction Scheme for Object Delivery," IETF RFC5053, October 2007.

[25] M. Luby, A. Shokrollahi, M. Watson, T. Stockhammer, and L. Minder, "RaptorQ Forward Error Correction Scheme for Object Delivery," IETF RFC6330, August 2011.

[26] M. Luby and M. Mitzenmacher, "Verification-Based Decoding for Packet-Based Low-Density Parity-Check Codes," *IEEE Trans. on Info. Theory*, vol. 51, no. 1, pp. 120–127, 2005.

[27] A. Lysyanskaya, R. Tamassia, and N. Triandopoulos, "Multicast authentication in fully adversarial networks," in *IEEE Sym. on Sec. and Priv.*, 2004, pp. 241–255.

[28] P. Maymounkov, "Online codes," Secure Computer Systems Group, NYU, Tech. Rep., 2002.

[29] S. Micali, C. Peikert, M. Sudan, and D. Wilson, "Optimal Error Correction Against Computationally Bounded Noise," *IEEE Tran. on Info. Theory*, vol. 56, no. 11, 2010.

[30] D. Newman and L. Shepp, "The double dixie cup problem," *American Mathematical Monthly*, vol. 67, no. 1, pp. 58–61, 1960.

[31] R. Ostrovsky, O. Pandey, and A. Sahai, "Private Locally Decodable Codes," in *ICALP*, ser. LNCS 4596, 2007, pp. 387–398.

[32] R. Palanki and J. Yedidia, "Rateless codes on noisy channels," in *ISIT*, July 2004.

[33] A. Pannetrat and R. Molva, "Efficient Multicast Packet Authentication," in *NDSS*, 2003.

[34] J. Perry, H. Balakrishnan, and D. Shah, "Rateless Spinal Codes," in *HotNets-X*, 2011, pp. 6:1–6:6.

[35] J. Perry, P. Iannucci, K. Fleming, H. Balakrishnan, and D. Shah, "Spinal Codes," *SIGCOMM*, vol. 42, no. 4, pp. 49–60, 2012.

[36] J. Perry, "libwireless and wireless-python," www.yonch.com/wireless, (2014/05/06).

[37] M. Saito and M. Matsumoto, "SIMD-Oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator," in *MCQMCM*, 2006, pp. 607–622.

[38] S. Sarkar and R. Safavi-Naini, "Proofs of Retrievability via Fountain Code," in *FPS*, 2012, pp. 18–32.

[39] H. Shacham and B. Waters, "Compact Proofs of Retrievability," in *ASIACRYPT*, 2008, pp. 90–107.

[40] E. Shi, E. Stefanov, and C. Papamanthou, "Practical Dynamic Proofs of Retrievability," in *CCS*, 2013, pp. 325–336.

[41] A. Shokrollahi, "Raptor codes," *IEEE/ACM Trans. on Networking*, vol. 14, no. SI, pp. 2551–2567, June 2006.

[42] A. Smith, "Scrambling Adversarial Errors Using Few Random Bits, Optimal Information Reconciliation, and Better Private Codes," in *SODA*, 2007, pp. 395–404.

[43] STMicroelectonics and INIRA, "LDPC FEC Codes," planete-bcast. inrialpes.fr/rubrique.php3?id_rubrique=5, 2006.

[44] C. Tartary and H. Wang, "Rateless Codes for the Multicast Stream Authentication Problem," in *Adv. in Info. and Comp. Sec.*, ser. LNCS 4266, 2006, pp. 136–151.

APPENDIX A
ERROR AND ASYMPTOTIC ANALYSES OF FalconSe

Error analysis. In FalconS, the symbols of each block are permuted together uniformly at random, giving us, via a balls-in-bins analysis, that the number of corruptions in a given block is binomially distributed. Thus, we cannot guarantee that each block receives a bounded number of corruptions. Hence, we add extra redundancy to absorb the variance in the number of corruptions per block, increasing the total redundancy by a factor of $(1+\tau)$, where τ is the tolerance rate. We wish to minimize τ while ensuring that the probability that we exceed the error-correction capacity of a block is negligible in λ.

Suppose, there are b blocks, where each block has k input symbols and is encoded into m output symbols, giving $n = bm$ total symbols. Suppose a γ-fraction of symbols are corrupted. If we encode a block so that it can tolerate $(1+\tau)\gamma m$ corruptions (were γm is the mean number of corruptions per block), then we must generate $m = \frac{1}{1-(1+\tau)\gamma}(1+\varepsilon)k$ symbols per block. We use a Chernoff-bound to bound the probability that there are more than $(1+\tau)\gamma$ corruptions in a given block. For a binomially distributed random variable X with mean μ and for some $\tau > 0$ it holds that,

$$P(X \geq (1+\tau)\mu) \leq \left(\frac{e^\tau}{(1+\tau)^{(1+\tau)}}\right)^\mu.$$

For our parameters, we have that $\mu = \gamma\frac{1}{b}n = \gamma m$. Suppose we want the probability to be less than some value q. Note that, $(\frac{e^\tau}{(1+\tau)^{(1+\tau)}})^\mu \leq q$ is not solvable algebraically in terms of τ. However, it can be solved numerically. Consider the following parameterization where the message consists of $k = 15000$ symbols in each of $b = 100$ blocks and the decoding overhead is $\varepsilon = 0.05$. The adversary $\mathcal{A}$ corrupts a $\gamma = 0.2$-fraction of the output symbols and we add a τ-fraction additional redundancy to each block. Thus, each block consists of $m = \frac{1}{1-0.2(1+\tau)}(1+0.05)15000 = \frac{15750}{0.8-0.2\tau}$ symbols and the average number of corrupted symbols is $\gamma m = \frac{3150}{0.8-0.2\tau}$. Suppose that we want $P(X \geq (1+\tau)\mu) \leq q = 2^{-128}$. Solving for τ, we have that $\tau \approx 0.21354$. If we calculate the tail of the distribution exactly, we find that the value of τ is close to 0.20788. So, the approximation overestimated the necessary redundancy by just 2.7%, and for larger values of q the overestimation is smaller.

Asymptotic efficiency. We wish for our FalconS codes to achieve the $O(k \log k)$ encoding/decoding time for each block that LT codes (and our Falcon codes) achieve. The permutation step can be performed in linear time by using the Fisher-Yates algorithm [10], but the extra redundancy added requires a careful analysis. We show next that the tolerance parameter τ is $o(1)$, and so we maintain $O(k \log k)$ encoding and decoding. Recall the Chernoff-bound above, where $\mu = \gamma m = \frac{\gamma}{1-(1+\tau)\gamma}(1+\varepsilon)k$ is the mean number of corruptions per block. Let $\mu' = \frac{\gamma}{1-\gamma}(1+\varepsilon)k$, then

$$\left(\frac{e^\tau}{(1+\tau)^{(1+\tau)}}\right)^\mu \leq \left(\frac{e^\tau}{(1+\tau)^{(1+\tau)}}\right)^{\mu'}.$$

If we bound the right-hand side by q, then we have,

$$\mu' \ln\left(\frac{e^\tau}{(1+\tau)^{(1+\tau)}}\right) \leq \ln q$$
$$\mu'(\tau - (1+\tau)\ln(1+\tau)) \leq \ln q$$
$$(1+\tau)\ln(1+\tau) - \tau \geq \frac{-\ln q}{\mu'}.$$

Since the left-hand side is monotonically increasing, to minimize τ we must set the two sides equal. Thus we have,

$$(1+\tau)\ln(1+\tau) - \tau = \frac{-\ln q}{\mu'} = \frac{-(1-\gamma)\ln q}{\gamma(1+\varepsilon)k}.$$

Since γ, q, and ε are constants, we have $(1+\tau)\ln(1+\tau) - \tau = O(\frac{1}{k}) = o(1)$. Since $\tau = o((1+\tau)\ln(1+\tau))$, this implies

$\tau = o(1)$; i.e., the amount of redundancy is bounded by a constant and we preserve $O(k \log k)$ encoding/decoding times.

APPENDIX B
ENCODERS/DECODERS

Input: 1^λ, keys k_{enc}, k_{mac}, master seed s, nonce ℓ, message length k, decoding failure prob. δ, degree distribution $\mathcal{D}$, overhead ε, authenticated codeword $c = (c_1, \ldots, c_N)$
Output: Message $m = (m_1, \ldots, m_k)$

1) Let $Rem = \emptyset$ (i.e., the empty set)
2) **for** $1 \le j \le N$ **do** ▷ Sieve out corrupted symbols
3) Parse $c_j = e_j \circ \ell' \circ \tau_j$
4) **if** $\ell' \neq \ell$ or $\mathsf{VerMac}(k_{mac}, e_j \circ \ell, \tau_j) = 0$ **then**
5) Discard c_j and continue
6) **else**
7) $\sigma_j \circ j' \leftarrow \mathsf{Dec}(k_{enc}, e_j)$, $Rem \leftarrow Rem \cup \{(\sigma_j, j')\}$
8) Set $s' \leftarrow f(s, \ell)$ ▷ f is a KDF
9) Output $m \leftarrow \mathsf{LT\text{-}Decode}(1^\lambda, s', k, \delta, \mathcal{D}, \varepsilon, Rem)$

Fig. 15. Decoder of main LT-coding scheme Falcon

Input: 1^λ, keys k_{enc}, k_{mac}, master seed s, nonce ℓ, symbols per block k, block decoding failure prob. δ, degree distribution $\mathcal{D}$, overhead per block ε, corruption rate γ, tolerance τ, number of blocks b, authenticated codeword $c = (c_1, \ldots, c_{bM})$
Output: message m or $\perp$

1) Set $M = \frac{1}{1-(1+\tau)\gamma}(1+\varepsilon)k$ and $s' \leftarrow f(s, \ell)$ ▷ f is a KDF
2) Seed PRG G using s' and generate a perm. Π over bM elements
3) ▷ Unused: need to sync the PRG state
4) Set $B_1 \leftarrow \emptyset, \ldots, B_b \leftarrow \emptyset$
5) **for** $1 \le i \le bM$ **do** ▷ De-permute and gather into blocks
6) Parse $c_i = (e_i, \ell', \tau_i)$
7) **if** $\ell' \neq \ell$ or $\mathsf{VerMac}(k_{mac}, e_i \circ \ell, \tau_i) = 0$ **then**
8) Discard c_i and continue
9) Set $\sigma_i \circ i' \circ j' \leftarrow \mathsf{Dec}(k_{enc}, e_i)$
10) Set $B_{i'} \leftarrow B_{i'} \cup \{(\sigma_{j'}, j')\}$
11) **for** $1 \le i \le b$ **do**
12) Generate a seed s_j using G
13) Use (weak) PRG H in LT-Decode
14) Set $\pi_j = (1^\lambda, s_j, k, \delta, \mathcal{D}, \varepsilon)$
15) Set $m_i = (m_{i,1}, \ldots, m_{i,k}) \leftarrow \mathsf{LT\text{-}Decode}(\pi, B_i)$
16) **if** $m_i = \perp$ **then** Output $\perp$ and exit
17) Output $m = (m_{1,1}, \ldots, m_{b,k})$

Fig. 16. Decoder of scalable LT-coding scheme FalconSe

Input: 1^λ, keys k_{enc}, k_{mac}, seed s, nonce ℓ, symbols per block k, block decoding failure prob. δ, degree distr. $\mathcal{D}$, overhead per block ε, number of blocks b, authenticated codeword c
Output: message m or $\perp$

1) Set $s' \leftarrow f(s, \ell)$ ▷ f is a KDF
2) Seed the PRG G using s'
3) Generate b seeds $s_1, \ldots, s_b$ using G
4) Set $\pi_i = (1^\lambda, s_i, k, \delta, \mathcal{D}, \varepsilon)$ for $1 \le i \le b$
5) Initialize decoders $\mathsf{LT\text{-}Decode}_1(\pi_1), \ldots, \mathsf{LT\text{-}Decode}_b(\pi_b)$
6) ▷ $\mathsf{LT\text{-}Decode}_i$ decodes the i-th block
7) **for** each symbol c_i in c **do**
8) Parse $c_i = (e_i, \ell', \tau_i)$
9) **if** $\ell' \neq \ell$ or $\mathsf{VerMac}(k_{mac}, e_i \circ \ell, \tau_i) = 0$ **then**
10) Discard c_i and continue
11) Set $\sigma_i \circ i' \circ j' \leftarrow \mathsf{Dec}(k_{enc}, e_i)$
12) Update $\mathsf{LT\text{-}Decode}_{i'}$ with σ_i as its j'-th symbol
13) **for** $1 \le i \le b$ **do**
14) Set $(m_{i,1}, \ldots, m_{i,k})$ to be the output of $\mathsf{LT\text{-}Decode}_i$
15) If $(m_{i,1}, \ldots, m_{i,k}) = \perp$, then output $\perp$ and exit
16) ▷ If any block fails to decode, output nothing
17) Output $m = (m_{1,1}, \ldots, m_{b,k})$

Fig. 17. Decoder of randomized LT-coding scheme FalconR