Falcon Codes: Fast, Authenticated LT Codes (Or: Making Rapid Tornadoes Unstoppable)

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

James Kelley*
NetApp, Inc.
Waltham MA, USA
James.Kelley@netapp.com

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

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

ABSTRACT

We introduce *Falcon codes*, a class of *authenticated error correcting codes* that are based on *LT codes* and achieve the following properties, for the first time simultaneously: (1) with high probability, they can correct *adversarial* corruptions of an encoded message, and (2) they allow very efficient encoding and decoding times, even *linear* in the message length. Our design framework encompasses a large number of such coding schemes. Through judicious use of simple cryptographic tools at the core LT-coding level, Falcon codes lend themselves to secure extensions of any LT-based fountain code, in particular providing Raptor codes that achieve resilience to adversarial corruptions while maintaining their fast encoding/decoding times. Falcon codes also come in three variants, each offering different performance trade-offs. For instance, one variant works well with small input messages (100s of KB to 10s of MB), but two other variants are designed to handle much larger messages (several GB). We study Falcon codes in a novel *adversarial model for rateless codes* over computational (corrupting) channels and prove their security under standard assumptions. We analyze the performance of our new coding schemes through a prototype implementation of their Raptor-code extension and a thorough experimental study that demonstrates their high efficiency in practice.

Applied to data transmission, Falcon codes can provably protect Raptor codes against targeted-erasure attacks, which were recently shown by Lopes and Neves [Oakland, 2014] to cause decoding failures of RaptorQ—the most advanced, standardized (IETF RFC6330) rateless code used in practice. Applied to data storage, Falcon codes can provide significant efficiency gainings as drop-in replacements of Reed-Solomon codes; in particular, a 35% speed-up over the state-of-the-art PoR scheme by Shi et al. [CCS, 2013].

Categories and Subject Descriptors

E.4 [**Coding and Information Theory**]: *error control codes*; C.2.0 [**Communication Networks**]: General—*Security and protection*

General Terms

Algorithms, Reliability, Security, Theory, Verification

Keywords

Authenticated error correcting codes; secure coding schemes; LT codes; adversarial channel; Raptor codes; proofs of retrievability

1. INTRODUCTION

By increasing the reliability of computing systems that may experience loss or corruption of data at rest or in transit, due to unreliable storage units or channels, error correcting codes are a particularly useful tool that finds numerous applications in distributed systems and network security. 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,[1] are employed widely in secure protocol design. Although their asymptotic efficiency depends on the implementation, RS codes typically involve encoding costs that are quadratic in the message size k; thus in reality they tend to be costly.[2]

Several alternative codes have been proposed to overcome the quadratic overheads of RS codes. For instance, using layered encoding, Tornado codes [5] achieve encoding/decoding speeds that are 10^2 to 10^4 times faster than RS codes. At the fastest end of the range lie LT codes [25], which achieve $O(k \log k)$ encoding/decoding times and are very practical. Based on LT codes, Raptor codes [46] and Online codes [30] are the first (rateless) codes to achieve *linear* encoding/decoding times. Each of these LT-based codes is a *rateless* (or *fountain*) code that can generate a practically unbounded stream of output code symbols. But this great efficiency comes with a qualitative drawback: Fountain codes have been designed and

*Work performed while this author was at Brown University.

CCS'15, October 12 - 16, 2015, Denver, CO, USA

ACM ACM 978-1-4503-3832-5/15/10 ... $15.00
DOI: http://dx.doi.org/10.1145/2810103.2813728 .

[1] An RS code breaks up the input message into fixed-sized pieces, regards these pieces as coefficients of a polynomial p, and produces the output symbols by repeatedly evaluating p on different points.
[2] RS codes with input size k and output size $n = O(k)$ have $O(k^2)$ complexity in practice: Even if specific configurations can asymptotically achieve $O(k \log k)$ encoding time, in most practical cases encoding with polynomial evaluation in $O(kn)$ time is faster.

analyzed over a *random (erasure) channel* rather than an adversarial (corruption) channel, essentially able to tolerating only random symbol *erasures* and no (or very limited) symbol corruption. And even current standardized implementations of these codes are easy to attack by adopting malicious (nonrandom) corruption strategies.

Vulnerabilities to adversarial corruptions. LT codes employ a random sparse bipartite graph to map message symbols, in one partition, into code symbols, in the other partition, via simple XORing (see Figure 1). By design, this graph provides enough coverage among symbols of the two partitions so that a belief-propagation decoding algorithm can recover the input symbols (with a small, encoder-determined probability of failure) despite random symbol erasures. But this algorithm will also readily *propagate (and amplify) any error* in the message encoding into the recovered message. This by itself is a serious problem, because LT codes provide no mechanism for checking symbol integrity. Thus, an attacker can trivially inflict a decoding failure or cause decoding errors that result in an incorrect (or even maliciously selected) recovered message.

Even worse, an attacker can exploit the graph structure to (covertly) increase the likelihood of a decoding failure by inflicting only a few adversarial symbol erasures, in a class of attacks we term *targeted-erasure attacks*. Here, the attacker's goal is to maliciously select those symbol erasures that are more likely to cause decoding failure. For instance, one can selectively erase symbols of high-degree nodes in the encoding graph so that with high probability not all input symbols are sufficiently covered by the surviving code symbols. Unfortunately, such targeted-erasure attacks have been neglected by existing RFCs (e.g., [27, 28]) that describe Raptor/RaptorQ codes for object delivery over the Internet. Indeed, to increase the practicality of these codes, their encoding graph is either completely deterministic or easily predictable by anyone,[3] thus trivially enabling high-degree symbol (or other targeted) erasures. This raises a serious threat for real-life applications that (will) employ RaptorQ codes—the most advanced fountain code that is being adopted for protecting digital media broadcast, cellular networks, and satellite communications. Surprisingly, until very recently, such vulnerabilities had been inadequately addressed in the literature.

In the context of secure P2P storage, Krohn et al. [19] identified *distribution attacks* (similar to targeted-erasure attacks) against Raptor-encoded data, but their mitigation was left as an open problem. Recently, in the context of data transmission, Lopes and Neves identified [23] and successfully implemented [24] such an attack against RaptorQ codes, by relating code symbols to input symbols and deriving *data-independent* erasure patterns that can increase the likelihood of decoding failure by *orders of magnitude*. Lopes and Neves [24] also informally proposed a basic remediation strategy that aims to make it harder to discover input-output symbol associations by employing a cryptographically strong pseudorandom generator (PRG) for mapping code symbols to input nodes in the encoding graph. In section 7, we explain why this strategy alone is inadequate to provide a secure solution, and further justify the importance of (provable) secure rateless codes against fully adversarial corruptions, in terms of both motivation and applications.

In view of this inconvenient trade-off between practicality and security, in this paper we consider the following natural questions: Is it possible to have codes that are simultaneously strongly tolerant of 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?

Contributions. 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 *Falcon codes*, a class of *authenticated error correcting codes*—i.e., error correcting codes that employ cryptography to withstand adversarial symbol corruptions, typically (but not exclusively) by authenticating the integrity of the code symbols—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., [4, 8, 18, 29, 31]) are typically RS codes, thus lacking efficiency or being costly in practice, and existing linear-time coding schemes (e.g., [30, 46]) are fountain codes that withstand only random erasures.

We develop the first adversarial model for analyzing the security of fountain codes against computationally bounded adversaries (previous adversarial models studied only fixed-rate codes). We first introduce *private LT-coding schemes*, which model the abstraction of authenticated rateless codes that combine the structure of LT codes with secret-key cryptography. 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 (i.e., a wrong message or no message is recovered).[4] 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; i.e., security holds when any *adversarial (corruption) channel* is effectively reduced to the *random (erasure) channel*. Section 2 introduces technical background on relevant coding theory and cryptography, and section 3 details our new security model for rateless codes.

We then provide three constructions of authenticated LT codes, called *core Falcon codes*, which achieve this new security notion against adversarial corruptions while preserving the efficiency of normal LT codes. Our main scheme, Falcon, (see Figure 3) leverages a simple combination of a strong PRG (to randomize and protect the encoding graph), a semantically secure cipher (to encrypt symbols and hide the graph), and an unforgeable MAC (to verify symbols). By partitioning the input message into *blocks* and applying the main scheme in each block, we extend Falcon to get two scalable and highly optimized (but more elaborate in terms of parameterization and analysis) schemes: a fixed-rate code FalconS and its rateless extension FalconR, which can produce unlimited code symbols.

Moreover, our core Falcon codes can be readily extended 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. 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. Section 4 details our core Falcon codes and section 5 (together with appendices B, C, and D) provides their security analysis.

In section 6, we perform an extensive experimental evaluation of the Raptor-extensions of our core Falcon codes (since Raptor codes are perhaps the fastest linear-time fountain code at present), showing that, indeed, they achieve practical efficiency with low overhead. In particular, our schemes can achieve encoding and decoding speeds up to 300MB/s on a Core i5 processor, several times faster than

[3] For instance, each code symbol contains a list of the indices of its covering input symbols or a seed for a PRG to generate these indices. Encrypting symbols and the associated index lists is not sufficient, because the RFCs contain explicit tables of random numbers to be used in the encoding process.

[4] As we explain, we impose minimal restrictions on the adversary: Symbol corruptions and erasures can be arbitrary, but inducing trivial decoding failures by destroying (almost) all symbols is disallowed.

Table 1: Comparison of core Falcon codes Falcon, FalconS, and FalconR with RS and LT codes. For Falcon, the message length is k; for FalconS and FalconR, the message is partitioned into b blocks of k symbols each; for RS codes, the codeword length is n.

Construction	Rateless	Efficiency	Strong PRG	Weak PRG	Tolerance to Adv. 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
$\mathsf{LT}_{\mathsf{Enc\&MAC}}$	yes	$O(k \log k)$	no	yes	no

the standard RS encoding coupled with encryption and MACs. 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%. Table 1 compares our constructions with a standard RS code, as well as the naïvely "secure" LT code $\mathsf{LT}_{\mathsf{Enc\&MAC}}$, also coupled with encryption and MACs, according to the following criteria: rateless property, asymptotic efficiency of encoding/decoding (FalconR's performance depends on the number b of blocks; see section 4), the use of a strong or weak PRG (FalconS can safely employ a weak, but fast, PRG for increased efficiency), and the achieved security related to the tolerance against adversarial data corruption.

In section 7, we discuss two main applications of our Falcon codes. In the secure transmission domain, we show how Falcon codes can remove the vulnerabilities that RaptorQ codes (IETF RFC6330) are known to have with targeted-erasure attacks, as recently demonstrated by Lopes and Neves [23], and discuss why their suggested remediations [23] fail to provide a fully secure solution. In the secure storage domain, we show how Falcon codes can significantly improve the performance of existing systems that employ RS codes. We examine existing proof-of-retrievability (PoR) protocols, whose security is known to be highly related to secure data encoding against adversarial channels [4], and thus typically employ sophisticated codes. In particular, for the recent dynamic PoR protocol by Shi et al. [45], which currently holds the record in terms of efficiency by employing an elaborate and highly optimized FFT-based coding scheme, we show experimentally that even a simple drop-in replacement of their coding scheme by our Falcon codes results in notable cost savings—on the order of 35%, at least for most configurations of interest, e.g., with very large files. We also show how Falcon codes can be applied to the general PoR design framework by Bowers et al. [4] to gain efficiency and simplicity (e.g., to avoid striping). In section 8, we review related work in the overlap of codes and security and present conclusions in section 9.

2. PRELIMINARIES

In what follows, we let λ denote the security parameter and PPT denote probabilistic polynomial-time. We also let $[\mathsf{Alg}(\pi)]$ denote the set of all possible outputs of a PPT algorithm Alg, running on input parameters π, and $\tau \leftarrow \mathsf{Alg}(\pi)$ the output derived by a specific random execution of $\mathsf{Alg}(\pi)$. Analogously, we let $x \xleftarrow{R} S$ (or $x \xleftarrow{D} S$) 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 $|S|$ the cardinality of a set S.

Cryptographic tools. To withstand adversarial errors, our schemes employ simple cryptographic tools, namely message authentication codes (MACs), symmetric ciphers (SCs), and pseudorandom generators (PRGs), with which basic familiarity is assumed. We next overview these tools together with their main security properties.

Keyed by secret $sk \leftarrow \mathsf{Gen}_1(1^\lambda)$, an *existentially unforgeable* MAC produces a tag $t = \mathsf{Mac}(sk, m)$ for message m, used to verify the integrity of m as $\mathsf{VerMac}(sk, m, t) \stackrel{?}{=} 1$, where no PPT adversary $\mathcal{A}$ can compute a verifiable fake tag τ^* for a message m^* not explicitly tagged by $\mathsf{Mac}(sk, \cdot)$. Keyed by secret $sk \leftarrow \mathsf{Gen}_2(1^\lambda)$, a *semantically secure* SC is an encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$ with $\mathsf{Dec}(sk, \mathsf{Enc}(sk, m)) = m$ for any message m, where no PPT $\mathcal{A}$ can determine a randomly selected bit b when challenged with $c_b = \mathsf{Enc}(sk, m_b)$ for m_1, m_2 of its own choosing. Finally, given a short random seed $s \xleftarrow{R} \{0,1\}^\lambda$, a *strongly secure*, or strong, PRG produces a long sequence of random-looking bits, where no PPT $\mathcal{A}$ can distinguish the PRG output from a string of truly random bits. *Weak* PRGs refer to faster but insecure pseudorandom generators.

Block ECCs. An *error correcting code* (ECC) is a message encoding scheme that can tolerate some corruption of the encoded data by allowing 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*.

Defined over a fixed, finite set of symbols (or *alphabet*) Σ and parameterized by integers k (the message length) and $n \geq k$ (the block length), a *fixed-rate* or *block* ECC specifies mappings between *messages* in Σ^k and *codewords* in Σ^n. (Example alphabets include $\{0,1\}^l$, the set of all l-bit strings, or a finite field $\mathbb{F}$.) A block code can encode any given message $m \in \Sigma^k$ to a corresponding *valid* codeword $c \in \Sigma^n$, and can decode any *invalid* codeword $\hat{c}$, derived as a bounded distortion of c back to the original message m. For $m \in \Sigma^k$ and $c \in \Sigma^n$, the components m_i and c_i are called *message* and *code symbols*, respectively. If the first k symbols of a valid codeword correspond to exactly the k message symbols, the code is called *systematic*. The *rate* of a block code is the ratio $R = k/n$; it captures the amount of information transmitted per codeword and controls the recovery strength of the code as follows. The *(minimum) distance* of a block code is the minimum number of symbol changes needed to transform one valid codeword into another, measured across all valid codewords. Specifically, a code has distance d if the Hamming distance $\Delta(c, c') = |\{i \mid 1 \leq i \leq n, c_i \neq c'_i\}|$ between any two valid codewords $c, c' \in \Sigma^n$, $c \neq c'$, is at least d; such a code allows for decoding invalid codewords that are distorted by up to $\lfloor d/2 \rfloor$ errors back to a unique (original) message.[5] Typically, smaller values of R imply a larger minimum distance d.

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

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

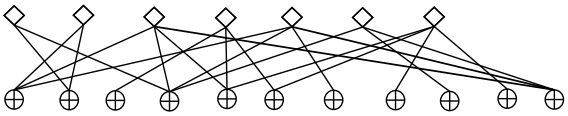


Figure 1: LT-encoding: Each code symbol (bottom) is the XOR of $O(\log k)$ randomly selected message symbols (top).

Rateless ECCs. Error correcting codes that employ no fixed block length n are called *rateless* or *fountain* codes. These codes can generate an unbounded stream of code symbols (i.e., a continuous "fountain"), thus lacking the notion of a code distance within which error correction is guaranteed. Instead, they typically operate over the *random erasure channel*, denoted by REC_p—where each code symbol is independently erased by REC_p with some fixed probability p—and only provide *probabilistic* message-recovery guarantees relative to this channel. (REC_p can be generalized with symbol-specific erasure probabilities p_j possibly dependent on code-symbol histories, but this does not affect our analyses.) Typically, a rateless code allows recovery of the original message, with probability at least $1-\delta$, from any subset of at least $(1+\varepsilon)k$ correct code symbols that are "sampled" (have survived erasure) by REC_p. Here, ε and δ measure the *overhead* and the *decoding failure probability* of the code, respectively, where smaller values of δ require larger values of ε and vice versa; we denote this relationship by $F(\delta, \varepsilon)$. Conveniently, the encoder needs no a priori knowledge of the value p.

DEFINITION 2. *A (k, δ, ε)-*rateless error correcting code *C over 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 elements of Σ^k to infinite sequences $\{c_i\}_{i=1}^{\infty}$, with $c_i \in \Sigma$, and for any $m \in \Sigma^k$ and any finite subsequence s of $\mathsf{Encode}(m)$ of length at least $(1+\varepsilon)k$ received over REC_p for some $p \in (0,1)$, $\mathsf{Decode}(s) = m$ with probability at least $1-\delta$.*

LT codes. Examples of rateless ECCs include LT codes [25] and their extensions, Raptor codes [46] and Online codes [30]. In LT codes, the encoding/decoding mapping takes a *degree distribution* $\mathcal{D}$ as an extra input parameter that is used to construct a sparse bipartite graph with 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 then this node's neighbors are selected uniformly at random among the message symbols. The code symbol of a given parity node is simply the XOR of the message symbols of the neighboring input nodes, and thus is very fast to compute. 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. (For notational simplicity, we leave this parameterization implicit.) Now, both δ and $\mathcal{D}$ determine the possible values of ε; we denote this relationship by $F(\delta, \mathcal{D}, \varepsilon)$.

Often, $\mathcal{D}$ instantiates to the *robust soliton* distribution [25], which ensures that the average node degree is $O(\log k)$. This implies that (1) encoding takes $O(k \log k)$ time; and (2) using a balls-in-bins analysis, with high probability, every message symbol is covered by at least one of the at least $(1+\varepsilon)k$ code symbols, and thus can be decoded in $O(k \log k)$ time, using a belief-propagation algorithm[6] to tolerate symbol erasures—but generally not symbol errors. Work on extending LT codes to withstand errors [26] has only studied channels of particular noise characteristics (e.g., additive white Gaussian or uniformly distributed noise) but not adversarial channels. LT codes were originally analyzed over the binary erasure channel [25], although this analysis easily generalizes to larger symbols.

Raptor codes. *Rap*id *tor*nado codes [46] use *precoding* of the input message $m \in \Sigma^k$ (before LT-encoding) to improve the performance of LT codes. 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 sufficiently many code symbols, each as the XOR of a random subset of the intermediate symbols, where each such subset (now drawn via a variant of the robust soliton distribution [46]) is of $O(1)$ size. Overall, Raptor codes feature $O(k)$ encoding/decoding time, i.e., high data encoding rates with low overhead. 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 are recovered by decoding the precode. Finally, based on LT codes, Raptor codes are essentially erasure codes, not tolerating symbol corruptions well. Analysis of Raptor codes over noisy channels [35, 40] is also restricted to specific random errors. IETF standard RFC5053 [27] and its amendment RFC6330 [28] suggest using a simple checksum (e.g., CRC32) to detect any random corruptions of code symbols: While possibly sufficient for small, random errors, this method will crumble quickly under malicious attacks.

[6]Starting by setting the values of message symbols that connect to degree-1 code symbols, the algorithm XORs newly set values to degree-ℓ neighboring code symbols, $\ell > 1$, removes all such propagation edges and iterates until no degree-1 code symbols remain.

3. SECURITY MODEL

Our main goal is to extend LT codes to endure *adversarial* corruptions inflicted by a PPT adversary. 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 secret-key cryptography to resist errors introduced by an adversarial (corruption) channel. We also introduce a corresponding new security notion that we call *computationally secure rateless encoding*. We use secret-key LT-coding schemes only for simplicity of analysis and presentation. Public-key versions can be readily defined and corresponding secure LT-coding schemes can be supported by employing public-key encryption and signatures, coupled (when needed) with a public-key key-agreement protocol (over a noiseless channel) for distribution of the PRG seed. Our security model is general enough to also capture security for block codes.

3.1 Private LT-coding Schemes

Prior work formalizing PPT adversaries acting against message encodings only considered block codes. In particular, the seminal work on *feasible channels* by Lipton [22], (α, β)*-networks* by Lysyanskaya et al. [29], *computationally bounded noise* by Micali et al. [31], and *adversarial codes* by Bowers et al. [4], model adversarial corruptions of a *bounded* constant fraction of the code symbols. Such modeling is intrinsically tied to block codes because it measures (and bounds) the number of changes an adversary $\mathcal{A}$ can perform to inflict decoding errors (or failures). By explicitly bounding the fraction of all code symbols that can be corrupted, however, existing security models cannot capture corruptions against rateless codes. Indeed, since fountain codes can produce an unbounded number of symbols, the rate of corruption introduced by $\mathcal{A}$ can continually grow *arbitrarily close* to 1, thus directly contradicting any bounded corruption rate.

A more accurate modeling of errors against rateless codes is to *lower bound* the amount of *non-corruption* (rather than upper bound the amount of corruption) as an absolute number, typically defined by the message length k (and not as a fraction of the code symbols). This ensures that a *minimum number* of "good" code symbols remain intact, allowing the remainder to be "bad." In block codes, an upper bound on badness implies a lower bound on goodness (and vice versa), but this symmetry breaks in rateless codes.

We thus define *private LT-coding schemes*, a class of "crypto-enabled" rateless codes that are based on LT codes. This class generically captures fountain codes that can produce an unbounded number of code symbols using LT-encoding over a set of "input" symbols (not necessarily the message symbols) using a proper degree distribution $\mathcal{D}$ (thus encompassing LT, Raptor, and Online codes).

DEFINITION 3. *A $(k, \delta, \mathcal{D}, \varepsilon)$-private LT-coding scheme over alphabet Σ, with 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 prob. δ, (4) degree distribution $\mathcal{D}$, (5) overhead ε, and (6) message $m \in \Sigma^k$, outputs an infinite sequence $\{c_i\}_{i=1}^{\infty}$ (with $c_i \in \Sigma$), called 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, and (6) a string $c \in \Sigma^*$, where $|c| \geq (1+\varepsilon)k$, outputs a string $m' \in \Sigma^k$ or fails 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 $1-\delta$, whenever c is a finite subsequence of $\mathsf{Encode}(sk, \ell, \delta, \mathcal{D}, \varepsilon, m)$, of length at least the decoding threshold $(1+\varepsilon)k$, received over REC_p for some $p \in (0,1)$.

Definition 3 can be viewed as an extension of the block private codes of Micali et al. [31] to LT-based rateless codes, but it is general enough to also capture two types of "crypto-enabled" block codes. First, any block 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, *block* versions of LT-coding schemes that simply produce codewords of *fixed size* (above the decoding threshold $(1+\varepsilon)k$) can also be captured: A $(k, n, \delta, \mathcal{D}, \varepsilon)$-*private block LT-coding scheme* is defined as a $(k, \delta, \mathcal{D}, \varepsilon)$-private LT-coding scheme where Encode, given an additional input parameter n, produces codewords of size $|c| = n \geq (1+\varepsilon)k$. 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 $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$, $\mathcal{LTS}_n = (\mathsf{Gen}, \mathsf{Encode}, \mathsf{Decode}, \pi, n)$ its n-symbol block version, and $\mathcal{LTS}_*$ any (rateless or block) LT-coding scheme.

As in existing works, in the context of data transmission, both the sender and the receiver are assumed to know the secret key sk, and nonces are used to prevent replay attacks. 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. Due to its dependence on $\mathcal{D}$, 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. (For Raptor codes, k is in the tens of thousands or greater, whereas smaller values of k (e.g., in the hundreds) require careful design [46].)

3.2 Security Definition

We define the security of private LT-coding schemes against PPT adversaries as tolerance against adversarial code symbol corruptions.

Adversarial channel. We thus consider a transmission channel that is fully controlled by a PPT adversary $\mathcal{A}$. That is, as new code symbols are produced by algorithm Encode of a private LT-coding scheme $\mathcal{LTS}$, $\mathcal{A}$ can maliciously corrupt any new or past such symbols. Moreover, $\mathcal{A}$ is allowed to *adaptively* interact with $\mathcal{LTS}$; that is, to examine (1) code symbols of its choice in the encoding

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

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 $\pi' = (s, \ell, \delta, \mathcal{D}, \varepsilon)$ and initialize oracle $\mathcal{O}_{m_i}$ to provide access to code symbols of $\mathsf{Encode}(\pi', m_i)$
   (d) $\mathcal{A}(\pi, \psi', m_i)^{\mathcal{O}_{m_i}(\cdot)} \rightarrow (\psi'', c_i)$ ▷ Codeword corruption where $c_i = (\sigma_1, \ldots, \sigma_{N_i})$
   (e) $\mathsf{Decode}(\pi', c_i) \rightarrow r_i$ ▷ Message recovery
   (f) Set $m_{\mathcal{R}} \leftarrow r_i$, $\psi \leftarrow \psi''$, $i \leftarrow i+1$
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 $\pi' = (s, \ell, \delta, \mathcal{D}, \varepsilon)$ and initialize oracle $\mathcal{O}_{m_a}$ to provide access to code symbols of $\mathsf{Encode}(\pi', m_a)$
10. $\mathcal{A}(\pi, \psi', m_a)^{\mathcal{O}_{m_a}(\cdot)} \rightarrow (\psi'', c_a)$
11. $\mathsf{Decode}(\pi', c_a) \rightarrow r_a$
12. If $\perp \neq r_a \neq m_a$ then output 1
13. If $r_a = \perp$ and $|c_a \cap Q_{m_a}| \geq (1+\varepsilon)k$ then output 1
14. Output 0

Figure 2: Security game for private LT-coding schemes.

(produced by Encode) of any message of its choice, and (2) the decoded message (produced by Decode) on any corrupted set of code symbols of its choice. Finally, $\mathcal{A}$ is *stateful*, remembering any past selected messages, their encodings, their chosen symbol corruptions and the corresponding recovered messages, and depending current actions on the full such past history. We call adversaries that do not keep such state for prior rounds of encoding/decoding *stateless*. (Note, however, that $\mathcal{A}$ is never given the bipartite graph underlying any of the LT-encodings in $\mathcal{LTS}$.)

Security game. We define security of a private LT-coding scheme $\mathcal{LTS}_* = (\mathsf{Gen}, \mathsf{Encode}, \mathsf{Decode}, \pi)$, $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$, through the game $\mathsf{Exp}_{\mathcal{A},\mathcal{LTS}_*}(\pi)$ of Figure 2, played between (the encoder Encode and decoder Decode of) $\mathcal{LTS}_*$ and an adversary $\mathcal{A}$.

For a stateful $\mathcal{A}$, the learning phase consists of a sequence of at most a polynomial number of rounds, where in the i-th round:

1. $\mathcal{A}$ selects a message m_i and Encode initializes itself with m_i;
2. $\mathcal{A}$ queries different symbols from the encoding of m_i through access to oracle $\mathcal{O}_{m_i}$ which, given as input an index j, returns the j-th code symbol produced by Encode (for block codes, if $j > n$ then $\mathcal{O}_{m_i}(j)$ outputs $\perp$);
3. $\mathcal{A}$ provides Decode a (corrupted) codeword c_i consisting of N_i symbols in $\Sigma \cup \{\perp\}$, where $\perp$ denotes symbol erasure, and Decode returns to $\mathcal{A}$ decoded message r_i.

The attack phase consists of a special final round where $\mathcal{A}$ selects an attack message m_a, queries code symbols of the encoding of m_a by querying oracle $\mathcal{O}_{m_a}(\cdot)$, and computes a corresponding corrupted codeword c_a that decodes into message r_a (by Decode).

For simplicity, we assume that $\mathcal{A}$ always provides at least $(1+\varepsilon)k$ code symbols, good or bad, to Decode (or else the latter trivially fails and outputs $\perp$). The adversary wins by either causing a *decoding failure*, when $r_a = \perp$, or by causing a *decoding error*, when $\perp \neq r_a \neq m_a$. However, for a decoding failure, we require that c_a contains at least $(1+\varepsilon)k$ intact code symbols to exclude trivial attacks. Thus, if Q_m denotes the set of symbols queried by $\mathcal{A}$ from $\mathcal{O}_m$, then (abusing notation) we require that $|c_a \cap Q_{m_a}| \geq (1+\varepsilon)k$.

Secure LT-coding schemes. In normal operation of $\mathcal{LTS}_*$ over the random erasure channel REC_p, the probability that Decode

succeeds in correctly decoding an input message is lower bounded by $1-\delta$. We want to ensure that no PPT adversary $\mathcal{A}$ can cause Decode to either (1) output an incorrect message; or (2) fail to even decode with probability significantly greater than δ.

We do this by defining computationally *secure rateless or block LT-coding schemes*, which ensure that any PPT $\mathcal{A}$ is only negligibly more likely to cause a decoding error or failure than an adversary that attacks codewords only with random erasures. We define security relative to REC_p rather than in absolute terms because an LT code itself reduces REC_p to a noiseless channel. Overall, an adversarial channel is reduced to a random one (by use of cryptography) which is further reduced to a noiseless one (by LT-coding).

We capture this property by comparing the winning advantage of $\mathcal{A}$ in our security game (just described) with the winning advantage of a *random adversary* $\mathcal{R}_p$ that interacts in a more restricted manner with $\mathcal{LTS}_*$ in the same security game. Specifically, $\mathcal{R}_p$ is parameterized by probability p (which augments π) and proceeds as follows, outputting symbols that are distributed identically to REC_p:

1. $\mathcal{R}_p$ directly chooses an attack message $m_a \in \Sigma^k$ (i.e., outputs $(\perp, m_a)$ in step 7);
2. $\mathcal{R}_p$ queries $\mathcal{O}_{m_a}$ sequentially for code symbols, erasing each new such symbol with probability p or otherwise adding it to (corrupted) codeword c_a;
3. $\mathcal{R}_p$ provides Decode codeword c_a of size at least $(1+\varepsilon)k$ symbols (or at least $(1-p)n$ symbols for block $\mathcal{LTS}_*$).

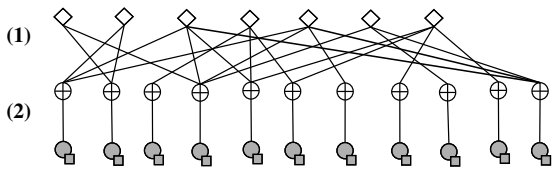
The output of $\mathcal{R}_p$ is distributed identically to REC_p.

To technically exclude extreme degenerate symbol-corruption patterns, we further restrict PPT adversaries $\mathcal{A}$ and $\mathcal{R}_p$ as follows. First, we disallow $\mathcal{A}$ from querying the oracle for symbols located arbitrarily far along in the stream of code symbols that can be produced by $\mathcal{LTS}_*$ on any given message, to avoid situations where $\mathcal{A}$ queries symbols that are infeasible for PPT Encode to produce sequentially or even index.[7] In our game, we call a query $\mathcal{O}_m(i)$ for the i-th symbol (τ,p)-*feasible* if $q_p(i) \geq \tau$, where $q_p(i) = \Pr[\mathsf{Encode}$ outputs at least i symbols over $REC_p]$; and, further, we call τ-*admissible for a given* p any $\mathcal{A}$ making only (τ,p)-feasible queries. Second, we restrict p so that $1-p$ is a non-negligible function of λ, calling such values of p *feasible*, to ensure that a non-negligible fraction of the code symbols will survive erasures by $\mathcal{R}_p$; thus transmission over REC_p is feasible. (For non-feasible p, $q_p(i)$ is non-negligible only for super-polynomially large values of i.) For a block code with message and block lengths k, n, we assume that $p \in (0, 1-k/n)$ and define p to be feasible if it is non-negligibly different from $1-k/n$. Finally, we call $\mathcal{A}$ *admissible* if it is τ-admissible for all feasible p and τ is non-negligible in λ.

Let $\mathcal{LTS}_*$ be a private (rateless or block) LT-coding scheme, let $\mathsf{Exp}_{\mathcal{R}_p,\mathcal{LTS}_*}(\pi,p)$ be the experiment in our security game run with random adversary $\mathcal{R}_p$ and a feasible erasure probability p, and let $\mathsf{Exp}_{\mathcal{A},\mathcal{LTS}_*}(\pi)$ be the same experiment run with a PPT admissible adversary $\mathcal{A}$. Let $\mathsf{Adv}_{\mathcal{A},\mathcal{LTS}_*}(\pi,p) = |\Pr[\mathsf{Exp}_{\mathcal{A},\mathcal{LTS}_*}(\pi)=1] - \Pr[\mathsf{Exp}_{\mathcal{R}_p,\mathcal{LTS}_*}(\pi,p)=1]|$ be $\mathcal{A}$'s advantage over $\mathcal{R}_p$.

DEFINITION 4. *We say that a private rateless (or block) LT-coding scheme $\mathcal{LTS}_*$ with parameters $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$ is* secure *if, for all PPT admissible $\mathcal{A}$ and for all feasible $p \in (0,1)$ (or $p \in (0, 1-k/n)$), $\mathsf{Adv}_{\mathcal{A},\mathcal{LTS}_*}(\pi,p)$ is negligible in λ.*

[7] If unrestricted, even a PPT $\mathcal{A}$ would be given unrealistically large power. For instance, if $\mathcal{LTS}_*$ employs a PRG with finite state, $\mathcal{A}$ could simply query symbols at multiples of the period of the PRG, thus getting identical code symbols (produced by the same random bits) and rendering decoding impossible.



(1) Apply LT code with a strong PRG **(2) Encrypt & MAC encoded symbols**

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 strong PRG G
2. Set $i \leftarrow 0$ and $\pi \leftarrow (1^\lambda, s', k, \delta, \mathcal{D}, \varepsilon, m)$
3. Initialize LT-Encode, G, Enc, Mac with π, s', k_{enc}, k_{mac}
4. **for** as long as required **do**
5. Set $\sigma_i \leftarrow$ LT-Encode(π, i) ▷ i-th LT code symbol σ_i
6. Set $e_i \leftarrow \mathsf{Enc}(k_{enc}, \sigma_i \circ i)$ ▷ Encrypt
7. Set $\tau_i \leftarrow \mathsf{Mac}(k_{mac}, e_i \circ \ell)$ ▷ MAC
8. Output $c_i \leftarrow e_i \circ \ell \circ \tau_i$
9. $i \leftarrow i+1$

Figure 3: Encoder of main LT-coding scheme Falcon.

4. CORE FALCON CODES

We present three variant private LT-coding schemes. They extend LT codes to achieve strong error-correction capabilities in our new adversarial model, while preserving their asymptotic efficiency. We refer to these three schemes as *core Falcon codes*, because they provide the design framework for devising corresponding secure extensions, resilient to malicious errors, of any other LT-based code.

Our first main scheme, Falcon, is a particularly simple crypto-enhanced extension of an LT code. Our second scheme, FalconS, is a *scalable* extension but also a *block* refinement of the main scheme based on data stripping and code scrambling [12]. Our third scheme, FalconR, is a *randomized rateless* extension of our scalable scheme. For brevity, we focus only on our schemes' encoders.

4.1 Main LT-Coding Scheme

In our main scheme, Falcon, we apply three cryptographic tools in a simple and rather intuitive manner during LT-encoding (Figure 3). First, we perform a key change in Encode compared to standard LT-encoding by using a strongly secure PRG to select the degree and the neighbors of each parity node. Also, a per-encoding nonce value is employed to prevent replays. During (or after) LT-coding, we encrypt each produced code symbol and compute a MAC tag for each such encrypted symbol and the nonce. (Alternatively, authenticated encryption with the nonce as additional authenticated data can be used.) The index i of each code symbol in the stream of symbols is also processed to help with reorder and deletion attacks, unless transmission occurs over a FIFO channel where the receiver learns about symbol erasures. The subroutine LT-Encode is initialized with parameters PRG seed s', message size k, decoding failure probability δ, degree distribution D, overhead ε, message m, and on input an index i, it outputs the i-th code symbol. Seed s' is derived from a master seed s (derived itself by secret key sk) and the nonce ℓ via a key-derivation function (KDF) f that produces long enough outputs to seed the strong PRG G. We denote this scheme as $(k,\delta,\mathcal{D},\varepsilon)$-Falcon.

This simple design satisfies our security definition. Intuitively, encryption, MACs, and the strong PRG work together to maintain the "goodput" of the channel by ensuring that each (intact) received symbol is just as "helpful" for decoding as when sent over a random erasure channel. First, the use of MACs ensures that symbol

corruptions are detected so that they can be discarded, a well-known technique for reducing errors to erasures by authenticating code symbols. We thus, informally, refer to our schemes as *authenticated LT codes*. However, although a necessary condition, symbol verification is not sufficient to achieve security: Input and parity symbols are interrelated through the underlying bipartite graph, so corruption of select parity symbols may seriously disrupt the recovery of the message. 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.[8] Encryption ensures that symbol contents (and any information about the graph structure contained therein) is concealed. Similarly, the strong PRG ensures that the adversary cannot exploit any biases or weaknesses in the PRG (e.g., output that is initially biased, such as RC4) to aid in guessing what the digraph structure may be. Overall, because the structure of the graph is unpredictable, an adversary will, intuitively, be unable to do better than random corruptions and erasures.

As a byproduct of our use of encryption, Falcon also provides message privacy. If the encoded data is encrypted by the channel (e.g., tunneling over SSH) or also authenticated by the channel (e.g., IPsec), then Falcon needs to use only a PRG and a MAC or only a PRG, greatly reducing the overhead of our main scheme. (However, "PRG-only" Falcon would be vulnerable to malicious proxies.)

Batching. Computing a MAC for each symbol, while conceptually simple, leads to large space and computational overheads when encoding with small symbols. Also, authenticating multiple symbols with a single MAC (as a single *batch*) is a reasonable implementation in many circumstances—e.g., when grouping symbols together to be sent in a single network packet. However, this results in corruption amplification during decoding (a single symbol corruption can cause many other possibly valid symbols to be discarded). However, in channels where errors are of low rate or are *bursty*, authenticating a batch of symbols would increase throughput because the cryptographic overhead is reduced. For example, on an Intel Core i5 processor, given a 2MB file with 4-byte symbols, a batch size of 100 increased the throughput by $\approx 23\%$, and a batch size of 50 for a 16MB file with 64-byte symbols increased throughput by $\approx 24\%$. (The exact amount saved depends on the input and symbol sizes.)

Batching also loosens the lower bound on the symbol size and instead requires that the batch is at least λ bits. Note that batching increases the decoding overhead ε because $(1+\varepsilon)k$ is not necessarily divisible by the batch size. With a batch of b symbols, the decoder must receive $(1+\varepsilon)k \leq \lceil (1+\varepsilon)k/b \rceil b \leq (1+\varepsilon)k+b-1$ symbols, giving an effective overhead of $\varepsilon \leq \varepsilon' \leq \varepsilon + \frac{b-1}{k}$. Because the presence or absence of batching does not affect our security analyses (except our block code, detailed next), we assume batches of size 1. Finally, appending MACs to parity symbols increases the space overhead per symbol. For both block and rateless codes, the number of raw bits transmitted increases by a factor of $(1 + m/s)$, where m is the MAC size and s is the symbol size. For instance, if $s = m$ (the minimum symbol size), then the size of an output symbol is doubled. If $m \ll s$, then this overhead is small (e.g., for $s = 1024$ bytes and $m = 16$ bytes, the overhead is $\approx 1.5\%$). Batching b symbols per MAC decreases overhead to $m/(bs + m)$.

4.2 Scalable Block LT-Coding Scheme

Although simple and efficient, our main scheme may suffer from scalability problems in certain cases. In particular, it can require a lot of memory when applied to large files. It must keep the entire input resident during encoding, and must maintain a very large graph, in addition to all of the partially recovered data during decoding.

To mitigate such limitations, we adopt a divide-and-conquer strategy on our main scheme toward designing a new *scalable* scheme, FalconS: We divide message symbols into blocks and encode each block independently, using Falcon. This immediately increases data locality during message encoding, generally providing better scalability, and further allows easy parallelization (see section 6). But such data stripping introduces a new security challenge: Depending on parameterization, an adversary may put its effort into corrupting parity symbols in a single block or few selected blocks, thus increasing its chances of causing a decoding failure, even if all block encoders have produced symbols beyond the LT-decoding threshold.

To defend against such attacks, we adopt Lipton's code scrambling technique [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 faster weak PRG when producing the Falcon-encoding of an individual block, 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—in fact, a useful byproduct of our new scheme (see section 7.2 for an application of FalconS to a PoR).

However, employing (rateless) Falcon within (block) FalconS, as just described, requires careful encoding parameterization. Consider applying a random permutation Π over all code symbols derived from all blocks and assume an adversarial *corruption rate* γ against our fixed-rate FalconS-encoding. Because a γ-fraction of the permuted symbols become corrupted, the number of corruptions per-block is binomially distributed with an average of a γ-fraction of the symbols. Then, to absorb any variance in per-block errors, we add to each block Falcon-encoding extra (beyond the LT-decoding threshold) redundancy, expressed by *tolerance rate* τ. (One must bound the probability that a block receives "too many" corruptions, or else block (and also total) decoding will fail; note that in-block symbol losses can be mitigated via erasure pre-coding of the input before its block partition, but this can only tolerate a certain number of block losses.) Then, given rate γ and scheme parameters π, we can approximate safe values (i.e., ensuring negligible decoding-failure probability) for rate τ using a Chernoff bound, and we can show that a safe τ is always $o(1)$ (see appendix A). Thus, encoding time is $O(k \log k)$ in Falcon and $O(bk(\log k + \log b))$ in FalconS with b blocks, since $O(bk)$ symbols are permuted.

FalconS comes in two versions. Detailed in Figure 4, FalconSe applies permutation Π *explicitly* over the final, encrypted and MACed, code symbols (of all blocks). (Note that a strong PRG G derives seeds for block encoding using weak PRG H, and Falcon encoding integrates block indices into code symbols to mitigate symbol reordering attacks.) Alternatively, FalconSi applies Π *implicitly*: First, Π is generated and then parity symbols are produced by the appropriate block Falcon-encoder *in the order of their final position* (after Π would have been applied). The implicit permutation provides an interesting trade-off in performance, because it allows outputting parity symbols in a streaming manner and avoids bufferring code symbols until they are *all* generated in order for Π to be applied (which reduces the memory required). But it also introduces some overheads related to preprocessing of the randomness and keeping extra state for Π and Falcon-encoders. Here, being able to easily and quickly reset the PRG state of each encoder (e.g., as in Salsa20 [2]) is desirable, so that we may access the needed pseudorandom bits in the stream that would have been used in a sequential encoding.

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

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, tolerance rate τ, number of blocks b, message m
Output: Authenticated codeword c

1. Set $s' \leftarrow f(s, \ell)$ ▷ f is a KDF; use s' to seed strong PRG G
2. Set $M \leftarrow \frac{1}{1-(1+\tau)\gamma}(1+\varepsilon)k$ ▷ Parameterization
3. Generate permutation Π of $b * M$ elements using G seeded by s'
4. Partition m into blocks $m_1, \ldots, m_b$
5. **for** $1 \le i \le b$ **do**
6. Generate a seed s_i using G; set $\pi \leftarrow (1^\lambda, s_i, k, \delta, \mathcal{D}, \varepsilon, M)$
7. Init. LT-Encode, weak PRG H, Enc, Mac w/ $\pi, s_i, k_{enc}, k_{mac}$
8. Set $(\sigma_{i,1}, \ldots, \sigma_{i,M}) \leftarrow \mathsf{LT\text{-}Encode}(\pi, m_i)$
9. **for** $1 \le j \le M$ **do**
10. Set $e_{i,j} \leftarrow \mathsf{Enc}(k_{enc}, \sigma_{i,j} \circ i \circ j)$
11. Set $\tau_{i,j} \leftarrow \mathsf{Mac}(k_{mac}, e_{i,j} \circ \ell)$
12. Set $c_{i,j} \leftarrow e_{i,j} \circ \ell \circ \tau_{i,j}$
13. Map each $c_{i,j}$ to position $\Pi(i*M+j)$ to get $c' = (c'_1, \ldots, c'_{bM})$
14. Output $c' = (c'_1, \ldots, c'_{bM})$

Figure 4: Encoder of scalable LT-coding scheme FalconSe.

Otherwise, all such bits should be pre-computed for degree and neighbor selection of parity nodes; however, storing pre-computed bits can still be much more efficient than storing final symbols, especially in storage applications with a large symbol size, e.g., 1KB. When batching is employed, due to corruption amplification from corrupted MACs, we must apply batching *after* the symbol permutation in order to restrict the adversary to only corrupt a γ-fraction of batches. For b blocks, corruption and tolerance rates γ, τ and parameters as discussed earlier, we denote our schemes as $(k, \delta, \mathcal{D}, \varepsilon, \gamma, \tau, b)$-FalconS, where FalconS can be either FalconSe or FalconSi.

Decoding failure. Suppose that an overall decoding failure probability of δ is needed for FalconS. Decoding fails when any of the individual blocks fails to decode because either fewer than $(1+\varepsilon)k$ intact symbols survived corruption in a block; or the code symbols in a block did not cover all of its input symbols. The probability of the first event can be made negligible via the tolerance rate τ; the second event is intrinsic to the schemes themselves. Note that in the latter case, the failures are *independent* and so are the successes. Therefore the probability of decoding success is $(1-\delta')^b = 1-\delta$; thus $\delta' = 1 - \sqrt[b]{1-\delta}$. For instance, for $b = 100$ and $\delta = 0.05$, we have $\delta' = 1 - \sqrt[100]{1-0.05} \approx 0.000512$. The smaller value of δ' implies that the overhead ε' of each block is increased.

4.3 Randomized Scalable LT-Coding Scheme

Although our block-oriented code, FalconS, is more scalable than Falcon (e.g., with a speedup of over a 50% for inputs of 32MB—see section 6), it is inherently no longer a rateless code, because explicitly/implicitly permuting all code symbols across all blocks requires knowing their total number. We present an alternative approach, still block-oriented and thus scalable, that yet allows rateless encoding.

Detailed in Figure 5, our new scheme, FalconR, breaks the input up into blocks and then applies Falcon-encoding to each block, as before; but now final code symbols are produced by employing Falcon "in parallel" and in a *randomized* manner. Specifically, for each block i an independent instance of Falcon is initialized with strong PRG G (seeded by s_i). Then, another strong PRG G' (seeded by s') is used to iteratively select a block at random whose corresponding Falcon-encoder simply outputs its next symbol. Importantly, this process is repeated (at least) until *every* block encoding has reached the required decoding threshold. 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

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; use s' to seed strong PRG G'
2. Generate seeds $s_1, \ldots, s_b$ of strong PRG G using G' seeded by s'
3. Divide m into b blocks $m_1, \ldots, m_b$
4. For $1 \le i \le b$, set $\pi_i \leftarrow (1^\lambda, s_i, k, \delta, \mathcal{D}, \varepsilon, m_i)$ and initialize $\mathsf{LT\text{-}Encode}_i$, strong PRG G, Enc, Mac with π, s_i, k_{enc}, k_{mac}
5. **for** as long as required **do**
6. Sample a block index i using G'
7. Let σ be a new code symbol of $\mathsf{LT\text{-}Encode}_i$ using G
8. Set $e \leftarrow \mathsf{Enc}(k_{enc}, \sigma \circ i \circ j)$, $\tau \leftarrow \mathsf{Mac}(k_{mac}, e \circ \ell)$
9. Output $e \circ \ell \circ \tau$

Figure 5: Encoder of randomized LT-coding scheme FalconR.

symbol deletion and reordering attacks. For b blocks and parameters, as before, we denote our scheme as $(k, \delta, \mathcal{D}, \varepsilon, b)$-FalconR.

Note the subtle difference between codes FalconSi and FalconR. In FalconSi, code symbols in a block are produced in a random order *induced by the permutation* Π, whereas in FalconR they 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 although FalconS can use a weak PRG when encoding an individual block, here each instance of Falcon must use a strong PRG.

Efficiency. The asymptotic efficiency of FalconR 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, asking how many balls must be thrown into b bins to get at least m balls in each bin. It is known that for a sufficiently large number of blocks b, the expected number of balls thrown is $O(b \log b + b(m-1)\log\log b) + O(b)$ [33]. Thus, on average, the overhead in encoding/decoding a file is a $O(\log \log b)$ factor. The same analysis applies when batching of symbols is used, except each encoder outputs a batch at a time. However, for small values of b (e.g., $b = 10$), since k must be sufficiently large (and therefore 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, because we would like this scheme to scale up to very large files (e.g., 10s of gigabytes or more). Suppose that 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). Because 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.

5. SECURITY ANALYSES

Our various constructions of Falcon codes seek to reduce an adversarial corrupting channel to a random erasure channel (REC). Through such a reduction, our LT-coding schemes inherit many of the properties of the original LT codes (e.g., the overhead ε and failure probability δ). In this section we provide proof outlines of reductions to REC for our core Falcon code schemes. For simplicity

and clarity, we use asymptotic security statements, rather than exact security statements that quantify an adversary's $\mathcal{A}$ resources. Full proofs are available in appendices B, C, and D.

The adversary $\mathcal{A}$ can win the game Exp, described in section 3, by causing a decoding error (Decode outputs an incorrect message) or a decoding failure (Decode outputs $\perp$). Because 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 our 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 use memoryless channels: The proofs extend in a straightforward way to stateful channels by simply utilizing the nonces. Recall that, as stated in section 3, we only consider *admissible* adversaries $\mathcal{A}$ that are compared to random erasure channels REC_p 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 λ is the security parameter. For all sufficiently large k and for all PPT $\mathcal{A}$, the probability that $\mathcal{A}$ wins $\mathsf{Exp}_{\mathcal{A},\mathcal{LTS}}$ 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 this contradicts the unforgeability of the MAC (except with negligible probability).

LEMMA 2. *Let $C = (\mathsf{Gen}_2, \mathsf{Enc}, \mathsf{Dec})$ be a semantically secure symmetric cipher and G a strong PRG used to encrypt the symbols and randomize the encoding of a $(k, \delta, \mathcal{D}, \varepsilon)$-*Falcon *$\mathcal{LTS}$, where $F(\delta, \mathcal{D}, \varepsilon)$ holds, and λ is the security parameter. For all sufficiently large k and for all PPT admissible $\mathcal{A}$, the probability that $\mathcal{A}$ wins $\mathsf{Exp}_{\mathcal{A},\mathcal{LTS}}$ via a decoding failure is negligibly different from δ in λ.*

Here, we reduce security against decoding failure to the security of the strong 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 the same information about m possessed by $\mathcal{A}$, outputs a set of indices of symbols to be erased, such that Decode fails to decode with probability $\approx \mathcal{A}$'s advantage over δ. Because 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, the probability of $\mathcal{A}'$ causing a decoding failure is exactly δ. Finally, the security of the PRG allows us to use pseudorandom bits instead of truly random bits while only giving $\mathcal{A}$ a negligible additional advantage. Thus, we have reduced $\mathcal{A}$ to REC. The following theorem summarizes the security of our scheme, which follows directly from Lemmas 1 and 2.

THEOREM 1. *Let $\mathcal{LTS}$ be a $(k, \delta, \mathcal{D}, \varepsilon)$-*Falcon *code that uses an existentially unforgeable MAC M, a semantically secure symmetric cipher C, and a strong PRG G, and let $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$. Then, for all sufficiently large k, for all PPT admissible $\mathcal{A}$, and for all feasible p, $\mathcal{A}$'s advantage $\mathsf{Adv}_{A,\mathcal{LTS}}(\pi, p)$ is negligible in λ.*

The above theorem and lemmas apply to *nonsystematic* 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 *nonrandom* against these symbols. That is, $\mathcal{A}$ has some a priori knowledge of the underlying encoding graph, and can adjust its strategy accordingly. Although we believe that our construction is secure in this case, we leave proving its security to future work.

For our scalable block FalconS codes (with both implicit and explicit permutations), the security of the scheme follows from the results of Lipton's work on scrambled codes [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) rather than erasing with probability γ. Recall that each block has $N = \frac{1}{1-(1+\tau)\gamma}(1+\varepsilon)k$ symbols with a corruption-tolerance parameter of τ.

THEOREM 2. *Let $\mathcal{LTS}_{bN}$ be a $(k, \delta, \mathcal{D}, \varepsilon, \gamma, \tau, b)$-*FalconS *code that divides the input into b blocks, generates N symbols per block using an existentially unforgeable MAC M and a semantically secure symmetric cipher C, and permutes code symbols using a strong PRG G; and let $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$. Then, for all sufficiently large k and for all PPT $\mathcal{A}$ that corrupt up to a γ-fraction of symbols, $\mathcal{A}$'s advantage $\mathsf{Adv}_{\mathcal{A},\mathcal{LTS}_{bN}}(\pi, \gamma)$ is negligible in λ.*

Finally, for our scalable randomized Falcon codes, we use 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 Theorem 3. This result, however, does not follow directly from Lipton's work as Theorem 2, 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—e.g., $\mathcal{A}$ 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 divides the input into b blocks and uses an existentially unforgeable MAC M, a semantically secure symmetric cipher C, a strong PRG G, and a secure $(k, \delta, \mathcal{D}, \varepsilon)$-*Falcon *code $\mathcal{LTS}$ to generate code symbols; and let $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$. Then, for all sufficiently large k, for all PPT admissible $\mathcal{A}$, and for all feasible p, $\mathcal{A}$'s advantage $\mathsf{Adv}_{A,\mathcal{LTS}_R}(\pi, p)$ is negligible in λ.*

6. EXPERIMENTS

This section details an experimental evaluation that demonstrates the practicality of Falcon codes. The experiments were run on two machines, one with "abundant" resources and a more pedestrian, consumer-grade machine, to measure the speed of our schemes in both "ideal" and "typical" configurations. The majority of the experiments were run 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. Benchmarks were also run on a 2.6GHz, Intel Core i5 processor with 12GB 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 later in this section) used pthreads—and were compiled with the '-O2' optimization flag. We use the Salsa20 stream cipher [2] as strong PRG, SFMT as weak PRG [42], and PBKDF2 for key derivation [15]. The Jerasure v2.0 library [41] was used for the Reed-Solomon erasure code.

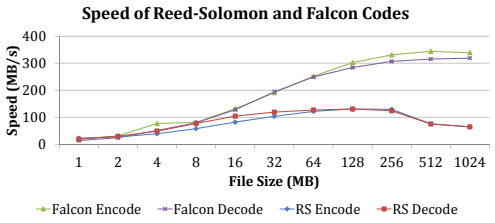


Figure 6: Falcon **vs. Reed-Solomon: Encoding throughput.**

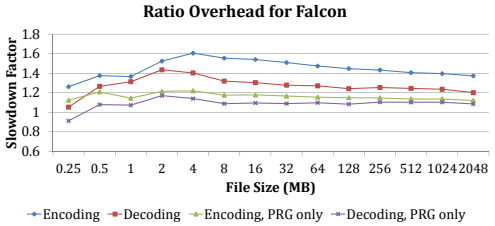


Figure 7: Falcon **vs. insecure Raptor: Slowdown factor.**

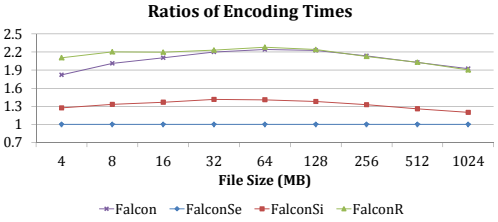


Figure 8: Falcon **codes: Relative encoding speeds.**

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 larger than 2KB in size and AES-GCM to be faster for smaller symbols.[9] 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 with a batch size of 1.

We benchmark Raptor-extensions of Falcon codes: We prepend an erasure-coding step to any instantiation of our Falcon-encoder in schemes Falcon, FalconS, and FalconR (where an erasure code is applied to the input data) to get corresponding secure Raptor codes. Raptor codes are among the most efficient erasure codes available, and we show in this section that our Raptor-based Falcon codes themselves achieve high efficiency. Our implementations of both Falcon codes and (standard) Raptor codes are based on the `libwireless` code written by Jonathan Perry [37]. We used an LDPC-Triangle code as the precoding step [48]. Unless noted otherwise, all encoding and decoding was performed with 1KB symbols and by adding 25% redundancy; the numbers given are an average of 10 trials.

Main scheme. Figure 6 compares Falcon scheme to Reed-Solomon (RS) codes, the de facto standard for encoding a file to withstand adversarial corruption, in terms of encoding/decoding speeds on the Core i5 for various file sizes. 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 used a MAC-authenticated systematic erasure code with $k = 204$ and $n = 255$ (for 20% redundancy) over $GF(2^8)$ using striping to increase both encoding and decoding speeds.[10] This optimized configuration makes it possible to quickly detect and discard any tampered symbols and correct up to $n - k$ errors, the theoretical maximum. Clearly, our scheme can achieve high throughput, reaching over 300MB/s for both encoding and decoding, and is several times faster than the RS encoder. (The RS code's slowdown for large files is due to an increased miss-rate in the L3 cache, going from $\approx 5.8\%$ to $\approx 11.8\%$ when moving from 256MB input to 512MB input, as measured by the `cachegrind` tool of `valgrind` [32].) Note that for inputs larger than 16MB, the throughput of Falcon can saturate a 1Gb network link.

Figure 7 compares Falcon against a standard, insecure Raptor code where no encryption, MAC, or strong PRG is used. The numbers shown are the average of 50 trials. Our main scheme is generally slower than the insecure Raptor by a factor between 1.25 and 1.6; this slowdown is usually less than 1.5. Note that the use of the strong PRG results in a slowdown of approximately 10-15% (shown in the lower two lines). For larger files, the cryptographic slowdown declines as the LT and precode encoding take a larger percentage of the total time. The "increase" in speed for decoding a 256KB file when using the strong PRG instead of a weak one is due to environmental noise and the very short encoding times (the encoding times of these two differ by only 20 to 30 μs).

Abundant resources. Figures 8 and 9 show that, indeed, FalconSe and FalconSi are more scalable than Falcon. Note that FalconR performs worse than Falcon on smaller inputs and almost identically on larger inputs. Falcon is slower than FalconSe and FalconSi primarily because it uses a strong PRG in the LT-encoding and it has worse locality of reference: When performing the LT-encoding, 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 for both code and data, because 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 because it (1) undermines locality by switching between encoders, (2) uses a strong PRG in the LT-encoding, in contrast to the FalconS schemes, and (3) generates more symbols than all other schemes. However, FalconR does have lower memory requirements than FalconSe because it does not need to buffer any symbols before outputting them.

Figure 10 validates our last argument (item (3) in the previous paragraph), showing the growth in the total number of symbols generated by FalconR, as compared to FalconSe, for encoding a 2GB file of 4KB symbols. FalconSe generates exactly the number of symbols required for a given overhead parameter ε. In contrast, FalconR requires generation of extra symbols to ensure that each of the b encoded blocks contains the minimum number of symbols m required for proper block decoding (plus any additional redundancy that is desired). Recall that the expected number of generated symbols is $O(bm)$ for smaller b, but increases to $O(b \log b + mb \log \log b)$ for larger b. This is evident in the graph; for $b < 40$ the number of symbols is close to optimal but increases with b.

On the same input used in Figure 10, Figure 11 compares FalconR to FalconSe in terms of speed: As the number of blocks increases, the block size decreases, allowing individual blocks to be encoded faster and FalconSe to achieve increased throughput. In contrast, the speed of FalconR decreases, because it must generate more symbols.

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

[10] Instead of performing large-symbol field operations, striping involves dividing the file into symbols over a much smaller field (while using the same k and n), encoding it in small "stripes" (batches), and grouping small symbols together to produce large output symbols.

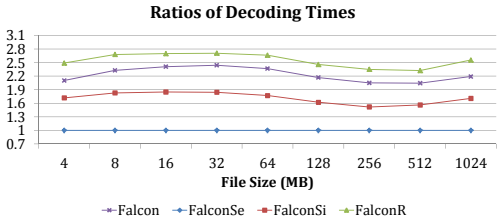


Figure 9: Falcon **codes: Relative decoding speeds.**

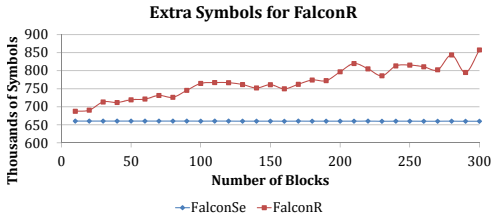


Figure 10: FalconSe **vs.** FalconR**: # generated code symbols.**

Parallelism. Falcon codes allow some level of parallelism and can take advantage of multiple CPU processing cores. Our scalable schemes are trivially parallelizable at the block level and Falcon at the symbol level: Once the degree and neighbors of a given symbol are computed, it can be encoded independently of the other symbols.

Figure 12 shows the results of running a multi-threaded version of FalconSe on the Opteron machine, using a 2GB file as input with various numbers of blocks and threads. In each case, the performance increases until there are 8 threads, and then it levels off. This plateau is due to the AES-GCM encryption in the OpenSSL library reaching its peak throughput for the machine. In particular, we measured the performance of AES-GCM on the Opteron 6282SE for 1024 byte inputs and found a peak throughput of approximately 620MB/s, including key setup. The cost of the remainder of the encoding process (e.g., XORing symbols, allocating and deallocating memory, etc.) accounts for the difference between the measured throughput of 500-600MB/s and the theoretical maximum of 620MB/s. The slow decline in performance as the number of threads increases is at least partly due to increased contention for the data caches. In particular, the fraction of data references that miss the last level cache increases from 19% to 23.8% when increasing from 8 to 32 threads, as measured by Linux's `perf` utility, averaging over 20 runs with a standard deviation between 0.1% and 0.7%, depending on the number of threads. (Usually, it was around 0.3%.)

Overall, this shows that the scalable Falcon encoders can, indeed, achieve very high performance using multiple threads and are limited primarily by the efficiency of the cryptography used.

7. APPLICATIONS

Our goal is to design fast, authenticated (rateless) codes that offer strong tolerance to adversarial symbol corruptions while achieving practical encoding rates. The problem lies at the intersection of security and coding theory and is motivated by a security vs. efficiency trade-off observed in two wide application areas:

- **Data transmission systems:** For fast data recovery against lossy channels, several data transmission systems employ linear-time rateless codes, thus being very practical but protecting data in transit only against random symbol erasures.
- **Distributed/cloud storage systems:** For better fault-tolerance guarantees against server failures or security guarantees against malicious cloud providers, several storage systems employ (adversarial) error correcting codes, which are typically RS codes, thus protecting against malicious corruptions of data at rest but often being less practical.

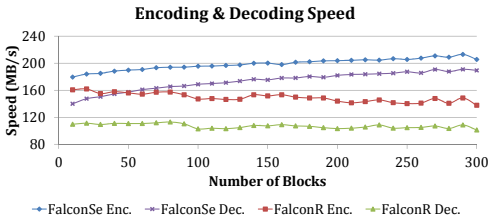


Figure 11: FalconSe **vs.** FalconR**: Encoding & decoding speeds.**

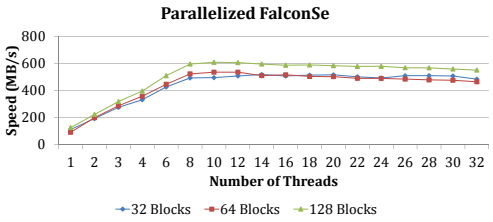


Figure 12: Throughput of parallelized FalconSe**.**

7.1 Falcon Applied in RaptorQ

Lopes and Neves [24] illustrate the practicality of a simple targeted-erasure attack against RaptorQ codes. They show *data-independent* attacks on encoded data such that the probability of decoding failure is increased by several orders of magnitude. The attack is as follows. RaptorQ codes include unique identifiers with each symbol to allow easy reconstruction of the encoding graph by the decoder. Each message symbol is associated to an "Internal Symbol ID" (ISI) label and each parity symbol to an "Encoded Symbol ID" (ESI) label. (RaptorQ is a systematic code, so some ISI and ESI labels will coincide.) RaptorQ codes generate a parity symbol's degree and associated neighbors by feeding the ISI and the message length k into a "tuple generator," which uses a (very) weak PRG (using a fixed table of random values). The attack leverages the fact that given the ESI it is possible to derive the ISI and thereby determine how a given parity symbol was encoded. Then a brute-force algorithm is used to find minimal erasure patterns for selectively deleting symbols to ensure (with some probability) that decoding fails.

Lopes and Neves [24] also informally suggest (without any correctness or security analysis) some possible mitigations, based on an observation that simply using a strong PRG for the encoding is sufficient to protect against malicious erasures. Their suggested mitigations are to: (M1) Encrypt and permute the symbols; (M2) Give each ISI a random ESI and then randomly "interleave" the symbols; or (M3) Give each ESI a random ISI (and do not permute). Although probably sufficient to mitigate the specific performed attack [24], these suggested strategies seem insufficient to provide a viable and secure solution in the design of authenticated LT codes, because they: (R1) Rely on a random permutation (or random interleaving) that undermines ratelessness; (R2) Only hide the input to the weak PRG (tuple generator), but leave the PRG itself unmodified, whose predicatable output can still be exploited to derive the encoding graph structure; and (R3) Do not protect against *data-dependent* attacks (i.e., inferring graph structure from message contents).

We next show that it is possible for an attacker to look at the symbols to infer the encoding graph structure and again enact the targeted-erasure attack. Although difficult in general, the low-degree symbols can be readily identified for structured data or attacker-known input messages. Symbols of degree 1 are trivial to identify in transit and, with a known input message, the attacker can take all pairs of message symbols and compute all degree 2 symbols. Then

with this knowledge, the attacker can perform three different attacks: (A1) Identify the low-degree symbols and delete all others; with high probability these low-degree symbols will not provide enough coverage of the input symbols and decoding will fail; (A2) If the receiver is known to use a belief-propagation decoder, the attacker only needs to delete the symbols of degree 1 to prevent decoding from even starting;[11] (A3) If any of the input symbols is distinguished (e.g., the only one with the most significant bit set), then the code symbols containing that input symbol can be trivially identified and erased to guarantee decoding failure. Thus, we must provide additional protections (i.e., encryption) to prevent these data-dependent attacks. Overall, given the many subtleties in such complicated structures, a formal proof of security is required for any suggested solution.

Falcon codes directly protect against all the previously discussed attacks. Also, Falcon codes can be applied to secure against corruptions in any application where Raptor codes are used, including 3GPP, MBMS, streaming media [1], IP TV, IP Datacast over DVB-H [10], or satellite communications [9].

7.2 Falcon Applied in a PoR

Falcon codes can be used in the dynamic proof-of-retrievability (PoR) scheme given by Shi et al. [45]. In this scheme, updates to data are maintained in a hierarchical log structure where each level is twice the size of the level below it. New blocks are written to the lowest level; when l consecutive levels become full, they are combined together to produce the $(l+1)$-th level and levels $1, \ldots, l$ are emptied. To encode each level of the hierarchy, the scheme utilizes an ECC that is structured in a butterfly network and employs the Fast Fourier Transform to optimize code symbol generation. This code allows to *reuse* the work done to encode levels $1, \ldots, l$ to help encode level $l+1$, greatly increasing efficiency. We term this optimized code by Shi et al. [45], the FFT encoder.

In the PoR scheme by Shi et al. [45], we can remove the FFT encoder, and substitute a Falcon encoder with only small changes. In the original scheme, whenever a level becomes full, it is combined with the lower levels to produce an encoding for the next level up: The incrementalism of the FFT encoder allows this step to happen quickly. Falcon codes are not incremental and so during these rebuilds, each level must be separately decoded, concatenated together, and then re-encoded to form the upper-most level. Although this may seem costly, for large files our Falcon is fast enough to produce an overall speed-up (see Tables 2 and 3). Moreover, the butterfly network in the FFT encoder precludes level-based parallelism: Before level i can be encoded, level $i-1$ must be fully encoded, creating an implicit serialization of the level encodings. Using Falcon, however, allows us to encode each of the levels independently and in parallel.

Table 2: Rebuild time (in sec) using Falcon vs. FFT encoder.

File size	Falcon	FFT encoder
128MB	0.96	0.92
256MB	1.59	1.83
512MB	2.83	3.67
1024MB	5.04	7.45
2048MB	9.53	14.77

Table 2 shows the time taken to rebuild the hierarchical log structure in the PoR scheme by Shi et al. [45]: Using Falcon with level-based parallelism, instead of the FFT encoder, gives a 32% and 35% speed up for 1GB and 2GB input sizes, respectively. The file sizes where Falcon codes are faster are well matched to typical usage of PoR protocols for reliable outsourced storage, where bulk data and large files are stored and periodically checked (e.g., high-volume archival or backup data).

[11] A decoder solving the system of linear equations that describe the encoding graph via Gaussian elimination would likely still succeed.

Table 3: Falcon vs. FFT encoder: Coding throughput (in MB/s).

File size	Encode	Decode	Decode pre-comp.	FFT `Mix`
512KB	13.89	8.61	14.12	277.6
1MB	26.74	16.58	26.25	291.37
4MB	63.69	51.22	72.86	285.40
16MB	157.48	147.19	179.17	291.75
64MB	296.16	271.30	294.52	287.25
256MB	378.42	343.39	351.79	287.45
512MB	398.32	358.14	362.63	286.95

Table 3 compares the encoding/decoding speeds achieved by Falcon and the FFT encoder by Shi et al. [45]. Specificially, we time the `Mix` function of the FFT encoder, denoted as FFT `Mix`, which allows an incremental encoding of the data. The speed given is for a *single pass* of the function over the data and does *not* include the cost of encoding the entire data (while the speeds for Falcon do). Although FFT `Mix` achieves high speeds for all input sizes, Falcon scales better to larger inputs; e.g., Falcon is $\approx 39\%$ faster than FFT `Mix` for a 512MB file. This crossover in performance comes from the fact that we hold the number of symbols constant across all input sizes, which gives a fixed cost for the encoder and decoder setup times. Also, the creation and initialization of our decoder can be precomputed before any of the rebuilds must take place, which gives a good boost to the throughput for smaller inputs (see 4th column).

The PoR code was written in C#, compiled using Visual Studio 2013, and run on Windows 7, because this setup gave the highest speed for the FFT encoder. (Running the encoder using Mono in Linux resulted in an 18% slowdown on all inputs.)

Applying FalconS in a PoR. Bowers et al. [4] describe a general framework for constructing PoRs. The framework includes two phases: (1) a challenge-response phase where the client sends challenges to a (possibly malicious) server to ensure (with high probability) that at most an ϵ-fraction of the input has been corrupted; and (2) an extract phase that, given an adversary that corrupts an ϵ-fraction of the file, executes a series of challenges and responses that allow the client to recover the original file. Accordingly, the framework derives PoR schemes that use two layers of ECCs, an outer code applied to the original file, and an inner code used in the challenge-response phase. The inner code ensures that the adversary corrupts at most an ϵ-fraction of the file (else, this excess corruption is detected), while the outer code ensures that the client can correct up to an ϵ-fraction of file corruption. The outer code must be what is termed an *adversarial error correcting code* (AECC) [4], which intuitively operates by turning a computationally bounded adversary into a random one (as we also do): Namely, an $[n, k]$ ECC is said to be a (β, δ)-bounded AECC if the probability that the adversary can produce two valid codewords that are at most βn apart is at most δ.

Falcon codes reduce adversarial corruptions to random erasures but do not precisely fit into the definition of an AECC [4]. Specifically, Falcon codes are rateless codes rather than block codes and so there is no fixed n. However, if we take the FalconS variant of Falcon codes, then we have that a $(k, \delta, \mathcal{D}, \varepsilon, \gamma, \tau, b)$-FalconS code F is a $(1, bN\varepsilon_{mac})$-bounded AECC, where ε_{mac} is the probability of forging a MAC for the MAC scheme used in F and $N = \frac{1}{1-(1+\tau)\gamma}(1+\varepsilon)k$.

Thus, Falcon codes can be used in the framework by Bowers et al. [4] as secure outer codes to simplify (e.g., avoid striping) and enhance (e.g., get faster speeds) known RS-based PoR schemes [4].

8. PREVIOUS WORK

Computationally bounded channels. Lipton [22] introduces the study of PPT adversarial channels, proposing code scrambling to transform any code with success probability q over the binary symmetric channel into a code that is resilient to adversarial errors. Langberg [21] designs private codes that employ a secret key to reduce an adversarial channel into a random one (with no use of cryptography). Lysyanskaya et al. [29] study secure data transmission through adversarial (α, β)-networks and provide a cryptographic ECC construction that transforms RS list-decoding into unique decoding. Micali et al. [31] define adversarial channels as stateful adversaries and provide provably secure codes (similar to [29]). Similar techniques to protect against adversarial channels appear in [12, 13, 26, 47]. However, all such works use block, not rateless, codes.

Secure LT codes. Krohn et al. [19] provide constructions for online verification (via homomorphic hashing) of data that is erasure-encoded by rateless codes (e.g., Raptor or Online codes) for content distribution in P2P networks. The possibility of targeted-erasure attacks, called "distribution attacks," is observed and left as future work. Recently, Lopes and Neves [24, 23] demonstrated the practicality of targeted-erasure attacks against RaptorQ codes [28], by devising data-independent erasure patterns that can cause δ to increase by several orders of magnitude, and suggested some mitigations which have certain drawbacks (discussed in section 7).

Codes and cryptography. There are several examples of schemes that combine cryptography and ECCs. Krawczyk [18] combines hashing, encryption, and erasure codes to implement computational secret sharing (an application of distributed fingerprints [17]). Also, Perry et al. [38, 39] 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 (e.g., [7, 34]) and efficient schemes for multicast authentication, such as list-decodable authenticated RS codes [29] and their LT-extensions [49], secure erasure codes [36], and distillation codes [16].

Proofs of Retrievability. Introduced by Juels and Kaliski [14], PoR protocols employ cryptography and ECCs to check the availability of outsourced data. PoR protocols have been studied widely, e.g., using homomorphic authenticators [44] or studying security against mobile adversaries [3]. Bowers et al. [4] present a general design framework for PoR protocols, which can provide improved variants of existing schemes [14, 44], and a new notion of adversarial ECCs (along with a concrete adversarial RS code that was independently developed by Curtmola et al. [8]), which, however, cannot model secure rateless coding schemes. Applying Raptor codes to the audit phase of a PoR protocol is known to improve efficiency [43]. The state-of-the-art PoR scheme by Shi et al. [45] uses a hierarchical-log structure combining erasure codes, Merkle trees and MACs to support efficient updates.

Secure storage. Cao et al. [6] use LT codes and homomorphic MACs in a storage system that allows efficient encoding/decoding, as well as repair of the data if any servers are lost. Targeted-erasure attacks on LT codes are noted, but the proposal to check whether all subsets of size k out of n code symbols can be successfully decoded, and re-encode the whole file if not, is not viable in practice.

9. 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 encoding/decoding for Falcon Raptor codes. Their efficiency makes them a useful general-purpose security tool for many practical applications such as secure data transmission and secure data storage over malicious channels. We identified two important applications of Falcon codes, for protecting RaptorQ codes against known attacks and for improving the efficiency of the currently best-known PoR protocol.

Acknowledgments

Research supported in part by the National Science Foundation under grants CNS–1012060 and CNS–1228485. We thank Charalampos Papamanthou and Elaine Shi for providing help and feedback on the application of Falcon codes to their PoR scheme [45], as well as reference code for use in benchmarks. We also thank the anonymous reviewers for their many insightful comments and helpful feedback.

10. REFERENCES

[1] 3GPP. Multimedia Broadcast/Multimedia Service (MBMS); Protocols and codecs. Technical Report ETSI TS 26.346, 3GPP, Sep. 2014. v12.3.0, Rel. 12.

[2] D. Bernstein. The Salsa20 family of stream ciphers. In *New Stream Cipher Designs*. 2008.

[3] K. D. Bowers, A. Juels, and A. Oprea. HAIL: A High-Availability and Integrity Layer for Cloud Storage. In *CCS*, 2009.

[4] K. D. Bowers, A. Juels, and A. Oprea. Proofs of Retrievability: Theory and Implementation. In *CCSW*, 2009.

[5] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A Digital Fountain Approach to Reliable Distribution of Bulk Data. *SIGCOMM*, 28(4), October 1998.

[6] N. Cao, S. Yu, Z. Yang, W. Lou, and Y. T. Hou. LT Codes-based Secure and Reliable Cloud Storage Service. In *INFOCOM*, March 2012.

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

[8] R. Curtmola, O. Khan, and R. Burns. Robust Remote Data Checking. In *Proc. of StorageSS '08*, 2008.

[9] Digital Video Broadcasting Project. Interaction channel for satellite distribution systems. Technical Report ETSI EN 301 790, ETSI, May 2009. v1.5.1.

[10] Digital Video Broadcasting Project. IP Datacast over DVB-H. Technical Report ETSI TS 102 591-1, ETSI, Feb. 2010. v1.3.1.

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

[12] P. Gopalan, R. J. Lipton, and Y. Z. Ding. Error correction against computationally bounded adversaries, April 2004. Unpublished manuscript.

[13] V. Guruswami and A. Smith. Codes for Computationally Simple Channels: Explicit Constructions with Optimal Rate. In *FOCS*, 2010.

[14] A. Juels and B. S. Kaliski, Jr. PoRs: Proofs of Retrievability for Large Files. In *CCS*, 2007.

[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. D. Tygar. Distillation Codes and Applications to DoS Resistant Multicast Authentication. In *NDSS*, 2004.

[17] H. Krawczyk. Distributed Fingerprints and Secure Information Dispersal. In *PODC*, 1993.
[18] H. Krawczyk. Secret Sharing Made Short. In *CRYPTO*. 1994.
[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.
[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.
[22] R. J. Lipton. A new approach to information theory. In *STACS*. 1994.
[23] J. Lopes and N. Neves. Robustness of the RaptorQ FEC Code Under Malicious Attacks. In *Inforum*, 2013.
[24] J. Lopes and N. Neves. Stopping a Rapid Tornado with a Puff. In *IEEE Sym. on Sec. and Priv.*, 2014.
[25] M. Luby. LT Codes. In *FOCS*, 2002.
[26] M. Luby and M. Mitzenmacher. Verification-Based Decoding for Packet-Based Low-Density Parity-Check Codes. *IEEE Trans. on Info. Theory*, 51(1), 2005.
[27] M. Luby, A. Shokrollahi, M. Watson, and T. Stockhammer. Raptor Forward Error Correction Scheme for Object Delivery. IETF RFC5053, October 2007.
[28] M. Luby, A. Shokrollahi, M. Watson, T. Stockhammer, and L. Minder. RaptorQ Forward Error Correction Scheme for Object Delivery. IETF RFC6330, August 2011.
[29] A. Lysyanskaya, R. Tamassia, and N. Triandopoulos. Multicast authentication in fully adversarial networks. In *IEEE Sym. on Sec. and Priv.*, 2004.
[30] P. Maymounkov. Online codes. Technical report, Secure Computer Systems Group, NYU, 2002.
[31] S. Micali, C. Peikert, M. Sudan, and D. A. Wilson. Optimal Error Correction Against Computationally Bounded Noise. *IEEE Tran. on Info. Theory*, 56(11), 2010.
[32] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *PLDI*, 2007.
[33] D. J. Newman and L. Shepp. The double dixie cup problem. *American Mathematical Monthly*, 67(1), 1960.
[34] R. Ostrovsky, O. Pandey, and A. Sahai. Private Locally Decodable Codes. In *ICALP*. 2007.
[35] R. Palanki and J.S. Yedidia. Rateless codes on noisy channels. In *ISIT*, July 2004.
[36] A. Pannetrat and R. Molva. Efficient Multicast Packet Authentication. In *NDSS*, 2003.
[37] J. Perry. libwireless and wireless-python. `www.yonch.com/wireless`, (2014/05/06).
[38] J. Perry, H. Balakrishnan, and D. Shah. Rateless Spinal Codes. In *HotNets-X*, 2011.
[39] J. Perry, P. A. Iannucci, K. E. Fleming, H. Balakrishnan, and D. Shah. Spinal Codes. *SIGCOMM*, 42(4), 2012.
[40] H. Pishro-Nik and F. Fekri. On raptor codes. In *IEEE Int. Conf. on Comm.*, volume 3, June 2006.
[41] J. S. Plank and K. M. Greenan. Jerasure: A Library in C Facilitating Erasure Coding for Storage Applications – Version 2.0. Technical Report UT-EECS-14-721, U. of Tenn., 2014.
[42] M. Saito and M. Matsumoto. SIMD-Oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator. In *MCQMCM*. 2006.
[43] S. Sarkar and R. Safavi-Naini. Proofs of Retrievability via Fountain Code. In *FPS*, 2012.
[44] H. Shacham and B. Waters. Compact Proofs of Retrievability. In *ASIACRYPT*, 2008.
[45] E. Shi, E. Stefanov, and C. Papamanthou. Practical Dynamic Proofs of Retrievability. In *CCS*, 2013.
[46] A. Shokrollahi. Raptor codes. *IEEE/ACM Trans. on Networking*, 14(SI), June 2006.
[47] A. Smith. Scrambling Adversarial Errors Using Few Random Bits, Optimal Information Reconciliation, and Better Private Codes. In *SODA*, 2007.
[48] STMicroelectonics and INIRA. LDPC FEC Codes, 2006.
[49] C. Tartary and H. Wang. Rateless Codes for the Multicast Stream Authentication Problem. In *Adv. in Info. and Comp. Sec.* 2006.

APPENDIX

A. ANALYSIS 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. Therefore, 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 want to minimize τ while ensuring that the probability that we exceed the error-correction capacity of a block is negligible in λ.

Suppose that there are b blocks, where each block has k input symbols and is encoded into m output symbols, giving $n = bm$ total symbols. Suppose that a γ-fraction of symbols are corrupted. If we encode a block so that it can tolerate $(1+\tau)\gamma m$ corruptions (where 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 that 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 want 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 [11], but the extra redundancy added requires a careful analysis. We show next that the tolerance parameter τ is $o(1)$, and so we main-

tain $O(k \log k)$ encoding and decoding. Recall the Chernoff bound, 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.

B. PROOF OF THEOREM 1

Here we present the full, formal proof for Theorem 1, starting with the proof for the first lemma.

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 λ is the security parameter. For all sufficiently large k and for all PPT $\mathcal{A}$, the probability that $\mathcal{A}$ wins $\mathsf{Exp}_{\mathcal{A},\mathcal{LTS}}$ via a decoding error is negligible in λ.*

PROOF. First, suppose that the adversary $\mathcal{A}$ that causes a decoding error, with $\mathcal{A}$ running in time T, making q queries in the learning phase (each of which is composed of at most n symbols), and succeeding with advantage ϵ. Since $\mathcal{A}$ seeks to cause a decoding *error*, it will not make corruptions in an attempt to cause a decoding *failure*. So the probability that decoding fails is still δ and the probability of a decoding error is ϵ. We will construct an algorithm $\mathcal{A}'$ that breaks M, runs in polynomial time—making at most $(q+1)n$ MAC oracle queries—and succeeds with probability at least $\frac{\epsilon}{n}$ (and at most ϵ).

$\mathcal{A}'$ is given access to a MAC oracle $\mathcal{O}_{mac}$ and simulates the Exp game, using $\mathcal{A}$ as a subroutine. Given an $\mathcal{LTS}$ code with parameters $k, \varepsilon, \mathcal{D}, \delta$ and security parameter 1^λ, on input 1^λ $\mathcal{A}'$ runs the Exp with parameters $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$. For $\mathcal{A}'$ to produce a successful forgery, we require that none of its queries to $\mathcal{O}_{mac}$ are used as the final message output by $\mathcal{A}'$.

To simulate $\mathsf{Exp}(\pi)$, $\mathcal{A}'$ uses $\mathcal{O}_{mac}$ to create the MACs for the code symbols when simulating Encode. To simulate Decode, $\mathcal{A}'$ tracks the symbol-MACs pairs from the queries from $\mathcal{A}$ and uses this list to filter out the symbols output by $\mathcal{A}$ that were either not previously queried or that have a MAC that is different from the result given by $\mathcal{O}_{mac}$ (i.e., the symbols that contain forged MACs). After q rounds in the learning phase, $\mathcal{A}$ outputs m_a for the attack phase. After simulating Encode and giving the result to $\mathcal{A}$, the algorithm $\mathcal{A}$ outputs the corrupted code symbols $(\sigma_1, \ldots, \sigma_{n'})$ (for some $(1+\varepsilon)k \leq n' \leq n$).

We require that the output of $\mathcal{A}'$ was not queried to $\mathcal{O}_{mac}$ and so can exclude those symbol-MAC pairs that match the Encode oracle queries. But there are at most n bad symbols to choose from. $\mathcal{A}'$ selects one symbol-MAC pair from the remaining pairs at random and outputs it as the attempted forgery. Since $\mathcal{A}$ succeeds with advantage at most ϵ, $\mathcal{A}'$ succeeds with probability at least $\frac{\epsilon}{n}$. Thus, $\mathcal{A}'$ runs in time $T + O(qn)$, making at most $(q+1)n$ queries to $\mathcal{O}_{mac}$ (with T and q polynomially bounded), and succeeds with advantage $\frac{\epsilon}{n} \leq \epsilon' \leq \epsilon$. Since M is existentially unforgeable, it must be that ϵ' is negligible; hence, ϵ is negligible. □

Now we will prove the second lemma; namely, that the probability of causing a decoding failure is negligibly more than for a random channel. This is proven by reducing security against (additional) decoding failure to the security of the PRG and the semantic security of the cipher. For simplicity, we use a memoryless channel in this proof. The proof extends in a straightforward way to stateful channels by simply using nonces.

LEMMA 2. *Let $C = (\mathsf{Gen}_2, \mathsf{Enc}, \mathsf{Dec})$ be a semantically secure symmetric cipher and G a strong PRG used to encrypt the symbols and randomize the encoding of a $(k, \delta, \mathcal{D}, \varepsilon)$-*Falcon *$\mathcal{LTS}$, where $F(\delta, \mathcal{D}, \varepsilon)$ holds, and λ is the security parameter. For all sufficiently large k and for all PPT admissible $\mathcal{A}$, the probability that $\mathcal{A}$ wins $\mathsf{Exp}_{\mathcal{A},\mathcal{LTS}}$ via a decoding failure is negligibly different from δ in λ.*

PROOF. Suppose that we have an adversary $\mathcal{A}$ that causes a decoding failure with probability μ. Initially, assume that $\mathcal{A}$ only erases symbols instead of corrupting them (we will remove this assumption later). Let $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$ be the parameters for a Falcon coding scheme $\mathcal{LTS}$. Define $h(m)$ (the "history" of m) to be $h(m) = (\pi, \psi', m)$.

Recall that $\mathcal{O}_m$ is an encoding oracle for $\mathcal{LTS}$ initialized to encode message m, where on input i, $\mathcal{O}_m$ outputs the i-th code symbol for the encoding of m. $\mathcal{O}_m$ is reinitialized with a new m chosen by $\mathcal{A}$ at each round. $\mathcal{A}$ takes as input π, ψ', runs in time t, and is given access to $\mathcal{O}_m$. After interacting with $\mathcal{O}_m$, $\mathcal{A}$ outputs $(\psi'', \sigma'_1, \ldots, \sigma'_n)$, where $\sigma'_i \in \Sigma \cup \{\perp\}$ (Σ is the code alphabet and $\perp$ is an erasure).

Define $\mathcal{B}$ such that, on input (π, ψ', m) and access to $\mathcal{O}_m$, it outputs $(i_1, \ldots, i_{n'})$, where $n' \leq n - (1+\varepsilon)k$, n is the total number of symbols requested from $\mathcal{O}_m$ by $\mathcal{A}$, and each i_j is the index of an erased symbol. Note that we can construct $\mathcal{B}$ by simply using $\mathcal{A}$ as a subroutine and outputting the indices of all $\perp$ symbols. We define $\mathcal{B}$ to be *successful* if, after erasing the symbols corresponding to the i_j's in the encoding of m, and giving the result to Decode, decoding fails. Note that $\mathcal{B}$ succeeds exactly when $\mathcal{A}$ does and runs in time $t_{\mathcal{B}} = t + O(n)$.

Since we are using a semantically secure cipher C, there exists $\mathcal{B}'_1$ such that on input $h(m)$ and *without* access to the first code symbol, outputs $(i_1, \ldots, i_{n'})$, where n' and each i_j are as before. Let γ denote the difference in the advantage of $\mathcal{B}'_1$ and $\mathcal{B}$. Note that $\mathcal{B}'_1$ runs in time $t_{\mathcal{B}} + o$ where o is the "overhead" in running time needed by $\mathcal{B}'_1$ to compute without access to the first code symbol (o is polynomially bounded). Now define $\mathcal{B}'_i$ such that it does not have access to the first i symbols of the encoding of m. By induction, we have that $\mathcal{B}'_i$'s probability of success is at most $i\gamma$ different from $\mathcal{B}$ and its running time is at most $t_{\mathcal{B}} + io$.

Note that $\mathcal{B}'_n$ succeeds with probability at most $n\gamma$ different from μ without seeing the encoding of m at all. This implies that $\mathcal{B}'_n$ can be run and select which symbols to erase *independently* of the encoding of m. Hence, after applying the output of $\mathcal{B}'_n$ to c, the remaining unerased symbols form a *random graph* over the symbols of m, where the degree distribution for the uncorrupted symbols is identical to the distribution used to encode m. Thus we have that, given $(1+\varepsilon)k$ uncorrupted symbols, the probability of decoding failure is at most δ. This implies that $|\mu - \delta| \leq n\gamma$, and since C is semantically secure, the advantage of $\mathcal{A}$ in winning the game is at most negligible.

This discussion assumes that the randomness used to encode m is true randomness. The actual definition of Encode uses the pseudorandom generator G. By definition of a PRG, for any PPT algorithm Alg running in time t_G,

$$|P[x \leftarrow U_\lambda; \mathsf{Alg}(1^\lambda, x) = 1] - P[s \leftarrow U_\lambda; x \leftarrow G(1^\lambda, s); \mathsf{Alg}(1^\lambda, x) = 1]| = \gamma'$$

where U_λ is the uniform distribution over strings of length λ and γ' is negligible. Thus, by replacing true randomness in Encode with the output of G, the advantage of $\mathcal{A}$ increases by at most γ'. Therefore the advantage of $\mathcal{A}$ in causing a decoding failure is $\gamma n + \gamma'$, which is negligible.

Finally, if we allow $\mathcal{A}$ to corrupt symbols instead of just erasing, then the result still holds. That is, corrupting gives $\mathcal{A}$ no additional advantage. To see this, note that $\mathcal{A}$ must output at least $(1+\varepsilon)k$ uncorrupted symbols for a decoding failure to count as a win in Exp. If any corrupted symbols are accepted by Decode (i.e., a MAC is successfully forged), then their acceptance *cannot* increase the probability of decoding failure. Rather, forgeries will *decrease* the probability of decoding failure by giving the LT-decoder more code symbols to use in decoding and thereby increase the coverage of message symbols by code symbols. This directly undermines the effort to cause a decoding failure. Therefore it is to $\mathcal{A}$'s advantage to only erase symbols. □

THEOREM 1. *Let $\mathcal{LTS}$ be a $(k, \delta, \mathcal{D}, \varepsilon)$-*Falcon *code that uses an existentially unforgeable MAC M, a semantically secure symmetric cipher C, and a strong PRG G, and let $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$. Then, for all sufficiently large k, for all PPT admissible $\mathcal{A}$, and for all feasible p, $\mathcal{A}$'s advantage $\mathsf{Adv}_{A,\mathcal{LTS}}(\pi, p)$ is negligible in λ.*

PROOF. Suppose not, then there exists a PPT adversary $\mathcal{A}$ such that $\mathsf{Adv}_{\mathcal{LTS},\mathcal{A}}(1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$ is non-negligible in λ and k; call this advantage γ. $\mathcal{A}$ can win the game Exp by causing a decoding failure (Decode outputs $\perp$) or a decoding error (Decode outputs the incorrect message). Because these are mutually exclusive events, we can consider them separately. By Lemma 1 and Lemma 2 we know that each event happens with negligible advantage. Therefore γ is negligible. □

C. PROOF OF THEOREM 2

Recall that the adversary may corrupt a γ-fraction of the encoded data, each block has $N = \frac{1}{1-(1+\tau)\gamma}(1+\varepsilon)k$ symbols, and the corruption-tolerance parameter is τ.

THEOREM 2. *Let $\mathcal{LTS}_{bN}$ be a $(k, \delta, \mathcal{D}, \varepsilon, \gamma, \tau, b)$-*FalconS *code that divides the input into b blocks, generates N symbols per block using an existentially unforgeable MAC M and a semantically secure symmetric cipher C, and permutes code symbols using a strong PRG G; and let $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$. Then, for all sufficiently large k and for all PPT $\mathcal{A}$ that corrupt up to a γ-fraction of symbols, $\mathcal{A}$'s advantage $\mathsf{Adv}_{\mathcal{A},\mathcal{LTS}_{bN}}(\pi, \gamma)$ is negligible in λ.*

PROOF. As shown in [22], the random permutation applied to the code symbols combined with perfectly secure encryption reduces $\mathcal{A}$ to a random channel. The use of G to produce the pseudorandomness to generate the permutation gives $\mathcal{A}$ at most a negligible advantage. Similarly, the semantic security of C hides all but a negligible amount of information in the code symbols and gives $\mathcal{A}$ at most a negligible advantage over a random erasure channel. Finally, the existential unforgeability of M ensures that $\mathcal{A}$ can produce a decoding error with at most negligible probability. □

D. PROOF OF THEOREM 3

Here we prove the security of Theorem 3. Recall that FalconR encodes multiple blocks in parallel, using Falcon to encode individual blocks.

THEOREM 3. *Let $\mathcal{LTS}_R$ be a $(k, \delta, \mathcal{D}, \varepsilon, b)$-*FalconR *code that divides the input into b blocks and uses an existentially unforgeable MAC M, a semantically secure symmetric cipher C, a strong PRG G, and a secure $(k, \delta, \mathcal{D}, \varepsilon)$-*Falcon *code $\mathcal{LTS}$ to generate code symbols; and let $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$. Then, for all sufficiently large k, for all PPT admissible $\mathcal{A}$, and for all feasible p, $\mathcal{A}$'s advantage $\mathsf{Adv}_{A,\mathcal{LTS}_R}(\pi, p)$ is negligible in λ.*

PROOF. First, we assume that any PPT $\mathcal{A}$ cannot distinguish between MACs and ciphertexts produced by different keys. That is, any PPT $\mathcal{A}$ cannot tell that $\mathsf{Mac}(k_1, m_1)$ and $\mathsf{Mac}(k_2, m_2)$ were produced using different keys, and similarly for ciphertexts. Although this is a limitation on the MACs and ciphers used, any symmetric cipher cipher or MAC used in practice will satisfy this requirement.

Suppose that we have a PPT $\mathcal{A}$ that can win $\mathsf{Exp}_{\mathcal{LTS}_R,\mathcal{A}}$ with non-negligible advantage. Then we will construct $\mathcal{A}'$, which will break $\mathcal{LTS}$ with non-negligible advantage. First, assume that G produces perfect randomness. Construct $\mathcal{A}'$ as follows. $\mathcal{A}'$ generates keys k_{enc} and k_{mac} and runs $\mathcal{A}$ as a subroutine with parameters $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon, b)$. $\mathcal{A}'$ then simulates the FalconR encoder by instantiating b separate instances of $\mathcal{LTS}$ and encoding each query of $\mathcal{A}$ appropriately.

Eventually, $\mathcal{A}$ outputs its challenge message m. $\mathcal{A}'$ selects one of the blocks at random (call it B) and then queries it to the Falcon oracle $\mathcal{O}_B$. (Note, $\mathcal{A}'$ skips the query phase entirely.) $\mathcal{A}'$ then initializes the encoders for the other blocks as before. $\mathcal{A}'$ then responds to $\mathcal{A}$'s symbol requests by either replying with a symbol from one of its encoders or, if the symbol is in B, sends it to $\mathcal{O}_B$. Eventually $\mathcal{A}$ outputs the code symbols $(\sigma_1, \ldots, \sigma_n)$ for the challenge message. $\mathcal{A}'$ selects the symbols that are for B, call these $(\sigma'_1, \ldots, \sigma'_{n'})$, where $n' \leq n$, and outputs only those as its corruption of B. Note that, with high probability, $n' \geq (1+\varepsilon)k$; call this probability p. If $n' < (1+\varepsilon)k$, then $\mathcal{A}'$ fails.

If $\mathcal{A}$ succeeds, then at least one of the blocks in the decoded message will either fail to decode or have a decoding error. If $\mathcal{A}'$ guessed B correctly, then $\mathcal{A}'$ succeeds as well. Therefore, if $\mathcal{A}$ has advantage ϵ in succeeding, then $\mathcal{A}'$ succeeds with advantage at least $p\epsilon/b$ and at most ϵ. Since we assume that Falcon is secure, ϵ is negligible and so is the advantage of $\mathcal{A}'$. If we change G to use pseudorandomness instead, then the advantages of $\mathcal{A}$ and $\mathcal{A}'$ increase by only a negligible amount. □