Securing Bulk Content Almost for Free

John Byers, Jeffrey Considine, Gene Itkis, Mei Chin Cheng, and Alex Yeung
Computer Science Department, Boston University
Boston, MA 02215, USA*

Abstract

Content providers often consider the costs of security to be greater than the losses they might incur without it; many view "casual piracy" as their main concern. Our goal is to provide a low cost defense against such attacks while maintaining rigorous security guarantees.

Our defense is integrated with and leverages efficient forward error correcting codes, such as Tornado codes, which are widely used to facilitate reliable delivery of rich content. We tune one such family of codes – while preserving their original desirable properties – to guarantee that *none* of the original content can be recovered whenever a key subset of encoded packets is missing. Ultimately we encrypt only these key codewords (only 4% of all transmitted packets), making the security overhead negligible.

1 Introduction

The cost of securing bulk content for delivery is often viewed as prohibitively expensive. Many content providers prefer to sidestep security issues in favor of optimizing transfer speed and client capacity so as to maximize performance to the paying customers to which content providers must cater. We propose a new method of securing content that significantly lowers the costs of security for both the server and its clients while still maintaining rigorous security guarantees.

Our techniques are leverage well known properties of certain fast forward error correction codes (FEC) such as LT codes [14] and online codes [16]. These FEC codes are designed to tolerate packet loss, and thus referred to as erasure-resilient codes. Erasure-resilient codes can be used to facilitate the delivery of bulk content, particularly in multicast scenarios, where they can be used to implement the digital fountain paradigm [6].

One property common to these codes is that nearly all of the input data is not decoded until nearly all of the encoded data is received. This behavior is similar to that of all-or-nothing transforms [22], and by making minor modifications to the codes and encrypting about 4% of all transmissions[1], we ensure that an adversary intercepting the remaining plain-text transmissions cannot determine *any* information about a particular block of the file – individual input blocks are completely undetermined by the non-encrypted packets. This is essentially the same security guarantee as that of [25, 22] as we describe in more detail in Section 3.2.

Similar ideas of using pre-processing (some special scrambling or an all-or-nothing transform[22]) with the subsequent encryption of only a small part of the content has been proposed before, see e.g. [13, 3]. The distinguishing feature of our work is that the pre-processing step we use is really motivated by other considerations (namely, forward error-correction requirements); we need only to slightly modify it, preserving the originally desired features. Thus, the

*J. Byers and J. Considine supported in part by NSF grants ANI-0093296 and ANI-9986397. G. Itkis supported in part by NSF grant CCR-0311485. E-mail: {byers,jconsidi,itkis,meicheng,ayeung}@cs.bu.edu.

[1] These 4% of the transmissions can be secured may be secured in a variety of ways, encryption is but one. Alternatively, they can be communicated by a separate secure channel, out-of band, etc.

pre-processing step can be considered as already "paid for", and adding security becomes almost free. Furthermore, in contrast to previous pre-processing steps, this encoding can actually be computed on-the-fly, so these techniques are well-suited for content on-demand.

Since only 4% of the information sent is encrypted, there are clear advantages on both the server and client sides. Reducing the amount of computational work on the server side is particularly important for busy servers delivering popular content, but the savings are also significant for thin clients, such as embedded devices or mobile palmtops. Moreover, while speeding up off-line encryption in group key settings is an important performance optimization, reducing the work spent encrypting packets by a factor of 25 can be a truly significant reduction.

These techniques motivate a study of specific approaches such as integration of security, fault-tolerance, compression and other aspects of data communication, which are typically considered separately. For example, since our techniques send most packets non-encrypted and expose significant partial information, it is important for the entropy of the input to be high to avoid basic statistical attacks. Fortunately, compression is already used for most bulk content delivery and most schemes are geared to provide the highest entropy output. In particular, analysis of our methods makes explicit the amount of information exposed when plain-text is leaked. As a result, it becomes possible to rigorously address the security provided and begin to examine the tradeoffs as less encryption is used.

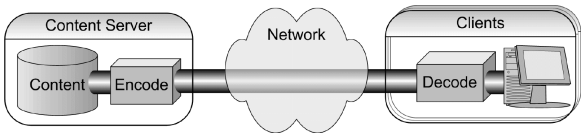
Lightweight methods for authentication have already been considered [27], but lightweight security has not previously been considered. We show a clear tradeoff between expensive per-packet encryption and our techniques without losing rigorous security guarantees. Additionally, our techniques are modular enough that they can be implemented as black box "filters" on the outgoing packets of a server and the incoming packets of a client thus providing security with minimum knowledge of and interaction with encoding functionality. Our solution gives a novel, cheap and easily applicable technique for adding encryption to encoded content and opens a number of important research directions in providing rigorous security guarantees at lower cost than full per-packet encryption. We also introduce the concepts and definitions intended to make the study of light security more formal and precise.

Moving beyond simple client-server scenarios, these techniques can also be applied in parallel download settings such as in [5, 4] with an interesting twist. Using the techniques of this section, a content provider could leverage a content distribution network to provide the unencrypted content. The content provider would only need to serve the small encrypted fraction while the vast majority was handled by the content distribution network. Note that the content provider does not need to entrust encryption keys with any other parties – only the content provider encrypts symbols in this scenario.

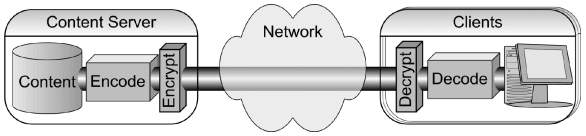
The rest of the paper is organized as follows. We begin in Section 2 by describing the general implications of our approach in content delivery scenarios, and then develop three specific scenarios which stand to benefit. Next, in Section 3, we give a technical presentation of the sparse parity-check codes and the encryption methods which we use in our work. We demonstrate how to design a provably secure encoding, and provide several key new definitions and theorems regarding the level of security which can be obtained by our methods as well as future similar techniques. In Section 4, we present the performance of our experimental system which integrates the components presented in Section 3, focusing in on the performance of our modified codes and the performance of dovetailing encoding and encryption. Various extensions to these methods are discussed briefly in Section 5 before we conclude with future directions and open problems in Section 6.

2 Bulk Content Delivery Scenarios and Requirements

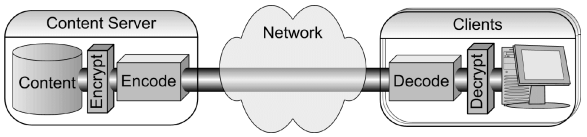
In the scenarios we wish to secure, the goal is to reliably transmit rich (bulky) content to large audiences with high concurrency. The primary challenge associated with delivering content to large audiences



(a) A typical system using FEC encoding and no encryption



(b) One naive way to secure FEC-encoded data streams



(c) Another naive way to secure data streams

Figure 1: (a) shows a typical system using FEC encoding. (b) and (c) show two naive ways to secure data streams in combination with FEC encoding.

is scalability. As session sizes grow, server side resources are strained, often leaving little time for separate encryption of each packet across each session.

A canonical approach to deal with issues of scale is to use a multicast paradigm, whereby each of the server's transmissions are replicated inside the network and delivered to the entire multicast group. Within such a paradigm, using FEC codes (see Figure 1) is a natural solution for dealing with issues of reliability while avoiding the well-known feedback implosion problem that arises when all members in a set of receivers must acknowledge each transmission [18, 6]. As for securing content under the multicast paradigm, the most common approach is to use a shared group key (i.e. one known to all clients) to encrypt the data. The group key is initially distributed in some (out-of-band) fashion, and then it can be maintained using secure in-band key updates. Efficient group key management techniques have been proposed e.g. [26] and efficiently implemented [12]. Note that when the group key does not change over time and there is no per-connection rekeying, it is possible to encrypt content off-line, though the burden of decryption by receiving clients is not alleviated. Alternatively, the content can be encrypted off-line using a random data encryption key (DEK), which will then be send to the receivers encrypted with the group key, used as the key encrypting key (KEK). Such a system can be implemented in either architecture of Figure 1.

One potential issue with using the multicast channel for all encrypted packets is that a content provider may prefer to have finer-grained control over who receives content. Thus, providers may prefer to employ connections which provide a more personalized service than multicast connections for ease of accounting (e.g. for billing purpose). Such personalized connections can use any generic encrypted point-to-point channel, such as SSL [24], but they avoid the benefits of multicast when every transmission must be individually encrypted for each client. A hybrid approach becomes more realistic when only 4% of the content must be encrypted and transmitted over a dedicated connection. While the ultimate scalability of the approach is still that of the single server approach, the constant factors involved become much better when 96% of the content may be multicast in the clear. We note that personalized connections can also be used as a simple means of distributing the group key while the encrypted content is still sent over the shared multicast channel. Namely, the KEK's can be used to provide personalization, while the data encrypted with DEK is distributed over the shared channel.

Actual content delivery implementations which approximately realize the benefits of the multicast paradigm can be divided into three categories. The first, IP multicast [10], provides a best effort IP multicast service, on top of which reliability can be supplied. IP multicast facilitates packet replication inside the network to reach the set of multicast clients so it substantially reduces both network traffic and the load at the server. Unfortunately, IP multicast

has not yet seen wide adoption over much of the Internet.

An alternative to IP multicast which has the potential to reduce server load, but does not provide any reduction in network traffic, is the use of individual unicast connections to deliver popular content. Of course, the server must then perform the extra work of sending packets to each client individually. One way to offset some of the associated processing costs is to use congestion-controlled UDP drawing FEC encoding packet payloads from a shared pool [23] (in this case the data flow of Figure 1 is flowing over UDP). This method can also achieve asynchronous reliability while minimizing I/O and computation costs at the server. In [23], it is also described how these mechanisms can be applied to efficiently deliver popular, encoded content over TCP.

A hybrid approach is to use an overlay network [7, 1] and simulate a multicast network using a collection of unicast tunnels. Each node, including the origin server, maintains low to moderate fanout in the virtual multicast topology, thus each sends data to only a few hosts, greatly distributing the delivery load across the system. However, these nodes are heterogeneous and may not be computationally powerful, so minimizing use of resources continues to be crucial. In [4], FEC codes are used over an overlay network to efficiently distribute content with better network utilization and fault tolerance.

For the first two of these approaches (true multicast and true unicast), encryption is handled in essentially the same fashion. FEC encoded content is generated at the origin server and passed to the encryption filter. This filter decides whether encryption is necessary and sends it through the appropriate channel. On the other side of the network, the client filter decrypts any encrypted packets received and sends both sets of packet to the FEC decoder which eventually outputs the data. In the overlay approach, the possibilities are richer, first since intermediate nodes may be able to forward encrypted content to other nodes when personalized encryption from the origin server is not a requirement. Also, in some scenarios, it is more efficient for overlay nodes to re-encode content.[4] This particular enhancement can be used completely transparently with our proposed encryption methods provided only un-encrypted packets are used in re-encoding.

3 Secure FEC Codes and Cryptographic Primitives

The modules which will be used as building blocks in our secure content-delivery architecture (see Figure 2) are FEC encoding and decoding modules as well as encryption and decryption modules. While we are able to use any of a variety of encryption algorithms (we will use them in a purely block-box fashion), the FEC codes which we employ must be specifically customized for our integrated security approach. We describe the class of codes we employ, the properties we need from them, and the customizations we apply to achieve these properties first.

3.1 Background: Sparse Parity-Check Codes

Rabin [20] defined an Information Dispersal Algorithm (IDA) which disperses a file F of length ℓ into n pieces F_i, $1 \leq i \leq n$ each of length ℓ/m such that the original file F can be reconstructed from *any* m pieces, where $n > m$. Rabin cited many applications for this procedure, ranging from dispersity routing, to storing files on disks which are prone to failure, to fault-tolerant broadcast. While the general approach proposed by Rabin is very powerful, a limitation of the approach is the computational complexity of the encoding and decoding operations he used. Most IDA approaches use Reed-Solomon codes, which rely on cumbersome finite field operations. The fastest in practice implementations of these require quadratic encoding and decoding times. Thus, while IDA works well for reconstructing a file stored as n pieces distributed over n remote sites when n is 14 and m is 10, its decoding inefficiency is prohibitive when multicasting an encoded file spanning n packets when n is 10,000 and m is 5,000. An alternative approach relies on the use of sparse codes based on parity-check principles, which can be encoded and decoded in nearly linear time, at the expense of relaxing the optimal decoding guarantee provided in IDA [15, 6].

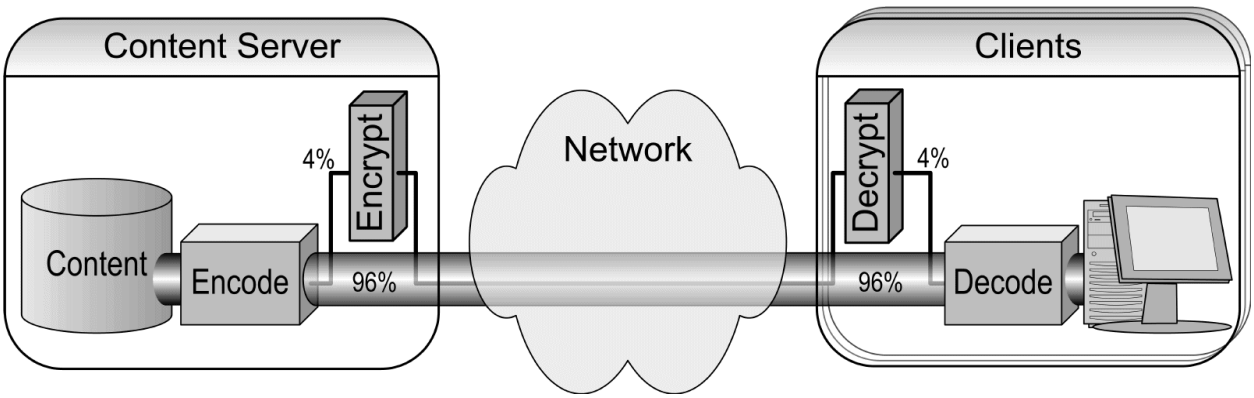


Figure 2: Our system architecture: Most data flows as in Figure 1, with only a 4% fraction of the packets pipelined through a secure channel.

Sparse parity-check codes facilitate fast encoding and decoding and are well suited to use with large files and highly dispersed encodings. A document is divided into a collection of m fixed-length *input symbols*, or data packets, $x_1, \ldots, x_m$. An *encoder* produces a potentially unbounded sequence of *output symbols*, or encoding packets[2], $y_1, y_2, \ldots$ from the set of data packets. With parity-check codes, each output symbol y_i is simply the bitwise exclusive-or of a subset of the input symbols, prepended with a list of the corresponding input symbols from which y_i is produced. Alternatively, the list of the input symbols is derived pseudo-randomly from the id i of the output symbol y_i.[3] Previous work on packetizing these output symbols [15, 6] describes how to represent this list in compressed form into a small, fixed-width field (4 bytes is typical). For a given output symbol, we refer to the cardinality of the subset of corresponding input symbols used to produce it as its *degree*, i.e. $y_3 = x_3 \oplus x_4$ has degree 2. We say that an encoding is *on-the-fly* or *rateless* if the random subset of input symbols used to produce each output symbol is drawn identically and independently from the same distribution $\mathcal{D}$ [14]. The time to produce an encoding packet from a set of data packets is proportional to the degree of an output symbol, while decoding from a subset of output symbols takes time proportional to the sum of the degrees of the output symbols in the subset. Thus, while variable degree distributions $\mathcal{D}$ may introduce some packets of large degree, it is important to note that encoding (and decoding) times are a function of the *average* degree, not the maximum. It is therefore essential that the codes used are sparse, i.e. the average degree is low.

A *decoder* attempts to recover the entire set of input symbols from a subset of the output symbols. Sparse codes typically relax the ideal decoding guarantees provided with IDA and Reed-Solomon codes. Such an erasure-resilient code is said to have average *decoding overhead* ϵ if on average, a randomly chosen subset of length $(1+\epsilon)m$ output symbols is sufficient to reconstruct the original file F. To minimize decoding overhead, the distribution $\mathcal{D}$ of the output symbols degrees is of crucial importance. High performance can be achieved using variable degree graphs, i.e. graphs in which degrees of output symbols are non-uniform [15]. The same paper shows that a certain family of heavy-tailed degree distributions is a particularly good choice.

To summarize the ideas introduced so far, the use of sparse erasure-resilient codes provides an algorithm which disperses a large file F of length ℓ into packets F_i, $1 \leq i \leq n$, each of length $p = \ell/m$

[2]The packets in this sequence are not necessarily distinct.

[3]In fact, an extra layer of security can be added here, by using a secret seed to derive this list of the input symbols. However, we do not include this extra protection in our analysis below.

such that the the original file F can be encoded in time $O(dn)$, where d is the average degree and can be reconstructed from a randomly chosen subset of $(1+\epsilon)m$ (independent of n) packets with high probability and in $O(dm)$ time on average.[4]

A set of n output symbols can be interpreted as a system of linear equations in m unknowns (input symbols) with 0-1 coefficients. A canonical representation of which output symbols are in the set would be in the form of an $n \times m$ 0-1 matrix. The ith row of the matrix represents the input symbols combined to form the ith output symbol where 0's represent the input symbols not used and 1's represent the input symbols that were exclusive-ored together. In this representation, encoding a set of output symbols $\vec{y}$ from a set of input symbols $\vec{x}$ and matrix Y can be described by $\vec{y} = Y\vec{x}$ where multiplication is multiplication by a scalar and addition is bit-wise exclusive-or. We call Y a *sparse parity-check encoding scheme*, or *spc-encoding scheme* for short and say that $\vec{y}$ is an *spc-encoding* of $\vec{x}$.

Given this representation as a set of linear equations, Gaussian elimination can be employed as a quadratic time decoding algorithm. However, the advantage of sparse codes in decoding is that the following (and computationally inexpensive) recovery rule can be employed instead: Find any equation with exactly one variable, recover the value of the variable by setting it equal to the value of the equation, then remove the newly recovered variable from any other equations that it appears in by exclusive-or'ing its value into each of these equations. With this approach, one can prove [15] that the total number of exclusive-or operations used to decode a set of output symbols is equal to the sum of the degrees of the output symbols, i.e. achieving linear decoding time with high probability, given a suitable degree distribution.

3.2 Securing Sparse Codes

Definition 1 *Function f is* packet-independent (pi-) secure *if for all v and uniformly distributed $\vec{x} = \langle x_1, \ldots, x_m \rangle$, $\Pr[x_i = v | f(\vec{x})] = \Pr[x_i = v]$.*

[4]The codes we employ in this paper have $\epsilon = .04$ and $d = 11$ when applied to a 32 MB file with symbol size 1400B.

This definition is essentially equivalent to the security definition of [25] which deals with all-or-nothing transforms [22] in the unconditional security model. In this case, $f(\vec{x})$ represents information sent as plaintext but no information about an individual x_i is known without additional information (which is sent encrypted). Note that later work with all-or-nothing transforms such as [3] add semantic security guarantees, thus avoiding information leaks about $\vec{x}$ present in [25] and our scheme.

In contrast to the schemes which had to do their own preprocessing [3, 13], we achieve pi-security in the context of spc-encoding without additional overhead. One way to do this is to make sure that Y has rank less than m and thus does not allow recovery of any x_i. In general these requirements may be hard to combine with the on-the-fly property of the encoding. However, a simple special case fulfils both requirements without any expensive verification: just assure that all rows of Y have even number of 1s. Namely, call an spc-encoding scheme Y an *spc2-encoding* if Y has an even number of 1s in each row.

Lemma 1 *Spc2-encoding is pi-secure.*

Proof: Let Y be an spc2-encoding scheme. Then the vector space spanned by the rows of Y (under bit-wise exclusive-or) contains only even degree vectors (i.e. those with an even number of 1's). Indeed, the sum (coordinate-wise exclusive-or) of any two vectors with even number of 1's, has an even degree of 1's. Thus any odd degree vector is independent of the row-vectors of Y. In particular, a vector containing 0s everywhere except i-th position (which contains 1) is independent from row-vectors of Y. Let us add this vector as a row vector (say as the 0th row) to Y and call the resulting matrix Y'. And let us add an output symbol v as an output symbol $y_0 = v$, $\vec{y'} = \langle y_0 = v, y_1, y_2, \ldots \rangle$. Since, for the given $\vec{y}$, the equation $Y\vec{u} = \vec{y}$ had a solution for $\vec{u}$, and Y' was obtained by adding an independent row vector, then $Y'\vec{u} = \vec{y'}$ also has a solution for $\vec{u}$. Moreover, this solution is guaranteed to have $u_i = v$. ■

If an encoding is pi-secure, it protects the content, but it also prevents its decoding. In order to enable decoding we must augment the pi-secure encoding

with some additional output symbols. Such additional output symbols must be communicated over a separate secure channel, in order to preserve security. We now specify one natural way to provide pi-security, beginning with a definition.

Definition 2 *A* repeated random pad encryption, *or rp-encryption for short, of $\vec{x}$ with a random pad p of fixed-length is $\vec{z} = \langle p \oplus x_1, \ldots, p \oplus x_m \rangle$.*

Corollary 1 *Repeated random pad encryption is pi-secure.*

Indeed, rp-encryption is equivalent to adding $x_0 = p$ to the vector $\vec{x}$ to obtain vector $\vec{v'}$, and computing the rp-encryption $z = Y\vec{x'}$, where Y is $(m+1) \times m$ 0-1 matrix with i-th row containing exactly two 1s: in position 0 and i. ■

This corollary highlights the relative weakness of pi-security in comparison to that of strong encryption – it can be provided by "weak" encryption schemes. Nonetheless, this paper explores how to achieve pi-security by restricting all encryption ("weak" or "strong") to apply only to a small fraction of all the data. It is our hope that new techniques can and will be developed to provide at least incremental improvements of our approach as compared to the "weak" encryption schemes such as rp-encryption.

Specifically, we propose to achieve pi-security by modifying the distributions discussed in Section 3.1, so that all the output symbols are either degree-1 or are even-degree, while keeping decoding overhead low, and keeping the proportion of singleton output symbols around 4% or less. Thus, the content can be protected by a pi-secure spc2-encoding plus encrypting of some singletons. If we assume that the singletons are properly secured by the encryption, then the security of the content is reduced to the security of the spc2-encoding. Thus, we can at least guarantee pi-security with our scheme. In fact, rp-encryption can be viewed as a special case of such a construction, but missing the FEC properties of the codes in Section 3.1. Our preferred method is to derive spc2-encoding which is very close to the distributions in Section 3.1.

The next two lemmas show that rp-encryption and spc2-encoding are polynomially equivalent in their security in a strong way: namely, whatever can be learned about $\vec{x}$ from a repeated pad encoding $\vec{z}$ can be learned from the even degree encoding $\vec{y}$, and vice versa. In fact, we make the polynomial equivalences more precise below. First we show that the encoding $\vec{y}$ itself can be computed from the rp-encryption $\vec{z}$ in time $O(md)$, where d is the average packet degree. Since this is the same running time used to rpc-encode the content, if the spc2-encoding can be broken (in any sense) efficiently, than breaking the rp-encryption will incur at most additive overhead $O(md)$, i.e. the rp-encryption is not much more secure than the spc2-encoding.

Lemma 2 *Let $\vec{z}$ be an rp-encryption of $\vec{x}$ and let matrix Y be an spc2-encoding scheme. Then $\vec{y} = Y\vec{x}$ can be computed from $\vec{z}$ and Y in $O(md)$ steps.*

Proof: From $\vec{z}$ generate $\vec{y} = Y\vec{z} = Y\vec{x}$. The second equality holds because the pads cancel each other out, as there are an even number of 1s in each row of Y. ■

The relationship in the other direction is slightly more complex since the correspondence between rp-encryption and spc2-encoding is one-to-many: a given rp-encoding determines spc2-encoding, while there could be many rp-encryptions corresponding to a given spc2-encoding. The next Lemma shows that an rp-encryption of any $\vec{x}$ with a uniformly distributed random pad can be computed from an spc-2 encoding of $\vec{x}$ in $O(m^2)$ steps. We assume that the number of output symbols $n = O(m)$.

Lemma 3 *Let Y be a 0-1 matrix of rank $m-1$, corresponding to an spc2-encoding scheme. Then given an spc2-encoding $\vec{y} = Y\vec{x}$ of some x, rp-encryption of $\vec{x}$ with a uniformly distributed random pad can be computed in time $O(m^2)$.*

Proof: The space of all even degree vectors has rank $(m-1)$. Thus, Y contains a basis for the space of all even degree vectors. In particular, the vectors corresponding to output symbols $\tilde{x_i} = x_1 \oplus x_i$ for $1 \leq i \leq m$ would be in that space. Thus, these output symbols could be obtained by Gaussian elimination in $O(m^2)$ steps.

Now, pick any output packet value r uniformly at random, and assume that $r = z_1 = x_1 \oplus p$ for some unknown, but also uniformly distributed pad p. The rp-encryption $\vec{z}$ of $\vec{x}$ with this p can now be computed as $z_i = z_1 \oplus \vec{x}_i$ in $m - 1$ steps. ■

The above two lemmas imply that the security of even degree encoding is polynomially equivalent to the security of repeated pad encryption. While the two schemes are polynomially equivalent in their security, it is still possible that spc2-encoding is more secure than the rp-encryption. For example, by Lemma 2, whatever can be extracted from the encoding in linear time can also be extracted from the rp-encryption, also in linear time. But if some information can be computed from the rp-encryption in linear time, it is not clear how to extract this information from the encoding faster than in quadratic time (due to Gaussian elimination) as given by Lemma 3.

So, in the worst case, by resorting to Gaussian elimination, spc2-codes can be reduced to repeated pad. If Gaussian elimination is necessary, this could provide some additional security – since for large files Gaussian elimination could be prohibitively expensive. In this specific case, it appears that a faster reduction might be possible (e.g. if more output symbols are collected), but other methods of increasing security might be possible.

Our final definition quantifies precisely how much encoded content we must encrypt in order to enable the remainder to be pi-secure:

Definition 3 *An encoding is called* α-securable, $0 \leq \alpha \leq 1$, *if for a randomly selected set of* n *output symbols, there exists a pi-secure subset of* $(1 - \alpha)n$ *output symbols (i.e. the encoding can be secured by securing only an* α *fraction of output symbols).*

How can we achieve pi-security of the code in Section 3.1? Encrypting all odd degree symbols is one possibility, but it still requires that a large fraction of codewords be encrypted. The degree distributions we start with contain approximately 37% odd degree symbols so they are 0.37-securable. The approach that we propose and ultimately adopt is to modify or adapt the degree distribution so that the overwhelming majority of output symbols have even degree, while the rest are of degree one. To secure such a code, only the degree one symbols need be encrypted; all the rest may be sent as plaintext. Our experimental results in Section 4 show that we can modify our output symbol distributions so that 1) all outcomes of a given odd degree can be rounded up or down to the nearest even degree, 2) a low frequency of degree one packets can be maintained and 3) the other desirable properties of the original encoding, such as low decoding overhead, can all be maintained. The upshot is a significantly lowered cost of securing a code against Gaussian elimination, since only approximately 4% of all packets need to be encrypted to achieve this goal, which we state this as our main result.

Theorem 1 *The content delivery scheme we have described is* .04*-securable.*

3.3 Encryption Schemes

In this paper we consider three efficient symmetric key encryption schemes: the two block ciphers Data Encryption Standard (DES) [17] and AES/Rijndael [9] cipher, and the stream cipher RC-4 [21]. The latter two are among the fastest modern cipher algorithms. AES is the the recently adopted Advanced Encryption Standard block cipher, which is intended to replace the veteran DES cipher. RC-4 is one of the fastest and most commonly used stream ciphers.

AES/Rijndael. Rijndael was recently selected as the Advanced Encryption Standard, approved by the National Institute of Standards and Technology, after extensive scrutiny and open analysis by experts, based both on its strength and efficiency. In our experiments, we use CBC (Cipher Block Chaining) mode with key size 128 bits and 128 bit block-size.

RC4. RC4 is a fast stream cipher. It requires a key with any size between 1 and 256 bytes to initialize a 256-byte state table. Then a pseudo-random stream is generated by the state table to be exclusive-or'ed with the plain-text for encrypting and with the cipher-text for decrypting (such use of a pseudo-random one time pad is typical of stream ciphers). In

this work, we use 128 bit (16 byte) key to initialize the state table.

DES. DES is one of the most used ciphers. It is widely deployed in applications ranging from Internet (SSL) to pay-TV, implemented in hardware (for which it was originally designed) and software. It uses 56-bit keys and 64-bit blocks. We also use CBC for DES. In its nearly 35 years of existence, it has demonstrated extraordinary resistance to cryptanalysis. Only in recent years have the limits set by its key size proven somewhat problematic, as successful attacks based largely on exhaustive search are now feasible. However, due to their high cost and other technical requirements, such attacks may not pose an immediate threat for many applications we focus on.

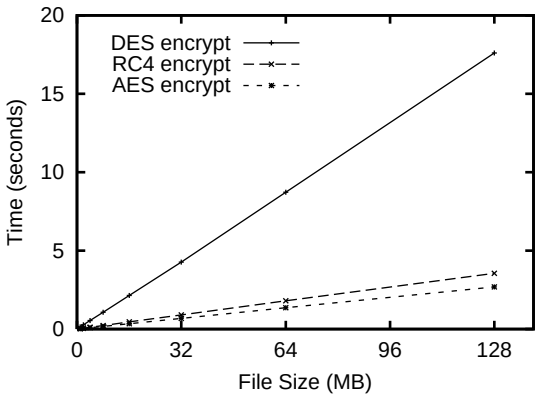


Figure 3: Comparison of Cypher Speeds.

Figure 3 compares the encrpytion times of these cyphers for encryption. Decryption times are omitted since they are virtually identical. Both RC4 and AES use 128 bit keys while DES uses 56 bit keys. The RC4 and DES implementations are from the SSLeay library [19] (more commonly seen as part of OpenSSL now) and the AES implementation is by Brian Gladman [11] with our own implementation of CBC. Of these three cyphers, AES is the fastest while DES is always the slowest.[5] We note that DES was originally designed for hardware implementations, while the speed of software implementations was a secondary consideration after security in the choice of the AES standard. Since RC4 is intermediate in speed, we will omit further results for RC4 since the times are easily interpolated from those of AES and DES. As we will demonstrate, using slower cipher algorithms like DES only strengthen the benefits of our results.

4 Experimental Results

Our experimental results are divided into two categories. The first set of results describes our methods for converting arbitrary degree distributions of sparse parity-check codes into distributions which can be secured by encrypting degree one output symbols and quantifies the performance cost of doing so. The second set describes our methods for combining erasure codes and encryption and quantifies the performance costs of using secure codes.

All of our experiments were run on a 2.8GHz Pentium 4 with 1GB of RAM running RedHat Linux (kernel version 2.4.20-9). Measurements were taken for file sizes ranging from 1MB to 128MB. The block size was fixed at 1KB, leaving plenty of room for packet headers without fragmentation. The time used for I/O operations was not included; only the time performing encoding and decoding or encrypting and decrypting was measured. Finally, each data point presented is the average over 100 samples.

The degree distributions we use in our experiments are based on a heuristic using oracles, which we do not detail here, but which is fully described in [8]. This approach enables a server to compute the output symbol degree most likely to advance a client's decoding process given that the client has recovered k input symbols. The server then iteratively refines the original degree distribution based on these computed values. This approach can generate degree distributions with empirical decoding overheads comparable to provable decoding guarantees in the literature such as Tornado Codes [15] though not quite as

[5]We note that the relative speeds of software implementations of the cyphers are very dependent on the underlying architecture. Earlier experiments running on Pentium 3 machines showed AES took 60% longer than RC4. These constant factors do not change the underlying message of this work.

low as some recently devised codes [14]. While the degree distributions we employ lack provable guarantees, they are easily adapted to our degree restrictions and our experimental results show that the desirable properties such as low decoding overhead and average degree are preserved.

4.1 Making Sparse Codes Securable

Our first set of experiments deals with adapting the degree distribution heuristic to only use degrees which are one or even. Only a minor modification to the code was necessary here – if the output symbol degree computed was odd and more than one, it was merely incremented. While this may seem biased towards higher degrees and delaying decoding, later iterations will compensate since they take longer to reach that point in the decoding process. That is, the higher degrees chosen will increase the all-or-nothing-like behavior, but the all-or-nothing-like behavior also delays those higher degrees being chosen again. Our experiments bear out our intuition that the heuristic is able to compensate for the one-or-even restriction, even when implemented in such an arbitrary fashion.

Figure 4 shows the results of applying the one-or-even restriction to the degree distribution heuristic. Figure 4(a) shows that the decoding overhead is comparable, both around two percent, except for the smallest file sizes. Figure 4(b) shows that the average degrees are comparable, so they are equally fast for encoding and decoding. All of these degree distributions choose degree one with $3.5 - 4\%$ probability, so the corresponding sparse parity check codes are 0.04-securable.

4.2 Adding Encryption to Securable Codes

Our remaining experiments deal with incorporating encryption into securable codes, in scenarios where both the encoding and encryption are done online. We consider the following ways of adding encryption to the transmission pipeline.

No Encoding: File is encrypted but not encoded. [6]

Encode Decode Only: File is encoded but not encrypted.

100% Encrypted: File is encoded with a securable code and *all* output symbols are encrypted.

4% Encrypted: File is encoded with a securable code and all degree one output symbols are encrypted.

Figure 5 summarizes the experiments adding encryption to securable codes from the point of view of the server. In each experiment, the server generates $2m$ output symbols and encrypts them as appropriate. The extra output symbols are generated to provide for losses and help guarantee that clients are able to decode. Errorbars showing the 1st and 99th percentiles are shown to illustrate variation in the encoding process. As expected, **4% Encrypted** is significantly cheaper than **100% Encrypted**. Also, we note that the slowest cipher with **4% Encrypted** is faster than the fastest cipher with **100% Encrypted**.

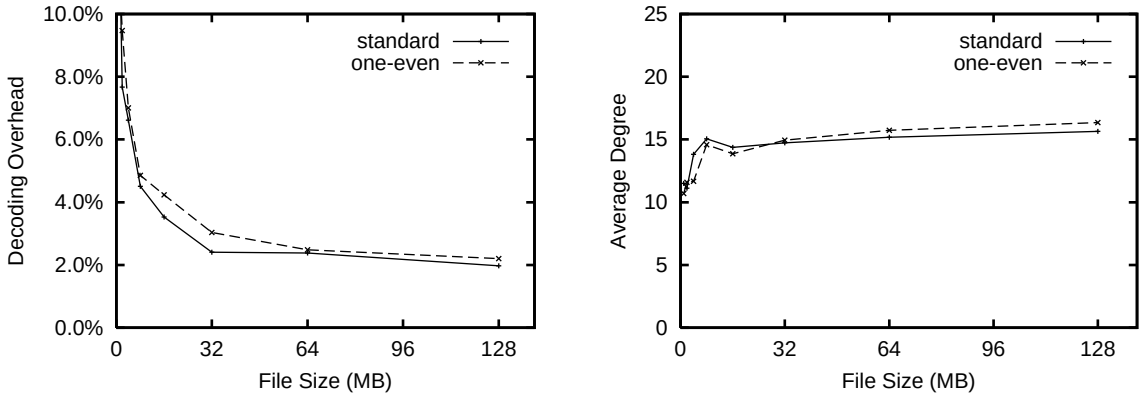
Benefits on the client side are similar and omitted for space. We do note that the speedup on the client is less due to the greater expense of decoding over encoding.[7] Table 1 summarizes the speedups of **4% Encrypted** over **100% Encrypted**, the most dramatic one being a speedup of almost five on the server side using DES. We emphasize that these experiments are conducted when both the FEC encoding and the encryption is performed online, although as noted earlier, for many applications it is natural to conduct encoding offline, in which case the speedups are even more dramatic.

5 Extensions

The construction we have focused upon so far is a very simple combination of sparse parity check codes

[6] By encoding, we specifically mean applying an erasure code.

[7] Encoding symbols is a simple memory-less process while decoding requires some bookkeeping for an efficient implementation.



(a) Decoding Overhead

(b) Average Degree

Figure 4: Adapting degree distributions to the one-even requirement.

	Server Side			Client Side		
	Secure-Code 100%	Secure-Code 4%	Speedup	Secure-Code 100%	Secure-Code 4%	Speedup
AES	$12.5s$	$7.8s$	1.6	$12.2s$	$10.7s$	1.1
RC4	$14.9s$	$7.5s$	2.0	$13.6s$	$9.9s$	1.4
DES	$40.7s$	$8.6s$	4.7	$27.0s$	$10.3s$	2.6

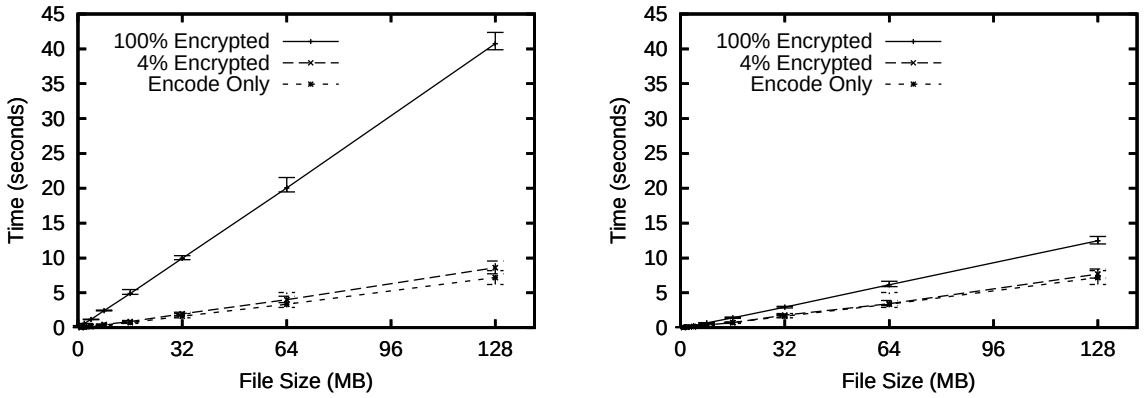
Table 1: System evaluation of CPU time and speedups for 128MB files.

and encryption. Despite this simplicity, we have already shown that this combination is both cheap and effective. We now briefly discuss further improvements, at the cost of some additional complications in the scheme.

As seen in Section 3.2, decrypting or otherwise recovering just one plaintext symbol suffices for an adversary to recover all of the symbols, given enough even degree symbols. Meanwhile, an honest client will need to decrypt approximately $0.04n$ symbols for recovery, much more than that required of the adversary. It would certainly be astonishing if a practical system could be improved by an honest client decrypting one symbol to simulate "breaking" the encryption and then switching to the adversary's methods to avoid further decryption operations. This could easily be the case in the scenarios of [2], where a smartcard is used to provide security with minimal computation and communication, which motivate work such as [3, 13].

We now introduce a new layer of pre-processing to provide pi-security such that a legitimate client performs exactly one decryption operation and the server performs exactly one encryption per key. We then generalize this pre-processing to increase the security so that exactly c encryption/decryption operations are necessary and c plain-text exposures are necessary to recover the entire file.

Recall that a document is divided into a collection of m fixed-length input symbols $x_1, \ldots, x_m$. Our new-preprocessing step constructs a new collection of input symbols $x'_1, \ldots, x'_m$, such that $x'_1 = E(x_1)$, where $E()$ is an appropriately keyed encryption function, and $x'_{i \geq 2} = x_1 \oplus x_i$.



(a) Encrypting with DES

(b) Encrypting with AES

Figure 5: System evaluation based upon server CPU time.

Theorem 2 *The set of symbols* $x_1, x'_2, \ldots, x'_m$ *is* $1/m$*-securable.*

Proof: First note that $x'_2, \ldots, x'_m$ is an spc2-encoding, so it is pi-secure. Since there is one other symbol, x_1, the theorem follows. ■

Given this theorem, we can safely transmit $x'_1, \ldots, x'_m$ in the fashion of our choosing while preserving the security guarantees so far. For example, we can apply any erasure encoding to $x'_1, \ldots, x'_m$ and transmit the resulting symbols. A client receiving these symbols recovers $x'_1, \ldots, x'_m$ using the normal decoding operations. Then x_1 is decrypted from x'_1, and $x_2, \ldots, x_m$ are easily recovered.

There are many ways to implement this construction on the server, though most naive methods suffer from storage requirements or excessive computational costs starting up. First, $x'_2, \ldots, x'_m$ can be pre-computed and stored before-hand, with x'_1 being computed once for each client or encryption key being used, and stored as long as it is needed. Once $x'_2, \ldots, x'_m$ has been stored, this implementation is no less efficient for the server than a plain erasure-encoding. The startup period generating $x'_2, \ldots, x'_m$ can be avoided by calculating $x'_2, \ldots, x'_m$ on an as-needed basis. That is, each $x'_{i\geq 2}$ is computed and stored the first time it is requested by the encoding process. Since all $x'_{i\geq 2}$ symbols have degree two, encoded symbols initially sent to clients will have their degrees effectively doubled while the $x'_{i\geq 2}$ symbols are being computed. This doubling disappears by the time the first $(1+\epsilon)$ encoded symbols are generated since all x'_i's have been used by then with high probability. Besides computational startup costs, the other disadvantage of this approach is that additional storage as large as the original file will be necessary to store $x'_2, \ldots, x'_m$.

A more on-the-fly approach is based upon the observation that combining an even number of $x'_{i\geq 2}$ symbols cancels out all of the x_1 symbols, leaving only an even number of $x_{i\geq 2}$ symbols. Similarly, combining an odd number of $x'_{i\geq 2}$ symbols cancels out all but one x_1 symbol. Thus, ignoring the small number of encoding symbols involving x'_1, only odd degree symbols involve x_1. Since our original degree distributions use about 37% odd degree symbols, this translates to only an increase of 0.37 in the average degree, which is only a few percent since the average degree of these distributions is around 15. For most sparse parity check codes, an individual input symbol only appears in $O(\log m)$ out of $(1+\epsilon)m$ encoding symbols (most occur only $O(1)$) times) with high probability, so the remaining overhead is negligible.

Besides minimizing the amount of encryption and decryption operations, the use of $x'_1, \ldots, x'_m$ is also useful in that it allows any erasure encoding to be applied. This means that codes such as LT codes [14] and online codes [16] with strong probabilistic guarantees on their behavior can be applied. However, we do note that one-level parity check codes such as LT codes will be slightly easier to implement efficiently. Online codes seem to require the pre-computation of $x'_2, \ldots, x'_m$ for the outer encoding, before being able to efficiently generate an inner encoding of unbounded length.[8]

A further generalization of the use of $x'_1, \ldots, x'_m$ can increase the security to that of c independent random pads of the same size as the input symbols. Basically, the input symbols $x_1, \ldots, x_m$ are partitioned into c disjoint sets. In each of these c sets, one element is encrypted and its plaintext is used to scramble the remainder of the set. As an example, $x'_{i \leq c} = E(x_i)$ where $E()$ is again the encryption function, and $x'_{i>c} = x_i \oplus x_{i\%c}$. Clearly, the pi-security of symbols in one partition is independent of the recovery of symbols in the other partitions. As before, any erasure encoding can be applied to $x'_1, \ldots, x'_m$ without compromising security. We do note that the on-the-fly implementation is slightly more expensive than the $c = 1$ special case discussed above, as the symbol degree may now increase by up to c, while the overhead of the other implementation strategies discussed before remains the same.

6 Conclusions

In this paper, we consider a problem which has not yet received much attention in either the networking or security literature, namely, providing a level of security for content delivery applications which is strong enough to thwart casual piracy, yet does not require the full computational horsepower of packet-level encryption. Such a method is especially useful for securing popular content, when server resources are truly at a premium. Using conventional transmission of content, it is not clear how to specify such a middle ground whereby a tradeoff between security guarantees and computational resources is possible. However, when fast forward error correcting codes are employed as part of the content delivery scheme, we demonstrate that certain encoding schemes lend themselves naturally to "partial encryption," whereby only a small fraction of the codewords generated are encrypted, while the rest are transmitted in the clear. Such an approach has many advantages, notably 1) the encoding-encryption pipeline is modular and easily configurable, 2) the encryption and decryption costs are minimal and 3) reconstructing any given piece of the plain-text is provably hard (albeit not as hard as when full encryption is employed).

We view our current work as only a first step in this direction, noting that there are a number of potential improvements that can be made to both our definitions and our general approach as evidenced by Section 5. Our future work will also consider securing content delivery applications when fast forward error correcting codes are not immediately applicable. For example, is there a general principle that can render an arbitrary byte stream amenable to *"partial encryption"* of the form described earlier? Such a mechanism could be useful in providing security guarantees for live streams, among other applications.

[8] Using the recommended parameters of [16] which give a decoding overhead similar to ours, generating symbols without pre-computing their outer encoding results in an effective degree twice that of ours, while generating symbols after pre-computing it results in an effective degree half of ours. This comparison does not involve the use of $x'_1, \ldots, x'_m$ for simplicity, though they would complicate matters further.

7 Acknowledgments

The third author would like to thank Leonid Reyzin for useful discussions and suggestions, esp. for Section 3.2, and simplifications of Lemma 3 in particular.

References

[1] Andersen, D., Balakrishnan, H., Kaashoek, F., and Morris, R. Resilient overlay networks. In *Proc. of ACM Symposium on Operating Systems Principles* (Banff, Canada, October 2001).

[2] Blaze, M. High-bandwidth encryption with low-bandwidth smartcards. In *Fast Software Encryption* (1996), no. 1039 in Springer Lecture Notes in Computer Science, pp. 33–40.

[3] Boyko, V. On the security properties of OAEP as an all-or-nothing transform. In *CRYPTO* (1999), pp. 503–518.

[4] Byers, J. W., Considine, J., Mitzenmacher, M., and Rost, S. Informed content delivery across adaptive overlay networks. In *Proc. of ACM SIGCOMM* (Pittsburgh, PA, August 2002), pp. 47–60.

[5] Byers, J. W., Luby, M., and Mitzenmacher, M. Accessing multiple mirror sites in parallel: Using tornado codes to speed up downloads. In *Proceedings of IEEE INFOCOM '99* (March 1999), pp. 275–83.

[6] Byers, J. W., Luby, M., and Mitzenmacher, M. A digital fountain approach to asynchronous reliable multicast. *IEEE Journal on Selected Areas in Communications 20*, 8 (2002), 1528–1540.

[7] Chu, Y., Rao, S., Seshan, S., and Zhang, H. A case for end system multicast. *IEEE Journal on Selected Areas in Communications 20*, 8 (2002), 1456–1471.

[8] Considine, J. Generating good degree distributions for erasure codes using oracles. Tech. Rep. BUCS-TR-2001-019, Boston University, CS Dept, Boston, MA 02215, October 2001.

[9] Daemen, J., and Rijmen, V. *The Design of Rijndael*. Springer-Verlag, 2002.

[10] Deering, S., and Cheriton, D. Multicast routing in datagram inter-networks and extended LANs. *ACM Transactions on Computer Systems 8* (May 1990), 85–110.

[11] Gladman, B. Cryptography Technology. `http://fp.gladman.plus.com/cryptography_technology/index.htm`.

[12] Hamandi, H., and Itkis, G. Group key manager on a smart-card. Available: `http://www.cs.bu.edu/groups/aces/gkm/`.

[13] Jakobsson, M., Stern, J. P., and Yung, M. Scramble all, encrypt small. In *Fast Software Encryption* (1999), pp. 95–111.

[14] Luby, M. LT Codes. In *Proceedings of 43rd Symposium on Foundations of Computer Science (FOCS 2002)* (Vancouver, BC, Canada, November 2002).

[15] Luby, M. G., Mitzenmacher, M., Shokrollahi, M. A., and Spielman, D. A. Efficient erasure correcting codes. *IEEE Trans. Info. Theory IT-47* (February 2001), 569–584.

[16] Maymounkov, P., and Mazieres, D. Rateless codes and big downloads. In *Proceedings of 2nd International Workshop on Peer-to-Peer Systems* (February 2003).

[17] National Bureau of Standards. Data Encryption Standard. Federal Information Processing Standards Pub. 46, Washington, D.C., 1977.

[18] Nonnenmacher, J., Biersack, E. W., and Towsley, D. Parity-based loss recovery for reliable multicast transmission. *IEEE/ACM Transactions on Networking (TON) 6*, 4 (1998), 349–361.

[19] The OpenSSL Project. `http://www.openssl.org`.

[20] Rabin, M. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM 38* (1989), 335–348.

[21] Rivest, R. The RC4 encryption algorithm. RSA Data Security, 1992.

[22] Rivest, R. L. All-or-nothing encryption. In *Fast Software Encryption* (1997), no. 1267 in Springer Lecture Notes in Computer Science, pp. 210–218.

[23] Rost, S., Byers, J., and Bestavros, A. The cyclone server architecture: Streamlining delivery of popular content. In *Proceedings of the 6th International Web Caching and Content Delivery Workshop (WCW)* (Boston, MA, June 2001).

[24] SSL 3.0 Specification. Available at: `http://www.netscape.com/eng/ssl3/`.

[25] Stinson, D. Something about all-or-nothing (transforms). *Designs, Codes and Cryptography 22* (2001), 133–138.

[26] Wallner, D., Harder, E., and Agee, R. Key management for multicast: Issues and architectures. internet request for comments 2627, 1999. Available: `ftp.ietf.org/rfc/rfc2627.txt`.

[27] Wong, C. K., and Lam, S. S. Digital signatures for flows and multicasts. *IEEE/ACM Transactions on Networking (TON) 7*, 4 (1999), 502–513.