On-the-Fly Verification of Rateless Erasure Codes for Efficient Content Distribution

Maxwell N. Krohn
MIT
krohn@mit.edu

Michael J. Freedman
NYU
mfreed@cs.nyu.edu

David Mazières
NYU
dm@cs.nyu.edu

Abstract— **The quality of peer-to-peer content distribution can suffer when malicious participants intentionally corrupt content. Some systems using simple block-by-block downloading can verify blocks with traditional cryptographic signatures and hashes, but these techniques do not apply well to more elegant systems that use rateless erasure codes for efficient multicast transfers. This paper presents a practical scheme, based on homomorphic hashing, that enables a downloader to perform on-the-fly verification of erasure-encoded blocks.**

I. Introduction

Peer-to-peer content distribution networks (P2P-CDNs) are trafficking larger and larger files, but end-users have not witnessed meaningful increases in their available bandwidth, nor have individual nodes become more reliable. As a result, the transfer times of files in these networks often exceed the average uptime of source nodes, and receivers frequently experience download truncations.

Exclusively unicast P2P-CDNs are furthermore extremely wasteful of bandwidth: a small number of files account for a sizable percentage of total transfers. Recent studies indicate that from a university network, KaZaa's 300 top bandwidth-consuming objects can account for 42% of all outbound traffic [1]. Multicast transmission of popular files might drastically reduce the total bandwidth consumed; however, traditional multicast systems would fare poorly in such unstable networks.

Developments in practical erasure codes [2] and *rateless* erasure codes [3], [4], [5] point to elegant solutions for both problems. Erasure codes of rate r (where $0 < r < 1$) map a file of n *message blocks* onto a larger set of n/r *check blocks*. Using such a scheme, a sender simply transmits a random sequence of these check blocks. A receiver can decode the original file with high probability once he has amassed a random collection of slightly more than n unique check blocks. At larger values of r, senders and receivers must carefully coordinate to avoid block duplication. In rateless codes, block duplication is much less of a problem: encoders need not pre-specify a value for r and can instead map a file's blocks to a set of check blocks whose size is exponential in n.

When using low-rate or rateless erasure codes, senders and receivers forgo the costly and complicated feedback protocols often needed to reconcile truncated downloads or to maintain a reliable multicast tree. Receivers can furthermore collect blocks from multiple senders simultaneously. One can envision an ideal situation, in which many senders transmit the same file to many recipients in a "forest of multicast trees." No retransmissions are needed when receivers and senders leave and reenter the network, as they frequently do.

A growing body of literature considers erasure codes in the context of modern distributed systems. Earlier work applied fixed-rate codes to centralized multicast CDNs [6], [7]. More current work considers rateless erasure codes in unicast, multi-source P2P-CDNs [8], [9]. Most recently, SplitStream [10] has explored applying rateless erasure codes to overlapping P2P multicast networks, and Bullet [11] calls on these codes when implementing "overlay meshes."

There is a significant downside to this popular approach. When transferring erasure-encoded files, receivers can only "preview" their file at the very end of the transfer. A receiver may discover that, after dedicating hours or days of bandwidth to a certain file transfer, he was receiving incorrect or useless blocks all along. Most prior work in this area assumes honest senders, but architects of robust, real-world P2P-CDNs cannot make this assumption.

This paper describes a novel construction that lets recipients verify the integrity of check blocks immediately, before consuming large amounts of bandwidth or polluting their download caches. In our scheme, a file F is compressed down to a smaller hash value, $H(F)$, with which the receiver can verify the integrity of any possible check block. Receivers then need only obtain a file's hash value to avoid being duped during a transfer. Our function H is based on a discrete-log-based, collision-resistant, homomorphic hash function, which allows receivers to compose hash values in much the same way that encoders compose message blocks. Unlike more obvious constructions, ours is independent of encoding rate and is therefore compatible with rateless erasure codes. It is fast to compute, efficiently verified using probabilistic batch verification, and has provable security under the discrete-log assumption. Furthermore, our implementation results suggest this scheme is practical for real-world use.

In the remainder of this paper, we will discuss our setting in more detail (Sections II and III), describe our hashing scheme (Section IV), analyze its important properties (Sections V and VI), discuss related works (Section VII), and conclude (Section VIII).

II. Brief Review of Erasure Codes

In this paper, we consider the non-streaming transfer of very large files over erasure channels such as the Internet. Typically, a file F is divided into n uniformly sized blocks, known

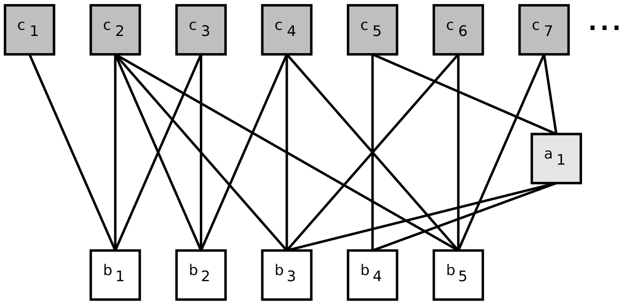

Fig. 1. Example Online encoding of a five-block file. b_i are message blocks, a_1 is an auxiliary block, and c_i are check blocks. Edges represent addition (via XOR). For example, $c_4 = b_2 + b_3 + b_5$, $a_1 = b_3 + b_4$, and $c_7 = a_1 + b_5$.

as *message blocks* (or alternatively, input symbols). Erasure encoding schemes add redundancy to the original n message blocks, so that receivers can recover from packet drops without explicit packet retransmissions.

Though traditional forward error correction codes such as Reed-Solomon are applicable to erasure channels [12], decoding times quadratic in n make them prohibitively expensive for large files. To this effect, researchers have proposed a class of erasure codes with sub-quadratic decoding times. Examples include Tornado Codes [7], LT Codes [3], Raptor Codes [5] and Online Codes [4]. All four of these schemes output *check blocks* (or alternatively, output symbols) that are simple summations of message blocks. That is, if the file F is composed of message blocks b_1 through b_n, the check block c_1 might be computed as $b_1 + b_2$. The specifics of these linear relationships vary with the scheme.

Tornado Codes, unlike the other three, are fixed-rate. A sender first chooses a rate r and then can generate no more than n/r check blocks. Furthermore, the encoding process grows more expensive as r approaches zero. For multicast and other applications that benefit from lower encoding rates, LT, Raptor and Online codes are preferable [9]. Unlike Tornado codes, they feature rateless encoders that can generate an enormous sequence of check blocks with state constant in n. LT codes are decodable in time $O(n \ln(n))$, while Tornado, Raptor and Online Codes have linear-time decoders.

This paper uses Online Codes when considering the specifics of the encoding and decoding processes; however, all three rateless techniques are closely related, and the techniques described are equally applicable to LT and Raptor Codes.

Online Codes. Online Codes consist of three logical components: a precoder, an encoder and a decoder. A sender initializes the encoding scheme via the precoder, which takes as input a file F with n message blocks and outputs $n\delta k$ *auxiliary blocks*. k is small constant such as 3, and δ, a parameter discussed later, has a value such as .005. The precoder works by adding each message block to k distinct randomly-chosen auxiliary blocks. An auxiliary block is thus the sum of $1/\delta$ message blocks on average. This process need not be random in practice; the connections between the message and auxiliary blocks can be a deterministic function of the input size n, and the parameters k and δ. Finally, the n message blocks and the $n\delta k$ auxiliary blocks are considered together as a composite file F' of size $n' = n(1 + \delta k)$, which is suitable for encoding.

To construct the i^{th} check block, the encoder randomly samples a pre-specified probability distribution for a value d_i, known as the check block's *degree*. The encoder then selects d_i blocks from F' at random, and computes their sum, c_i. The outputted check block is a pair $\langle x_i, c_i \rangle$, where x_i describes which blocks were randomly chosen from F'. In practice, an encoder can compute the degree d_i and the meta-data x_i as the output of a pseudo-random function on input (i, n). It thus suffices to send $\langle i, c_i \rangle$ to the receiving client, who can compute x_i with knowledge of n, the encoding parameters, and access to the same pseudo-random function. See Figure 1 for a schematic example of precoding and encoding.

To recover the file, a recipient collects check blocks of the form $\langle x_i, c_i \rangle$. Assume a received block has degree one; that is, it has meta-data x_i of the form $\{j\}$. Then, c_i is simply the j^{th} block of the file F', and it can be marked *recovered*. Once a block is recovered, the decoder subtracts it from the appropriate *unrecovered* check blocks. That is, if the k^{th} check block is such that $j \in x_k$, then b_j is subtracted from c_k, and j is subtracted from x_k. Note that during this subtraction process, other blocks might be recovered. If so, then the decoding algorithm continues iteratively. When the decoder receives blocks whose degree is greater than one, the same type of process applies; that is, all recovered blocks are subtracted from it, which might in turn recover it.

In the encoding process, auxiliary blocks behave like message blocks; in the decoding process, they behave like check blocks. When the decoder recovers an auxiliary block, it then adds it to the pool of unrecovered check blocks. When the decoder recovers a message block, it simply writes the block out to a file in the appropriate location. Decoding terminates once all n message blocks are recovered.

In the absence of the precoding step, the codes are expected to recover $(1-\delta)n$ message blocks from $(1+\epsilon)n$ check blocks, as n becomes large. The auxiliary blocks introduced in the precoding stage help the decoder to recover the final δn blocks. A sender specifies δ and ϵ prior to encoding; they in turn determine the encoder's degree distribution and consequently the number of block operations required to decode.

Online Codes, like the other three schemes, use bitwise exclusive OR for both addition and subtraction. We note that although XOR is fast, simple, and compact (*i.e.*, XORing two blocks does not produce carry bits), it is not essential. Any efficiently invertible operation suffices.

III. Threat Model

Deployed P2P-CDNs like KaZaa consist of nodes who function simultaneously as publishers, mirrors, and downloaders of content. Nodes transfer content by sending contiguous file chunks over point-to-point links, with few security guarantees. We imagine a similar but more powerful network model:

When a node wishes to publish F, he uses a collision-resistant hash function such as SHA1 [13] to derive a succinct cryptographic file handle, $H(F)$. He then pushes F into the network and also publicizes the mapping of the file's name $N(F)$ to its key, $H(F)$. Mirrors maintain local copies of the file F and transfer erasure encodings of it to multiple clients simultaneously. As downloaders receive check blocks, they can forward them to other downloaders, harmlessly "down-sampling" if constrained by downstream bandwidth. Once a downloader fully recovers F, he generates his own encoding of F, sending "fresh" check blocks to downstream recipients. Meanwhile, erasure codes enable downloaders to collect check blocks concurrently from multiple sources.

This setting differs notably from traditional multicast settings. Here, internal nodes are not mere packet-forwarders but instead are active nodes that produce unique erasure encodings of the files they redistribute.

Unfortunately, in a P2P-CDN, one must assume that adversarial parties control arbitrarily many nodes on the network. Hence, mirrors may be frequently malicious.[1] Under these assumptions, the P2P-CDN model is vulnerable to a host of different attacks:

Content Mislabeling. A downloader's original lookup mapped $N(F) \rightarrow H(\tilde{F})$. The downloader will then request and receive the file $\tilde{F}$ from the network, even though he expected F.

Bogus-Encoding Attacks. Mirrors send blocks that are not check blocks of the expected file, with the intent of thwarting the downloader's decoding. This has also been termed a pollution attack [14].

Distribution Attacks. A malicious mirror sends valid check blocks from the encoding of F, but not according to the correct distribution. As a result, the receiver might experience degenerate behavior when trying to decode.

Deployed peer-to-peer networks already suffer from malicious content-mislabeling. A popular file may resolve to dozens of names, only a fraction of which are appropriately named. A number of solutions exist, ranging from simply downloading the most widely replicated name (on the assumption that people will keep the file if it is valid), to more complex reputation-based schemes. In more interesting P2P-CDNs, trusted publishers might sign file hashes. Consider the case of a Linux vendor using a P2P-CDN to distribute large binary upgrades. If the vendor distributes its public key in CD-based distributions, clients can verify the vendor's signature of any subsequent upgrade. The general mechanics of reliable filename resolution are beyond the scope this paper; for the most part, we assume that a downloader can retrieve $H(F)$ given $N(F)$ via some out-of-band and trusted lookup.

This work focuses on the bogus-encoding attack. When transferring large files, receivers will talk to many different mirrors, in series and in parallel. At the very least, the receiver should be able to distinguish valid from bogus check blocks at decoding time. One bad block should not ruin hundreds of thousands of valid ones. Moreover, receivers have limited bandwidth and cannot afford to communicate with all possible mirrors on the network simultaneously. They would clearly benefit from a mechanism to detect cheating as it happens, so they can terminate connections to bad servers and seek out honest senders elsewhere on the network.

To protect clients against encoding attacks, P2P-CDNs require some form of source verification. That is, downloaders need a way to verify individual check blocks, given a reliable and compact hash of the desired file. Furthermore, this verification must not be interactive; it should work whether or not the original publisher is online. The question becomes, should the original publisher authenticate file blocks before or after they are encoded? We consider both cases.

A. Hashing All Input Symbols

A publisher wishes to distribute an n-block file F. Assuming Online Codes, he first runs F through a precoder, yielding an n'-block file F'. He then computes a Merkle hash tree of F' [15]. The file's full hash is the entirety of the hash tree, but the publisher uses the hash tree's root for the file's succinct cryptographic handle. To publish, he pushes the file and the file's hash tree into the network, all keyed by the root of the hash tree. Note that although the hash tree is smaller than the original file, its size is still linear in n.

To download F, a client maps $N(F)$ to $H(F)$ as usual, but now $H(F)$ is the root of the file's hash tree. Next, the client retrieves the rest of the hash tree from the network, and is able to verify its consistency with respect to its root. Given this information, he can verify check blocks as the decoding progresses, through use of a "smart decoder." As check blocks of degree one arrive, he can immediately verify them against their corresponding leaf in the hash tree. Similarly, whenever the decoder recovers an input symbol b_j from a check block $\langle x_i, c_i \rangle$ of higher degree, the receiver verifies the recovered block b_j against its hash. If the recovered block verifies properly, then the receiver concludes that $\langle x_i, c_i \rangle$ was generated honestly and hence is *valid*. If not, then the receiver concludes that it is bogus.

In this process, the decoder only XORs check blocks with validated degree-one blocks. Consequently, valid blocks cannot be corrupted during the decoding process. On the other hand, invalid check blocks which are reduced to degree-one blocks are easily identified and discarded. Using this "smart decoder," a receiver can trivially distinguish bogus from valid check blocks and need not worry about the download cache pollution described in [14]. The problem, however, is that a vast majority of these block operations happen at the very end of the decoding process—when almost n check blocks are available to the decoder. Figure 2 exhibits the average results for decoding a file of $n = 10,000$ blocks, taken over 50 random Online encodings. According to these experiments, when a receiver has amassed $.9n$ check blocks, he can recover

[1]We do not explicitly model adversaries controlling the underlying physical routers or network trunks, although our techniques are also robust against these adversaries, with the obvious limitations (*e.g.*, the adversary can prevent a transfer if he blocks the downloader's network access).

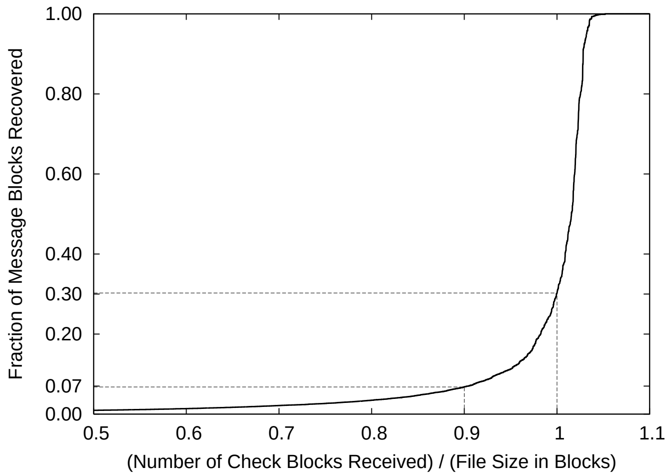

Fig. 2. Number of blocks recoverable as function of number of blocks received. Data collected over 50 random encodings of a 10,000 block file.

only $.068n$ message blocks; when he has amassed n check blocks, he can recover only $.303n$ message blocks. In practice, a downloader could dedicate days of bandwidth to receiving gigabytes of check blocks, only to find that most are bogus.

B. Hashing Check Blocks

Instead of hashing the input to the erasure encoder, publishers might hash its output. If so, the P2P-CDN is immediately limited to fixed-rate codes. Recall that the publisher is not directly involved in the file's ultimate distribution to clients and therefore cannot be expected to hash and sign check blocks on-the-fly. Thus, the publisher must pre-specify a tractable rate r and "pre-authorize" n/r check blocks. In practice, the publisher might do this by generating n/r check blocks, computing their hash tree, and keying the file by its root. When mirrors distribute the file, they distribute only those check blocks that the publisher has preauthorized. With the benefit of the hash tree taken over all possible check blocks, the receiver can trivially verify check blocks as they arrive. Section V-D explores this proposal in more detail. We simply observe here that it becomes prohibitively expensive for encoding at low rates, in terms of the publisher's computational resources and the bandwidth required to distribute hashes.

IV. Homomorphic Hashing

Our solution combines the advantages of the previous section's two approaches. As in the first scheme, our hashes are reasonably-sized and independent of the encoding rate r. As in the second, they enable receivers to authenticate check blocks on the fly.

We propose two possible authentication protocols based on a homomorphic collision-resistant hash function (CRHF). In the *global hashing* model, there is a single way to map F to $H(F)$ by using global parameters. As such, one-time hash generation is slow but well-defined. In the *per-publisher hashing* model, each publisher chooses his own hash parameters, and different publishers will generate different hashes for the same file. We will later show that the per-publishing model enables publishers to generate hashes more efficiently, although the downloader's verification overhead is the same.

In today's file-sharing systems, there may be multiple publishers for the same content—*e.g.*, different individuals may rip the same CD—thus these publishers may use global hashing so that all copies look identical to the system. In other environments, content has a single, well-known publisher, and the per-publisher scheme is more appropriate. While the latter might be ill-suited for copyright circumvention, it otherwise is more useful, allowing publishers to sign file hashes and clients to authenticate file name to file hash mappings. Many Internet users could benefit from cheap, trusted, and efficient distribution of bulk data: anything from Linux binary distributions to large academic data sets could travel through such a network.

A. Notation and Preliminaries

In the following discussion, we will be using scalars, vectors and matrices defined over modular subgroups of $\mathbb{Z}$. We write scalars in lowercase (*e.g.*, x), vectors in lowercase boldface (*e.g.*, $\mathbf{x}$) and matrices in uppercase (*e.g.*, X). Furthermore, for the matrix X, the j^{th} column is a vector written as $\mathbf{x}_j$, and the ij^{th} cell is a scalar written as x_{ij}. Vectors might be row vectors or column vectors, and we explicitly specify them as such. All additions are assumed to be taken over $\mathbb{Z}_q$, and multiplications and exponentiations are assumed to be taken over $\mathbb{Z}_p$, with q and p selected as described in the next subsection. Finally, we invent one notational convenience concerning vector exponentiation. That is, we define $g^{\mathbf{r}} = \mathbf{g}$ component-wise: if the row vector $\mathbf{r} = (r_1\ r_2\ \cdots\ r_m)$, then the row vector $g^{\mathbf{r}} = (g^{r_1}\ g^{r_2}\ \cdots\ g^{r_m})$.

B. Global Homomorphic Hashing

In global homomorphic hashing, all nodes on the network must agree on hash parameters so that any two nodes independently hashing the same file F should arrive at exactly the same hash. To achieve this goal, all nodes must agree on security parameters λ_p and λ_q. Then, a trusted party globally generates a set of hash parameters $G = (p, q, \mathbf{g})$, where p and q are two large random primes such that $|p| = \lambda_p$, $|q| = \lambda_q$, and $q|(p-1)$. The hash parameter $\mathbf{g}$ is a $1 \times m$ row-vector, composed of random elements of $\mathbb{Z}_p$, all order q. These and other parameters are summarized in Table I.

In decentralized P2P-CDNs, such a trusted party might not exist. Rather, users joining the system should demand "proof" that the group parameters G were generated honestly. In particular, no node should know i, j, x_i, x_j such that $g_i^{x_i} = g_j^{x_j}$, as one that had this knowledge could easily compute hash collisions. The generators might therefore be generated according to the algorithm PickGroup given in Figure 3. The input $(\lambda_p, \lambda_q, m, s)$ to the PickGroup algorithm serves as a heuristic proof of authenticity for the output parameters, $G = (p, q, \mathbf{g})$. That is, unless an adversary exploits specific properties of SHA1, he would have difficulty computing a seed s that yields generators with a known logarithmic relation. In practice, the seed s might be chosen globally, or even chosen

TABLE I

SYSTEM PARAMETERS AND PROPERTIES

Name	Description	*e.g.*
λ_p	discrete log security parameter	1024
λ_q	discrete log security parameter	257
p	random prime, $\lvert p\rvert = \lambda_p$	
q	random prime, $q\vert(p-1)$, $\lvert q\rvert = \lambda_q$	
β	block size in bits	16 KB
m	$= \lceil \beta/(\lambda_q - 1)\rceil$ (number of "sub-blocks" per block)	512
$\mathbf{g}$	$1 \times m$ row vector of order q elts in $\mathbb{Z}_p$	
G	hash parameters, given by $(p, q, \mathbf{g})$	
n	original file size	1 GB
k	precoding parameter	3
δ	fraction of unrecoverable message blocks (without the benefit of precoding)	.005
n'	precoded file size, $n' = (1 + \delta k)n$	1.015 GB
ϵ	asymptotic encoding overhead	.01
d	average degree of check blocks	~ 8.17

per file F such that $s = \mathsf{SHA1}(N(F))$. Either way, the same parameters G will always be used when hashing file F.

File Representation. As per Table I, let β be the block size, and let $m = \lceil \beta/(\lambda_q - 1)\rceil$. Consider a file F as an $m \times n$ matrix, whose cells are all elements of $\mathbb{Z}_q$. Our selection of m guarantees that each element is less than $2^{\lambda_q - 1}$, and is therefore less than the prime q. Now, the j^{th} column of F simply corresponds to the j^{th} message block of the file F, which we write $\mathbf{b}_j = (b_{1,j}, \ldots, b_{m,j})$. Thus:

$$F = (\mathbf{b}_1\ \mathbf{b}_2\ \cdots\ \mathbf{b_n}) = \begin{pmatrix} b_{1,1} & \cdots & b_{1,n} \\ \vdots & \ddots & \vdots \\ b_{m,1} & \cdots & b_{m,n} \end{pmatrix}$$

We add two blocks by adding their corresponding column-vectors. That is, to combine the i^{th} and j^{th} blocks of the file, we simply compute:

$$\mathbf{b}_i + \mathbf{b}_j = (b_{1,i} + b_{1,j}, \ldots, b_{m,i} + b_{m,j}) \bmod q$$

Precoding. Recall that the precoding stage in Online Codes produces auxiliary blocks that are summations of message blocks, and that the resulting composite file has the original n message blocks, and the additional $n\delta k$ auxiliary blocks. The precoder now proceeds as usual, but uses addition over $\mathbb{Z}_q$ instead of the XOR operator.

We can represent this process succinctly with matrix notation. That is, the precoding stage is given by a binary $n \times n'$ matrix, $Y = (I|P)$. The matrix Y is the concatenation of the $n \times n$ identity matrix I, and the $n \times n\delta k$ matrix P that represents the composition of auxiliary blocks. All rows of P sum to k, and its columns sum to $1/\delta$ on average. The precoded file can be computed as $F' = FY$. The first n columns of F' are the message blocks. The remaining $n\delta k$ columns are the auxiliary blocks. For convenience, we refer to auxiliary blocks as $\mathbf{b}_i$, where $n < i \le n'$.

Encoding. Like precoding, encoding is unchanged save for the addition operation. For each check block, the encoder picks an n'-dimensional bit vector $\mathbf{x}$ and computes $\mathbf{c} = F'\mathbf{x}$. The output $\langle \mathbf{x}, \mathbf{c}\rangle$ fully describes the check block.

Algorithm $\mathsf{PickGroup}(\lambda_p, \lambda_q, m, s)$
  Seed PRNG $\mathcal{G}$ with s.
  do
    $q \leftarrow \mathsf{qGen}(\lambda_q)$
    $p \leftarrow \mathsf{pGen}(q, \lambda_p)$
  while $p = 0$ **done**
  for $i = 1$ **to** m **do**
    do
      $x \leftarrow \mathcal{G}(p-1) + 1$
      $g_i \leftarrow x^{(p-1)/q} \pmod p$
    while $g_i = 1$ **done**
  done
  return $(p, q, \mathbf{g})$

Algorithm $\mathsf{qGen}(\lambda_q)$
  do
    $q \leftarrow \mathcal{G}(2^{\lambda_q})$
  while q is not prime **done**
  return q

Algorithm $\mathsf{pGen}(q, \lambda_p)$
  for $i = 1$ to $4\lambda_p$ **do**
    $X \leftarrow \mathcal{G}(2^{\lambda_p})$
    $c \leftarrow X \pmod{2q}$
    $p \leftarrow X - c + 1$ // Note $p \equiv 1 \pmod{2q}$
    if p is prime **then return** p
  done
  return 0

Fig. 3. The seed s can serve as an heuristic "proof" that the hash parameters were chosen honestly. This algorithm is based on that given in the NIST Standard [16]. The notation $\mathcal{G}(x)$ should be taken to mean that the pseudo-random number generator $\mathcal{G}$ outputs the next number in its pseudo-random sequence, scaled to the range $\{0, \ldots, x-1\}$.

Hash Generation. To hash a file, a publisher uses a CRHF, secure under the discrete-log assumption. This hash function is a generalized form of the Pederson commitment scheme [17] (and from Chaum et al. [18]), and it is similar to that used in various incremental hashing schemes (see Section VII). Recall that a CRHF is informally defined as a function for which finding any two inputs that yield the same output is difficult.

For an arbitrary message block $\mathbf{b}_j$, define its hash with respect to G:

$$h_G(\mathbf{b}_j) = \prod_{i=1}^{m} g_i^{b_{i,j}} \bmod p \tag{1}$$

Define the hash of file F as a $1 \times n$ row-vector whose elements are the hashes of its constituent blocks:

$$H_G(F) = (h_G(\mathbf{b}_1)\ h_G(\mathbf{b}_2)\ \cdots\ h_G(\mathbf{b}_n)) \tag{2}$$

To convey the complete hash, publishers should transmit both the group parameters and the hash itself: $(G, H_G(F))$. From this construction, it can be seen that each block of the file is β bits, and the hash of each block is λ_p bits. Hence, the hash function H_G reduces the file by a factor of β/λ_p, and therefore $|H_G(F)| = |F|\lambda_p/\beta$.

Hash Verification. If a downloader knows $(G, H_G(F))$, he can first compute the hash values for the $n\delta k$ auxiliary blocks. Recall that the precoding matrix Y is a deterministic function of the file size n and the preestablished encoding parameters δ and k. Thus, the receiver computes Y and obtains the hash over the composite file as $H_G(F') = H_G(F) \cdot Y$. The hash of the auxiliary blocks are the last $n\delta k$ cells in this row vector.

To verify whether a given check block $\langle \mathbf{x}, \mathbf{c} \rangle$ satisfies $\mathbf{c} = F'\mathbf{x}$, a receiver verifies that:

$$h_G(\mathbf{c}) = \prod_{i=1}^{n'} h_G(\mathbf{b_i})^{x_i} \tag{3}$$

h_G functions here as a *homomorphic hash function*. For any two blocks $\mathbf{b}_i$ and $\mathbf{b}_j$, $h_G(\mathbf{b}_i + \mathbf{b}_j) = h_G(\mathbf{b}_i)h_G(\mathbf{b}_j)$.

Downloaders should monitor the aggregate behavior of mirrors during a transfer. If a downloader detects a number of unverifiable check blocks above a predetermined threshold, he should consider the sender malicious and should terminate the transfer.

Decoding. Decoding proceeds as described in Section II. Of course, XOR is conveniently its own inverse, so implementations of standard Online Codes need not distinguish between addition and subtraction. In our case, we simply use subtraction over $\mathbb{Z}_q$ to reduce check blocks as necessary.

Despite our emphasis on Online Codes in particular, we note that these techniques apply to LT and Raptor codes. LT Codes do not involve preprocessing, so the above scheme can be simplified. Raptor Codes involve a two-stage precoding process, and probably are not compatible with the implicit calculation of auxiliary block hashes described above. In this case, we compute file hashes over the output of the precoder, therefore obtaining slightly larger file hashes.

C. Per-Publisher Homomorphic Hashing

The per-publisher hashing scheme is an optimization of the global hashing scheme just described. In the per-publisher hashing scheme, a given publisher picks group parameters G so that a logarithmic relation among the generators $\mathbf{g}$ is known. The publisher picks q and p as above, but generates $\mathbf{g}$ by picking a random $g \in \mathbb{Z}_p$ of order q, generating a random vector $\mathbf{r}$ whose elements are in $\mathbb{Z}_q$ and then computing $\mathbf{g} = g^{\mathbf{r}}$.

Given the parameters g and $\mathbf{r}$, the publisher can compute file hashes with many fewer modular exponentiations:

$$H_G(F) = g^{\mathbf{r}F} \tag{4}$$

The publisher computes the product $\mathbf{r}F$ first, and then performs only one modular exponentiation per file block to obtain the full file hash. See Section V-B for a more complete running-time analysis. The hasher must be careful to never reveal g and $\mathbf{r}$; doing so allows an adversary to compute arbitrary collisions for H_G.

Aside from hash parameter generation and hash generation, all aspects of the protocol described above hold for both the per-publisher and the global scheme. A verifier does not distinguish between the two types of hashes, beyond ensuring that the party who generated the parameters is trusted.

D. Computational Efficiency Improvements

We have presented a bare-bones protocol that achieves our security goals but is expensive in terms of bandwidth and computation. The hash function H_G is orders of magnitude slower than a more conventional hash function such as SHA1. Our goal here is to improve verification performance, so that a downloader can, at the very least, verify hashes as quickly as he can receive them from the network. The bare-bones primitives above imply that a client must essentially recompute the hash of the file $H_G(F)$, but without knowing $\mathbf{r}$.

We use a technique suggested by Bellare, Garay, and Rabin [19] to improve verification performance. Instead of verifying each check block $\mathbf{c}_i$ exactly, we verify them probabilistically and in batches. Each downloader picks a batch size t such as 256 blocks, and a security parameter l such as 32.

The downloader runs a probabilistic batch verifier given by $\mathcal{V}$. The algorithm takes as input the parameter array $(H_G(F'), G, X, C)$. As usual, $H_G(F')$ is the hash of the precoded file F' and G denotes the hash parameters. The $m \times t$ matrix C represents the batch of t check blocks that the downloader received; for convenience, we will write the decomposition $C = (\mathbf{c}_1 \ \cdots \ \mathbf{c}_t)$, where a column $\mathbf{c}_i$ of the matrix represents the i^{th} check block of the batch. The $m \times t$ matrix X is a sparse binary matrix. The cell x_{ij} should be set to 1 if the j^{th} check block contains the message block $\mathbf{b}_i$ and should be 0 otherwise. In other words, the j^{th} column of the matrix X is exactly $\mathbf{x}_j$.

Algorithm $\mathcal{V}(H_G(F'), G, X, C)$

1) Let $s_i \in \{0,1\}^l$ be chosen randomly for $0 < i \leq t$, and let the column vector $\mathbf{s} = (s_1, \ldots, s_t)$.
2) Compute column vector $\mathbf{z} = C\mathbf{s}$
3) Compute $\gamma_j = \prod_{i=0}^{n'} h_G(\mathbf{b}_i)^{x_{ij}}$ for all $j \in \{1, .., t\}$. Note that if the sender is honest, then $\gamma_j = h_G(\mathbf{c}_j)$.
4) Compute $y' = \prod_{i=1}^{m} g_i^{z_i}$, and $y = \prod_{j=1}^{t} \gamma_j^{s_j}$
5) Verify that $y' \equiv y \bmod p$

This algorithm is designed to circumvent the expensive computations of $h_G(\mathbf{c}_i)$ for check blocks in the batch. $\mathcal{V}$ performs an alternative and roughly equivalent computation with the product y in Step 4. The key optimization here is that the exponents s_j are small (l bits) compared to the much larger λ_q-bit exponents used in Equation 1.

Batching does open the receiver to small-scale attacks: a receiver accepts a batch worth of check blocks before closing a connection with a malicious sender. With our example parameters, each batch is 4 MB. However, a downloader can batch over multiple sources. Only once a batch fails to verify might the downloader attempt per-source batching to determine which source is sending corrupted check blocks. Finally, downloaders might tune the batching parameter t based upon their available bandwidth or gradually increase t for each source, so as to bound its overall fraction of bad blocks.

E. Homomorphic Hash Trees

As previously noted, hashes $H_G(F)$ are proportional in size to the file F and hence can grow quite large. With our sample hash parameters, an 8 GB file will have a 64 MB hash—a sizable file in and of itself. If a downloader were to use traditional techniques to download such a hash, he would be susceptible to the very same attacks we have set out to thwart, albeit on a smaller scale.

To solve this problem, we construct homomorphic hash trees—treating large hashes themselves as files, and repeatedly hashing until an acceptably small hash is output. We also use a traditional hash function such as SHA1 to reduce our hashes to standard 20-byte sizes, for convenient indexing at the network and systems levels.

First, pick a parameter L to represent the size of the largest hash that a user might download without the benefit of on-the-fly verification. A reasonable value for L might be 1 MB. Define the following:

$$\begin{aligned} H_G^0(F) &= F \\ H_G^i(F) &= H_G(H_G^{i-1}(F)) \text{ for } i > 0 \\ I_G(F) &= (G, j, H_G^j(F)) \\ &\qquad \text{for minimal } j \text{ such that } |I_G(F)| < L \\ J_G(F) &= \mathsf{SHA1}(I_G(F)) \end{aligned}$$

That is, $H_G^i(F)$ denotes i recursive applications of H_G. Note that J outputs hashes that are the standard 20 bytes in size. Now, the different components of the system are modified accordingly:

Filename to Hash Mappings. Lookup services map $N(F) \to J_G(F)$, for some G.

File Publication. To publish a file F, a publisher must compute the hashes chain of hashes $H_G^1(F), \ldots, H_G^j(F)$, and also the hashes $I_G(F)$ and $J_G(F)$. For $i \in \{0, ..., j-1\}$, the publisher stores $H_G^i(F)$ under the key $(J_G(F), i)$, and he additionally stores $I_G(F)$ under the key $(J_G(F), -)$.

File Download. To retrieve a file F, a downloader first performs the name-to-hash lookup $N(F) \to J_G(F)$, for some G. He then uses the peer-to-peer routing layer to determine a set of sources who serve the file and hashes corresponding to $J_G(F)$. The downloader queries one of the mirrors with the key $(J_G(F), -)$, and expects to receive $I_G(F)$. This transfer can be at most L big. Assuming the hash $J_G(F) = \mathsf{SHA1}(I_G(F))$ correctly verifies, the downloader knows the value j, and the j^{th} order hash $H_G^j(F)$. He can then request the next hash in the sequence simultaneously from all of the mirrors who serve F. The downloader queries these servers with the key $(J_G(F), j-1)$, expecting the hash $H_G^{j-1}(F)$ in response. This transfer also can be completed using erasure encoding and our hash verification protocol. The downloader iteratively queries its sources for lower-order hashes until it receives the 0^{th} order hash, or rather, the file itself.

In practice, it would be rare to see a j greater than 3. With our sample hash parameters, the third-order hash of a 512 GB file is a mere 32 KB. However, this scheme can scale to arbitrarily large files. Also note that because each application of the hash function cuts the size of the input by a factor of β/λ_p, the total overhead in hash transmission will be bounded below a small fractional multiple of the original file size, namely:

$$\begin{aligned} \frac{\text{overhead}}{\text{filesize}} &= \sum_{i=1}^{j} \left(\frac{\lambda_p}{\beta}\right)^i < \sum_{i=1}^{\infty} \left(\frac{\lambda_p}{\beta}\right)^i \\ &< \frac{1}{1 - \lambda_p/\beta} - 1 = \frac{\lambda_p}{\beta - \lambda_p} \end{aligned}$$

With our example parameters, $\lambda_p/(\beta - \lambda_p) \approx 0.79\%$.

V. Analysis

In this section, we analyze our hashing scheme and report performance numbers of a sample implementation.

A. Correctness

We first claim that homomorphic hashing scheme coupled with the batch verifier given in Section IV-D guarantees correctness. That is, a verifier should always accept the encoded output from an honest sender. Our proof is in Appendix I. Given the proof of this more involved probabilistic verifier, it is easy to see that the naïve verifier is also correct.

B. Running Time Analysis

In analyzing the running time of our algorithms, we count the number of multiplications over $\mathbb{Z}_p^*$ and $\mathbb{Z}_q$ needed. For instance, a typical exponentiation y^x in $\mathbb{Z}_p^*$ requires $1.5|x|$ multiplications using the "iterative squaring" technique. $|x|$ multiplications are needed to produce a table of values y^{2^z}, for all z such that $1 \le z < |x|$. Assuming data compression, half of the bits of x on average will be 1, thus requiring $|x|/2$ multiplications of values in the table. In our analysis, we denote $\mathsf{MultCost}(p)$ as the cost of multiplication in $\mathbb{Z}_p^*$, and $\mathsf{MultCost}(q)$ as the cost of multiplication in $\mathbb{Z}_q$.

Note that computations of the form $\prod_{i=1}^{m} g_i^{x_i}$ are computed at various stages of the different hashing protocols. As mentioned above, the precomputation of the $g_i^{2^z}$ requires $m\lambda_q$ multiplications over $\mathbb{Z}_p^*$. But the product itself can be computed in $(m\lambda_q/2)\,\mathsf{MultCost}(p)$ computations—and not the $(m\lambda_q/2 + m - 1)\,\mathsf{MultCost}(p)$ one might expect—by keeping a "running product."

We recognize that certain operations like modular squaring are cheaper than generic modular multiplication. Likewise, multiplying an element of $\mathbb{Z}_q$ by a 32-bit number is less expensive than multiplying two random elements from $\mathbb{Z}_q$. In our analysis, we disregard these optimizations and seek only simplified upper bounds.

Per-Publisher Hash Generation. Publishers first precompute a table g^{2^z} for all z such that $1 \le z < \lambda_q$. This table can then be used to compute $H_G(F)$ for any file F. Here and throughout this analysis, we can disregard the one-time precomputation, since $n \gg m$. Thus, the n-vector exponentiation in Equation 4 requires an expected $n\lambda_q/2$ multiplications in $\mathbb{Z}_p^*$. To compute $\mathbf{r}F$ as in Equation 4, mn

Algorithm $\mathsf{FastMult}$ $((y_1, s_1), \ldots, (y_t, s_t))$
$y \leftarrow 1$
for $j = l - 1$ **down to** 0 **do**
  for $i = 1$ to t **do**
    if $s_i[j] = 1$ **then** $y \leftarrow yy_i$
  done
  if $l > 0$ **then** $y \leftarrow y^2$
done
return y

Fig. 4. Algorithm for computing $\prod_{i=1}^{t} y_i^{s_i}$. Each s_i is an l-bit number, and the notation $s_i[j]$ gives the j^{th} bit of s_i, $s_i[0]$ being the least significant bit. This algorithm is presented in [19], although we believe there to be an off-by-one-error in that paper, which we have corrected here.

multiplications are needed in $\mathbb{Z}_q$. The total cost is therefore $mn\,\mathsf{MultCost}(q) + n\lambda_q\,\mathsf{MultCost}(p)/2$.

Global Hash Generation. Publishers using the global hashing scheme do not know $\mathbf{r}$ and hence must do multiple exponentiations per block. That is, they must explicitly compute the product given in Equation 1, with only the benefit of the precomputed squares of the g_i. If we ignore these costs, Global Hash Generation requires a total of $nm\lambda_q\,\mathsf{MultCost}(p)/2$ worth of computation.

Naïve Hash Verification. Hash verifiers who chose not to gain batching speed-ups perform much the same operations as the global hash generators. That is, they first precompute tables of squares, and then compute the left side of Equation 3 for the familiar cost of $m\lambda_q\,\mathsf{MultCost}(p)/2$. The right side of the equation necessitates an average of d multiplications in $\mathbb{Z}_p^*$, where d, we recall, is the average degree of a check block $\mathbf{c}$. Thus, the expected per-block cost is $(m\lambda_q/2+d)\mathsf{MultCost}(p)$.

Fast Hash Verification. We refer to the algorithm described in Section IV-D. In Step 2, recall that C is a $m \times t$ matrix, and hence the matrix multiplication costs $mt\,\mathsf{MultCost}(q)$. $\mathcal{V}$ determines γ_j in Step 3 with d multiplications over $\mathbb{Z}_p^*$, at a total cost of $td\,\mathsf{MultCost}(p)$. In Step 4, computing y' costs $m\lambda_q/2\,\mathsf{MultCost}(p)$ with access to precomputed tables of the form $g_i^{2^x}$. For y, no such precomputations exist; the bases in this case are γ_j, of which there are more than n. To compute y efficiently, we suggest the $\mathsf{FastMult}$ algorithm described in Figure 4, which costs $(tl/2 + l - 1)\,\mathsf{MultCost}(p)$.[2] Summing these computations and amortizing over the batch size t yields a per-block cost of:

$$m \cdot \mathsf{MultCost}(q) + \left[d + \frac{l}{2} + \frac{m\lambda_q/2 + l - 1}{t}\right] \cdot \mathsf{MultCost}(p)$$

C. Microbenchmarks

We implemented a version of these hash primitives using the GNU MP library, version 4.1.2. Table II shows the results of our C++ testing program when run on a 3.0 GHz Pentium 4, with the sample parameters given in Table I and the batching parameters given in Section IV-D. On this machine, $\mathsf{MultCost}(p) \approx 6.2$ μsecs and $\mathsf{MultCost}(q) \approx 1.0$ μsecs. Our results are reported in both cost per block and overall throughput. For comparison, we include similar computations for SHA1 and for the Rabin signature scheme with 1024-bit keys [20]. We also include disk bandwidth measurements for reading blocks off a Seagate 15K Cheetah SCSI disk drive (in batches of 64), and maximum theoretical packet arrival rate on a T1. We will call on these benchmarks in the next section.

Although batched verification of hashes is almost an order of magnitude slower than a more conventional hash function such as SHA1, it is still more than an order of magnitude faster than the maximum packet arrival rate on a good Internet connection. Furthermore, by adjusting the batch parameter t, downloaders can upper-bound the amount of *time* they waste receiving bad check blocks. That is, receivers with faster connections can afford to download more potentially bogus check blocks, and can therefore increase t (and thus verification throughput) accordingly.

Our current scheme for global hash generation is rather slow, but publishers with reasonable amounts of RAM can use k-ary exponentiation to achieve a four-fold speedup (see Appendix III for details). Our performance analysis focuses on the per-publisher scheme, which we believe to be better-suited for copyright-friendly distribution of bulk data.

D. Performance Comparison

In Section III-A, we discussed other strategies for on-the-fly verification of check blocks in peer-to-peer networks. We now describe these proposals in greater detail, to examine how our scheme compares in terms of bandwidth, storage, and computational requirements. There are three schemes in particular to consider:

High-Degree SHA1 Hash Tree. The publisher generates n/r check blocks, and then hashes each one. Since this collection of hashes might be quite large, the publisher uses the recursive scheme described in Section IV-E to reduce it to a manageable size. The publisher distributes the file, keyed by the root of the hash tree. Downloaders first retrieve all nodes in the hash tree and then can verify check blocks as they arrive.

Binary SHA1 Hash Tree. As before, the publisher generates n/r check blocks, but then computes a binary hash tree over all check blocks. The publisher keys the file by the root of its hash tree. In this scheme, mirrors need access to the entire hash tree, but clients do not. Rather, when the mirrors send check blocks, they prepend the "authentication path" describing the particular check block's location in the hash tree. If downloaders know the hash tree's root *a priori*, they can, given the correct authentication path, verify that a received check block is one of those intended by the publisher.

Sign Every Block. A publisher might generate n/r blocks and simply sign every one. The hash of the file is then the SHA1 of the filename and the publisher's public key. The mirrors must download and store these signatures, prepending them to check blocks before they are sent to remote clients. To retrieve the file, clients first obtain the publisher's public key

[2] $\mathsf{FastMult}$ offers no per-block performance improvement for naïve verification, thus we only consider it for fast verification.

TABLE II
MICROBENCHMARKS

Operation on 16 KB block $\mathbf{b}$	time (msec)	throughput (MB/sec)
Per-publisher computation of $h_G(\mathbf{b})$	1.39	11.21
Global computation of $h_G(\mathbf{b})$	420.90	0.037
Naïve verification of $h_G(\mathbf{b})$	431.82	0.038
Batched Verification of $h_G(\mathbf{b})$	2.05	7.62
$\mathsf{SHA1}(\mathbf{b})$	0.28	56.25
Sign $\mathbf{b}$ with Rabin-1024	1.98	7.89
Verify Rabin-1024 Signature of $\mathbf{b}$	0.29	53.88
Receiving $\mathbf{b}$ on a T1	83.33	0.186
Reading $\mathbf{b}$ from disk (sequentially)	0.27	57.87

from the network, and verify this key against the hash of the file. When they arrive from mirrors, the check blocks contain their own signatures and are thus easily verified.

These three schemes require a suitable value of r. For codes with rate r, a file with n message blocks will be expanded into n/r check blocks. For simple lower bounds, assume that any set of n of these check blocks suffices to reconstruct the file. In a multicast scenario, a client essentially collects these blocks at random, and the well-known "coupon collector bound" predicts that he will receive $-(n/r)\ln(1-r)$ check blocks on average before collecting n unique check blocks.[3] Using this bound, we can estimate the expected additional transmission overheads due to repeated check blocks:

r	$-(1/r)\ln(1-r)$
1/2	0.3863
1/4	0.1507
1/8	0.0683
1/16	0.0326
1/32	0.0160

That is, with an encoding rate $r=1/2$, a receiver expects an additional 39% overhead corresponding to duplicate blocks. In many-to-many transmission schemes, small encoding rates are essential to achieving good bandwidth utilization.

We now present a performance comparison of the three fixed-rate schemes and our homomorphic hashing proposal, focusing on key differences between them: hash generation costs incurred by the publisher, storage requirements at the mirror, bandwidth utilization between the mirror and downloader, and verification performance.

1) Hash Generation: Fixed-rate schemes such as the three presented above can generate signatures only as fast as they can generate check blocks. Encoding performance depends upon the file's size, but because we wish to generalize our results to very large files, we must assume that the publisher cannot store the entire input file (or output encoding) in main memory. Hence, secondary storage is required.

Our implementation experience with Online Codes has shown that the encoder works most efficiently if it stores the relevant pieces of the encoding graph structure and a fixed number of check blocks in main memory.[4] The encoder can make several sequential passes through the file. With each pass, it adds message blocks from disk into check blocks in memory, as determined by the encoding graph. As the pass is completed, it flushes the completed batch of check blocks to the network, to disk, or to functions that compute hashes or signatures. This technique exploits the fact that sequential reads from disk are much faster than random seeks.

Our current implementation of Online Codes can achieve encoding throughputs of about 21 MB/sec (on 1 GB files, using roughly 512 MB of memory). However, to better compare our system against fixed-rate schemes, we will assume that an encoder exists than can achieve the maximum possible throughput. This upper bound is $ae/(\beta n)$, where the file has n blocks, the block size is β, the amount of memory available for storing check blocks is a, and the disk's sequential read throughput is e.

When publishers use fixed-rate schemes to generate hashes, they must first precompute n/r check blocks. Using the encoder described above, this computation requires $n\beta/(ra)$ scans of the entire file. Moreover, each scan of the file involves n block reads, so $n^2\beta/(ra)$ block reads in total are required. Concurrent with these disk reads, the publisher computes hashes and signatures of the check blocks and the implied hash trees if necessary.

The theoretical requirements for all four schemes are summarized in Table III. In the final three columns, we have attempted to provide some concrete realizations of our theoretical bounds. Throughout, we assume (1) a 1 GB file, broken up into $n=2^{16}$ blocks, each of size $\beta=16$ KB, (2) the publisher has $a=512$ MB of memory for temporary storage of check blocks, and (3) disk throughputs can be sustained at 57.87 MB/sec as we observed on our machine. Under these conditions, an encoder can achieve theoretical encoding throughputs of up to 28.9 MB/sec. We further assume that (4) looking to keep overhead due to redundant check blocks below 5%, the publisher uses an encoding rate of $r=1/16$ and (5) a publisher can entirely overlap disk I/O and computations and therefore only cares about whichever takes longer. In the right-most column, we present reasonable lower bounds on hash generation performance for the four different schemes.

Despite our best efforts to envision a very fast encoder, the results for the three fixed-rate schemes are poor, largely due to the cost encoding of n/r file blocks. Moreover, in the sign-every-block scheme, CPU becomes the bottleneck due to the expense of signature computation.

By contrast, the homomorphic hashing scheme can escape excessive disk accesses, because it hashes data before it is encoded. It therefore requires only one scan of the input file to generate the hashes of the message blocks. The publisher's subsequent computation of the higher-level hashes $H^2(F), H^3(F), \ldots$ easily fit into memory. Our prototype can compute a homomorphic hash of a 1 GB file in 123.63 seconds, reasonably close to the lower bound of 91.81 seconds predicted in Table III.

Of course, performance for the three fixed-rate schemes worsens as r becomes smaller or n becomes larger. It is

[3] This asymptotic bound is within a 10^{-5} neighborhood of the exact probability when $n=2^{16}$.

[4] With little impact on performance, our implementation also stores auxiliary blocks in memory.

TABLE III

HASH GENERATION

Scheme	Block Reads	DLog Hashes	SHA1 Hashes	Sigs	Disk (sec)	CPU (sec)	Lower Bound (sec)
Homomorphic Hashing	n	$n\beta/(\beta-\lambda_p)$	1	1	17.69	91.81	91.81
Big-Degree SHA1 Hash Tree	$n^2\beta/(ra)$	0	$(n/r)\beta/(\beta-160)$	1	566.23	293.96	566.23
Binary SHA1 Hash Tree	$n^2\beta/(ra)$	0	$2n/r$	1	566.23	587.20	587.20
Sign Every Block	$n^2\beta/(ra)$	0	0	n/r	566.23	2076.18	2076.18

TABLE IV

ADDITIONAL STORAGE REQUIREMENTS FOR MIRRORS

Scheme	Overhead	Storage (MB)
Homomorphic Hash	0.008	8.06
Big-Degree Tree	0.020	20.02
Binary Tree	0.039	40.00
Sign Every Block	0.125	128.00

possible to ameliorate these problems by raising the block size β or by striping the file into several different files, but these schemes involve various trade-offs that are beyond the scope of this paper.

2) Mirror's Encoding Performance: In theory, the homomorphic hashing scheme renders encoding more computationally expensive because it substitutes XOR block addition for more expensive modular additions. We have measured that our machine computes the exclusive OR of two 16 KB check blocks in 8.5 μsecs. By comparison, our machine requires 37.4 μsecs to sum two blocks with modular arithmetic. The average check-block degree in our implementation of Online Codes is 8.17, so check-block generation on average requires 69.5 μsecs and 305 μsecs under the two types of addition. This translates to CPU-bound throughputs of 224.8 MB/sec and 51.3 MB/sec, respectively. However, recall that disk throughput and memory limitations combine to bound encoding for both schemes at only 28.9 MB/sec. Moreover, these throughputs are quite large relative to typical network throughput; many P2P-CDN mirror nodes would be happy with T1-rates at 1.5 Mbit/sec.

3) Storage Required on the Mirror: Mirrors participating in P2P-CDNs agree to donate disk space for content distribution, though usually they mirror files they also use themselves. All four verification schemes require additional storage for hashes and signatures. With homomorphic hashing, the mirror should store the hash that the publisher provides. Regenerating the hash is theoretically possible but computationally expensive. Similarly, mirrors in the two SHA1 hash tree schemes should retrieve complete hash trees from the publisher and store them to disk, or otherwise must dedicate tremendous amounts of disk I/O to generate them on-the-fly. Finally, in the sign-every-block scheme, the mirror does not know the publisher's private key and hence cannot generate signatures. He has no choice but to store all signatures. We summarize these additional storage requirements in Table IV, again assuming a 1 GB input file and an encoding rate of $r = 1/16$.

4) Bandwidth: The bandwidth requirements of the various schemes are given in terms of up-front and per-block costs. These results are considered in Table V. The new parameter λ_σ describes the size of signatures, which is 1024 bits in our examples. In multicast settings, receivers of fixed-rate codes incur additional overhead due to repeated blocks (reported as "penalty"). At an encoding rate of $r = 1/16$, the coupon collector bound predicts about 3.3% overhead. In all four schemes, downloaders might see duplicate blocks when reconciling partial transfers with other downloaders. That is, if two downloaders participate in the same multicast tree, and then try to exchange check blocks with each other, they will have many blocks in common. This unavoidable problem affects all four schemes equally and can be mitigated by general set-reconciliation algorithms [8] and protocols specific to peer-to-peer settings [9].

The binary SHA1 tree and the sign-every-block scheme allow downloaders to retrieve a file without up-front transfer of cryptographic meta-data. Of course, when downloaders become full mirrors, they cannot avoid this cost. In the former scheme, the downloader needs the hash tree in its entirety, adding an additional 3.9% overhead to its total transfer. In the latter, the downloader requests all those signatures not already received. This translates to roughly 11.7% additional overhead when $r = 1/16$.

5) Verification: Table VI summarizes the per-block verification costs of the four schemes. For our homomorphic hashing scheme, we assume batched verification with the parameters given in Section IV-D. The Rabin signature scheme was specially chosen due to its fast verification time, as shown. Surprisingly, verifying a check block using a SHA1 binary tree is more than twice as slow as using our homomorphic hashing protocol, due to the height of the tree.

E. Discussion

For encoding rates such as $r = 1/16$, each of the three fixed-rate schemes has important strengths and weaknesses. Though the sign-every-block scheme is bandwidth-efficient, requires no up-front hash transfer, and has good verification performance, its hash generation costs are prohibitive and its storage costs are higher. Similarly, though the binary hash tree method has no up-front transfer, its bandwidth, storage and verification costs make it less attractive than hash trees with larger fan-out. The homomorphic hashing scheme entails no such tradeoffs, as it performs well across all categories considered. Homomorphic hashing ranks less favorably when considering verification throughput, but as argued in Section V-C, tuning batch size allows throughput to scale with available bandwidth.

TABLE V
BANDWIDTH

Scheme	Up-Front Predicted (bits)	Up-Front *e.g.* (MB)	Per-Block Predicted (bits)	Per-Block *e.g.* (KB)	Total (GB)	Total w/ Penalty (GB)
Homomorphic Hashing	$\lambda_p n\beta/(\beta-\lambda_p)$	8.06	$\beta+m$	16.06	1.0118	1.0118
Big-Degree SHA1 Hash Tree	$160n\beta/(\beta-160)$	20.02	β	16.00	1.0196	1.0528
Binary SHA1 Hash Tree	0	0	$\beta+160\log_2(n/r)$	16.39	1.0244	1.0578
Sign Every Block	0	0	$\beta+\lambda_\sigma$	16.13	1.0078	1.0407

TABLE VI
PER-BLOCK VERIFICATION PERFORMANCE

Scheme	Batch	SHA1	Rabin	Total (msec)
Homomorphic Hash	1	0	0	2.05
Big-Degree Tree	0	1	0	0.28
Binary Tree	0	$\log_2(n/r)$	0	5.60
Sign Every Block	0	0	1	0.29

VI. SECURITY

In modern real-world P2P-CDNs, an honest receiver who wishes to obtain the file F from the network communicates almost exclusively with untrusted parties. As mentioned in Section III, a crucial stage of the file transmission protocol—mapping file names to file hashes—is beyond the scope of this paper. In our analysis, we assume that the receiver can reliably resolve $N(F) \rightarrow J_G(F)$ through a trusted, out-of-band channel. We wish to prove, however, that given $J_G(F)$, the downloader can recover F from the network while recognizing certain types of dishonest behavior almost immediately.

A. Collision-Resistant Hash Functions

First, we formally define a collision-resistant hash function (CRHF) in the manner of [21]. Recall that a family of hash functions is given by a pair of PPT algorithms $\mathcal{F} = (\mathsf{HGen}, \mathcal{H})$. HGen denotes a hash generator function, taking an input of security parameters $(\lambda_p, \lambda_q, m)$ and outputting a description of a member of the hash family, G. $\mathcal{H}_G$ will hash inputs of size $m\lambda_q$ to outputs of size λ_p, exactly as we have seen thus far. A hash adversary $\mathcal{A}$ is a probabilistic algorithm that attempts to find collisions for the given function family.

Definition 1: For any CRHF family $\mathcal{F}$, any probabilistic algorithm $\mathcal{A}$, and security parameter $\lambda = (\lambda_p, \lambda_q, m)$ where $\lambda_q < \lambda_p$ and $m \leq \mathrm{poly}(\lambda_p)$, let

$$\begin{aligned}\mathsf{Adv}^{\text{col-atk}}_{\mathcal{F},\lambda}(\mathcal{A}) = \Pr\big[&G \leftarrow \mathsf{HGen}(\lambda); (x_1, x_2) \leftarrow \mathcal{A}(G) : \\ &\mathcal{H}_G(x_1) = \mathcal{H}_G(x_2) \ \wedge\ x_1 \neq x_2\big]\end{aligned}$$

$\mathcal{F}$ is a (τ, ε)-secure hash function family if, for all PPT adversaries $\mathcal{A}$ with time-complexity $\tau(\lambda)$, $\mathsf{Adv}^{\text{col-atk}}_{\mathcal{F},\lambda}(\mathcal{A}) < \varepsilon(\lambda)$, where $\varepsilon(\lambda)$ is negligible in λ and $\tau(\lambda)$ is polynomial in λ.

Our definition of the hash primitive h per Section IV fits naturally into this definition. In fact, the PickGroup algorithm is a reasonable candidate for the function HGen(). See [21] for a proof that the function family h_G is collision-resistant hash function, assuming that the discrete log problem is hard over the group parameterized by (λ_p, λ_q).

B. Security of Encoding Verifiers

We can now define a notion of security against bogus-encoding attacks. For simplicity, we assume erasure codes that have a precoding algorithm $\mathcal{P}$, and an encoder amenable to succinct matrix representation; as discussed in Section II, examples include LT, Raptor, and Online Codes.

As usual, consider an adversary $\mathcal{A}$ against an honest verifier $\mathcal{V}$. The adversary $\mathcal{A}$ succeeds in a bogus-encoding attack if he can convince the verifier $\mathcal{V}$ to accept blocks from "forged" or bogus file encodings. When making the decision of whether or not to accept a given encoding, $\mathcal{V}$ can only access the hash $H_G(F')$ of the precoded file F' he expects. In this definition, the adversary has the power to generate the file F, which is the precoded as normal to obtain F'.

Definition 2 (Secure Encoding Verifier): For any CRHF $\mathcal{H}$, any probabilistic algorithm $\mathcal{A}$, any honest verifier $\mathcal{V}$, any $m, n > 0$, any batch size $t > 1$, let:

$$\begin{aligned}&\mathsf{Adv}^{\text{enc-atk}}_{\mathcal{H},\mathcal{V},m,n,t}(\mathcal{A}) = \\ &\quad \Pr\big[G \leftarrow \mathsf{HGen}(\mathcal{H});\ (F, X, C) \leftarrow \mathcal{A}(G, m, n, t); \\ &\qquad F' \leftarrow \mathcal{P}(F);\ b \leftarrow \mathcal{V}(\mathcal{H}_G(F'), G, X, C) : \\ &\qquad F \text{ is } m \times n \ \wedge\ F' \text{ is } m \times n' \ \wedge\ X \text{ is } n' \times t \\ &\qquad\ \wedge\ C \text{ is } m \times t \ \wedge\ F'X \neq C \ \wedge\ b = \mathsf{Accept}\big]\end{aligned} \tag{5}$$

The encoding verifier $\mathcal{V}$ is (τ, ε)-secure if, $\forall\ m, n > 0$, $t > 1$ and PPT adversaries $\mathcal{A}$ with time-complexity $\tau(m, n, t)$, $\mathsf{Adv}^{\text{enc-atk}}_{\mathcal{H},\mathcal{V},m,n,t}(\mathcal{A}) < \varepsilon(m, n, t)$.

Our definition requires that $\mathcal{V}$ be (τ, ε)-secure for all values of $t > 1$. Thus, a protocol that uses a secure encoding verifier can tune t as desired to trade computational efficiency for communication overhead. From here, we can prove that the batch verification procedure presented previously is secure. See Appendix II.

Theorem 1: Given security parameters l, λ_p, λ_q, batch size t, number of generators m, and the (τ, ε)-secure hash family h generated by $(\lambda_q, \lambda_p, m)$, the batched verification procedure $\mathcal{V}$ given above is a (τ', ε')-secure encoding verifier, where $\tau' = \tau - mt(\mathsf{MultCost}(q) + \mathsf{MultCost}(p))$ and $\varepsilon' = \varepsilon + 2^{-l}$.

We do not state or prove the corresponding theorem for the naïve verifier, but it is straightforward to check that it has equivalent or stronger properties than that of the batch verifier. The security of the recursive hashing scheme outlined in Section IV-E follows from an inductive application of Theorem 1.

C. Future Work and End-To-End Security

These security guarantees, while necessary, are not sufficient for all multicast settings. In Section III, we proposed both the bogus-encoding attack and the distribution attack. While we have solved the former, one can imagine malicious encoders who thwart the decoding process through an incorrect distribution of well-formed check blocks. Because Tornado, Raptor, Online, and LT Codes are all based on irregular graphs, their output symbols are not interchangeable. Bad encoders could corrupt degree distributions; they could also purposefully avoid outputting check blocks derived from some particular set of message blocks. Indeed, the homomorphic hashing scheme and the three fixed-rate schemes discussed in Section V-D are all vulnerable to the distribution attack.

In future work, we hope to satisfy a truly end-to-end definition of security for encoding schemes. For the end-to-end model, we envision an experiment in which the adversary can chose to supply the recipient with either its own check blocks, or those from an honest encoder. The X and C parameters of the verifier function now correspond to the entire download history, not just the most recent batch of blocks. The verifier outputs Reject if it believes it is talking to a malicious encoder, in which case the experiment discards the batch of blocks just received. In the end, the experiment runs the decoder on all retained check blocks after receiving $(1+\epsilon)n' + aB$ total blocks, where a is a constant allowance for wasted bandwidth per bad encoder, and B is the number of times the verifier correctly output Reject after receiving blocks from the adversary. The adversary succeeds if this decoding fails with non-negligible probability.

One approach toward satisfying such a definition might be to require a sender to commit to a pseudo-random sequence determined by a succinct seed, and then send check blocks whose x_i portions are entirely determined by the pseudo-random sequence. But in the context of non-reliable network transport or multicast "downsampling," a malicious sender can drop particular blocks in the sequence and place the blame on congestion. If, for example, the sender drops all degree-one blocks, or drops all check blocks that mention a particular message block, decoding will never succeed.

A more promising approach involves validating an existing set of check blocks by simulating the receipt of future check blocks. Given an existing set of check blocks $\langle \mathbf{x}_1, \mathbf{c}_1 \rangle, \langle \mathbf{x}_2, \mathbf{c}_2 \rangle, \ldots, \langle \mathbf{x}_Q, \mathbf{c}_Q \rangle$, the verifier can run the encoder (without the contents of F) to generate a stream of block descriptions $\mathbf{x}_{Q+1}, \mathbf{x}_{Q+2}, \ldots$. If the file would not be recoverable given $\mathbf{c}_{Q+1}, \mathbf{c}_{Q+2}, \ldots$, this is evidence that the distribution of $\mathbf{x}_1, \ldots, \mathbf{x}_Q$ has been skewed. If the file would be recoverable, the verifier can repeat the experiment several times to amplify its confidence in $\mathbf{x}_1, \ldots, \mathbf{x}_Q$. To be of any use, such a verifier can do no more than $O(\log n)$ operations per check block received. Thus simulated streams should be re-used for efficiency, with the effects of the first simulated block $\mathbf{x}_{Q+1}$ replaced by those of the next real block received. The feasibility of efficiently "undoing" encoding remains an open question; therefore we leave the description and analysis of an exact algorithm to future work.

VII. Related Works

Multicast source-authentication is well-studied problem in the recent literature; for a taxonomy of security concerns and some schemes, see [22]. Preexisting solutions fall into two broad categories: (1) sharing secret keys among all participants and MACing each block, or (2) using asymmetric cryptography to authenticate each block sent. Unfortunately, the former lacks any source authentication, while the latter is costly with respect to both computation resources and bandwidth.

A number of papers have looked at providing source authentication via public key cryptography, yet amortizing asymmetric operations over several blocks. Gennaro and Rohatgi [23] propose a protocol for stream signatures, which follows an initial public-key signature with a chain of efficient one-time signatures, although it does not handle block erasures (*e.g.*, from packet loss). Wong and Lam [24] delay consecutive packets into a pool, then form an authentication hash and sign the tree's root. Rohatgi [25] uses reduced-size online/offline k-time signatures instead of hashes. Recent tree-based [26] and graph-based [27] approaches reduce the time/space overheads and are designed for bursty communication and random packet loss. More recent work [28], [29] makes use of *trusted* erasure encoding in order to authenticate blocks, while most schemes, including our own, try to authenticate blocks in spite of *untrusted* erasure encoding.

Another body of work is based solely on symmetric key operations or hash functions for real-time applications. Several protocols used the delayed disclosure of symmetric keys to provide source authentication, including Chueng [30], the Guy Fawkes protocol [31], and more recently TESLA [32], [33], by relying on loose time synchronization between senders and recipients. The recent BiBa [34] protocol exploits the birthday paradox to generate one-time signatures from k-wise hash collisions. The latter two can withstand arbitrary packet loss; indeed, they were explicitly developed for Digital Fountain's content distribution system [6], [7] to support video-on-demand and other similar applications. Unfortunately, these delayed-disclosure key schemes require that publishers remain online during transmission.

In the traditional settings considered above, the publisher and the encoder are one in the same. In our P2P-CDN setting, untrusted mirrors generate the check blocks; moreover a trusted publisher cannot explicitly authenticate every possible check block, since their number grows exponentially with file size. Thus, a publisher must generate its authentication tokens on the initial message blocks, and we require a hash function that preserves the group structure of the encoding process.

Our basic homomorphic hashing scheme is complementary to existing threads of work that make use of homomorphic group operations. One-way accumulators [35], [36] and incremental hashing [21], based on RSA and DL constructions respectively, examine commutative hash functions that yield an output independent of the operations' order. Improvements to the schemes' efficiency [37], [38], [39], however, largely focus on dynamic or incremental changes to the elements being hashed/authenticated, *e.g.*, the modification of an entry of an authenticated dictionary. More recent work has investigated homomorphic signature schemes for specific applications: undirected transitive signatures [40], authenticated prefix aggregation [41], redactable signatures [42], and set-union signatures via accumulators [42]. We use similar techniques to maintain group structure across applications of cryptographic functions, but to different ends. Composing homomorphic signatures with traditional hash functions such as SHA1 [13] would not solve our problem, as the application of the traditional hash function would destroy the group structure we hope to preserve.

VIII. Conclusions

Current peer-to-peer content distribution networks, such as the widely popular file-sharing systems, suffer from unverified downloads. A participant may download an entire file, increasingly in the hundreds of megabytes, before determining that the file is corrupted or mislabeled. Current downloading techniques can use simple cryptographic primitives such as signatures and hash trees to authenticate data. However, these approaches are not efficient for low encoding rates, and are not possible for rateless codes.

To our knowledge, this paper is the first to consider non-interactive, on-the-fly verification of rateless erasure codes. We present a discrete-log-based hash scheme that provides useful homomorphic properties for verifying the integrity of downloaded content. Because recipients can compose hashes just as encoders compose message blocks, they can verify any possible check block. Using batching techniques to improve verification efficiency, we provide implementation results that suggest this scheme is practical for real-world use. A tight reduction proves our scheme secure under standard cryptographic assumptions. We leave formalization of end-to-end security and protection against distribution attacks as interesting open problems.

Acknowledgments

We thank Michael Walfish for first alerting us to the distribution attack. We also thank Petar Maymounkov and Benny Pinkas for helpful discussions, and our shepherd Dan Wallach for his feedback. This research was conducted as part of the IRIS project (`http://project-iris.net/`), supported by the NSF under Cooperative Agreement No. ANI-0225660. Maxwell Krohn is supported by an MIT EECS Fellowship, Michael Freedman by an NDSEG Fellowship, and David Mazières by an Alfred P. Sloan Research Fellowship.

References

[1] S. Saroui, K. P. Gummadi, R. J. Dunn, S. D. Gribble, and H. M. Levy, "An analysis of Internet content delivery systems," in *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Oct. 2002.

[2] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann, "Practical loss-resilient codes," in *Proc. 29th Annual ACM Symposium on Theory of Computing (STOC)*, El Paso, TX, May 1997.

[3] M. Luby, "LT codes," in *Proc. 43rd Annual Symposium on Foundations of Computer Science (FOCS)*, Vancouver, Canada, Nov. 2002.

[4] P. Maymounkov, "Online codes," NYU, Tech. Rep. 2002-833, Nov. 2002.

[5] A. Shokrollahi, "Raptor codes," Digital Fountain, Inc., Tech. Rep. DF2003-06-001, June 2003.

[6] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege, "A Digital Fountain approach to reliable distribution of bulk data," in *Proc. ACM SIGCOMM '98*, Vancouver, Canada, Sept. 1998.

[7] J. Byers, M. Luby, and M. Mitzenmacher, "Accessing multiple mirror sites in parallel: Using Tornado codes to speed up downloads," in *Proc. IEEE INFOCOM '99*, New York, NY, Mar. 1999.

[8] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost, "Informed content delivery across adaptive overlay networks," in *Proc. ACM SIGCOMM '02*, Aug. 2002.

[9] P. Maymounkov and D. Mazières, "Rateless codes and big downloads," in *Proc. 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, Feb. 2003.

[10] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "Splitstream: High-bandwidth multicast in a cooperative environment," in *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton's Landing, NY, Oct. 2003.

[11] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat, "Bullet: High bandwidth data dissemination using an overlay mesh," in *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton's Landing, NY, Oct. 2003.

[12] L. Rizzo, "Effective erasure codes for reliable computer communication protocols," *ACM Computer Communication Review*, vol. 27, no. 2, Apr. 1997.

[13] FIPS 180-1, *Secure Hash Standard*, U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, Apr. 1995.

[14] C. Karlof, N. Sastry, Y. Li, A. Perrig, and J. Tygar, "Distillation codes and applications to DoS resistant multicast authentication," in *Proc. 11th Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, Feb. 2004.

[15] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology—CRYPTO '87*, Santa Barbara, CA, Aug. 1987.

[16] National Institute of Standards and Technology, "Digital Signature Standard (DSS)," Federal Information Processing Standards Publication 186-2, U.S. Dept. of Commerce/NIST, 2000.

[17] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *Advances in Cryptology—CRYPTO '91*, Santa Barbara, CA, Aug. 1991.

[18] D. Chaum, E. van Heijst, and B. Pfitzmann, "Cryptographically strong undeniable signatures, unconditionally secure for the signer," in *Advances in Cryptology—CRYPTO '91*, Santa Barbara, CA, Aug. 1991.

[19] M. Bellare, J. Garay, and T. Rabin, "Fast batch verification for modular exponentiation and digital signatures," in *Advances in Cryptology—EUROCRYPT 98*, Helsinki, Finland, May 1998.

[20] M. O. Rabin, "Digitalized signatures and public key functions as intractable as factorization," MIT Laboratory for Computer Science, Tech. Rep. TR-212, Jan. 1979.

[21] M. Bellare, O. Goldreich, and S. Goldwasser, "Incremental cryptography: The case of hashing and signing," in *Advances in Cryptology—CRYPTO '94*, Santa Barbara, CA, Aug. 1994.

[22] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas, "Multicast security: A taxonomy and some efficient constructions," in *Proc. IEEE INFOCOM '99*, New York, NY, 1999.

[23] R. Gennaro and P. Rohatgi, "How to sign digital streams," in *Advances in Cryptology—CRYPTO '97*, Santa Barbara, CA, Aug. 1997.

[24] C. K. Wong and S. S. Lam, "Digital signatures for flows and multicasts," in *Proc. IEEE International Conference on Network Protocols*, Austin, TX, Oct. 1998.

[25] P. Rohatgi, "A compact and fast hybrid signature scheme for multicast packet authentication," in *Proc. 6th ACM Conference on Computer and Communication Security (CCS)*, Singapore, Nov. 1999.

[26] P. Golle and N. Modadugu, "Authenticated streamed data in the presernce of random packet loss," in *Proc. Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2001.

[27] S. Miner and J. Staddon, "Graph-based authentication of digital streams," in *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, May 2000.

[28] A. Pannetrat and R. Molva, "Efficient multicast packet authentication," in *Proc. Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2003.

[29] J. M. Park, E. K. P. Chong, and H. J. Siegel, "Efficient multicast stream authentication using erasure codes," *ACM Trans. Inf. Syst. Secur.*, vol. 6, no. 2, 2003.

[30] S. Cheung, "An efficient message authentication scheme for link state routing," in *Proc. 13th Annual Computer Security Applications Conference*, San Diego, CA, Dec. 1997.

[31] R. Anderson, F. Bergadano, B. Crispo, J.-H. Lee, C. Manifavas, and R. Needham, "A new family of authentication protocols," *Operating Systems Review*, vol. 32, no. 4, Oct. 1998.

[32] A. Perrig, R. Canetti, D. Song, and D. Tygar, "Efficient authentication and signature of multicast streams over lossy channels," in *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, May 2000.

[33] ——, "Efficient and secure source authentication for multicast," in *Proc. Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2001.

[34] A. Perrig, "The BiBa one-time signature and broadcast authentication protocol," in *Proc. 8th ACM Conference on Computer and Communication Security (CCS)*, Philadelphia, PA, Nov. 2001.

[35] J. Benaloh and M. de Mare, "One-way accumulators: A decentralized alternative to digital sinatures," in *Advances in Cryptology—EUROCRYPT 93*, Lofthus, Norway, May 1993.

[36] N. Barić and B. Pfitzmann, "Collision-free accumulators and fail-stop signature schemes without trees," in *Advances in Cryptology—EUROCRYPT 97*, Konstanz, Germany, May 1997.

[37] M. Bellare and D. Micciancio, "A new paradigm for collision-free hashing: Incrementality at reduced cost," in *Advances in Cryptology—EUROCRYPT 97*, Konstanz, Germany, May 1997.

[38] J. Camenisch and A. Lysyanskaya, "Dynamic accumulators and application to efficient revocation of anonymous credentials," in *Advances in Cryptology—CRYPTO 2002*, Santa Barbara, CA, Aug. 2002.

[39] G. Tsudik and S. Xu, "Accumulating composites and improved group signing," in *Advances in Cryptology—ASIACRYPT-2003*, Taipei, Taiwan, Nov. 2003.

[40] S. Micali and R. Rivest, "Transitive signature schemes," in *Progress in Cryptology — CT-RSA 2002*, San Jose, CA, Feb. 2002.

[41] S. Chari, T. Rabin, and R. Rivest, "An efficient signature scheme for route aggregation," Feb. 2002.

[42] R. Johnson, D. Molnar, D. Song, and D. Wagner, "Homomorphic signature schemes," in *Progress in Cryptology — CT-RSA 2002*, San Jose, CA, Feb. 2002.

Appendix I
Correctness of Batched Verification

Consider the batched verification algorithm given in Section IV-D. To prove it correct (*i.e.*, that correct check blocks will be validated), let us examine an arbitrary hash $(G, H_G(F))$. For notational convenience, we write y and y' computed in Step 4 in terms of an element $g \in \mathbb{Z}_p$ of order q and row vector $\mathbf{r}$ such that $g^{\mathbf{r}} = \mathbf{g} \bmod p$. These elements are guaranteed to exist, even if they cannot be computed efficiently. Thus,

$$y' = \prod_{i=1}^{m} g_i^{z_i} = \prod_{i=1}^{m} g^{r_i z_i} = g^{\sum_{i=1}^{m} z_i r_i} = g^{\mathbf{rz}}$$

By the definition of $\mathbf{z}$ from Step 2, we conclude $y' = g^{\mathbf{r}C\mathbf{s}}$.

Now we examine the other side of the verification, y. Recalling Equation 1, rewrite hashes of check blocks in terms of a common generator g:

$$h_G(\mathbf{c}_j) = \prod_{i=1}^{m} g^{r_i c_{i,j}} = g^{\sum_{i=1}^{m} r_i c_{i,j}} = g^{\mathbf{r}\mathbf{c}_j}$$

As noted in Step 3, for an honest sender, $\gamma_j = h_G(\mathbf{c}_j)$. Thus, we can write that $\gamma_j = g^{s_j \mathbf{r}\mathbf{c}_j}$. Combining with the computation of y in Step 4:

$$y = \prod_{j=1}^{t} g^{s_j \mathbf{r}\mathbf{c}_j} = g^{\sum_{j=1}^{t} s_j \mathbf{r}\mathbf{c}_j} = g^{\mathbf{r}C\mathbf{s}}$$

Thus we have that $y' \equiv y \bmod p$, proving the correctness of the validator.

Appendix II
Proof of Theorem 1

We now prove the security of the batched verification scheme by proving Theorem 1 given in Section VI-B. Our proof follows that from [19], with some additional complexity due to our multi-dimensional representation of a file.

Consider the hash function family h parameterized by $(\lambda_p, \lambda_q, m)$. For any file size n, batch size $t < n$, consider an arbitrary adversary $\mathcal{A}'$ that (τ', ε')-attacks the encoding verifier $\mathcal{V}$. Based on this adversary, define a CRHF-adversary $\mathcal{A}(G)$ that works as follows:

Algorithm $\mathcal{A}(G)$
1) $(F, X, C) \leftarrow \mathcal{A}'(G, m, n, t)$
2) If F is not $m \times n$ or X is not $n' \times t$ or C is not $m \times t$ then Fail.
3) $F' \leftarrow \mathcal{P}(F)$
4) If $F'X = C$, then Fail
5) If $\mathcal{V}(H_G(F'), G, X, C) =$ Reject, then Fail.
6) If $H_G(F'X) \neq H_G(C)$, then Fail.
7) Find a column j such that $F'\mathbf{x}_j \neq \mathbf{c}_j$. Return $(F'\mathbf{x}_j, \mathbf{c}_j)$.

By our selection of the adversary $\mathcal{A}'$, running it in Step 1 will require time complexity τ' and will succeed in the experiment given in Definition 2 with probability ε'. By construction, $\mathcal{A}$ corresponds naturally to the steps of our definitional experiment in Equation 5. Step 2 enforces appropriate dimensionality. Step 4 enforces the requirements that $\langle X, C \rangle$ not be a legal encoding, given in Equation 5 by $F'X \neq C$. Step 5 requires that the verifier $\mathcal{V}$ accepts the "forged" input. We can conclude that the Algorithm $\mathcal{A}$ will arrive at Step 6 with probability ε'.

We now argue that $\mathcal{A}$ fails at Step 6 with probability 2^{-l}. To arrive at this step, the verifier $\mathcal{V}$ as defined in Section IV-D must have output Accept. Using the same manipulations as those given in Appendix I, we take the fact that $\mathcal{V}$ accepted to mean that:

$$g^{\mathbf{r}F'X\mathbf{s}} \equiv g^{\mathbf{r}C\mathbf{s}} \bmod p \quad (6)$$

Note that the exponents on both sides of the equation are scalars. Because g has order q, we can say that these exponents are equivalent $\bmod q$; that is $\mathbf{r}F'X\mathbf{s} \equiv \mathbf{r}C\mathbf{s} \bmod q$, and rearranging,

$$\mathbf{r}\,(F'X - C)\,\mathbf{s} \equiv 0 \bmod q. \quad (7)$$

If the algorithm $\mathcal{A}'$ fails at Step 6, then $H_G(F'X) \neq H_G(C)$. Rewriting these row vectors in terms of the g and $\mathbf{r}$, we have that $g^{\mathbf{r}F'X} \not\equiv g^{\mathbf{r}C} \bmod p$. Recalling that g is order q and that exponentiation of a scalar by a row vector is defined component-wise, we can write that $\mathbf{r}F'X \not\equiv \mathbf{r}C \bmod q$, and consequently:

$$\mathbf{r}\left(F'X - C\right) \not\equiv \mathbf{0} \bmod q \tag{8}$$

For convenience, let the $1 \times t$ row vector $\mathbf{u} = \mathbf{r}(F'X - C)$. Equation 8 gives us that $\mathbf{u} \not\equiv \mathbf{0} \bmod q$; thus some element of $\mathbf{u}$ must be non-zero. For simplicity of notation, say that u_1 is the first non-zero cell, but our analysis would hold for any index. Equation 7 gives us that $\mathbf{us} \equiv \mathbf{0} \bmod q$. Since $u_1 \neq 0$, it has a multiplicative inverse, u_1^{-1}, in $\mathbb{Z}_q^*$. Therefore:

$$s_1 \equiv -\left(u_1^{-1}\right) \sum_{j=2}^{t} u_j s_j \bmod q \tag{9}$$

Referring to Step 1 of verifier $\mathcal{V}$, s_1 was selected at random from 2^l possible values; consequently, the probability of its having the particular value in Equation 9 is at most 2^{-l}. Thus, $\mathcal{A}$ can fail at Step 6 with probability at most 2^{-l}.

Combining our results, we have that algorithm $\mathcal{A}$ will reach Step 7 with probability $\varepsilon' - 2^{-l}$. At this point in the algorithm, $\mathcal{A}$ is assured that $F'X \neq C$, since execution passed Step 4. If we consider this inequality column-wise, we conclude there must be some $j \in \{1, ..., t\}$ such that $F'\mathbf{x}_j \neq \mathbf{c}_j$, where $\mathbf{x}_j$ and $\mathbf{c}_j$ are the j^{th} columns of X and C, respectively. Because Step 6 guarantees that $H_G(F'X) = H_G(C)$ at this point in the algorithm, we can use the definition of H_G to claim that for all j, $h_G(F'\mathbf{x}_j) = h_G(\mathbf{c}_j)$. Thus, $(F'\mathbf{x}_j, \mathbf{c}_j)$ represents a hash collision for the hash function h_G.

Analyzing the time-complexity of $\mathcal{A}$, Step 1 completes with time-complexity τ', the matrix multiplication $F'X$ in Step 4 requires mt multiplications in $\mathbb{Z}_q$, and the hash computations in Step 6 each require $tm/2$ multiplications in $\mathbb{Z}_p^*$, assuming the usual precomputations. Therefore, $\mathcal{A}$ has a time complexity given by $\tau = \tau' + mt(\mathsf{MultCost}(q) + \mathsf{MultCost}(p))$.

Therefore, we have shown that if an adversary $\mathcal{A}'$ exists that is successful in a (τ', ε')-attack against $\mathcal{V}$, then another adversary $\mathcal{A}$ exists that is (τ, ε)-successful in finding collisions for the hash function h, where $\tau' = \tau - mt(\mathsf{MultCost}(q) + \mathsf{MultCost}(p))$ and $\varepsilon = \varepsilon' + 2^{-l}$. This completes the proof of Theorem 1. ∎

Appendix III
k-ary exponentiation

In order to speed up global hash generation, one can make an exponential space-for-time tradeoff, using k-ary exponentiation. That is, we can speed up each exponentiation by a factor of $x/2$ while costing a factor of $(2^x - 1)/x$ in core memory. For simplicity, assume that $x | (\lambda_q - 1)$:

1) For $1 \leq i \leq m$, for $0 < j < 2^x$, for $0 \leq k < (\lambda_q - 1)/x$, precompute $g_i^{j2^{kx}}$. Store each value in an array A under the index $A[i][j][k]$.
2) To compute $g_i^{z_i}$, write z_i in base 2^x:

$$z_i = a_0 + a_1 2^x + a_2 2^{2x} + \cdots + a_{(\lambda_q - 1)/x - 1} 2^{\lambda_q - x}$$

Let $K = \{k \mid a_k \neq 0\}$. Then compute the product:

$$g_i^{z_i} = \prod_{k \in K} A[i][a_k][k]$$

The storage requirement for the table A is $m(2^x - 1)(\lambda_q - 1)\lambda_p/x$ bits, which is exponential in x. Disregarding the one-time precomputation in Step 1, the computation of z_i in Step 2 costs $(\lambda_q - 1)\,\mathsf{MultCost}(p)/x$. Compared to the conventional iterative-squaring technique, this method achieves a factor of $x/2$ speed-up.

Setting $x = 8$, the size of the tables $|A| = 510$ MB, and we can hash a 1 GB file with global parameters in less than 2 hours (of course hashing is much faster in the per-publisher model).