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

Maxwell N. Krohn, Michael J. Freedman, David Mazières
New York University
{max,mfreed,dm}@cs.nyu.edu

Abstract

The quality of peer-to-peer content distribution can suffer from the malicious behavior of participants that corrupt or mislabel content. While systems using simple block-by-block downloading can verify blocks using traditional cryptographic signatures, these same techniques do not apply to more elegant systems that use rateless erasure codes for efficient multi-source, multicast download. This paper presents a practical scheme, based on homomorphic hashing, that enables a downloader to perform on-the-fly verification of erasure-encoded blocks.

1 Introduction

Peer-to-peer content distribution networks 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.

These exclusively unicast networks are furthermore extremely wasteful of bandwidth, in that a small number of files account for a sizable percentage of total transfers. Recent studies indicate that on the Kazaa network, the 300 top bandwidth-consuming objects can account for 42% of all outbound traffic [2]. Multicast transmission of popular files might drastically reduce the total bandwidth consumed; however, traditional multicast systems would not fare well in such unstable networks.

Developments in efficient *rateless erasure codes* [11, 12, 25] can point to elegant solutions for both of these problems. These codes allow a sender to encode a file as a nearly infinite, non-repeating stream of blocks; a receiver need only collect a random, fixed-size subset of these blocks to recover the original file. If senders and receivers agree to encode their files with a rateless erasure-encoding scheme, they can avoid the costly and complicated feedback protocols often needed to resume a truncated download or to maintain a reliable multicast tree. Receivers can furthermore collect blocks from multiple senders simultaneously. One can imagine an ideal situation in which M senders are transmitting the same file to N recipients in a "forest of multicast trees." No retransmissions are needed when receivers and senders leave and reenter the network, as they are likely to do.

There is a significant problem to this approach. When transferring erasure-encoded files, receivers can only "preview" their file at the very end of the transfer. It would be unacceptable for a receiving node to dedicate hours of bandwidth to a certain file transfer, only to find out that she was receiving a corrupted or mislabeled file all along.

Rateless erasure codes have been studied in the context of P2P-CDNs [14]. Erasure codes with fixed expansion factors have been applied to more centralized multicast content distribution networks [5, 4]. Other work has explored adding multicast capabilities to P2P-CDNs [7]. However, most previous work assumes honest source nodes. Modern peer-to-peer content distribution networks cannot make this assumption. This paper shows how to construct codes that allow recipients to detect invalid data streams immediately, without first consuming large amounts of bandwidth or polluting their download caches.

The Problem. Consider a simplified example, in which a receiver $\mathcal{R}$ wishes to obtain a file $\mathbf{F}$ from a peer-to-peer content distribution network (P2P-CDN). Most popular systems have $\mathcal{R}$ lookup $\mathbf{F}$ by its name $n(\mathbf{F})$, yielding a set of source nodes $(\mathcal{S}_1, \mathcal{S}_2, ..\mathcal{S}_n)$ who can transfer to $\mathcal{R}$ either $\mathbf{F}$ in its entirety, or portions of $\mathbf{F}$ for in the cases of multi-source and resumed download. Note that this abstraction applies equally well to unstructured systems (*e.g.*, Gnutella and Kazaa) and structured systems based on efficient distributed hash tables (*e.g.*, Chord [26], Pastry [24], and Kademlia [13]).

Assuming that the receiver and senders use rateless erasure codes, the source nodes would then proceed to "shower" $\mathcal{R}$ with randomly-encoded packets of $\mathbf{F}$, chosen from the very large (and for practical purposes, infinite) set $E(\mathbf{F})$, according to a given probability distribution. $\mathcal{R}$ would be able to recover the file as soon as it has received some number of packets only slightly larger than $|\mathbf{F}|$.

The P2P-CDN described above is vulnerable to host of different attacks:

1. **Content Mislabeling.** The original lookup incorrectly mapped $n(\mathbf{F}) \rightarrow \mathbf{F}'$. This might be the result of a malicious user of the system trying to promote his own content; it might otherwise be the result of innocent user error. Either way $\mathcal{R}$ will request and receive the file $\mathbf{F}'$ from the network, even though she expected $\mathbf{F}$.

2. **Encoding Corruption Attacks.** One or more of the senders $\mathcal{S}_i$ might send packets to $\mathcal{R}$ not from the expected encoding $E(\mathbf{F})$ but rather from some other maliciously-chosen encoding $E'(\mathbf{F})$, with the intent of thwarting $\mathcal{R}$'s decoding of $\mathbf{F}$. Unlike error-correcting codes, erasure codes cannot recover correct output symbols from corrupted input symbols.

3. **Bandwidth Wasting Attack.** As before, a malicious $\mathcal{S}_i$ sends packets to $\mathcal{R}$ from some other encoding $E'(\mathbf{F})$, but now with the intent of wasting $\mathcal{R}$'s network resources. $\mathcal{S}_i$ figures that $\mathcal{R}$ can only begin to decode the file $\mathbf{F}$ at the very end of the transfer, and hence, even if $\mathcal{R}$ can correctly discern which packets are members of $E(\mathbf{F})$ and which are not, she still will have received a potentially massive amount of useless data.

4. **Distribution Attack.** A malicious $\mathcal{S}_i$ will send $\mathcal{R}$ packets from $E(\mathbf{F})$, not according to the probability distribution required by the coding scheme, but rather from a maliciously-chosen one. As a result, $\mathcal{R}$ might experience degenerate behavior when trying to recover $\mathbf{F}$.

In order for $\mathcal{R}$ to find multiple download sources of $\mathbf{F}$, it makes sense for a P2P-CDN to add a layer of indirection and track files by content hash. This technique, already employed by Kazaa, among others, requires another layer of indexing for file names. $\mathcal{R}$ first maps $n(\mathbf{F}) \rightarrow \mathcal{H}(\mathbf{F})$, for some is a collision-resistant hash function $\mathcal{H}$, then locates and downloads $\mathbf{F}$ based on this hash value.

Malicious content-mislabeling is already a noticeable problem in peer-to-peer networks. In part because systems give users incentive to upload as much data as possible, a popular file name may resolve to dozens of hashes, a fraction of which are appropriately named and the remainder of which correspond to users who do not have the file but nonetheless wish to gain upload credit. A number of solutions exist, ranging from simply downloading the most widely replicated hash (on the assumption that people will keep the file if it is valid), to obtaining hashes from a trusted web page, to more complex reputation based-schemes. The mechanics of these solutions are beyond the scope this paper, however.

This work focuses on the other three problems. That is, assume that $\mathcal{R}$ can correctly map $n(\mathbf{F}) \rightarrow \mathcal{H}(\mathbf{F})$. She then looks up $\mathcal{H}(\mathbf{F})$ and obtains a list of servers $(\mathcal{S}_1, .., \mathcal{S}_n)$ who can transfer packets from the encoding $E(\mathbf{F})$. Knowing $\mathcal{H}(\mathbf{F})$, how can $\mathcal{R}$ verify the packets that she receives from $\mathcal{S}_i$ are actually from the encoding $E(\mathbf{F})$, chosen according to the appropriate probability distribution? Moreover, how can she do this *on-the-fly*, with each packet received?

We propose verifying erasure-encoded blocks through the use of a discrete-log-based, collision-resistant, homomorphic hash function. That is, given the hash $\mathcal{H}(\mathbf{F})$ of the original file, $\mathcal{R}$ has the ability to compute the hash value of any possible encoding block in $E(\mathbf{F})$, and can then use hash to verify the actual block. Our hash function can be efficiently verified using probabilistic batch verification, and has provable security under the discrete log assumption. We furthermore present implementation results that suggest this scheme is practical for real-world use.

The remainder of this paper is organized as follows: Section 2 describes related work, Section 3 reviews erasure codes, Section 4 describes our scheme, Section 5 analyzes its security and performance, including implementation results, and Section 6 concludes.

2 Related Works

Multicast source-authentication is well-studied problem in the recent literature; for a taxonomy of security concerns and some schemes, see [6]. The naïve solution is either to have a secret key shared among all participants used to MAC each packet, or to have the sender use an asymmetric key to sign each packet of the stream. 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 packets. Gennaro and Rohatgi [9] 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 packet loss. Wong and Lam [27] delay consecutive packets into a pool, then form an authentication hash and sign the tree's root. Rohatgi [23] uses reduced-size online/offline k-time signatures instead of hashes. More recent tree-based [10] and graph-based [15] approaches reduce the time/space overheads and are explicitly designed for bursty communication in addition to random packet loss.

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 [8], the Guy Fawkes protocol [1], and more recently TESLA [20, 21], by relying on loose time synchronization between senders and recipients. The recent BiBa [19] 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 [5, 4] to support video-on-demand and other similar applications.

None of these protocols provide a solution to our problem: They require that the trusted publisher explicitly authenticate every packet. Additionally, the delayed-disclosure key schemes require that publishers remain online during transmission. However, our system uses erasure codes with effectively *infinite expansion*: the publisher of the hash could not possibly prepare an authentication token (even using efficient hash trees) for every possible check block, since the number of possible check blocks explodes exponentially with the size of the file. Furthermore, we cannot require that the original publisher of the hash remain online indefinitely; hashes should be verifiable without any online interaction.

We require a mechanism where, given some authentication tokens for the initial file blocks generated by the sender, a valid encoding of file blocks into check blocks can be accompanied by derived authentication tokens that can be validated in real-time.

3 Brief Review of Erasure Codes

Consider a large file $\mathbf{F}$, comprised of n uniformly-sized blocks, $B_1||B_2||\ldots||B_n$, known as *message blocks*. Many efficient erasure-encoding schemes [4, 11, 12, 25] use the bitwise XOR operation ($\oplus$) over message blocks to produce *check blocks*. For instance, a check block C_1 might be defined as $C_1 = B_2 \oplus B_{10}$. A check block's *degree* is defined to be the number of message blocks that compose it. In this example, the degree of

name	description	*e.g.*
λ_p	discrete log security parameter	1024
λ_q	discrete log security parameter	257
p	random prime, $\|p\| = \lambda_p$	
q	random prime, $q\|(p-1)$, $\|q\| = \lambda_q$	
β	block size in bits	131702 (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 elements in $\mathbb{Z}_p$	
G	hash parameters, given by $(p, q, \mathbf{g})$	
d	average degree of check blocks	~ 10

Table 1: System Parameters and Properties

C is 2. Thus, message blocks can be thought of as check blocks with degree 1. An encoding of a file consists of a sequence of check blocks.[1] The properties of the particular code determine how the check blocks are constructed, and the probability distribution of the degrees of checkblocks is a critical part of the design. The *expansion factor* of a code is given by the ratio of the number of output check blocks to the number of input message blocks. Codes such as Tornado Codes [4] have fixed expansion factors, which for practical purposes should not grown much larger than 10. Other codes such as LT-Codes [11], Raptor Codes [25] and Online Codes [12] have no preset expansion factor, and hence can be called "rateless." See [14] for a discussion of why this latter category of codes is best suited for P2P-CDNs.

A downloader decodes a file by iteratively discovering message blocks. If she has received C_1 as above, and $C_2 = B_2$, then she can recover $B_{10} = C_1 \oplus B_2$. Consider a file $\mathbf{F}$ that is composed of n message blocks. The properties of a given erasure code ensure that downloader can reconstruct the original file $\mathbf{F}$ with high probability after receiving n' checkblocks. For Raptor Codes and Online Codes, $n' = (1+\epsilon)n$ for some arbitrarily small ϵ. For LT Codes, $n' = n + O\left(\sqrt{n}\ln^2(n/\delta)\right)$.

None of these codes require that the operation used to form check blocks be XOR. Any efficiently computable and invertible suffices.

4 Our Solution: Homomorphic Hashing

Consider a hypothetical network consisting of three types of players: publishers, mirrors and downloaders. Publishers introduce new content into the network. When a publisher wishes to publish file $\mathbf{F}$, she hashes the file to obtain $\mathcal{H}(\mathbf{F})$, and then pushes $\mathbf{F}$ into the network, keyed by $\mathcal{H}(\mathbf{F})$. Mirrors maintain local copies of the file $\mathbf{F}$ so that downloaders can retrieve it from multiple hosts concurrently. Downloaders who desire the file $\mathbf{F}$ first map the name of the file $n(\mathbf{F})$ to the hash of the file $\mathcal{H}(\mathbf{F})$ using an out-of-band and trusted lookup. Downloaders then request the file by its hash from the appropriate mirrors online in the network. In a P2P-CDN, all nodes can perform all three roles. Furthermore, downloaders can mirror a file as soon as they have received any part of it.

This section presents two simple protocols based on a homomorphic collision-resistant hash function (CRHF) that enable a downloader to verify encoded blocks on-the-fly. In the first, publishers use globally-defined hash parameters. As such, one-time-hash generation is slow but well-defined; that is, one file will map to one hash. Alternatively, we present a scheme in which each publisher determines her own hash parameters. This enables publishers to compute hashes more efficiently, but has the drawback that a file $\mathbf{F}$ when hashed by two different publishers will yield two different hashes. Such a one-to-many hash function

[1] The encoding also includes index information, either implicit or explicit, so that the receiver knows which message blocks comprise the check blocks. For instance, a block might be accompanied by the seed of a PRNG with which it was generated.

might be inappropriate for massively-decentralized networks with many publishers and frequent network partitions.

Both hash functions differ from more traditional hashes in that they do not output fixed-sized hashes. Rather, they reduce their input by a fixed factor. We also describe in this section a means by which our hash functions can be recursively applied, so that small, fixed-size hash values are output.

4.1 Notation

In the discussion that follows, we will be using vectors and matrices defined over modular subgroups of $\mathbb{Z}$; they will always be denoted in boldface, matrices written as capital letters (*e.g.*, $\mathbf{F}$) and vectors written as lowercase letters (*e.g.*, $\mathbf{g}$). Vectors might be row vectors or column vectors, and are explicitly specified as one or the other. All additions are assumed to be taken over $\mathbb{Z}_q$, and multiplications and exponentiations are assumed to be taken over $\mathbb{Z}_p$. Finally, we invent one notational convenience concerning vector exponentiation. That is, $g^{\mathbf{r}} = \mathbf{g}$ is defined component-wise: if $\mathbf{r} = (r_1\ r_2\ \cdots\ r_m)$, then $g^{\mathbf{r}} = (g^{r_1}\ g^{r_2}\ \cdots\ g^{r_m})$.

4.2 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 $\mathbf{F}$ 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 1. The system property d is listed in this table for convenience but is determined by the underlying erasure-encoding scheme and therefore not directly tunable by content publishers or the trusted party tasked with generating the hash parameters.

In the decentralized P2P-CDNs that we are considering, such a trusted party might not exist. Rather, users joining the system should demand "proof" that the group parameters G were generated honestly, and that some malicious node does not know some g and $\mathbf{r}$ such that $g^{\mathbf{r}} = \mathbf{g}$. The generators might be generated according to the algorithm PickGroup given in Figure 1. The input $(\lambda_p, \lambda_q, s)$ to the PickGroup algorithm serves as a proof of authenticity for the output parameters, $G = (p, q, \mathbf{g})$. The seed s might be chosen globally, or even chosen per file $\mathbf{F}$ such that $s = \mathsf{SHA1}(n(\mathbf{F}))$. Either way the same parameters G will always be used when hashing file $\mathbf{F}$.

Encoding. As per Table 1, call the block size to be used β, and let $m = \beta/(\lambda_q - 1)$. To encode a file $\mathbf{F}$, consider it as an $m \times n$ matrix in $\mathbb{Z}_q^{m \times n}$, where $n = \lceil |\mathbf{F}|/\beta \rceil$. Each file block $\mathbf{b}_j$ then consists of columns of the matrix, or rather, of an $m \times 1$ column vector whose elements are in $\mathbb{Z}_q$ and less than $2^{\lambda_q - 1}$. We write $\mathbf{b}_j = (b_{1,j}, \ldots, b_{m,j})$, thus:

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

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

$$\mathbf{b_i} + \mathbf{b_j} = (b_{1,i} + b_{1,j}, \ldots, b_{m,i} + b_{m,j}) \bmod q$$

Now, proceed with erasure encoding as normal, using vector addition over $\mathbb{Z}_q$ instead of the $\oplus$ operator. Note that some erasure codes such as Online Codes involve a *preprocessing* stage, in which "auxiliary"

Algorithm $\mathsf{PickGroup}(\lambda_p, \lambda_q, s)$
  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

Figure 1: The seed s can serve as a "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 $[0, x)$.

blocks are concatenated to the file. The resulting composite file is then fed through the erasure encoder. In such a scheme, we would perform the above encoding over the composite file.

Hash Generation. To hash a file, we use a collision-resistant hash function (CRHF), secure under the discrete-log assumption. This hash function is a generalized form of the Pederson commitment scheme [18]. Recall that a CRHF is informally defined as a function for which finding any two inputs that yield the same hash-value is difficult.

For an arbitrary file 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 a complete file $\mathbf{F}$ as a $1 \times n$ row-vector whose elements are the hashes of its constituent blocks:

$$H_G(\mathbf{F}) = (h_G(\mathbf{b_1})\ h_G(\mathbf{b_2})\ \cdots\ h_G(\mathbf{b_n})) \tag{2}$$

To convey the complete hash, publishers should transmit both the group parameters and the hash itself: $(G, H_G(\mathbf{F}))$. 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 $|H_G(\mathbf{F})| = |\mathbf{F}|\lambda_p/\beta$.

Hash Verification. If a downloader knows $(G, H_G(\mathbf{F}))$, he can verify any check block of the form

$$\mathbf{c} = \sum_{k \in X} \mathbf{b_k} \text{ for any } X \subset \{1, .., n\}$$

by verifying that

$$h_G(\mathbf{c}) = \prod_{k \in X} h_G(\mathbf{b_k}). \tag{3}$$

h_G functions here as a *homomorphic hash function*. We are using the fact that addition distributes over multiplication in $\mathbb{Z}_p$ to verify check blocks.

name	description
g	a random element in $\mathbb{Z}_p$ of order q.
$\mathbf{r}$	random row vector in $\mathbb{Z}_q^m$.
$\mathbf{g}$	$= g^{\mathbf{r}}$, a row vector in $\mathbb{Z}_p^m$

Table 2: Additional Per-Publisher Parameters

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. A downloader can reconstruct the file $\mathbf{F}$ after receiving a sufficient number of check blocks, $n' = (1+\epsilon)n$, for small ϵ in the case of Raptor and Online Codes. Call these check blocks $\mathbf{c_1}, \ldots, \mathbf{c_{n'}}$. Recall that conventional implementations of erasure codes combine message blocks with the XOR operation, which is conveniently its own inverse. With our modular addition operation, we simply use vector subtraction. The decoding algorithm proceeds as normal, but with the use of additive inverses over the m-vectors. Given $\mathbf{c} = \mathbf{b_j} + \mathbf{b_k}$ and $\mathbf{b_j}$, $\mathbf{b_k} = \mathbf{c} - \mathbf{b_j}$.

4.3 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, the publisher picks group parameters G so that a logarithmic relation among the generators $\mathbf{g}$ is known. A publisher $\mathcal{P}$ would pick q and p as above, but would generate $\mathbf{g}$ as shown in Table 2.

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

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

$\mathcal{P}$ computes the product $\mathbf{rF}$ first, and then performs only one modular exponentiation per file block to obtain the full file hash. See Section 5.2 for a more complete running-time analysis. Note that $\mathcal{P}$ must be careful to never reveal g and $\mathbf{r}$. So doing would allow an adversary to compute arbitrary collisions for the hash function 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. In particular, a verifier does not distinguish between the two types of hashes, beyond ensuring that the party who generated the parameters is the trusted publisher.

4.4 Computational Efficiency Improvements

We have presented a bare-bones protocol that achieves our security goals, but at a price in terms of bandwidth and computation. First, the XOR block-combining operation is faster to compute than modular addition. More important, our hash function H_G is orders of magnitude slower than a more conventional hash function such as SHA1. Our most important goal here is to improve the performance of a downloader verifying the hash of a file $(G, H_G(\mathbf{F}))$, 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 $H_G(\mathbf{F})$, but without knowing $\mathbf{r}$.

A technique suggested by Bellare, Garay, and Rabin [3] is used to improve verification performance. Instead of verifying each check block $\mathbf{c}_i$ exactly, we verify them probabilistically and in batches. Each downloader picks batching parameters, as given in Table 3.

name	description	*e.g.*
t	batch size	256 blocks
l	security parameter	32

Table 3: Batching parameters

To verify, the downloader collects t check blocks of the file $\mathbf{F}$, which for convenience, we will call $\mathbf{C} = (\mathbf{c}_1 \ \cdots \ \mathbf{c}_t)$. For each $j \in \{1, .., t\}$ the downloader knows the supposed set $X_j \subset \{1, .., n\}$ such that $\mathbf{c}_j = \sum_{k \in X_j} \mathbf{b_k}$. She then proceeds as follows:

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 $\mathbf{z} = \mathbf{Cs}$
3. Compute $\gamma_j = \prod_{k \in X_j} h_G(\mathbf{b_k})$ 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 $y' \equiv y \bmod p$

Batching does open the receiver to small-scale bandwidth wasting attacks. A receiver needs to accept 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 the overall fraction of bad blocks it can transmit.

4.5 Hash Size Considerations

Our substitution of modular arithmetic for the XOR also results in slight message dilations. Message blocks are β bits long, but check blocks require $\beta+m$ bits to encode because $q > 2^{\lambda-1}$. Thus, to completely decode a file $\mathbf{F}$, a Raptor or Online Code downloader must receive a total amount of data at roughly $|\mathbf{F}|(1+\epsilon+m/\beta)$ in size.

Another consideration is that hashes $H_G(\mathbf{F})$ are proportional in size to the file $\mathbf{F}$, 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. File hashes are often used as lookup keys at the peer-to-peer routing level, especially in structured systems; internally to each node, they are also used as indices into disk and memory-resident file caches. A 64 MB key is horribly impractical in any of these settings. Moreover, if a downloader were to use traditional techniques to download such a hash, he would be susceptible to the very same attacks we've set out to avoid, albeit on a smaller scale.

To solve these problems, we apply our encoding and hashing scheme recursively — 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:

$$\begin{aligned} H_G^i(\mathbf{F}) &= H_G(H_G^{i-1}(\mathbf{F})) \text{ and } H_G^0(\mathbf{F}) = \mathbf{F} \\ \mathcal{H}_G(\mathbf{F}) &= (G, k, H_G^k(\mathbf{F})) \text{ for minimal } k \text{ such that } |\mathcal{H}_G(\mathbf{F})| < L \\ \mathcal{I}_G(\mathbf{F}) &= \mathsf{SHA1}(\mathcal{H}_G(\mathbf{F})) \end{aligned}$$

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

- **Filename to Hash Mappings** . Filename-to-hash lookup services will now map $n(\mathbf{F}) \rightarrow \mathcal{I}_G(\mathbf{F})$, for some G.
- **File Publication** . To publish a file $\mathbf{F}$, a publisher $\mathcal{P}$ must compute the hashes $H_G(\mathbf{F}), H_G^2(\mathbf{F}), ..., H_G^k(\mathbf{F})$, and trivially the hashes $\mathcal{H}_G(\mathbf{F})$ and $\mathcal{I}_G(\mathbf{F})$. For $i \in (0, ..., k-1)$, $\mathcal{P}$ stores $H_G^i(\mathbf{F})$ under the key $(\mathcal{I}_G(\mathbf{F}), i)$. $\mathcal{P}$ additionally stores $\mathcal{H}_G(\mathbf{F})$ under the key $(\mathcal{I}_G(\mathbf{F}), -)$.
- **File Download** . To retrieve a file $\mathbf{F}$, a downloader $\mathcal{D}$ first performs the name-to-hash lookup $n(\mathbf{F}) \rightarrow \mathcal{I}_G(\mathbf{F})$, for some G. $\mathcal{D}$ then uses the peer-to-peer routing layer to determine a set of sources who serve the file and hashes corresponding to $\mathcal{I}_G(\mathbf{F})$. The downloader queries one of the servers with the key $(\mathcal{I}_G(\mathbf{F}), -)$, and expects to receive $\mathcal{H}_G(\mathbf{F})$. This transfer can be at most L big, and the downloader can verify $\mathcal{H}_G(\mathbf{F})$ with the hash $\mathcal{I}_G(\mathbf{F})$ it obtained from the name-to-hash lookup. Assuming that $\mathcal{D}$ correctly verifies $\mathcal{I}_G(\mathbf{F}) = \mathsf{SHA1}(\mathcal{H}_G(\mathbf{F}))$, $\mathcal{D}$ knows the value k, and the kth order hash $H_G^k(\mathbf{F})$. $\mathcal{D}$ can then request the next hash in the sequence simultaneously from all of the servers who serve the file $\mathbf{F}$. $\mathcal{D}$ queries these servers with the key $(\mathcal{I}_G(\mathbf{F}), k-1)$, and expects the hash $H_G^{k-1}(\mathbf{F})$ in response. This transfer can be completed with erasure encoding, and on-the-fly hash verification as given above. $\mathcal{D}$ iteratively queries its server nodes for lower order hashes until it receives the 0th order hash, or rather, the file itself.

In practice, it would be rare to see a k 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 reduces its 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:

$$\frac{\text{overhead}}{\text{filesize}} = \sum_{i=1}^{k} \left(\frac{\beta}{\lambda_p}\right)^i < \sum_{i=1}^{\infty} \left(\frac{\beta}{\lambda_p}\right)^i = \frac{1}{1-\lambda_p/\beta} - 1 = \frac{\lambda_p}{\beta - \lambda_p}$$

5 Analysis

In this section, we sketch the running time and security analysis of our homomorphic hashing scheme, and report performance numbers of a sample implementation.

5.1 Security

In Appendix A, we offer formal proofs that the batched verification scheme described above protects downloaders from both the Encoding Corruption and Bandwidth Wasting attacks. We show that the fast hash verification scheme is correct, and use the hardness of the discrete log problem to prove the scheme's security — that the probability of a false-negative is negligible in the security parameter.

A full discussion of Distribution Attacks is beyond the scope of this paper, but we have considered minor changes to the above protocols to make them resilient to adversaries who intentionally disregard the

operation	$\mathsf{MultCost}(q)$ per block	$\mathsf{MultCost}(p)$ per block	*e.g.*
Per-Publisher Hash Generation	m	$\lambda_q/2$	1.3 msec
Global Hash Generation	0	$m\lambda_q/2$	407.9 msec
Naïve Verification	0	$m\lambda_q/2 + d$	408.0 msec
Fast Verification	m	$d + l/2 + (m\lambda_q/2 + l - 1)/t$	2.3 msec

Table 4: Expected computational costs required by our different algorithms. These values do not reflect the cost of precomputing the iterative squares of g_i, because m and λ_q are neglible when amortized over n. The last column details rough estimates for the absolute amount of computation needed to perform these operations. We assume the example hash parameters given throughout, and that on a 3.0 GHz Pentium 4, $\mathsf{MultCost}(p) \approx 6.2$ μsecs and $\mathsf{MultCost}(q) \approx 1.0$ μsecs.

prescribed probability distributions. First, we might require encoders not to generate check blocks entirely randomly, but instead according to some sequence defined by a seed and a pseudo-random number generator. If a sender discloses his one-time-random seed and the corresponding sequence number for each block, then the recipient can check that the received check block is the appropriate block in the sequence. This leaves us with the problem that a malicious sender might follow the appropriate sequence, while deleting certain blocks from the transmission, and placing the blame on network congestion. Thus, receivers should monitor the observed check block distribution for each source, and compare that distribution against the expected distribution given by the erasure code, perhaps using the χ^2 test. If the receiver observes a statistically significant deviation, then he should suspect malicious behavior.

5.2 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 $1 \leq z < |x|$. On average, half of the bits of x 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 $(m\lambda_q/2 + m - 1)\,\mathsf{MultCost}(p)$ we might expect. That is we can save $(m-1)$ multiplications by keeping a "running product" — as opposed to first computing all of the $g_i^{x_i}$ and then multiplying them together.

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 expense than multiplying two random elements from $\mathbb{Z}_q$. In our analysis below, we disregard these optimizations and seek only simplified upper bounds.

Table 4 lists the running time of the scheme's various stages, which we compute below.

- **Per-Publisher Hash Generation.** Publishers first precompute a table g^{2^z} for all z such that $1 \leq z < \lambda_q$. This table can then be used to compute $H_G(\mathbf{F})$ for any file $\mathbf{F}$. To compute $\mathbf{rF}$ as in Equation 4, mn multiplications are needed in $\mathbb{Z}_q$. Here and throughout this analysis, we can disregard the one-time precomputation of g^{2^z}, since $n \gg m$. The n-vector exponentiation in Equation 4 requires an expected $n\lambda_q/2$ multiplications in $\mathbb{Z}_p^*$. Thus, the total cost of hash generation is $mn\,\mathsf{MultCost}(q) + n\lambda_q\,\mathsf{MultCost}(p)/2$.

```
Algorithm FastMult ((y_1, s_1), ..., (y_t, s_t))
  y ← 1
  for j = l − 1 down to 0 do
    for i = 1 to t do
      if s_i[j] = 1 then y ← y y_i
    done
    if l > 0 then y ← y^2
  done
  return y
```

Figure 2: 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 jth bit of s_i, $s_i[0]$ being the least significant bit. This algorithm is presented in [3], although we believe there to be an off-by-one-error in that paper, which we have corrected here.

- **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 ingore 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 of naïve hash verification is $(m\lambda_q/2 + d)\mathsf{MultCost}(p)$.
- **Fast Hash Verification.** We refer to the algorithm described in Section 4.4. In Step 2, we recall that $\mathbf{C}$ is a $m \times t$ matrix, and hence the matrix multiplication costs $mt \,\mathsf{MultCost}(q)$. We determine γ_j in Step 3 with d multiplications over $\mathbb{Z}_p$, at a total cost of $td \,\mathsf{MultCost}(p)$. In Step 4, we compute y' at a cost of $m\lambda_q/2 \,\mathsf{MultCost}(p)$, because we have precomputed tables of the form $g_i^{2^x}$. For y, we cannot assume this precomputation; the bases in this case are γ_j, of which there are more than n. To compute y efficiently, we suggest the FastMult algorithm described in Figure 2, which costs $(tl/2 + l - 1)\,\mathsf{MultCost}(p)$.[2] Summing these computations together yields a total cost per batch of:

  $$mt \,\mathsf{MultCost}(q) + (td + m\lambda_q/2 + l + tl/2 - 1)\,\mathsf{MultCost}(p)$$

  The estimates reported in Table 4 simply amortize the total cost of a batch over the t constituent blocks.

5.3 Implementation Results

We implemented a version of the hash primitives above using the GNU MP library, version 4.1.2. Table 5 shows the results of our C++ testing program when run on a 3.0 GHz Pentium 4, and with the sample parameters given in Figures 1 and 3. Our results are reported in both cost per block, and also overall throughput. For comparison, similar computations for SHA1 and maximum theoretical packet arrival rate on a T1 are also included.

Although batched verification of hashes is an order of magnitude slower than a more conventional hash function such as SHA1, we still believe them to be practical in that they are an order of magnitude faster than the maximum packet arrival rate on an average Internet connection.

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

operation on 16 kByte block $\mathbf{b}$	msec to complete	throughput (MBytes/sec)
$\mathsf{SHA1}(\mathbf{b})$	0.14	55.66
Receiving $\mathbf{b}$ on a T1	83.33	0.186
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

Table 5: Performance benchmarks for our different algorithms

Unfortunately, our current scheme for global hash generation is rather slow. A publisher who wishes to introduce a 1 GB file into the network must perform over 7.5 hours of computation to derive the file's hash. For publishers with large amounts of RAM, k-ary exponentiation can be used bring the cost under two hours. See Appendix B for details.

Another approach to avoiding the initial computational cost of global hash generation is to alter the overlying P2P-CDN application-level protocols and enable per-publisher hashes. That is, a publisher $\mathcal{P}_2$ would not publish a file $\mathbf{F}$ if she notices that publisher $\mathcal{P}_1$ has already published it. In the case of an inopportune network partition, $\mathcal{P}_2$ might not see $\mathcal{P}_1$'s publication of $\mathbf{F}$, and hence might introduce $\mathbf{F}$ into the network under a hash $\mathcal{H}_{G_1}(\mathbf{F})$ such that $\mathcal{H}_{G_1}(\mathbf{F}) \neq \mathcal{H}_{G_2}(\mathbf{F})$. Assuming both of these hashes propagate through the network, a downloader might want to include mirrors with both hashes in his download of the file $\mathbf{F}$. Such an approach is possible if both publishers agree on λ_q and m. However, the complexity and bandwidth overhead of the protocol would increase with the number of different hashes used in a download.

6 Conclusion

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, the introduction of rateless erasure-encoding for more efficient content distribution invalidates these straightforward approaches.

This paper considers the problem of on-the-fly verification of erasure-encoded content. We present a hash scheme based on a discrete-log construction that provides useful homomorphic properties for verifying check blocks against authenticated file blocks. However, this basic construction is quite inefficient; therefore, we apply batching techniques for probabilistic verification that yield an efficient online algorithm.

To our knowledge, this paper is the first to consider such a verification problem. We think that the design of more efficient schemes, measured from both a computational or communication perspective, is an interesting open problem.

Acknowledgments

We wish to thank Petar Maymounkov and Benny Pinkas for helpful discussions. This research was conducted as part of the IRIS project (`http://project-iris.net/`), supported by the NSF under Cooperative Agreement No. ANI-0225660.

References

[1] R. J. Anderson, F. Bergadano, B. Crispo, J.-H. Lee, C. Manifavas, and R. M. Needham. A new family of authentication protocols. *Operating Systems Review*, 32(4):9–20, October 1998.

[2] Stefan Aroui, Krishna P. Gummadi, Richard J. Dunn, Steven D. Gribble, and Henry M. Levy. An analysis of internet content delivery systems. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 'O2)*, pages 315–329, Boston, MA, October 2002.

[3] Mihir Bellare, Juan Garay, and Tal Rabin. Fast batch verification for modular exponentiation and digital signatures. In Kaisa Nyberg, editor, *Advances in Cryptology—EUROCRYPT 98*, volume 1403 of *Lecture Notes in Computer Science*. Springer-Verlag, May 31–June 4 1998.

[4] John Byers, Michael Luby, and Michael Mitzenmacher. Accessing multiple mirror sites in parallel: Using Tornado codes to speed up downloads. In *Proceedings of IEEE INFOCOM '99*, pages 275–283, New York, NY, 1999.

[5] John Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A Digital Fountain approach to reliable distribution of bulk data. In *Proceedings of ACM SIGCOMM '98*, Vancouver, Canada, September 1998.

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

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

[8] S. Cheung. An efficient message authentication scheme for link state routing. In *Proc. 13th Annual Computer Security Applications Conference*, 1997.

[9] Rosario Gennaro and Pankaj Rohatgi. How to sign digital streams. In Burton S. Kaliski Jr., editor, *Advances in Cryptology—CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 180–197. Springer-Verlag, 17–21 August 1997.

[10] Philippe Golle and Nagendra Modadugu. Authenticated streamed data in the presernce of random packet loss. In Oorschot and Gligor [17].

[11] Michael Luby. LT codes. In *43rd Annual Symposium on Foundations of Computer Science*, Vancouver, Canada, November 2002. IEEE.

[12] Petar Maymounkov. Online codes. Technical Report 2002-833, NYU, November 2002.

[13] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS02)*, Cambridge, MA, March 2002.

[14] Petar Maymounkov and David Mazières. Rateless codes and big downloads. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, February 2003.

[15] Sara Miner and Jessica Staddon. Graph-based authentication of digital streams. In Reiter and Needham [22].

[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] Paul Van Oorschot and Virgil Gligor, editors. *Network and Distributed System Security Symposium*. ISOC, February 2001.

[18] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In J. Feigenbaum, editor, *Advances in Cryptology—CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer-Verlag, 1992, 11–15 August 1991.

[19] Adrian Perrig. The BiBa one-time signature and broadcast authentication protocol. In Pierangela Samarati, editor, *Eighth ACM Conference on Computer and Communication Security*, pages 28–37. ACM, November 5–8 2001.

[20] Adrian Perrig, Ran Canetti, Dawn Song, and Doug Tygar. Efficient authentication and signature of multicast streams over lossy channels. In Reiter and Needham [22].

[21] Adrian Perrig, Ran Canetti, Dawn Song, and Doug Tygar. Efficient and secure source authentication for multicast. In Oorschot and Gligor [17].

[22] Michael Reiter and Roger Needham, editors. *IEEE Symposium on Research in Security and Privacy*. IEEE, May 2000.

[23] Pankaj Rohatgi. A compact and fast hybrid signature scheme for multicast packet authentication. In Gene Tsudik, editor, *Sixth ACM Conference on Computer and Communication Security*, pages 93–100. ACM, November 1999.

[24] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware*, November 2001.

[25] Amin Shokrollahi. Raptor codes, 2003. Preprint.

[26] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. In *IEEE/ACM Transactions on Networking*, 2002.

[27] C. K. Wong and S. S. Lam. Digital signatures for flows and multicasts. In *Proc. IEEE International Conference on Network Protocols*, Austin, TX, October 1998.

A Security analysis

A.1 Correctness of Batched Verification

Consider the batched verification algorithm given in Section 4.4. To prove it correct (*i.e.*, that correct check blocks will be validated), examine an arbitrary hash $(G, H_G(\mathbf{F}))$. For notational convenience, we write y and y' computed in Step 4 in terms of an element $g \in \mathbb{Z}_p$ 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 $y' = g^{\mathbf{rCs}}$.

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:

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

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

$$y = \prod_{j=1}^{t} g^{s_j \mathbf{rc}_j} = g^{\sum_{j=1}^{t} s_j \mathbf{rc}_j} = g^{\mathbf{rCs}}$$

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

A.2 Security of Batched Verification

We use the hardness of the discrete log problem to prove the scheme's soundness against a malicious adversary. We can use the same proof for the per-publisher and the global hashing protocol by assuming — mainly for notational convenience — a hypothetical hash-parameter generation oracle, $\mathcal{O}$. This oracle takes as input the two discrete log security parameters (λ_p, λ_q), and outputs a random set of hash parameters G along the lines of Tables 1. That is, $G = (p, q, \mathbf{g})$, $|q| = \lambda_q$, $|p| = \lambda_p$, $q|(p-1)$, and $\mathbf{g}$ is a $1 \times m$ row vector of elements in $\mathbb{Z}_p$, order q.

Consider a discrete-log hash adversary $\mathcal{A}$ who takes as input a random set of hash parameters G and attempts to find a collision. Define the probability that $\mathcal{A}$ succeeds:

$$\mathsf{Adv}_0(\mathcal{A}) = \Pr\left[(p, q, \mathbf{g}) \leftarrow \mathcal{O}(\lambda_p, \lambda_q);\ (\mathbf{a}, \mathbf{a}') \leftarrow \mathcal{A}(p, q, \mathbf{g}) : \prod_{i=0}^{m} g_i^{a_i} \equiv \prod_{i=0}^{m} g_i^{a'_i} \bmod p \ \wedge\ \mathbf{a} \neq \mathbf{a}'\right]$$

Assuming the discrete-log problem is hard in G, $\mathsf{Adv}_0(\mathcal{A})$ is negligible in λ_p, for all PPT adversaries $\mathcal{A}$. The proof is canonical, and we do not reiterate it here.

The adversary in our model can chose the file $\mathbf{F}$, and should output a potentially "forged" erasure encoding of that file. Let us represent check blocks recipes as (sparse) $1 \times n$ row vectors, whose elements are simply bits. Given such a vector $\mathbf{x}$, the corresponding check block is then the vector $\mathbf{c} = \mathbf{F}\mathbf{x}$. Thus, an adversary hopes to output a file $\mathbf{F}$, a vector $\mathbf{x}$, and a forged check block $\mathbf{c}'$ such that $h_G(\mathbf{c}') = h_G(\mathbf{F}\mathbf{x})$ and $\mathbf{c}' \neq \mathbf{F}\mathbf{x}$. Considering such an adversary, $\mathcal{A}'$, we define the probability that it succeeds in its forgery:

$$\begin{aligned}\mathsf{Adv}_1(\mathcal{A}') = \Pr[G \leftarrow \mathcal{O}(\lambda_p, \lambda_q);\ (\mathbf{F}, \mathbf{x}, \mathbf{c}') \leftarrow \mathcal{A}'(G) :\\ h_G(\mathbf{F}\mathbf{x}) = h_G(\mathbf{c}') \ \wedge\ \mathbf{F}\mathbf{x} \neq \mathbf{c}']\end{aligned}$$

It is easy to see that given a PPT $\mathcal{A}'$, we can construct a corresponding algorithm $\mathcal{A}$ that finds hash-collisions. Given G, $\mathcal{A}$ runs $\mathcal{A}'$ on the input G to obtain the output $(\mathbf{F}, \mathbf{x}, \mathbf{c}')$. $\mathcal{A}$ then simply outputs the pair of row vectors $(\mathbf{F}\mathbf{x}, \mathbf{c}')$. By construction of the two preceding random experiments, we can state quite simply:

$$\mathsf{Adv}_1(\mathcal{A}') = \mathsf{Adv}_0(\mathcal{A}) \tag{5}$$

Now consider our verifier function $\mathcal{V}$. $\mathcal{V}$ takes as input an $m \times t$ batch $\mathbf{C}$, the hash parameters G, the hash of the entire file, and a random t-length column vector $\mathbf{s}$ of l-bit numbers. It then should output Accept if the hash of the batch matches that predicted by the complete file hash, and Reject otherwise. Define an adversary $\mathcal{A}''$ who attempts to trick $\mathcal{V}$ into wrongly verifying a batch of t check blocks. As above, we define its likelihood of success in terms of randomized experiment. The adversary $\mathcal{A}''$ takes as input the hash parameters G, and then chooses a file $\mathbf{F}$, a recipe for a batch of check blocks given by $\mathbf{X}$, and finally a forged batch of check blocks given by $\mathbf{C}'$. Here, $\mathbf{X}$ is a batch of t column vectors each of length n, or rather, an $n \times t$ sparse matrix whose cells are in $\{0, 1\}$. While the verifier expects the batch of check blocks given by $\mathbf{C} = \mathbf{F}\mathbf{X}$, the experiment feeds it the forged batch $\mathbf{C}'$ generated by $\mathcal{A}''$.

$$\begin{aligned}\mathsf{Adv}_2(\mathcal{A}'') = \Pr[G \leftarrow \mathcal{O}(\lambda_p, \lambda_q);\ (\mathbf{F}, \mathbf{X}, \mathbf{C}') \leftarrow \mathcal{A}''(G);\ \mathbf{s} \leftarrow (\mathbb{Z}_{2^l})^t :\\ \mathcal{V}(\mathbf{X}, \mathbf{C}', G, H_G(\mathbf{F}), \mathbf{s}) = \mathsf{Accept} \ \wedge\ \mathbf{F}\mathbf{X} \neq \mathbf{C}']\end{aligned}$$

Given an adversary $\mathcal{A}''$, we can define a derived adversary $\mathcal{A}'$ that attempts to forge check blocks as follows:

1. $\mathcal{A}'$ is given input G.
2. $(\mathbf{F}, \mathbf{X}, \mathbf{C}') \leftarrow \mathcal{A}''(G)$

3. If $\mathbf{FX} = \mathbf{C}'$ then Fail.

4. Pick $\mathbf{s}$ at random as in the experiment for Adv_2.

5. If $\mathcal{V}(\mathbf{X}, \mathbf{C}', G, H_G(\mathbf{F}), \mathbf{s}) = \mathsf{Reject}$ then Fail.

6. If $H_G(\mathbf{FX}) \neq H_G(\mathbf{C}')$ then Fail.

7. Find minimal j such that $\mathbf{F}\mathbf{x}_j \neq \mathbf{c}'_j$. Such a j must exist since $\mathbf{FX} \neq \mathbf{C}'$. Return $(\mathbf{F}, \mathbf{x}_j, \mathbf{c}'_j)$.

If the algorithm $\mathcal{A}'$ runs until the successful return at Step 7, then it will succeed in the experiment given by Adv_1, because Steps 3 and 6 guarantee that $\mathbf{FX} \neq \mathbf{C}'$ and $H_G(\mathbf{FX}) = H_G(\mathbf{C}')$, respectively. Recall that the hash H_G is a row vector of size t, and hence $H_G(\mathbf{FX}) = H_G(\mathbf{C}')$ implies that $h_G(\mathbf{F}\mathbf{x}_j) = h_G(\mathbf{c}'_j)$. Moreover, a quick analysis of this algorithm shows that the probability of making it past Step 5 (i.e., not failing in Step 3 or Step 5) is precisely $\mathsf{Adv}_2(\mathcal{A}'')$. In Lemma 1, we show that the probability of failing at Step 6 is at most 2^{-l}. We conclude that:

$$\mathsf{Adv}_1(\mathcal{A}') \geq \mathsf{Adv}_2(\mathcal{A}'') - 2^{-l} \tag{6}$$

Composing Equations 5 and 6, we have that for any PPT $\mathcal{A}''$ there exists a corresponding $\mathcal{A}$ such that: $\mathsf{Adv}_0(\mathcal{A}) \geq \mathsf{Adv}_2(\mathcal{A}'') - 2^{-l}$. By the discrete log assumption, $\mathsf{Adv}_0(\mathcal{A}) < \mathsf{negl}(\lambda_p)$ for all PPT $\mathcal{A}$. Thus, for all batch-forging adversaries $\mathcal{A}''$,

$$\mathsf{Adv}_2(\mathcal{A}'') \leq \mathsf{negl}(\lambda_p) + 2^{-l} \tag{7}$$

Lemma 1 *$\mathcal{A}'$ fails at Step 6 with probability 2^{-l}.*

Proof. To arrive at Step 6, the verifier $\mathcal{V}$ must have output Accept. Using the same manipulations as those given in Appendix A, we take the fact that $\mathcal{V}$ accepted to mean that:

$$g^{\mathbf{rFXs}} \equiv g^{\mathbf{rC's}} \bmod p \tag{8}$$

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 $\mathbf{rFXs} \equiv \mathbf{rC's} \bmod q$, and rearranging,

$$\mathbf{r}\left(\mathbf{FX} - \mathbf{C}'\right)\mathbf{s} \equiv \mathbf{0} \bmod q. \tag{9}$$

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

$$\mathbf{r}\left(\mathbf{FX} - \mathbf{C}'\right) \not\equiv \mathbf{0} \bmod q \tag{10}$$

For convenience, let the $1 \times t$ row vector $\mathbf{u} = \mathbf{r}(\mathbf{FX} - \mathbf{C}')$. Equation 10 gives us that $\mathbf{u} \not\equiv \mathbf{0} \bmod q$; thus some element of $\mathbf{u}$ most 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 9 gives us that $\mathbf{us} \equiv \mathbf{0} \bmod q$. Since $u_1 \neq 0$, it has a multiplicative inverse, u_1^{-1}, in $\mathbb{Z}_q^*$. We therefore have:

$$s_1 \equiv -\left(u_1^{-1}\right) \sum_{j=2}^{t} u_j s_j \bmod q \tag{11}$$

But since s_1 was selected at random from 2^l possible values, the probability of its having this value is at most 2^{-l}. Thus, $\mathcal{A}''$ can fail at Step 6 with probability at most 2^{-l}. ∎

B 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 $\mathbf{A}$ under the index $\mathbf{A}[i][j][k]$.

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:

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

The storage requirement for the table $\mathbf{A}$ is $m(2^x - 1)(\lambda_q - 1)\lambda_p/x$ bits, which is exponential in x. Disregarding 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, we achieve a factor of $x/2$ speed-up.

Setting $x = 8$, the size of the tables $|A| = 510$ MBytes, but we can hash a 1 GB file in less than 2 hours.