On the Practical and Security Issues of Batch Content Distribution Via Network Coding

Qiming Li
Computer & Information Science Dept
Polytechnic University
Email: qiming.li@ieee.org

Dah-Ming Chiu
Information Engineering Dept
Chinese University of Hong Kong
Email: dmchiu@ie.cuhk.edu.hk

John C.S. Lui
Computer Science & Eng. Dept
Chinese University of Hong Kong
Email: cslui@cse.cuhk.edu.hk

***Abstract*— File distribution via network coding has received a lot of attention lately. However, direct application of network coding may have security problems. In particular, attackers can inject "faked" packets into the file distribution process to slow down the information dispersal or even deplete the network resource. Therefore, content verification is an important and practical issue when network coding is employed. When network coding is used, it is infeasible for the source of the content to provide all the hash values or signatures required for verification, and hence the traditional "hash-and-sign" methods are no longer applicable. Recently, a new on-the-fly verification technique is proposed by Krohn et al. for rateless erasure codes [1]. However, their scheme requires a large number of hash values to be distributed in advance, and all of them are needed to verify even for a single packet. We propose a new batch delivery and verification scheme that is similar to the classical scenario where the authentication information of a message is embedded with the message and is sufficient for the verification purpose. We investigate how our technique can be applied when random linear network coding is employed, and show that both the computational and the bandwidth overhead can be greatly reduced by using a variant of the random network coding. We further show by simulation that this variant is sufficiently effective in practice.**

Keywords: Content distribution, security, verification, network coding.

I. Introduction

For the past few years, there has been an increasing interest on the application of network coding on file distribution. Various researchers have considered the benefit of using network coding on P2P networks for file distribution and multimedia streaming (such as [2], [3], [4], [5], [6]), while other researchers have considered using network coding on million of PCs around the Internet for massive distribution of new OS updates and software patches (i.e., the *Avalanche* project from Microsoft). What we are interested in is the "*security*" and "*practical*" aspect of using network coding on content delivery.

An important issue in practical large content delivery in a fully distributed environment is how to maintain the integrity of the data, in the presence of link failures, transmission errors, software and hardware faults, and even malicious attackers who can modify the data arbitrarily. Since the current Internet does not implement any network access control, leaving the network vulnerable to packet modification or packet injection attacks where a malicious node can inject a large number of *faked* packets into the network with the goal of depleting the resources of the legitimate PCs relaying the packets. Also, receivers which utilize networking coding may not be able to determine the validity of the received packets until they receive sufficient number of these packets. This may impede the progress and reduce the efficiency of the file distribution.

In traditional scenarios, intermediate nodes in the network only forward data without any modification. Hence, the task of verifying the integrity of the data can be accomplished using classical cryptographic techniques. In particular, given original data $\mathbf{X}$ and a collision-free hash function h, a hash value $h(\mathbf{X})$ is computed, which is then signed by employing a digital signature scheme S with some signing key k, and the signature $S_k(h(\mathbf{X}))$ is published. When some $\mathbf{Y}$ is received by a sink node T who already knows $S_k(h(\mathbf{X}))$, the node T computes $h(\mathbf{Y})$ and verify the digital signature with $h(\mathbf{Y})$, $S_k(h(\mathbf{X}))$, and the verification key obtained from either the sender or some trusted third party (e.g., a certificate authority). This is often referred to as the "hash-and-sign" paradigm.

Recently, there are a number of works (such as [3], [7], [8], [9]) that focus on how to apply *network coding* to achieve efficient content delivery. The seminal work on network coding was first studied by Ahlswede et al. [10], who showed that if the nodes in the network can perform coding instead of simply forwarding information, multiple sinks in a multicast session can achieve their maximum network flow simultaneously. Since then, the topic has been intensively studied, and there are many variations (such as [11], [12], [13], [14], [15], [4]). More details on the literature can be found in Section II.

In practical content delivery scenarios, the network can be very dynamic, in the sense that the topology of the network can change over time, nodes can join/leave the network, links and nodes can fail, and so on. In this case, classical theoretical results (such as [14]) would be difficult to be employed since they require the knowledge of the network topology during code construction, and require the link failures to follow certain predefined pattern for the code to be reliable.

Random linear network coding [15], on the other hand, avoids those problems by allowing each node in the network to make *local* decisions. In their setting, the original $\mathbf{X}$ is divided into n blocks $\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n$, and each node computes and forwards some random linear combination $\mathbf{p} = \sum_{i=1}^{n} c_i \mathbf{x}_i$ for each of its downstream nodes, together with the coefficients $\mathbf{c} = \langle c_1, \cdots, c_n \rangle$. We call the pair $(\mathbf{p}, \mathbf{c})$ a *packet*. When sufficient linearly independent packets are received, a node would be able to decode the original $\mathbf{X}$. It is clear that data integrity is even more important in this setting, since, without verification, a node T could combine a damaged (or maliciously modified) packet into all the packets that T generates, and hence all its downstream nodes would received only corrupted data.

Unfortunately, traditional "hash-and-sign" techniques cannot be easily applied with random linear network coding. When a node receives a packet $(\mathbf{p}, \mathbf{c})$ from an upstream node, it needs to be convinced that $\mathbf{p}$ is indeed the linear combination of the data blocks defined by $\mathbf{c}$, and not some garbage. With classical digital signature schemes, only the sender can produce the correct signature of the data. Hence, the sender may have to pre-compute and distribute the signatures for all possible linear combinations. The number of signatures required to be generated can be prohibitively large.

This problem of detecting malicious modifications at intermediate nodes, especially when it is infeasible for the sender to sign all the data being transmitted, is sometimes referred to as *on-the-fly Byzantine fault detection*. Freedman and Mazières [1] considered the problem in the context of large content distribution using rateless erasure codes (or fountain codes), and proposed a technique using homomorphic cryptographic hash functions [16]. Although their original intension was to apply their technique to rateless erasure codes, it could be used when random linear network code is employed, as noted by Gkantsidis and Rodriguez [7]. A simple and efficient verification method called *secure random checksum* was proposed by Gkantsidis et al. [9], which comes at the price of weaker security.

In the scheme in [1], given n data blocks $\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n$, the sender firstly computes the hash values $h_1, h_2, \cdots, h_n$ for those data blocks. Next, these hash values are distributed to all the nodes in the network. When a packet $(\mathbf{p}, \mathbf{c})$ is received by a node, the hash value $h_{\mathbf{p}}$ of $\mathbf{p}$ can be computed from the hash values $h_1, \cdots, h_n$ and the coefficients $\mathbf{c}$, and the packet is verified using $\mathbf{p}$ and $h_{\mathbf{p}}$. In this way, the sender would only have to compute n hash values. The details can be found in Section II.

It is worth to note that all hash values $h_1, \cdots, h_n$ need to be known by a node to be able to verify any packet. Hence, this verification technique is significantly different from the traditional message verification scenarios where the signature of a message is transmitted together with the message, and is sufficient for the verification of it. Their technique, in contrast, requires all the hash values to be transmitted separately and reliably in advance.

There are two disadvantages of the scheme in [1]. They are:

- **Problem 1:** The total size of the hash values is *proportional* to the number of blocks, which could be very large. Hence the requirement that all nodes have to have the knowledge of all the hash values beforehand would lead to *significant delay* at the beginning of a content distribution session, or when a node joins a network, which we call *start-up latency*. This problem becomes more serious for the intermediate nodes who only participate in the distribution of a small portion of the content, since they still need to know all the hash values regardless of the number of packets they need to verify. Furthermore, the large hash values themselves need to be distributed reliably, which increases the complexity of the system. It is proposed by Krohn et al. [1] that the hash values should be recursively divided into blocks and hashed using the same technique.
- **Problem 2:** The cryptographic hash function proposed in [1] is computationally expensive. To reduce the computational cost so that it would be feasible to be applied in real networks, a probabilistic batch verification method was proposed in [1] based also on the results in [16], where the verification is performed only once for a batch of packets. However, it is not straightforward to apply this batch verification method when network coding is used, since the existing network coding based content distribution schemes do not deliver content in batches.

In this paper, we propose a new paradigm for on-the-fly content verification. Our scheme is similar to the traditional techniques where the hash values travel with the messages, instead of being distributed in advance. To verify a packet, only *one hash value* and *some public parameters* are required, where the parameters are independent of the content, and are typically of small constant size. In this way, a node can perform verification of a packet as soon as it is received, with the knowledge of the public parameters. Hence, the start-up latency of our scheme is reduced significantly and the communication overhead for intermediate nodes is proportional only to the number of packets they verify.

Our method employs a new homomorphic hash function based on a new trapdoor function. The basic technique is explained in detail in Section III. We prove its security by showing that a successful attacker would be able to solve a variant of the discrete logarithm problem. Our scheme can be considered as a direct improvement on the on-the-fly verification technique in [1] for rateless erasure codes. Note that their batch verification technique can also be applied to our scheme with minor modifications.

We also propose a probabilistic batch verification scheme that further reduces both the computational complexity of the verifications and the network bandwidth overhead by allowing a node to send only a small number τ of hash values per batch for verification, where τ can be just 1 or 2.

One major drawback of the current content distribution schemes based on network coding, as pointed out by Bram [17], is that each node has to cache all the received packets, and has to scan all of them when the node computes a new packet. To reduce the computational cost, our scheme deviates from the classical network coding schemes in the sense that, we allow each node to distribute a combination of only a smaller number of data blocks. Details of the scheme can be found in Section IV.

To study the effectiveness of our scheme, we simulate the scheme under the classical settings with delay-free directed acyclic graphs, and show that the success rate for the content distribution does not suffer too much. Note that this success rate is only a reference to the actual performance of the scheme in real applications, where networks are typically not delay-free, unlikely to be unidirectional and seldom acyclic.

We give some detailed survey of previous works on network coding and error detection techniques in Section II. Our basic scheme is explained in Section III, and the proposed batch delivery and verification method can be found in Section IV. In Section V we give an analysis on the effectiveness of our batch delivery schemes using simulations. Section VI concludes.

II. Related Works

It is a well-known graph-theoretic result that the maximum capacity between a source and a sink connected through a network is the same as the maximum network flow f between them. When the network can be viewed as a directed acyclic graph with unit capacity edges, f is also the min-cut of the graph between the source and the sink. However, when there is a single source and multiple sinks, the maximum network flow f may not be achieved. A seminal work of *network coding*[10] reveals that if the nodes in a network can perform coding on the information they receive, it is possible for multiple sinks to achieve their max-flow bound simultaneously through the same network. This elegant result provides new insights into networking today since it now becomes possible to achieve the theoretical capacity bound if one allows the network nodes on the path to perform coding, instead of just the conventional tasks of routing and forwarding.

Later, Li et al. [11] showed that, although the coding performed by the intermediate nodes does not need to be linear, linear network codes are indeed sufficient to achieve the maximum theoretical capacity in acyclic synchronous networks. In their settings, each node computes some linear combination of the information it receives from its upstream nodes, and passes the results to its downstream nodes. However, to compute the network code (i.e., the correct linear combinations) that is to be performed by the nodes, the topology of the network has to be known beforehand, and has to be fixed during the process of content distribution. Furthermore, their algorithm is exponential in the number of edges in the network.

Koetter and Médard [12], [13] also considered the problem of linear network coding. They improved and extended the results by Li et al. [11], and considered the problem of link failures. They found that a static linear code is sufficient to handle link failures, if the failure pattern is known beforehand. However, as mentioned by Jaggi et al. [14], the code construction algorithm proposed by Koetter et al. still requires checking a polynomial identity with exponentially many coefficients.

Jaggi et al. [14] proposed the first centralized code construction algorithm that runs in polynomial time in the number of edges, the number of sinks, and the minimum size of the min-cut. They also noted that, although the results of Edmonds [18] shows that network coding does not improve the achievable transmission rate when all nodes except the source are sinks, finding the optimal multicast rate without coding is NP-hard. They also showed that if there are some nodes that are neither the source nor the sinks, then multicast with coding can achieve a rate that is $\Omega(\log|V|)$ times the optimal rate without coding, where $|V|$ is the number of nodes in the network. It is also shown in [14] that their method of code construction can handle link failures, provided that the failure pattern is known a priori.

Random network coding was proposed by Ho et al. [15] as a way to ensure the reliability of the network in a distributed setting where the nodes do not know the network topology, which could change over time. In their setting, each node would perform a random linear network coding, and the probability of successful recovery at the sinks can be tightly bounded. Chou et al. [3] proposed a scheme for content distribution based on random network coding in a practical setting, and showed that it can achieve nearly optimal rate using simulations. Recently, Gkantsidis and Rodriguez [4] proposed another scheme for large scale content distribution based on random network coding. They show by simulation that when applied to P2P overlay networks, using network coding can be 20 to 30 percent better than server side coding and 2 to 3 times better than uncoded forwarding, in terms of download time.

The problem of on-the-fly Byzantine fault detection in content distribution in P2P networks using random network coding is considered by Gkantsidis and Rodriguez [7], who noted that the verification techniques proposed by Krohn, Freedman and Mazières [1] can be employed to protect the integrity of the data without the knowledge of the entire content. The verification techniques were originally developed for content distribution using rateless erasure codes and were based on homomorphic cryptographic hash functions [16].

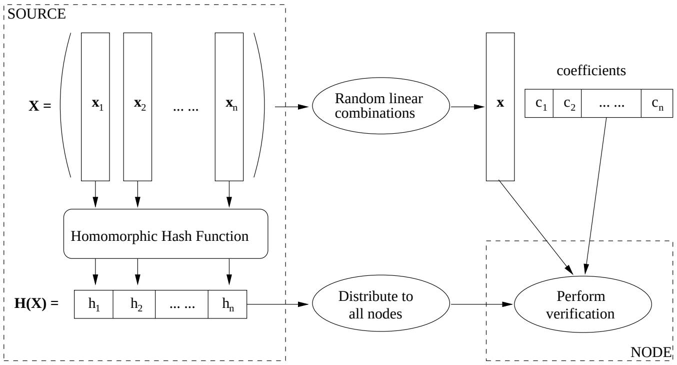

Fig. 1. On-the-fly Byzantine fault detection.

The overall picture of the on-the-fly detection technique in [1] is illustrated in Fig. 1. In their scheme, the content $\mathbf{X}$ is divided into n blocks $\mathbf{x}_1,\cdots,\mathbf{x}_n$, and a hash function $\mathscr{H}$ is applied on each blocks to obtain the hash values $h_1,\cdots,h_n$. The hash function $\mathscr{H}$ is homomorphic in the sense that for any $\mathbf{x}_i,\mathbf{x}_j$, it holds that $\mathscr{H}(\mathbf{x}_i)\mathscr{H}(\mathbf{x}_j) = \mathscr{H}(\mathbf{x}_i+\mathbf{x}_j)$. These hash values are distributed to all the nodes reliably using some other mechanisms. In fact, the authors propose to use the same technique recursively on the hash values until the final hash value is small enough to be distributed without coding. After receiving a coded block $\mathbf{x}$, which is a linear combination of the original n blocks with coefficients $C = \langle c_1, \cdots, c_n \rangle$, a node will be able to verify the integrity of $\mathbf{x}$ using $\mathbf{x}$, C, and the hash values $h_1, \cdots, h_n$, making use of the homomorphic property of $\mathscr{H}$. In particular, the node checks if the following holds

$$\mathscr{H}(\mathbf{x}) = \prod_{i=1}^{n} h_i^{c_i}. \tag{1}$$

As noted earlier, there are two major *limitations* when the above verification methods are applied. First of all, all nodes have to have the knowledge of all the hash values before any verification can be done. Unfortunately, the total size of the hash values is *proportional to the number of blocks* (e.g., for a 1 GB file, one has to distribute hash values of size around 8 MB). Hence there could be a significant delay before the nodes can receive and verify the actual data. Secondly, the computation of (1) is expensive, which hinders efficient verification with high speed networks. To make the scheme practical, the authors in [1] proposed a batch verification algorithm. In essence, they show that it is possible to verify a batch of packets by computing a random combination of the packets in the batch and check only the integrity of the combination. With this method, a corrupted packet can be detected with high probability.

Another simple and efficient on-the-fly verification scheme was proposed by Gkantsidis et al. [9]. Their scheme provides weaker security compared to that in [1], and it also suffers from the limitation that the size of the data required for verification is proportional to the size of the content, and they have to be distributed beforehand.

III. The Basic Security Scheme

In this section, we present our basic security scheme. We also provide the necessary background on a trapdoor homomorphic hash function, which we will use in later encoding and verification algorithms.

In Fig. 2 we illustrate our basic technique. In this basic scheme, the source node chooses a seed S, and feed it to a pseudo-random generator $\mathscr{G}$. Instead of computing the hash values for each data block, the source uses the random numbers $h_1, \cdots, h_n$ generated by $\mathscr{G}$ as the "intended" hash values. Next, given the original data $\mathbf{X}$ and the hash values $h_1, \cdots, h_n$, the source inverts the hash function $\mathscr{H}$ to get a list of paddings $d_1, \cdots, d_n$, such that when a block $\mathbf{x}_i$ is padded with d_i to form the new message block $\widehat{\mathbf{x}}_i$, it holds that $\mathscr{H}(\widehat{\mathbf{x}}_i) = h_i$. Note that $\mathscr{H}$ can only be inverted using a secret key that is known only by the source. Now, since the hash values can be computed from S and public function $\mathscr{G}$, there would be no need to distribute all the hash values, and it suffices if all the nodes knows S. In fact, S can be the SHA-1 hash of some public information of the content that is to be distributed (e.g., its unique identifier). Hence, even distributing S would be unnecessary.

A. A Trapdoor Permutation

As mentioned in the introduction, our scheme is based on a trapdoor homomorphic hash function using an invertible permutation that is built upon the one-way trapdoor permutation proposed in [19]. We will use this permutation as a building block in our construction.

Let $N = pq$ be the product of two large primes p and q. Let $\lambda = \mathrm{lcm}((p-1),(q-1))$ be the Carmichael's function of N, which is the maximal order for any element in $\mathbb{Z}_N^*$. Note that the Carmichael's function of N^2 is λN. Let $o(x)$ denote the order of x in the multiplicative group $\mathbb{Z}_{N^2}^*$, and we use $x \mid y$ to denote that x divides y. We choose $g, h \in \mathbb{Z}_{N^2}$ such that $o(g) = \alpha$ and $o(h) = N$, where $1 < \alpha < \lambda$ and $\gcd(\alpha, N) = 1$. Note that in this case, $o(gh) = \alpha N$.

We use $\mathrm{CRT}[(x_1, p_1), (x_2, p_2)]$ to denote the unique element $x \in \mathbb{Z}_{p_1, p_2}$ such that $x = x_1 \mod p_1$ and $x = x_2 \mod p_2$, where $\gcd(p_1, p_2) = 1$. Given x_1 and x_2, such x can be computed using the Chinese Remainder Theorem. Conversely, given any $x \in \mathbb{Z}_{p_1 p_2}$, we have $x_1 = x \mod p_1$ and $x_2 = x \mod p_2$. We follow [19] and define the function $L(\cdot)$ as $L(u) = (u-1)/N$.

Now we give our algorithms for the permutation and the inverse permutation.

Algorithm Permutation $\mathscr{P}(x, g, h, \alpha, N)$**:** Given $x \in \mathbb{Z}_{N^2}$, split x into x_1, x_2 and x_3 such that $x = \mathrm{CRT}[(x_1, \alpha), (x_2, N)] + \alpha N x_3$. That is, we compute $x_3 = \lfloor \frac{x}{\alpha N} \rfloor$, $x' = x \mod \alpha N$, $x_1 = x' \mod \alpha$, and $x_2 = x' \mod N$. Note that $x_1 \in \mathbb{Z}_\alpha$, $x_2, x_3 \in \mathbb{Z}_N$ and $x_3 \leq \lceil N/\alpha \rceil$. Next, compute permutation y as

$$y = g^{x_1} h^{x_2} x_3^N \mod N^2. \tag{2}$$

Note that $g^{x_1} h^{x_2} = (gh)^{\mathrm{CRT}[(x_1, \alpha), (x_2, N)]} \mod N^2$. This permutation essentially maps every point x in $\mathbb{Z}_{N^2}$ to another distinct point y in the same domain.

Algorithm Inverse Permutation $\mathscr{I}(y, g, h, \alpha, N, \lambda)$**:** Given $y \in \mathbb{Z}_{N^2}$, compute

$$x' = \frac{L(y^\lambda \mod N^2)}{L(h^\lambda \mod N^2)} \mod \alpha N \tag{3}$$

and compute x_1 and x_2 such that $x' = \mathrm{CRT}[(x_1, \alpha), (x_2, N)]$

$$x_1 = x' \mod \alpha, \qquad x_2 = x' \mod N. \tag{4}$$

Next, compute $y' = y(gh)^{-x'} \mod N$, and finally

$$x_3 = y'^{\frac{1}{N} \mod \lambda} \mod N. \tag{5}$$

Hence, x can be recovered as $x = \mathrm{CRT}[(x_1, \alpha), (x_2, N)] + \alpha N x_3$. In other words, given any point $y \in \mathbb{Z}_{N^2}$, we can efficiently find an x such that x is mapped to y under the permutation algorithm, provided that λ is known.

Our permutation is very similar to that proposed by Paillier in [19]. The difference is that Paillier chooses a random element in $\mathbb{Z}_{N^2}$ with order αN for some random α, whereas we choose the value of α and represent the element in the form of the product (gh). Hence the correctness of the permutation directly follows from that of Paillier's scheme.

The one-wayness of the permutation depends on the difficulty to find discrete logarithm in $\mathbb{Z}_{N^2}^*$. If we view $\mathscr{P}$ as a

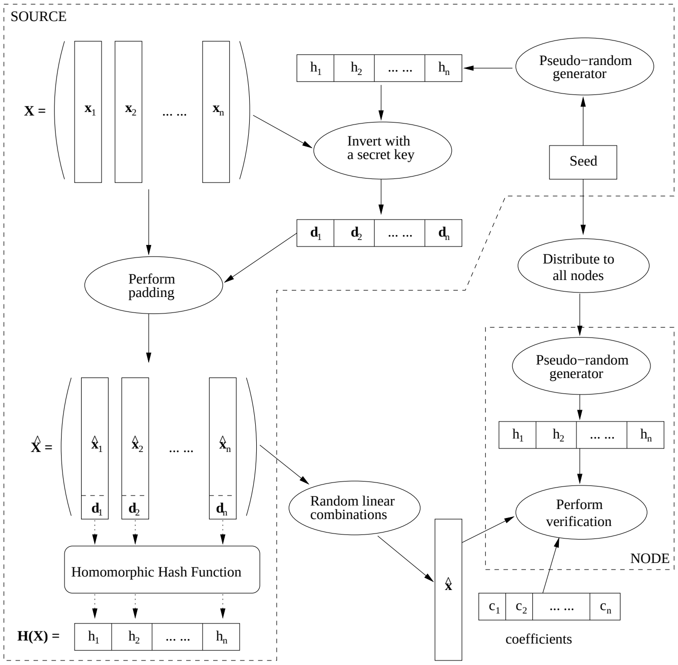

Fig. 2. The basic security scheme.

function of $x_1 = x \mod \alpha$, and make x_2 and x_3 public parameters, $\mathscr{P}$ remains one-way. Formally, we define $DL[g,\alpha,N^2]$ to be the computational problem: Given y, g and N, where $o(g) = \alpha$ in $\mathbb{Z}^*_{N^2}$, find an x such that $y = g^x \mod N^2$. Hence, we have

Theorem 1 *Given $x_2 \in \mathbb{Z}_N$, $x_3 \in \mathbb{Z}^*_N$ and $x_3 \leq \lfloor (N/\alpha) \rfloor$, $y \in \mathbb{Z}^*_{N^2}$, and public parameters g,h,α,N, it is computationally infeasible to find an x_1 such that $\mathscr{P}(x,g,h,N,\alpha) = y$ where $x = CRT[(x_1,\alpha),(x_2,N)] + \alpha N x_3$, if $DL[g,\alpha,N^2]$ is hard.*

Proof: To prove the theorem, it suffices to show that if the permutation can be efficiently inverted, we can solve the problem $DL[g,\alpha,N^2]$ efficiently. In particular, assume there is a polynomial time algorithm A such that given x_2, x_3, y as the above and public parameters g,h,α,N, the algorithm A outputs $x_1 = A(x_2,x_3,y)$ such that $\mathscr{P}(x,g,h,N,\alpha) = y$ with a probability p that is not negligible, where $x = \mathrm{CRT}[(x_1,\alpha),(x_2,N)] + \alpha N x_3$.

Now, given g,α,N and w, we construct a polynomial algorithm B which performs the following steps.

1) Find an h such that the order of h is N in $\mathbb{Z}^*_{N^2}$.
2) Randomly choose $x_2 \in \mathbb{Z}_N$ and $x_3 \in \mathbb{Z}^*_N$ such that $x_3 \leq \lfloor (N/\alpha) \rfloor$.
3) Compute $y = \mathscr{P}(x,g,h,N,\alpha)$.
4) Output $A(x_2,x_3,y)$.

It is clear that algorithm B runs in polynomial time and outputs x such that $w = g^x \mod N^2$ with probably at least p. ■

Essentially, the above theorem shows that, while it is easy to invert the permutation if λ (or equivalently, the factorization of N) is known, it would be infeasible to perform the inversion otherwise. Note that Theorem 1 is a necessary condition for the security of our scheme.

Homomorphic Property: The permutation $\mathscr{P}$ has the following homomorphic property: Given $x,y,z \in \mathbb{Z}_{N^2}$ such that $x = \mathrm{CRT}[(x_1,\alpha),(x_2,N)] + \alpha N x_3$, $y = \mathrm{CRT}[(y_1,\alpha),(y_2,N)] + \alpha N y_3$, and $z = \mathrm{CRT}[(z_1,\alpha),(z_2,N)] + \alpha N z_3$, we have $\mathscr{P}(x)\mathscr{P}(y) = \mathscr{P}(z)$ if and only if $z_1 = x_1 + y_1 \mod \alpha$, $z_2 = x_2 + y_2 \mod N$, and $z_3 = x_3 y_3 \mod N$.

Choosing the Parameters: To choose the appropriate parameters for the permutation the following conditions need to be satisfied.

- Factoring N is hard.
- $p-1$ and $q-1$ have large distinct prime factors p' and q' respectively. In this case the discrete logarithm on $\mathbb{Z}_p$ and $\mathbb{Z}_q$ is hard.
- α should be large enough and should contain only large prime factors. For example, one can choose $\alpha = p'q'$.

B. A Trapdoor Homomorphic Hash Function

We choose parameters p,q,N,α,g,h as in Section III-A. Furthermore, we randomly select $m+1$ numbers $u_0,u_1,\cdots,u_m$ from $\mathbb{Z}_\alpha$. Next, we compute $g_i = g^{u_i} \mod N^2$ for all $0 \leq i \leq m$. The public parameters of the hash function is N, $g_0,g_1,\cdots,g_m$. whereas the factorization of N (or equivalently λ) and $u_0,\cdots,u_m$ should be kept secret.

Assume that each message is of the form: $\mathbf{x} = (x_0,x_1,\cdots,x_m,d,r)$ where $x_i \in \mathbb{Z}_\alpha$ for $0 \leq i \leq m$, $d \in \mathbb{Z}_N$, and $r \leq \lfloor N/\alpha \rfloor$. The hash of $\mathbf{x}$ is computed as

$$\mathscr{H}(\mathbf{x}) = \left(\prod_{i=0}^{m} g_i^{x_i} \right) h^d r^N \mod N^2. \qquad (6)$$

Based on this construction, we have

$$\begin{aligned}\mathscr{H}(\mathbf{x}) &= g^{\sum_{i=0}^{m} u_i x_i} h^d r^N \mod N^2 \\ &= \mathscr{P}(\mathrm{CRT}[(\sum_{i=0}^{m} u_i x_i, \alpha), (d, N)] + \alpha N r, g, h, \alpha, N).\end{aligned} \tag{7}$$

For any two messages $\mathbf{x} = (x_0, \cdots, x_m, d_x, r_x)$ and $\mathbf{y} = (y_0, \cdots, y_m, d_y, r_y)$, we define the *addition* (represented by $\dotplus$) of $\mathbf{x}$ and $\mathbf{y}$ as

$$\begin{aligned}&\mathbf{x} \dotplus \mathbf{y} = (z_0, \cdots, z_m, d_z, r_z), \text{ where} \\ &\quad z_i = x_i + y_i \mod \alpha, \text{ for } 0 \le i \le m \\ &\quad d_z = d_x + d_y \mod N \\ &\quad r_z = r_x r_y \mod N.\end{aligned} \tag{8}$$

Hence, from the results in Section III-A, this hash function has the following homomorphic property.

$$\mathscr{H}(\mathbf{x})\mathscr{H}(\mathbf{y}) = \mathscr{H}(\mathbf{x} \dotplus \mathbf{y}). \tag{9}$$

The security of $\mathscr{H}$ is defined in terms of the difficulty in finding collisions. In particular, we have

Definition 1 *A hash function h is collision-free if it is computationally infeasible to find two messages x_1 and x_2 such that $h(x_1) = h(x_2)$.*

It can be shown that the hash function $\mathscr{H}$ is indeed collision-free if discrete logarithm on $\mathbb{Z}_{N^2}^*$ is hard, using an argument similar to that in [16] (proof of Theorem 3.4). In other words, it would be infeasible to find two messages having the same hash value without the knowledge of λ (or equivalently the factorization of N).

Theorem 2 *The hash function $\mathscr{H}$ is collision-free if $DL[g, \alpha, N^2]$ is hard.*

Proof: We prove this theorem by showing that if there is a polynomial time algorithm A that finds a collision in $\mathscr{H}$ with probability p that is not negligible, we can use it to construct a polynomial time algorithm B that solves the problem $DL[g, \alpha, N^2]$ with probability that is not negligible.

Given g, h, N, α and y, algorithm B chooses $v_0, \cdots, v_m \in \{0, 1\}$ and $u_0, \cdots, u_m \in \mathbb{Z}_N$ at random. For $i = 0, \cdots, m$, the algorithm computes

$$g_i = \begin{cases} g^{u_i} \text{ if } v_i = 0 \\ y^{u_i} \text{ if } v_i = 1. \end{cases} \tag{10}$$

Next, algorithm B invokes algorithm A and finds a collision on the hash function $\mathscr{H}$ with parameters $g_0, \cdots, g_m, h$. If A fails, B declares that it has failed and halt. Otherwise, let the distinct messages having the same hash values be

$$w = (w_0, \cdots, w_m, d_1, r_1) \text{ and } z = (z_0, \cdots, z_m, d_2, r_2).$$

Now, let $a = \sum_{v_i=1} u_i (w_i - z_i) \mod N$. If the inverse of a does not exist in $\mathbb{Z}_\alpha^*$, algorithm B declares that it has failed and halts. Otherwise, the algorithm computes an inverse b of a in $\mathbb{Z}_\alpha^*$, and outputs $x = b \cdot \sum_{v_i=0} u_i (z_i - w_i) \mod \alpha$ and halts. Note that the inverse can be computed (or its existence determined) using Euclid's algorithm in polynomial time.

Since B only invokes A once and other computations can all be done in polynomial time, B itself halts in polynomial time. Now we examine the probability that B succeeds.

Note that the distribution of $g_0, \cdots, g_m$ is uniform and independent, which is the same for $\mathscr{H}$. Hence the algorithm A succeeds with probability at least p. Since the two messages w and z forms a collision, we have

$$\left(\prod_{i=0}^{m} g_i^{w_i}\right) h^{d_1} r_1^N = \left(\prod_{i=0}^{m} g_i^{z_i}\right) h^{d_2} r_2^N \mod N^2.$$

Considering (10) and rearranging the items we have

$$\left(\prod_{v_i=1} y^{u_i(w_i - z_i)}\right) h^{d_1} r_1^N = \left(\prod_{v_i=0} g^{u_i(z_i - w_i)}\right) h^{d_2} r_2^N \mod N^2.$$

Suppose there is an x such that $y = g^x \mod N^2$, we have

$$g^{x \sum_{v_i=1} u_i(w_i - z_i)} h^{d_1} r_1^N = g^{\sum_{v_i=0} u_i(z_i - w_i)} h^{d_2} r_2^N \mod N^2.$$

However, since $\mathscr{P}$ is a permutation, we have

$$\begin{aligned} x \sum_{v_i=1} u_i(w_i - z_i) &\equiv \sum_{v_i=0} u_i(z_i - w_i) \mod \alpha \\ d_1 &\equiv d_2 \mod N \\ r_1 &\equiv r_2 \mod N. \end{aligned}$$

Therefore, we have

$$x \equiv b \cdot \sum_{v_i=0} u_i(z_i - w_i) \mod \alpha.$$

The algorithm B could fail, however, when the inverse of a does not exists in $\mathbb{Z}_\alpha$. This can happen only when (1) $a = 0$, or (2) $a > 0$ but $\gcd(a, \alpha) \ne 1$. As noted in the proof of Theorem 3.4 in [16], the probability that case 1 happens is at most $1/2$. Since we choose α in such a way that it only contains large prime factors, the probability p' that case 2 happens is negligible. Therefore, algorithm B will succeed with probability $(p/2 - p')$, which is not negligible. ■

C. The Basic Secured Encoding and Verification Schemes

Our proposed scheme consists of two algorithms, namely the *encoding* algorithm where the original data are coded for distribution, and the *verification* algorithm, which is used by individual nodes to verify the integrity of the received data.

Basic Encoding Algorithm: Suppose the data $\mathbf{X}$ we want to encode is of the form $\mathbf{X} = (\mathbf{x}_1, \cdots, \mathbf{x}_n)$, where $\mathbf{x}_i = (x_{i,1}, \cdots, x_{i,m})^T$ for all $1 \le i \le n$, and $|x_{i,j}| = \gamma$ for all $1 \le i \le n$ and $1 \le j \le m$. We require that $n < m$.

Firstly we choose the parameters p, q, N, α, g, h as in Section III-A, such that $\alpha > 2^\gamma$ and each $x_{i,j}$ can be represented by an element in $\mathbb{Z}_\alpha$. Next, we generate $g_0, \cdots, g_m$ as in Section III-B. Now, given data $\mathbf{X}$, the encoder performs the following steps.

1) Choose a random seed S and a pseudo-random number generator $\mathscr{G}$. [1]
2) Generate pseudo-random numbers $h_1,\cdots,h_n \in \mathbb{Z}^*_{N^2}$ from $\mathscr{G}$ with S.
3) For each $1 \le i \le n$, compute $e_i = \mathscr{I}(h_i)$, and write $e_i = \mathrm{CRT}[(e'_i,\alpha),(d_i,N)] + \alpha N r_i$.
4) For each $1 \le i \le n$, compute $x_{i,0} = (e'_i - \sum_{j=1}^m x_{i,j}u_j)u_0^{-1}$ mod α.
5) Let $\widehat{\mathbf{X}} = \left(\widehat{\mathbf{x}}_1,\cdots,\widehat{\mathbf{x}}_n\right)$, where $\widehat{\mathbf{x}}_i = (x_{i,0},x_{i,1},\cdots,x_{i,m},d_i,r_i)^T$ for all $1 \le i \le n$.
6) Output $\widehat{\mathbf{X}}$, the pseudo-random generator $(S,\mathscr{G})$, and the hash function $(g_0,\cdots,g_m,h,N,\alpha)$.

In summary, we randomly "choose" the hash values for each data block, and pad the original data such that the hash of each data block is the same as the chosen ones. Note that Step 3 is always possible since $\mathscr{P}$ is a permutation. In this way, each node only needs to know the seed S and the function $\mathscr{G}$ to compute the hash values. In practice, the need for distributing S can be further eliminated by using a public random function. For example, S can be the SHA-1 hash of the original file identifier, creation date, publisher, and other data that are public and should be known to all the receivers before the download session begins[2].

Basic Verification Algorithm: During verification, each network node is given a packet $\langle \mathbf{x},\mathbf{c}\rangle$ and public information $\mathbf{t}$. In the case where this packet is not tampered with, $\mathbf{c} = (c_1,\cdots,c_n)$ are the coefficients where each $c_i \in \mathbb{Z}_N$, $\mathbf{x}$ is the linear combination $\mathbf{x} = \sum_{i=1}^n c_i\widehat{\mathbf{x}}_i$, where the addition is $\dot{+}$ as defined in (8), and $\mathbf{t}$ represents public parameters S,$\mathscr{G}$,$g_0,\cdots,g_m$, h, N and α.

Each node can verify the integrity of the packet as the following.

1) From S and $\mathscr{G}$, compute $h_1,\cdots,h_n \in \mathbb{Z}^*_{N^2}$.
2) Compute the hash value $H_1 = \mathscr{H}(\mathbf{x})$.
3) Compute the hash value $H_2 = \prod_{i=1}^n h_i^{c_i} \mod N^2$.
4) Verify that $H_1 = H_2$.

D. Security Analysis of the Basic Schemes

Intuitively, an attack is considered as successful if the attacker can find a packet $(\mathbf{p},\mathbf{c})$ such that $\mathbf{p}$ is not a linear combination of the original data specified by $\mathbf{c}$ but the packet still passes the verification. Here we have

Definition 2 *The basic schemes are secure if it is computationally infeasible to find* $\widehat{\mathbf{X}} = (\widehat{\mathbf{x}}_1,\cdots,\widehat{\mathbf{x}}_n)$, $\mathbf{y}$ *and* $\mathbf{c} = (c_1,\cdots,c_n)$ *such that for* $\mathbf{x} \triangleq \sum_{i=1}^n c_i\widehat{\mathbf{x}}_i$, *we have* $\mathbf{y} \ne \mathbf{x}$ *and* $\mathscr{H}(\mathbf{x}) = \mathscr{H}(\mathbf{y})$.

We show that the basic schemes are secure by showing that otherwise we can easily find a collision in $\mathscr{H}$.

Theorem 3 *The basic schemes are secure if* $DL[g,\alpha,N^2]$ *is hard.*

Proof: Suppose on the contrary that an adversary A can find such $\widehat{\mathbf{X}}$, $\mathbf{y}$ and $\mathbf{c}$ with a probability p that is not negligible. Now we use A to construct an algorithm B as below.

- Invoke A, and let $\widehat{\mathbf{X}} = (\widehat{\mathbf{x}}_1,\cdots,\widehat{\mathbf{x}}_n)$, $\mathbf{y}$ and $\mathbf{c} = (c_0,\cdots,c_n)$ be the output.
- Output $\mathbf{x} \triangleq \sum_{i=0}^n c_i\widehat{\mathbf{x}}_i$ and $\mathbf{y}$.

Clearly, if A successfully finds $\widehat{\mathbf{X}}$, $\mathbf{y}$ and $\mathbf{c}$ such that $\mathbf{y} \ne \mathbf{x}$ and $\mathscr{H}(\mathbf{x}) = \mathscr{H}(\mathbf{y})$ with probability p, B would find a collision $\mathbf{x}$ and $\mathbf{y}$ in $\mathscr{H}$ with the same probability p, which is not negligible.

However, if $DL[g,\alpha,N^2]$ is hard, by Theorem 2 the hash function $\mathscr{H}$ is collision free, and thus p should be negligible, which is a contradiction. Therefore, the basic schemes are secure if the discrete logarithm $DL[g,\alpha,N^2]$ is hard. ■

IV. Extended Security Scheme with Batch Verification

Our main idea is the following. When a node computes the padding for the message blocks it is going to send to its downstream, it only computes some small padding for a batch of blocks instead of for every block. In this way, the communication overhead caused by the padding is reduced by a factor of k, where k is the number of blocks per batch.

In particular, our proposed batch verification is a challenge-response process between the verifier and its upstream node. The algorithm is as illustrated in Fig. 3. However, we note that the challenge step can be avoided with carefully designed protocols. We show such an example in Section IV-A.

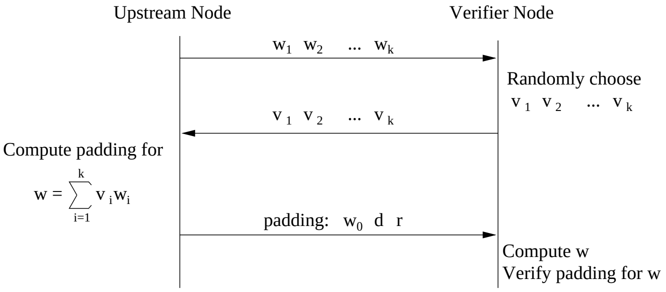

Fig. 3. Batch verification.

Suppose each batch contains the packets $\langle\mathbf{w}_1,\mathbf{c}_1\rangle,\cdots,\langle\mathbf{w}_k,\mathbf{c}_k\rangle$, where each $\mathbf{w}_i$ is a linear combination of the original data blocks. In particular, let $\mathbf{c}_i = (c_{i,1},\cdots,c_{i,n})$ be the coefficients, we have $\mathbf{w}_i = (w_{i,1},\cdots,w_{i,m})^T = \sum_{j=1}^n c_{i,j}\mathbf{x}_j$. We assume that the upstream node knows the correct padding for each of the packets, which can be computed from the paddings for the packets received from its own upstream nodes. That is, the upstream node keeps $w_{i,0} = \sum_{j=1}^n c_{i,j}x_{i,0}$, $d_i = \sum_{j=1}^n c_{i,j}d_i$, $r_i = \prod_{j=1}^n r_i^{c_{i,j}}$.

After receiving this batch of packets, the verifier generates a list of random coefficients as the challenge $\mathbf{v} = (v_1,\cdots,v_k)$.

[1] Generally $\mathscr{G}$ and S should be chosen such that the output is independent of other random coin tosses made by the encoder. For instance, $\mathscr{G}$ should not happen to reveal a collision of $\mathscr{H}$. In practice, it is sufficient to choose such $\mathscr{G}$ and S randomly.

[2] In case these data are not sufficiently random, however, the seed should be explicitly chosen and distributed by the source.

Next, the verifier sends $\mathbf{v}$ to its upstream from whom it received the batch of packets. After that, the upstream node computes the linear combination of the batch of packets using $\mathbf{v}$ as the coefficients. That is, $\mathbf{w} = \sum_{j=1}^{n} v_j \mathbf{w}_j$.

Finally, the upstream node generates the padding for $\mathbf{w}$ as $w_0 = \sum_{j=0}^{n} v_j w_{j,0}$, $d = \sum_{j=1}^{n} v_j d_j$, $r = \prod_{j=1}^{n} r_j^{v_j}$, and sends the tuple (w_0, d, r) to the verifier. The verifier computes $\mathbf{w}$ in the same way, and verify the padding $\mathbf{v}$ with $\mathbf{w}$ as in the basic scheme. It is worth to note that $\mathbf{w}$ is computed locally by both nodes and is never transmitted.

A. Eliminating the Challenge Step

The batch verification presented above can be further simplified and the challenge step can be avoided. This is possible because the purpose of the challenge step is to allow the verifier to pick a "random" linear combination of the received blocks and perform verification on that block. If the upstream node is allowed to choose arbitrary coefficients, it may be able to cheat. Nevertheless, if the upstream node can somehow "prove" that its choices of the coefficients are indeed random, the challenge can be avoided.

A standard technique to achieve such a proof is well studied in the literature of zero-knowledge proofs. Such a technique employs a random oracle accessible by both parties [20], which can be used to transform interactive protocols to equivalent non-interactive protocols. One simple way of implementing such a random oracle is to make use of a publicly known pseudo-random generator G, and let the seed to the generator depend on some random value. For example, the upstream node can calculate the SHA-1 hash value over all the blocks in the batch, and use that as the seed to generate the random coefficients.

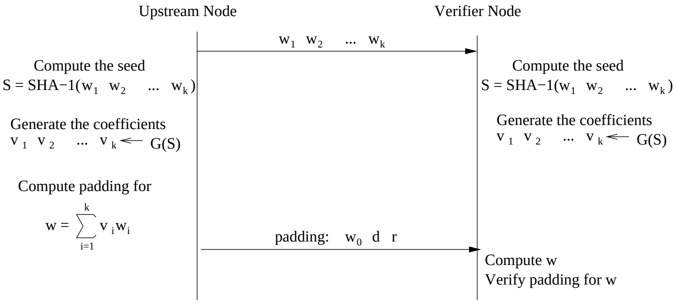
Upstream Node
Verifier Node
w_1 w_2 ... w_k
Compute the seed
$S = SHA-1(w_1\ w_2\ ...\ w_k)$
Generate the coefficients
$v_1\ v_2\ ...\ v_k \leftarrow G(S)$
Compute padding for
$w = \sum_{i=1}^{k} v_i w_i$
padding: w_0 d r
Compute the seed
$S = SHA-1(w_1\ w_2\ ...\ w_k)$
Generate the coefficients
$v_1\ v_2\ ...\ v_k \leftarrow G(S)$
Compute w
Verify padding for w

Fig. 4. Batch verification without challenge.

The verification process is illustrated as in Fig. 4. In particular, given a batch $\mathbf{w}_1, \cdots, \mathbf{w}_k$, the upstream node computes the SHA-1 digest S of all the data in the batch. Next, a random number generator G is applied to generate random coefficients $v_1, \cdots, v_k$, where the digest S is used as the seed. After that, a combination of the packets is computed using the random coefficients $\mathbf{w} = \sum_{i=1}^{k} v_i \mathbf{w}_i$. Finally, the padding (w_0, d, r) for $\mathbf{w}$ is computed. This batch and the padding is transmitted to the downstream node, who follow the same method used by the upstream node to compute the random coefficients and then verify the padding for the linear combination $\mathbf{w}$.

Remarks: Since all the nodes have the pseudo-random generator $\mathscr{G}$ as a public parameter, they can re-use it as the generator G in this case. Furthermore, although we used SHA-1 hash of the data block as the seed, in fact we can derive the seed in any one-way manner.

In Fig. 4, the verifier node obtains one piece of padding after the verification is done. However, we can let both nodes to generate more random coefficients using the seed, and hence allow the verifier node to verify multiple (say, τ) pieces of padding. Typically $\tau < k$.

B. A Practical Batch Content Distribution Scheme

Now we give a practical scheme where content is distributed and verified in batches. We assume that every node (except the source) has at least k upstream nodes at any point of time (e.g., $k = 16$). We also assume that each node sends packets in batches, and each batch contains k packets. Furthermore, we assume that all the nodes except the source are interested in the content, as in the case of most file-sharing applications.

For any node A, after receiving and verifying a batch of message blocks from one of its upstream nodes, node A knows the proper padding for τ random linear combinations of the message blocks. Now, since node A has at least k upstream nodes, it will receive at least k batches of packets. Hence, node A knows the padding for the τk linear combinations generated from the k batches of message blocks. In this way, node A can use those τk linear combinations to generate batches of packets and deliver them to its downstream nodes. This process continues until the content is no longer needed to be propagated.

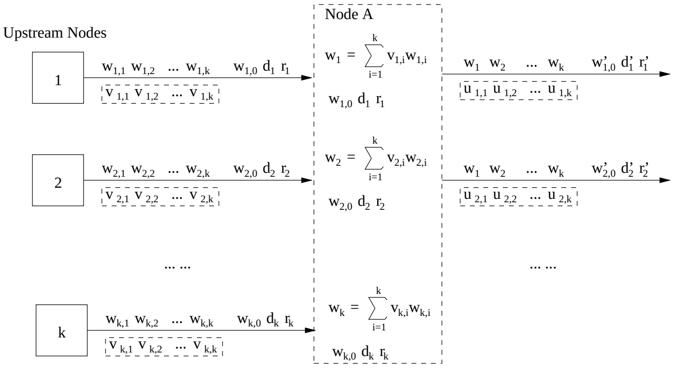
Upstream Nodes
Node A
1
2
k
$w_{1,1}$ $w_{1,2}$ ... $w_{1,k}$ $w_{1,0}$ d_1 r_1
$v_{1,1}$ $v_{1,2}$ ... $v_{1,k}$
$w_{2,1}$ $w_{2,2}$ ... $w_{2,k}$ $w_{2,0}$ d_2 r_2
$v_{2,1}$ $v_{2,2}$ ... $v_{2,k}$
... ...
$w_{k,1}$ $w_{k,2}$ ... $w_{k,k}$ $w_{k,0}$ d_k r_k
$v_{k,1}$ $v_{k,2}$ ... $v_{k,k}$
$w_1 = \sum_{i=1}^{k} v_{1,i} w_{1,i}$
$w_{1,0}$ d_1 r_1
$w_2 = \sum_{i=1}^{k} v_{2,i} w_{2,i}$
$w_{2,0}$ d_2 r_2
$w_k = \sum_{i=1}^{k} v_{k,i} w_{k,i}$
$w_{k,0}$ d_k r_k
w_1 w_2 ... w_k $w'_{1,0}$ d'_1 r'_1
$u_{1,1}$ $u_{1,2}$ ... $u_{1,k}$
w_1 w_2 ... w_k $w'_{2,0}$ d'_2 r'_2
$u_{2,1}$ $u_{2,2}$ ... $u_{2,k}$
... ...

Fig. 5. File-sharing with batch delivery ($\tau = 1$).

This process is illustrated in Fig. 5, where $\tau = 1$ for simplicity. From the i-th upstream node, node A receives a batch of message blocks $\mathbf{w}_{i,1}, \cdots, \mathbf{w}_{i,k}$, and the padding $(\mathbf{w}_{i,0}, d_i, r_i)$ for the linear combination $\mathbf{w}_i = \sum_{j=1}^{k} v_{i,j} \mathbf{w}_{i,j}$, where $v_{i,j}$'s are random coefficients generated during batch verifications. Hence, for the i-th downstream node, node A sends the batch of message blocks $\mathbf{w}_1, \cdots, \mathbf{w}_k$, and the padding $(\mathbf{w}'_{i,0}, d'_i, r'_i)$ for the linear combination $\mathbf{w}'_i = \sum_{j=1}^{k} u_{i,j} \mathbf{w}_j$, where $u_{i,j}$'s are the coefficients generated during the batch verification between node A and its i-th downstream node.

This scheme has the following advantages compared to previous schemes.

- Each node A only has to cache τk blocks of data (in the above case, $\mathbf{w}_1, \cdots, \mathbf{w}_k$), which are linear combinations of the received batches. Each node only delivers combinations of the cached data to its downstream nodes. Hence it reduces the storage and computational costs to τ/k of that would occur with previous schemes.
- The communication overhead for the verification is reduced to τ/k of that in the basic scheme, hence the overhead is $O(k/\tau)$ times less than that of the batch verification technique in [1].

V. Evaluation of the Batch Delivery Scheme

A. Communication Overhead

Here we study the communication overhead of our scheme. This overhead refers to the amount of data we need to distribute to each node for the purpose of the verification of the integrity of the application data. The actual communication overhead largely depends on the parameters chosen for the actual implementations.

In the scheme proposed by Krohn et al. [1], the file to be distributed is divided into blocks of 16 KB, and the parameters chosen for the homomorphic hash function would generate a hash value of size 1024 bits per data block. Hence, the total size of the "first-order" hash values would be about 0.78% of the original data. For a file of size 1 GB, their method would require hash values of size 8 MB. To distribute these hash values, the authors in [1] propose to recursively apply the same scheme on the 8 MB hash values, which would generate more "second" or higher order of hash values. The size of the high order hash values constitutes about 0.01% of the size of the original file. Hence the total overhead is about 0.79%, or about 8.1 MB for a 1 GB file.

For fair comparisons, we choose parameters that are comparable to those given in [1]. In particular, we choose N to be 1024 bit long, and the primes $|p'| = |q'| = 160$, hence $|\alpha| = 320$. We also choose 16 KB as the size of the data block. Recall that our scheme requires padding of three values x_0, d and r, such that $x_0 \in \mathbb{Z}_\alpha$, $d, r \in \mathbb{Z}_N$ and $r \leq \lfloor N/\alpha \rfloor$. Hence, the total padding for one packet is 2048 bit long, or 1.56% of the original data. However, since the padding is done only once for every batch, the overhead is reduced by a factor of k/τ. As we show later in this section, we conducted two sets of experiments with τ fixed to be 2, and $k = 8$ and $k = 16$ respectively. Hence, the communication overhead for the two sets of experiments would be $(2/8) \cdot 1.56\% \approx 0.39\%$ and $(2/16) \cdot 1.56\% \approx 0.20\%$ respectively. For a file of size 1 GB, the above percentiles are equivalent to about 4 MB and 2 MB respectively.

Note that the above calculation is done based on the assumption that every node in the network receives the entire file eventually. If some of the intermediate nodes receive only a fraction of the data, the overhead would be much less than 4 MB (resp. 2 MB), which would be 0.39% (resp. 0.20%) of the size of the data they receive. Whereas in the scheme in [1], each node has to obtain all 8 MB hash values regardless of the amount of actual data it needs to verify.

B. Start-Up Latency

At the beginning of a content distribution session, the source and all the nodes participating the distribution have to agree on the set of parameters used for the coding and verification.

Recall that the public parameters in our scheme are $(g_0, \cdots, g_m, h, N, \alpha)$. When we choose the block size to be 16 KB, $|N| = 1024$ and $|\alpha| = 320$, we have $m = 410$ and the total size of the parameters is approximately 16.3 KB. With these parameters it would be sufficient for any node to perform verification of any given padded packet. Assuming that the bandwidth between a node and the source (or any other node from which these parameters are distributed) is 1 Mbps, it would take less than 0.13 seconds before the node is ready to perform verification. The start-up latency in our scheme is fixed once the parameters for the hash function and the block size are chosen, and is independent of the size of the content to be distributed.

On the other hand, for the scheme in [1], the size of all the public parameters is the same as the size of the data in a packet, which is 16 KB and it takes 0.125 seconds to be transmitted on the same link. However, when the node needs to receive 8 MB hash values of a 1 GB file as in the example given in [1], it would require 64 seconds, with the same 1 Mbps link. The start-up latency is proportional with the size of the file. This would hinder the application of the scheme in latency critical scenarios such as real time streaming.

C. Effectiveness of the Batch Random Network Coding

Recall that our batch distribution deviates from the random linear network coding scheme [15] by allowing each node to compute and send the linear combination of only some of the packets it receives. Since it is difficult to analytically study the effectiveness of our scheme, we investigated the effectiveness using experiments.

Following the classical theoretical framework on random network coding (such as [15]), we model the network as a random directed acyclic graph. In contrast with classical assumption that each edge in the network is used for the transmission of one symbol, we assume that each edge is able to transmit a batch of k symbols. We also assume that each node (except for the source node) has at least k incoming edges, and the total number of symbols is k^2.

A content delivery session starts with the formation of a random network of n nodes, and the source node begins to distribute combinations of all its data to its downstream nodes in batches. When a node has received one batch of packets on every incoming edge, it will generate random linear combinations of the τk blocks in its cache and pass them to its downstream nodes. This process continues until all edges are used. At the end of a content delivery session, we check the data received by every node, and determine if the node can successfully recover the original data $\mathbf{X}$. We use the success rate (i.e., the probability that a node successfully recovers **X**) as the measure of the effectiveness of our scheme, which we estimated by computing the average number of successful nodes over $n-1$. In classical works on random linear network coding, this rate can be arbitrarily close to 1.

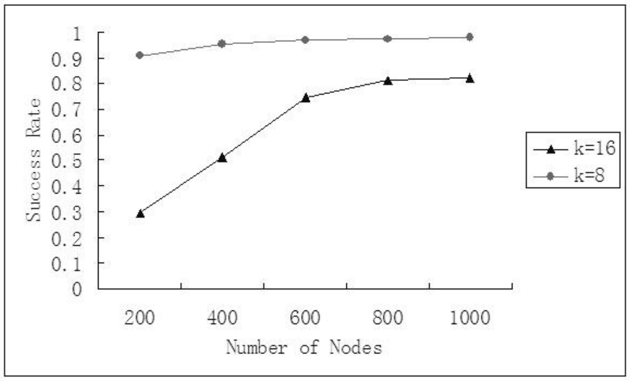

Fig. 6. Simulation results ($\tau = 2$).

In our simulations, we fix $\tau = 2$, and the modulo $p = 251$, which is large enough for the random linear network coding to have a success rate very close to 1. We conducted two sets of experiments for $k = 8$ and $k = 16$, and in each set we vary the number of nodes n from 200 to 1000. Each experiment is repeated for 15 times, and the average success rate is shown in Fig. 6. We have the following observations. From the figure it seems that our scheme works well when n is large or k is small, i.e., when the network is sparsely connected. It is not intuitive since more densely connected network should normally provide better capability in content distribution. However, in our scheme, the total number of symbols is k^2, which grows fast when k becomes larger. The better performance when k is small actually comes at the price of more computational and communication overhead. In fact, the ratio of the number cached blocks over total number of data blocks is τ/k and the overhead of verification can be considered as $c\tau/k$, where c is some constant that represents the overhead in the verification of one packet.

VI. Conclusion

In this paper we study the problem of on-the-fly detection of Byzantine errors during the content distribution process when the traditional "hash-and-sign" techniques are no longer feasible. In particular, we consider content distribution schemes utilizing network coding, where each "packet" of data consists of some linear combination of the original data to be distributed. A known technique proposed for rateless erasure codes uses homomorphic hash functions on the original data blocks, such that the hash value for any linear combination of blocks can be computed from the hash values for every individual block. However, this technique suffers from some major limitations that result in high start-up latency and inefficiencies in the verification.

We devise a new homomorphic hash function based on a modified trap-door one-way permutation as in [19]. We also propose a new paradigm where the hash values are generated from a pseudo-random number generator and the actual data are padded so that they are hashed to the pre-generated hash values. In this way, we allow each packet to carry its own authentication information so that high start-up latency can be avoided, and it also becomes possible to pad an entire batch of packets to save the bandwidth overhead. We further propose a batch distribution and verification scheme based on random linear network coding, such that each node can generate combined data blocks from a relatively smaller number of blocks. This not only allows each node to have a smaller cache, it also reduces the computational overhead. Although this deviates from the standard random linear network coding, we show by simulation that it is sufficiently effective in practice for small batches or large networks.

Acknowledgement: This work is supported in part by the Microsoft Research Fund and the MSRA-CUHK Lab.

References

[1] M. N. Krohn, M. J. Freedman, and D. Mazières, "On-the-fly verification of rateless erasure codes for efficient content distribution," in *IEEE Symp. Security and Privacy*, Oakland, CA, May 2004, pp. 226–240.

[2] S. Acedanski, S. Deb, M. Medard, and R. Koetter, "How good is random linear coding based distributed networked storage," in *Workshop on Network Coding, Theory and Applications*, Italy, April 2005.

[3] P. A. Chou, Y. Wu, and K. Jain, "Practical network coding," in *Allerton Conf. Commun., Contr., and Comput.*, October 2003, invited paper.

[4] C. Gkantsidis and P. R. Rodriguez, "Network coding for large scale content distribution," in *IEEE INFOCOM*, 2005, pp. 2235–2245.

[5] M. Wang, Z. Li, and B. Li, "A high-throughput overlay multicast infrastructure with network coding," in *IWQoS*, 2005.

[6] Y. Zhu, B. Li, and J. Guo, "Multicast with network coding in application-layer overlay networks," *IEEE JSAC*, vol. 22, pp. 107–120, 2004.

[7] C. Gkantsidis and P. Rodriguez, "Cooperative security for network coding file distribution," Microsoft Research, Tech. Rep., 2004.

[8] T. Ho, B. Leong, R. Koetter, M. Médard, M. Effros, and D. R. Karger, "Byzantine modification detection in multicast networks using randomized network coding," in *IEEE Intl. Symp. Inf. Theory*, 2004.

[9] C. Gkantsidis, J. Miller, and P. Rodriguez, "Anatomy of a P2P content distribution system with network coding," in *Intl. Workshop on Peer-to-Peer Systems*, Santa Barbara, CA, February 2006.

[10] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung, "Network information flow," *IEEE Trans. Inf. Theory*, vol. 46(4), pp. 1204–1216, 2000.

[11] S. R. Li, R. W. Yeung, and N. Cai, "Linear network coding," *IEEE Trans. Inf. Theory*, vol. 49, no. 2, pp. 371–381, 2003.

[12] R. Koetter and M. Médard, "Beyond routing: An algebraic approach to network coding," in *IEEE INFOCOM*, 2002, pp. 122–130.

[13] R. Koetter and M. Médard, "An algebraic approach to network coding," *IEEE/ACM Trans. Networking*, vol. 11, no. 5, pp. 782–795, 2003.

[14] S. Jaggi, P. Sanders, P. A. Chou, M. Effros, S. Egner, K. Jain, and L. M. Tolhuizen, "Polynomial time algorithms for multicast network code construction," *IEEE Trans. Inf. Theory*, vol. 51, no. 6, pp. 1973–1982, June 2005.

[15] T. Ho, R. Koetter, M. Médard, D. R. Karger, and M. Effros, "The benefits of coding over routing in a randomized setting," in *IEEE ISIT*, 2003.

[16] M. Bellare, O. Goldreich, and S. Goldwasser, "Incremental cryptography: The case of hashing and signing," in *CRYPTO*, 1994.

[17] B. Cohen, "Bram Cohen's comments on Microsoft's Avalanche," http://www.livejournal.com/users/bramcohen/20140.html.

[18] J. Edmonds, "Minimum partition of a matroid into independent sets," *J. Res. Nat. Bur. Standards Sect.*, vol. 869, pp. 67–72, 1965.

[19] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *CRYPTO*, ser. LNCS, vol. 1592, 1999, pp. 223–238.

[20] M. Bellare and P. Rogaway, "Random oracles are practical: a paradigm for designing efficient protocols," in *ACM CCS*, 1993, pp. 62–73.