Cooperative Security for Network Coding File Distribution

Christos Gkantsidis and Pablo Rodriguez Rodriguez
Microsoft Research
Cambridge, CB3 0FD, UK
Email: chrisgk@microsoft.com, pablo@microsoft.com

Abstract— **Peer-to-peer content distribution networks can suffer from malicious participants that intentionally corrupt content. Traditional systems verify blocks with traditional cryptographic signatures and hashes. However, these techniques do not apply well to more elegant schemes that use network coding techniques for efficient content distribution.**

Architectures that use network coding are prone to jamming attacks where the introduction of a few corrupted blocks can quickly result in a large number of bad blocks propagating through the system. Identifying such bogus blocks is difficult and requires the use of homomorphic hashing functions, which are computationally expensive.

This paper presents a practical security scheme for network coding that reduces the cost of verifying blocks on-the-fly while efficiently preventing the propagation of malicious blocks. In our scheme, users not only cooperate to distribute the content, but (well-behaved) users also cooperate to protect themselves against malicious users by informing affected nodes when a malicious block is found. We analyze and study such cooperative security scheme and introduce elegant techniques to prevent DoS attacks. We show that the loss in the efficiency caused by the attackers is limited to the effort the attackers put to corrupt the communication, which is a natural lower bound in the damage of the system. We also show experimentally that checking as low as 1-5% of the received blocks is enough to guarantee low corruption rates.

I. Introduction

Peer-to-Peer (P2P) networks have recently emerged as alternative to traditional Content Distribution solutions (e.g. Akamai [4]) to deliver large files. Such P2P networks create a fully distributed architecture where commodity PCs are used to form a cooperative network and share their resources (storage, CPU, bandwidth). By capitalizing the bandwidth of end-systems, P2P cooperative architectures offer great potential for providing a cost-effective distribution of software updates, critical patches, videos, and other large files to thousands of simultaneous users both Internet-wide and in private networks.

Despite their enormous potential and popularity, existing end-system cooperative schemes, that use a mesh-like architecture (e.g. [16]), suffer from a number of inefficiencies which decrease their overall performance [28], [33], [34]. Such inefficiencies arise from the fact that there is no central scheduler that decides how content should propagate through the overlay mesh, and nodes perform decisions in a large distributed setting with local information only. These inefficiencies are more pronounced for large and heterogeneous populations, when the publisher has limited resources, or when cooperative incentive mechanisms are in place.

Recent developments in *network coding*, have provided elegant algorithms for the efficient propagation of information in a large scale distributed system with no central scheduler. Network coding was first considered in the pioneering work by Alswede et al. [8], where they showed that a sender can communicate information to a set of receivers at the broadcast capacity of the network provided one allows network coding. The principle behind network coding is to allow intermediate nodes to encode packets.[1] A growing body of literature has recently considered network coding in the context of file distribution [2], [34], [35] and proposed practical network coding systems [5], [6], [34].

There is, however, a significant downside to using network coding for file distribution since any untrusted node is allowed to produce new encoded packets. A malicious node can generate corrupted packets and then distribute them to other nodes, which in turn use them to (unintentionally) create new encoded packets that are also corrupted. Observe that commonly used methods for protecting the integrity of each packet by using digital signatures do not work with network coding, since each peer produces unique encoded packets which cannot be signed by the server.

A receiver may discover after downloading the full file that it was receiving corrupted blocks from misbehaving or malicious nodes and cannot decode the file. We call attacks that alter and corrupt the content of the encoded blocks *jamming attacks*. As we shall see in this paper, jamming attacks can result in huge reduction in the performance of the distribution network, wasting bandwidth in distributing corrupted content, and increasing the time to download the entire file. Hence, there is a strong incentive to check blocks on the fly to prevent nodes from downloading corrupted blocks.

To prevent a jamming attack in a network coding system, we need a hashing scheme such that the hash of an encoded packet can be easily derived from the hashes of the packets contributing to the encoding. One such class of functions are *collision-resistant homomorphic hashing* functions.

In [21], homomorphic hashing functions were first introduced to allow nodes to check blocks on-the-fly in a system where content is encoded at the source using rateless codes

[1] In the rest of the paper we will use packets and blocks inter-exchangeable.

[]. However, such homomorphic hashes are *computationally expensive* and often require that nodes check blocks probabilistically to reduce the cryptographic overhead, opening the door for malicious blocks to infect a larger portion of the network.

We propose a novel *cooperative security* scheme where users not only cooperate to distribute content, but (well-behaved) users also cooperate to protect themselves against malicious users by alerting affected nodes when a malicious block is found. Even though each node checks for bad blocks infrequently, at any point in time there are many nodes in the system that perform such checks, and when they find corrupted blocks they alert the rest of the nodes. Hence, our system greatly reduces the computation overhead at each node and, at the same time, quickly eliminates corrupted blocks.

We study and analyze such cooperative security scheme, and show that the loss in the efficiency caused by the attackers is limited to the effort the attackers put to corrupt the communication, which is a natural lower bound in the damage of the system. Moreover, we show that nodes need check only $1-5\%$ of the packets and still be able to quickly find most of the malicious blocks. To ensure that nodes are not overloaded by bogus alert messages, we present a novel and light-weight scheme that prevents bogus alerts from traveling through the system. Alert messages are quickly verified against the data stored at each node using *random masks* and *mask-based* hashes and bogus alerts are quickly discarded thus preventing nodes from performing unnecessary expensive homomorphic hashing checks.

In this paper, we also address other possible attacks that are possible due to the use of network coding. More specifically, we discuss a particular attack that attempts to reduce the diversity of the encoded blocks in the system, and, hence, results in increased download times since the users do not find innovative content to download. We call such attacks *entropy attacks*. We present a practical mechanism that prevents such attacks by requiring the sender to transmit a short description of how the encoding is produced prior to the block download. Given this description, the receiver can check whether the encoding is innovative, and, hence, useful prior to downloading it.

The rest of the paper is organized as follows. In Section II, we give an overview of network coding. In Section III, we describe the threat model, and give a short overview of homomorphic hash functions, which can be used to verify network encoded blocks. In Section V, we present strawman approaches for improving computing time and discuss their limitations. In Section VI we present our cooperative security scheme to share the cryptographic work involved in checking for malicious blocks. We also present a mechanism to prevent DoS attacks from bogus alerts during the cooperation process. In Section VIII, we give analytical and experimental results related to the benefits of cooperate protection. We provide related work in Section IX and summarize in Section X.

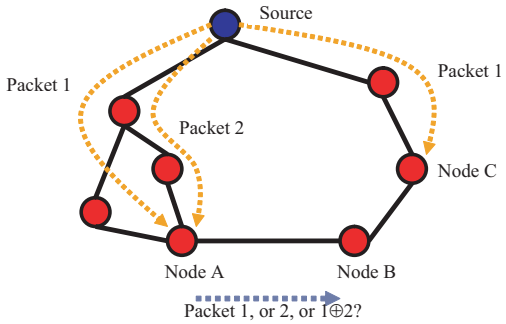

Fig. 1. Network Coding benefits even when nodes only have local information.

II. Brief Overview of Network Coding for Content Distribution

Network coding is a novel mechanism proposed in the last years to improve the throughput utilization of a given network topology [8]. The principle behind network coding is to allow intermediate nodes to re-encode packets. Compared to other traditional approaches, network coding makes optimal use of the available network resources and computing a scheduling scheme that achieves such rate is computationally easy. An overview of network coding and a discussion of possible Internet applications is given in [3].

With network coding, every time a client needs to send a packet to another client, the source client generates and sends a linear combination of all (or part) of the information available to it (similarly to XORing multiple packets). After clients receive enough linearly independent combinations of packets, they can reconstruct the original information. To illustrate how network coding improves the propagation of information without a global coordinated scheduler we consider the following (simple) example. In Figure 1 assume that Node A has received from the source packets 1 and 2. If network coding is not used, then, Node B can download either packet 1 or packet 2 from A with the same probability. At the same time that Node B downloads a packet from A, Node C independently downloads packet 1. If Node B decides to retrieve packet 1 from A, then both Nodes B and C will have the same packet 1 and, the link between them can not be used.

If network coding is used, Node B will download a linear combination of packets 1 and 2 from A, which in turn can be used with Node C. Obviously, Node B could have downloaded packet 2 from A and then use efficiently the link with C, however, without any knowledge of the transfers in the rest of the network (which is difficult to achieve in a large, complex, and distributed environment), Node B cannot determine which is the right packet to download. On the other hand, such a task becomes trivial using network coding. It is important to note that the decision on which packets to generate and send at given node does not require for nodes to keep information about what the other nodes in the network are doing, or how the information should propagate in the network, thus, greatly simplifying the content distribution effort.

Network coding can be seen as a generalization of Erasure

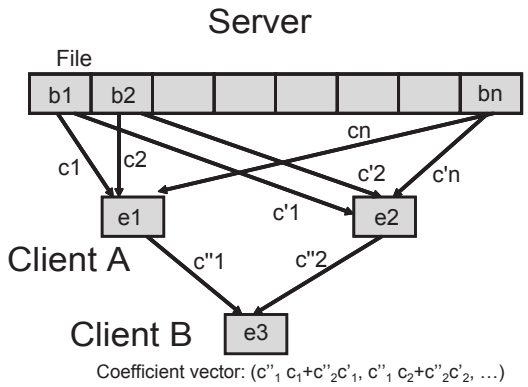

Fig. 2. Sample description of our network coding system.

Codes [23] [24] (e.g. Digital Fountain) since both the server and the end-system nodes perform information encoding. Note, however, that restricting erasure codes only to the origin server implies that intermediate nodes can only copy and forward packets. This results in *the same* erasure codes being blindly copied over from one node to another without knowing whether they will be of use to other nodes downstream.

A. Content Propagation with Network Coding

Assume a file F that originally exists at the server. File F is divided into n blocks $(b_1, b_2, \ldots b_n)$. Then each block b_i is subdivided into m codewords $b_{k,i}$, $k \in 1, .., m$. The file F is considered as an $m \times n$ matrix of elements of $\mathbb{Z}_q$, where m is a predetermined number of codewords.

$$F = (b_1, b_2, \ldots b_n) = \begin{pmatrix} b_{1,1} & b_{1,2} & \ldots & b_{1,n} \\ b_{2,1} & b_{2,2} & \ldots & b_{2,n} \\ \vdots & \ddots & & \vdots \\ b_{m,1} & b_{m,2} & \ldots & b_{m,n} \end{pmatrix}$$

With network coding, both the server and the users perform encoding operations. Whenever a node or the server needs to forward a block to another node, it produces a linear combination of all the blocks it currently stores. The operation of the system is best described in Figure 2.

Assume that initially all users are empty and that user A contacts the server to get a block. The server will combine all the blocks of the file to create an encoded block $e1$ as follows. The server will pick some random coefficients $c_1, c_2, \ldots, c_n$, then multiply each codeword of block b_i with c_i, and add the results of the multiplications together.

For the purpose of this paper, we will be using scalars, vectors, and matrices defined over modular subgroups of $\mathbb{Z}_q$, however, network coding operations can be performed in any finite field (e.g. Galois Fields are also possible [34]). Hence, addition of blocks is defined as component-wise addition of the corresponding block codewords. That is, to combine the blocks of the file with *coefficient vector* $\vec{c} = (c_i)$, $\sum_{i=1}^{i=n} c_i b_i$ the server computes:[2]

$$((c_1 b_{1,1} + \ldots + c_n b_{1,n}) \mathrm{mod} q\,, \ \ldots, (c_1 b_{m,1} + \ldots + c_n b_{m,n}) \mathrm{mod} q)$$

[2]Observe that this is a correction on the original paper that appeared on Infocom 2006, which stated that

$$\left(\begin{array}{c} (c_1 b_{1,1} + \ldots + c_n b_{1,n}) \mathrm{mod} q \\ \ldots \\ (c_1 b_{m,1} + \ldots + c_n b_{m,n}) \mathrm{mod} q \end{array} \right)$$

Many thanks to Man Liang for pointing out the mistake.

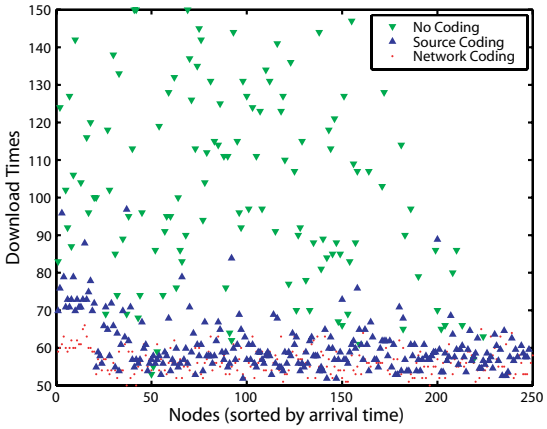

Fig. 3. Download times for each node (rounds) depending on the coding scheme used (no coding, source coding, network coding). Nodes arrive dynamically (20 nodes every 20 rounds). File is composed of 50 blocks.

The server will then transmit to user A the result of the linear combination and the coefficient vector $\vec{c}$. Assume now that user A has received another block of encoded information $e2$, either directly from the server or from another peer, with its associated vector of coefficients. If user A needs to transmit an encoded block $e3$ to user B, A generates a linear combination of its two blocks $e1$ and $e2$ as follows. User A picks two random coefficients c_1'' and c_2'', multiplies each element of block e_1 with the coefficient c_1'' and similarly for the second block e_2, and adds the results of the multiplication. The block transmitted to user B will be the addition of the multiplications $c_1'' \cdot e_1$ and $c_2'' \cdot e_2$. Note that the coefficient vector $\vec{c''}$ associated with the new block is equal to $c_1'' \cdot \vec{c} + c_2'' \cdot \vec{c'}$.

Observe that a node can recover the original file after receiving n blocks for which the associated coefficient vectors are *linearly independent* to each other. The reconstruction process is similar to solving a system of linear equations.

B. Network Coding Benefits

The benefit we expect to get by using network coding is due to the randomization introduced each time we generate a new encoded block. If at least one of the combined blocks is of use to other nodes down the path, then the linear combination will also be useful. Network coding minimizes the response time in the absence of a centralized scheduler that decides which node will forward which part of the content.

The benefits of network coding vs source coding and no coding are summarized in Figure 3. In Figure 3 we show the download times for each node in a well-connected mesh network of 250 nodes where nodes arrive randomly. The file size is 50 blocks (or equivalent rounds).

From this Figure we can see that network coding provides near-optimal response times since most nodes are able to

download the file soon after the full file is served once (note that there is a small number of rounds required for blocks to propagate into the network). However, with source coding and no coding at all, the time that it takes to download the file is on average much larger. Moreover, the performance experienced by nodes not using network coding is much more variable, resulting in some nodes having to wait for very long times to receive the full file. For instance, certain nodes using source coding have to wait 1.4 times longer than the worse behaving nodes with network coding. Given the clear benefits of network coding, in the rest of the paper we will study how to ensure high efficiency even in the presence of various types of security attacks.

III. Threat Model

Traditional P2P cooperative architectures can suffer from a number of attacks including disrupting the topology, blocking access to the tracker node, or steering nodes toward a certain set of malicious neighbors. In this paper we do not consider such types of attacks, which often require a more distributed peer-discovery protocol [47], [48]. Instead, we focus on those attacks that arise from the use of Network Coding for content distribution. Next we describe the threat model scenario.

We assume that an original source file F exists on a single server and a large population of users are interested in retrieving F. The users trust the source server but users form a large set of untrustworthy nodes. A subset of these users want to disrupt the distribution of the file by propagating corrupted blocks, so that legitimate users cannot decode the original file, and/or the rate of content dissemination is reduced [3].

When a server wishes to publish a file F, he encodes the file and distributes such encodings to multiple users simultaneously. Users download blocks from the server or other users and distribute new encoded blocks that are produced as linear combinations of all of the encoded blocks they hold. Thus, new encoded blocks are produced at each node even if the download has not been completed.

A node A only knows the identities and can upload(download) blocks from(to) a small number of other nodes. We call this subset of users the neighbors of A. The neighboring relationship is symmetric. Malicious users are a fraction of the total user population and can fully coordinate their activities. We do not address the issue of how to limit the number of malicious users in this paper; a possible approach could be to use a human verification test before a user joins the network.

We aim at providing a best effort security system. This means that we are not trying to identify and remove malicious nodes. Instead, we focus on making sure that the system provides high throughput even in the presence of such attackers. We do not require that peers build trust relationships. However, we do assume that legitimate users will disconnect from peers that frequently supply them with corrupted content. Hence, we assume that malicious users will try a mixed strategy in which they serve a mixture of valid and corrupted blocks.

Under these assumptions, the P2P cooperative model is vulnerable to a) **Entropy attacks** where malicious clients try to disrupt the diversity of the system and the tit-for-tat exchange balance and b) **Jamming attacks** where malicious clients try to inject bogus blocks in the content distribution process.

A. Entropy Attacks

With network coding a node does not need to worry about how to pick the block to transmit to another node; it combines all its available blocks. However, such encoded blocks are only useful to a given node if they carry new information, i.e. they are innovative. Determining innovation needs to take into account the coefficient vector used to generate the block; the encoded block downloaded can be different to each of the locally available encoded blocks, but, still, if its coefficient vector can be written as a linear combination of the vectors of the locally available blocks, then it is not innovative.

Using encoded blocks, would be very easy for an attacker to send non-innovative blocks that are trivial linear combinations of already existing blocks at the recipient. We call this an *entropy attack* since malicious users try to decrease the entropy or diversity in the system, reducing the opportunities that nodes have for making download progress and thus the rate of the system. Another side effect of such attacks is that a malicious user can easily become a free-rider even when incentive mechanisms like tit-for-tat are used by basically sending non-informative data and getting useful data in return.

To solve this problem we ensure that each node, prior to downloading a block, first downloads the coefficient vectors of all the blocks in the neighborhood. By using the neighbors' coefficient vector and its own coefficient vectors, a given node then calculates the *rank* of the combined matrices and determines which neighbors can provide innovative blocks and moreover how many blocks they can provide.

Rather than checking all the vectors from a potential sender, an alternative and often cheaper approach is to have the sender generate a random linear combination of all its coefficient vectors. The receiver checks whether the generated coefficient vector can be expressed as a linear combination of its coefficient vectors. If it cannot be written as a linear combination, then the sender has at least one innovative block that the receiver can download. If it can, then either the sender does not have any innovative blocks, or the random linear combination produced by the server fails to prove that the sender has indeed innovative information. This latter case is very rare and we can ignore such events.

Observe that the transfer of the coefficient vectors generates little overhead since the size of each vector can fit in a couple of packets, whereas the size each block is in the order of several hundreds of KBytes [18].

B. Jamming Attacks

In large scale P2P distributed systems, there exists the possibility that some nodes are malicious and inject bogus packets

[3]We do not explicitly consider attacks against the underlying physical routers or network links

in the network to *jam* the download. Jamming attacks happen when a malicious node sends a pair of an encoded block and a coefficient vector where either one does not carry valid information. The receiver will obtain corrupted information, and, if the receiver uses this information to create encoded blocks, it will inject (involuntarily) more corrupted blocks in the network.

Since receivers have limited bandwidth, they would clearly benefit from a mechanism that detects cheating as it happens, so they can terminate connections to bad neighbors and seek out honest nodes elsewhere in the network. Another alternative is to wait for the receiver to finish the download and try to decode the full file. However, if the file is infected with bad blocks, it is very difficult to identify such bad blocks at decoding time. Downloading extra blocks and performing multiple decoding operations with different combinations of blocks in an attempt to reconstruct a valid file has prohibitive cost. This is clearly unacceptable, especially for large downloads where the cost of multiple decodings becomes prohibitive. One bad block should not ruin hundreds or thousands of valid ones.

To protect clients against jamming attacks, P2P cooperative systems require some form of source verification. That is, downloaders need a way to verify individual check blocks. Furthermore, this verification should work whether or not the original publisher is online. In standard P2P systems, this is achieved by hashing each block and distributing the block hashes from a central trusted publisher. By comparing the hash of each downloaded block to the corresponding hash given by the publisher, a node can quickly check whether a block is valid or not.

When blocks are encoded only at the server, or at a limited set of servers, then the standard way to prevent an attack by a malicious user injecting bad data into the system is to require that valid senders sign all their encoded packets cryptographically [21].

However, using network coding, jamming attacks are particularly serious because undetected malicious blocks will be used to generate more malicious blocks, and quickly every block transmitted in the network is corrupted. Observe that each encoded block is unique and cannot be signed by a trusted authority, like for example the server. Thus, to prevent a jamming attack in an open system that uses network coding, one would need a hashing scheme such that the hash of an encoded packet can be easily derived from the hashes of the original packets and from the coefficient vector that describes the encoding. Homomorphic hashes have this property and are described in the following section.

IV. Homomorphic Hashes for Network Coding

To prevent bogus packets from jamming the system, we require special hashing functions that survive the construction of linear combinations of original blocks at intermediate nodes. We next describe such functions and show how they can be used to prevent jamming attacks in the presence of network coding.

In [21], the authors demonstrate how to use *homomorphic hash functions* to validate encoded blocks that were produces using rateless codes. In [21] coding takes place only at the source, or at receivers that have finished downloading. An encoding in their case is performed by adding sub-block wise a small subset of the blocks in $\mathbb{Z}_q$. In the case of network coding, we need to verify random linear combination of blocks, which can be thought as weighted additions of all or of a subset of the blocks. We next show how to extend the basic homomorphic hash function to deal with network coding. More details on the use of homomorphic hash functions can be found in [21].

A hash function, say $h(\cdot)$, maps a large input, which is a block of information, say b, to an output $h(b)$ typically of much smaller size. Function $h(\cdot)$ has the important property that given b it is difficult to find another input block b' with the same hash value $h(b') = h(b)$. Homomorphic hash functions have the additional property that the hash value of a linear combination of some input blocks can be constructed efficiently by a combination of the hashes of the input blocks. More specifically, if the original blocks are b_i, with $\forall i \in [1, n]$, then the hash value of the linear combination $b' = c_1 b_1 + c_2 b_2 + \ldots + c_n b_n$ is $h(b') = h^{c_1}(b_1) \cdot h^{c_2}(b_2) \cdot \ldots \cdot h^{c_n}(b_n)$. We prove that property in Lemma 4.1 below.

Recall from Section II-A that each block b_i is divided into m codewords $b_{k,i}$, $k = 1 \ldots m$. Before computing the hash of a block we need to decide on the hash parameters $G = (r, q, g)$. The parameters r and q are prime numbers of order λ_r and λ_q chosen such that $q|(r-1)$.

The parameter g is a vector of m numbers such that each of the elements of the vector can be written as $x^{(r-1)/q}$, where $X \in \mathbb{Z}_q$ and $x \neq 1$. The number of codewords m is such that each element is less than $2^{\lambda_q - 1}$. More details about the sizes of the prime numbers r and q and algorithms for the efficient construction of G can be found in [21] and in Table I.

TABLE I
HOMOMORPHIC HASHING FUNCTION PARAMETERS

Name	Description	e.g.
λ_r	discrete log security parameter	1024 bit
λ_q	discrete log security parameter	257 bit
r	random prime $\lvert r\rvert = \lambda_r$	
q	random prime $q\lvert(r-1)$, $\lvert q\rvert = \lambda_q$	
m	number of "sub-blocks" per block	512
g	$1 \times m$ row vector of order q in $\mathbb{Z}_q$	
β	block size	16 KB

We define a hash for each block $b_i = [b_{1,i} b_{2,i} \ldots b_{m,i}]$ as

$$h(b_i) = \prod_{k=1}^{m} g_k^{b_{k,i}} \bmod p$$

The hash of the file F is simply the vector of the hash of each b_i:

$$H(F) = (h(b_1), h(b_2), \ldots, h(b_n))$$

Whenever a node first joins the system, it downloads from the server the hash of the file $H(F)$ as well as the security parameters G. This hash will be used to check *encoded blocks*

on-the-fly.[4]

We now show that the hash of an encoded block can be constructed by the hashes of the original blocks. Hence, it is possible to check whether a received encoded block and coefficient vector are indeed correct.

Lemma 4.1 (Homomorphic hashing for network coding): The hash value of the encoded block $e = \sum_{i=1}^{n} c_i b_i$ can be computed by the hashes of the original blocks:

$$h(e) \equiv \prod_{i=1}^{n} h^{c_i}(b_i) \bmod p$$

Proof: Assume an encoded block $e = \sum_{i=1}^{n} c_i b_i$; all arithmetic operations are in the $\mathbb{Z}_q$ field. The hash of this block is:

$$\begin{aligned} h(e) &= h(\sum_{i=1}^{n} c_i b_i) \\ &= \prod_{k=1}^{m} g_k^{(\sum_{i=1}^{n} c_i b_{k,i}) \bmod q} \bmod r \end{aligned}$$

Observe that the sum in the exponent can be written as

$$\sum_{i=1}^{n} c_i b_{k,i} = q \cdot \text{quot} + (\sum_{i=1}^{n} c_i b_{k,i}) \bmod q$$

where quot is the quotient of dividing $\sum_{i=1}^{n} c_i b_{k_i}$ by q.

and, hence, $g_k^{\sum_{i=1}^{n} c_i b_{k,i}} \bmod r$ can be written as

$$(g_k^{q \cdot \text{quot}} \bmod r) \cdot (g_k^{(\sum_{i=1}^{n} c_i b_{k,i}) \bmod q} \bmod r) \bmod r$$

Recall that $g_k = x_k^{(r-1)/q}$, and from Fermat's Little Theorem,

$$g_k^{q \cdot \text{quot}} \bmod r = (x_k^{(r-1)} \bmod r)^{\text{quot}} \bmod r = 1$$

Thus, $h(e)$ can be expressed as

$$\begin{aligned} \prod_{k=1}^{m} g_k^{\sum_{i=1}^{n} c_i b_{k,i}} \bmod r &= \prod_{k=1}^{m} \prod_{i=1}^{n} (g_k^{b_{k,i}})^{c_i} \bmod r \\ = \prod_{i=1}^{n} (\prod_{k=1}^{m} g_k^{b_{k,i}})^{c_i} \bmod r &= \prod_{i=1}^{n} h(b_i)^{c_i} \bmod r \end{aligned}$$

■

V. Strawman Approaches

In the previous section we have shown how homomorphic hashing functions can be used with network coding to verify blocks as they are received at each node. However, such homomorphic hashing functions are computationally expensive and users cannot check every block. For instance checking rates for homomorphic hashing functions on a Pentium IV at 3GHz is approximately 300 Kbps, compared to 560 Mbps for SHA1 [21]. Our own implementation of homomorphic hashes produces rates of 160 Kbps on a slower 2GHz computer. These rates are one order of magnitude lower than the rate at which encoded packets can be produced, therefore, limiting the overall throughput of the over system. For instance, our implementation of network coding using modular arithmetic produces encoding rates greater than 8 Mbps for a 1 GByte file, which are sufficient relative to typical residential network throughput; many P2P nodes are restricted by their access capacity which is typically less than 2-3 Mbps.

Our goal is to improve the speed of checking, so that a downloader can, at least, verify blocks as new blocks can be produced. In this section, we consider some strawman approaches to improve the computation time and show how existing techniques that are applied with source-coding, do not work well with network coding.

A. Batching with Network Coding

One possible solution to improve computation time is to verify blocks in batches, either probabilistically or periodically. In such scheme nodes do not check every block, but they check a window of blocks all at once. This solution was first proposed in [32] and then used in [21] to improve the verification performance of homomorphic hashes for erasure codes. We next describe how batching can be done with network coding.

Batching is possible thanks to the homomorphic property of the hashing functions. Let's assume we have a window of L *encoded blocks* to verify all together. We can build a *batched* encoded block e_r as a linear combination of all L encoded blocks and check the resulting combination. Let $L = (\langle e_1, c_1 \rangle, \ldots \langle e_L, c_L \rangle)$, and let the resulting batched block $e_r = \sum_{i=1}^{L} e_i$, with combined coefficient vector $c_r = \sum_{i=1}^{L} c_i$. The advantages of using batching is that nodes only need to check one e_r block rather than L blocks.

However, this batching scheme is exposed to a specific byzantine attack, which we call the *pairwise byzantine attack* [32]. Such an attack consists of sending two blocks that make the batch verification scheme fail. For instance, assume $\langle e_1, c_1 \rangle$ and $\langle e_2, c_2 \rangle$ to be two correct messages hold by an attacker. The attacker now creates the two following corrupted blocks, $e_1' = e_1 + \epsilon$, and $e_2' = e_2 - \epsilon$. When this two corrupted vectors are checked together in batch, it is easy to see that the verifier will fail to capture the corrupted packets since the value of e_r will remain the same.

Our solution to this problem is to create a batched encoded vector e_r using a *random set of coefficients* $\vec{w}$: $e_r = \sum_{i=1}^{L} w_i e_i$, where $c_r = \sum_{i=1}^{L} w_i \times c_i$. Doing this, an attacker can only succeed with such pairwise byzantine attack if it is able to create two blocks that after being multiplied by random coefficients w_i and w_j will cancel each other, which is very unlikely.

Although batching can decrease the computation time almost linearly with the size of the batch, batching block verification has the risk of letting some malicious packets propagate since packets are exchanged without being checked. This opens the receiver to small-scale attacks in the case of

[4] Hashes are elements in $\mathbb{Z}_p$ (typically 128 bytes long), which is $\frac{1}{128}$ times the size of a block of size 16 KB. Thus, the hash of the file $H(F)$ will be typically 1% the total file size. To avoid smaller scale attacks during the hash download, nodes can use Merkle hash trees [45].

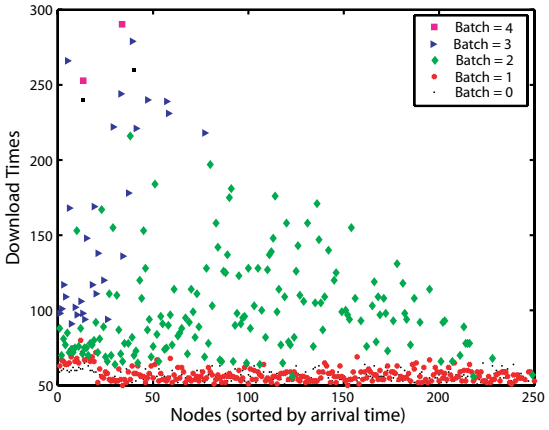

Fig. 4. Download times (rounds) for each node using network coding depending on the batching window size. Blocks in the batching windows are not used for network coding. Nodes arrive dynamically (20 nodes every 20 rounds). File is composed of 50 blocks.

source coding (a receiver accepts a batch worth of check blocks before closing a connection with a malicious sender). However, with network coding, batching can cause more serious effects since malicious packets are quickly re-combined with other valid packets at each node and corrupt a large portion of the download. For instance, if the batch size is equal to 256 blocks (as suggested in [32]), more than 97% of the blocks will be corrupted by an attacker that injects only 5% of malicious the packets. Thus, standard batching techniques do not work well with network coding. For more detailed analysis see Section VIII.

B. Isolating Unverified Blocks

To prevent unverified blocks in a batch from corrupting other blocks, an alternative solution is to isolate unverified blocks in the batch window, preventing them from being re-combined with other blocks. Only once blocks in the batch window have been verified then they are involved in the network coding process.

The benefit of this solution is that *all* packets are checked before they are forwarded to other nodes, thus, drastically limiting the scope of a possible infection. However, the problem with this approach is that new information is delayed at each node before it can be propagated to the rest of the network. Such delay can seriously impact the efficiency of the system since nodes may have to wait much longer to download missing blocks. In Figure 4 we highlight such effect.

Figure 4 shows the download times per node for a file of 50 blocks (each block is one time unit) in a network where nodes arrive dynamically at a rate of 20 nodes each 20 rounds. We show the impact of increasing batching windows sizes. When there is no batching, network coding provides optimal download times (i.e. 50 rounds). However, as the batching window size increases, blocks are delayed and the efficiency of network coding sharply decreases. In fact, if the batching window is greater than two packets, then, the efficiency of network coding decreases by a factor of three to four.[5] Thus, to ensure that the network coding efficiency remains high, blocks in the batching window should be made part of the network coding process as soon as they are received.

VI. Cooperative Security

To reduce the cryptographic work at each node while still preventing malicious packets from infecting large portions of the network, we propose a *cooperative* security scheme where nodes cooperate in checking for malicious blocks. Users not only cooperate to distribute the content, but (well-behaved) users also cooperate to protect themselves against malicious users by informing affected nodes when a malicious block is found. By having a large number of nodes checking at every point in time and making them cooperate, expensive homomorphic hashing can be applied less frequently without significantly weakening the resistance of the scheme to resource adversarial behavior. Next we describe the details of how such cooperative security system works.

We assume that nodes check blocks with probability p. Blocks that pass the check are marked as *safe*, while blocks that have not yet been checked are kept in an *insecure* window. Blocks are checked in batches. The batch window is equal to the insecure window. Whenever a node verifies its insecure window, valid blocks are marked as safe and the insecure window is reset (Refer to Section V-A for more detail about batch verification).

Nodes do not rely on other nodes to mark blocks as safe. However, they actively cooperate with other nodes to detect malicious blocks. Whenever a node detects a malicious block, it sends an *alert* message to all its neighbors. To prevent nodes that have not been infected from processing the alert message, a given node keeps an *insecure-activity* table with the ID of a) those nodes that downloaded blocks encoded with insecure window blocks, and b) those nodes that delivered the blocks inside the insecure window. Note that the state of this table is reset when the insecure window is checked. If an alert message is received from a node that is not in this table, the message is discarded.

Alert messages are propagated from one node to another until all infected nodes are informed. If the insecure window is empty, alert messages are not processed. Alert messages are processed as soon as they are received. However, alert messages are only propagated after the node is convinced that a malicious block exists (see Section VII-A for an efficient alert verification technique). Duplicated alert messages can be received for the same malicious packet since mesh overlays often contain loops. However, such duplicated messages will be discarded when a) the insecure window is empty, or b) the duplicate message comes from a node that is not in the insecure-activity table.

In addition to alerting its neighbors, a node takes the following actions: 1) it puts blocks in the insecure window *in*

[5]Note that some nodes in the Figure do not have a corresponding download time. The reason for this is because their download times are higher than 300 rounds, which is the maximum time considered in the experiment.

quarantine to be checked and cleaned in the background , 2) it stops using blocks in the insecure window for network coding, and 3) it starts checking blocks with probability one until the insecure window is secured and cleaned, thus, preventing new malicious blocks from infecting the system.

One drawback that arises during the quarantine period is that valid blocks in the insecure window are not part of the re-encoding process. Note, however, that network coding ensures a high level of representation of blocks in the network under such small temporal glitches, thus, maintaining a high level of efficiency in the content distribution system. To ensure that the insecure window is quickly cleaned and valid blocks are back into the system as soon as possible, nodes use a fast and effective search mechanism that rapidly discards malicious blocks.

One simple approach to clean the insecure window, is to check the hashes of each block individually. However, this may result in unnecessary checks since it is quite likely that most blocks will not be corrupted. Another more efficient approach is to use *binary batching trees*. Such batching trees work as follows. A node first verifies all blocks in the insecure window using batching. If this test does not find any malicious block, then the process is stopped and all packets are marked as safe. If the batch verification fails, then the insecure window is divided in two halves, which are then checked independently using batching. If one half of the insecure window has not been corrupted, then, all its blocks are marked as safe and they are not checked any more. Corrupted parts are again subdivided in two parts until the individual corrupted blocks are identified and discarded.

VII. Preventing Bogus Alert Attack

One potential risk of the cooperative security mechanism is that nodes may be exposed to a DoS attack where a malicious node sends bogus alert messages, i.e. alert messages that are not triggered from the discovery of a malicious block. Such behavior could force well-behaving nodes to check every block, defeating the purpose of the cooperative security scheme. Next we discuss a number of techniques that minimize the impact of bogus alerts.

As discussed in the previous section, alert messages from nodes that are not in the insecure-activity table will be discarded. For a sending node to appear in other node's insecurity-activity table, the sending node needs to have uploaded at least one block worth of data to the receiving one. This prevents a malicious node from sending bogus alerts at an arbitrary rate over the distribution network since malicious nodes need to upload blocks to well-behaving nodes before their alert messages are taken into account. Furthermore, even if bogus alert messages are accepted by the immediate neighbors around the attacker, bogus alerts will be slowed down by the next tier of non-malicious peers receiving such an alert. The reason for this is that non-malicious peers also need to upload content to their neighbors before their alerts are considered. Since non-malicious peers have a limited upload capability, this will essentially limit the rate of false alarm propagation.

Despite all this protection against bogus alerts that is already embedded in the system's design, adversaries may still be able to gather large amount of bandwidth (e.g. through peer zombies) and target well-connected nodes, which could result in a more effective bogus alert propagation. To effectively limit the propagation speed of bogus alerts, we now introduce the concept of *verifiable alerts*.

A. Verifying Alerts

We now present a scheme for quickly and cheaply verifying whether an alert message is correct. This scheme effectively blocks bogus alerts from getting into the content distribution network while ensuring that valid alerts are not slowed down. One way to verify alerts is by appending in the alert message some kind of proof for quick verification, thus, creating a self-verifiable alert. However, generating such self-verifiable alerts can be time consuming and expensive since a node needs to accurately determine where the corruption is happening to be able to provide a verifiable proof. This can significantly slow down the rate of propagation of valid alerts, thus, reducing the effectiveness of the cooperative security mechanism.

Rather than waiting for a node to generate a self-verifying proof, we propose to use instead a light-weighted scheme where each node is capable of quickly and independently testing for the validity of an alert. To this extend, receivers use a set of *random masks* and *mask-based hashes*. We define a random mask as a vector $\vec{t} = \{t_1, ..., t_m\}$, where t_k is random element in $\mathbb{Z}_q$ (recall that each block b_i is sub-divided into m sub-blocks, $\{b_{1,i}, ..., b_{m,i}\}$). We then define a mask-based hash for each block b_i as $f(b_i) = \sum_{k=1}^{m} t_k b_{k,i}$ and the corresponding mask-based vector as $\vec{f} = \{f(b_1), ..., f(b_n)\}$, where n is the total number of original blocks and $f(b_i)$ is in the same field as $b_{k,i}$. Please, note that each mask-based hash $f(b_i)$ per block is produced with the same random mask vector $\vec{t}$. Each mask-based hash is of the same size as each sub-block. Note that one can create a very large number of mask-based vectors for a given set of original blocks by selecting a different random mask $\vec{t}$.

Using such masks and hashes, a given node can quickly verify the validity of an alert on the fly. Next we detail such process. Before the download commences, each node downloads (using a secure channel) the random mask and corresponding hashes ($\vec{t}$,$\vec{f}$).[6] Each node will get a different set of randomly generated masks and hashes, which are kept secret from the other nodes. Whenever a node receives an alert, it checks whether the alert is valid or not by testing whether the blocks in its insecure window are corrupted or not using the random masks and hashes. To this extend, a node generates a combination block e of all the encoded blocks in the secure window. The resulting block has corresponding

[6]Note that there is no need to download the random mask as such $\vec{t}$. In practice, it suffices to download the seed that was used to generate those random masks, and use the same random generator to reproduce the same random mask.

coefficient vector $\vec{c}$, i.e. $e = \sum_{i=1}^{n} c_i b_i$. To verify whether a block in the invalid window is corrupted or not, the node applies the random mask to the e block and checks whether the following equation holds:

$$f(e) = \sum_{k=1}^{m} t_k e_k = \sum_{k=1}^{m} t_k (\sum_{i=1}^{n} c_i b_{k,i})$$

$$= \sum_{i=1}^{n} c_i (\sum_{k=1}^{m} t_k b_{k,i}) = \sum_{i=1}^{n} c_i f(b_i)$$

By applying the random mask to the encoded block and comparing it with a combination of mask-hashes weighted by the coefficient vector, a node can check whether any block within the insecure window is corrupted or not. If the alert verification fails, then, the node discards the alert and does not propagate it further. If in turn the alert verification succeeds, then the node forwards the alert to its neighbors and starts the process of cleaning up its insecure window.

Note that the field size of the mask-based hashes $\lambda_q - 1$ is smaller than the field size of the homomorphic collision resistant hashes λ_r, providing weaker security guarantees since fewer possibilities need to be tested during a brute-force attack. Hence, nodes use the mask-based hashes to quickly determine whether an alert is valid or not, but revert to the homomorphic hashes to commit a block as safe.

If the random mask check fails, then, valid alerts may be discarded (*false negatives*) and bad blocks may be temporarily identified as valid ones until the node performs the probabilistic homomorphic hash checking. The probability that a node discards a valid alert is $1/2^{(\lambda_q - 1)}$. However, even if few nodes fail to detect the corrupt block, most nodes will still be able to identify the corrupt blocks and propagate an alert since each node uses different randomly selected masks. Masks can also be periodically refreshed from the server.

B. Microbenchmark

We now quantify the overhead of using masks to prevent DoS attacks in terms of additional data downloaded and processing overhead. To this extend, we have implemented a version of the mask-based hash system in C++. We next present the results of the implementation running on a 2.0 GHz Pentium 4 with the sample parameters given in Table I, 1 GByte file, and 2^{16} blocks.

Using random masks to prevent DoS attacks requires downloading additional security information from the server, however, as we will detail next, the additional overhead compared to the data already downloaded in terms of homomorphic hashes is quite small. Each node first downloads a unique random mask vector $\vec{t}$ for the whole file. The size of each random element t_k is $\lambda_q - 1$, thus, each node needs to download $m \cdot |t_k|$ bytes, which accounts for 16 KByte of data. Rather than sending the random mask vector, in our implementation, each node downloads the seed used by the server to generate the random mask. Based on the seed, nodes can reproduce the random mask vector $\vec{t}$ locally. The size of the seed, typically 64 bits, is negligible.

TABLE II
EFFICIENCY DROP DEPENDING ON THE ALERT'S SPEED.

Alert Rate/Block Rate	Efficiency Drop
1/2	19%
1	15%
5	0%

In addition to the seed, each node also downloads a mask-based hash $f(b_i)$ per block. The size of the mask-based hash vector $\vec{f}$ is $n \cdot (\lambda_q - 1)$, which results in 2 MBytes of data. The amount of data that needs to be downloaded in terms of homomorphic hashes is roughly equal to 8 MBytes for the same number of blocks, which is four times the amount of data downloaded in terms of masks and mask-based hashes. Thus, adding protection against DoS attacks caused by bogus alerts increases the overhead of downloading security-related data by 25%.

Our current implementation of this mask-based scheme is able to generate and check masks at rates close to 160 Mbps. This throughput is about 20 times the rate at which network coded blocks can propagate in our current implementation and about one thousand times the rate at which we are able to check homomorphic hashes. These high throughputs permit each node to check alerts very fast, thus, efficiently blocking bogus alerts and adding a negligible delay on the propagation of valid alerts.

C. Impact of Alert speed propagation

Given that alert messages are very small and they require little processing at each node, valid alerts can propagate much faster than malicious blocks, thus, efficiently halting the malicious block propagation wave. However, if alerts did not propagate fast enough, then, the wave of malicious content could not be halted and the whole system would get easily infected.

To study the impact of the rate of alerts in the efficiency of the content distribution system, we consider a well connected network with average degree equal to 4 with a varying number of attackers and alert speeds. We considered various rates of infection varying from 5% to 20%, resulting in similar results. Table II shows the efficiency drop in terms of additional corrupted packets created as alerts are slowed down, compared to the case where alerts propagate at an infinite rate.

From table II we can see that if the speed of propagation of alert messages is 4-5 times the speed of block propagation, then the performance of the content propagation is almost the same as if the propagation of alert messages was instantaneous. It is reasonable to assume that in realistic scenarios alert messages will propagate at such speeds, specially given that the alert messages are only of small size (compared to the size of encoded blocks) and require little processing at each node. On the other hand, if the speed of alert propagation is the same as the propagation speed of encoded blocks or even lower, the drop in performance, measured as the ratio of correct blocks over the total blocks, is about 15-19 percentile points lower compared to the case of instantaneous propagation; such scenarios are, however, unrealistic.

VIII. Performance of cooperative protection

In this section we analyze the performance improvement of cooperation in detecting corrupted blocks of information when network coding is used. We measure the performance in terms of the percentage of correct blocks transmitted. Our cooperative scheme performs significantly better compared to non-cooperative schemes. We notice that a cooperative scheme provides significant benefits when content is encoded in the network but also when content is not encoded at all (or is encoded only at the server), although the benefits in the later case are smaller.

We start by presenting a simple model that captures many important properties of content propagation, and show analytically the benefits of cooperation. Then, we show using simulation of more complicated models, that cooperation is essential when network coding is used and we quantify the improvement gained by using such cooperative scheme.

A. Analysis

Assume that a non-malicious node gets infected at time $t = 1$. Then, the node with probability p checks the packet and discards the infected packet, and with probability of $1 - p$ sends infected packets to one or more of its neighbors at time (more accurately, round) $t = 2$. Let's denote with $\mathcal{P}(t)$ the number of nodes that are infected at time t, with $\mathcal{P}(1) = 1$ and $\mathcal{P}(0) = 0$.

We assume that each node checks independently with probability p and that the error is detected and corrected by all infected nodes when at least one user checks the packet and cooperation is used. If there is no cooperation for protecting against corrupted packets, then each node detects and discards the corrupted blocks only when it checks with probability p.

a) The case of cooperative protection: The probability of correcting the error at time t is equal to the probability that at time t (and not earlier) at least one infected node checked its content, discovered the corrupted packets, and informed the rest of the infected nodes. This probability is:

$$\Pr[\text{Detect and correct at time t}] = (1-p)^{\sum_{\tau=1}^{t-1}\mathcal{P}(\tau)}\left(1-(1-p)^{\mathcal{P}(t)}\right)$$

In order to compute the expected cost per bad packet introduced into the network, measured in terms of the wasted network capacity, we need to estimate the number of corrupted packets transmitted at each round. The total cost if the malicious packet is discovered at time t is

$$\text{Cost}(t) = \sum_{\tau=1}^{t}\mathcal{P}(\tau)$$

Thus, the expected cost per bad packet is:

$$\begin{aligned} E(\text{Cost}) &= \sum_{t=1}^{\infty}\mathcal{Q}(t)(1-p)^{\mathcal{Q}(t-1)}\left(1-(1-p)^{\mathcal{P}(t)}\right) \\ &< \sum_{t=1}^{\infty}\mathcal{Q}(t)(1-p)^{\mathcal{Q}(t-1)} \end{aligned} \tag{1}$$

with $\mathcal{Q}(t) = \sum_{\tau=1}^{t}\mathcal{P}(\tau)$.

Each term of the summation in Eq. 1 decreases fast as the number of infected users increases, which means that the loss of efficiency gets smaller at each time step. The reason for this is that even though the cost in terms of wasted blocks increases linearly with as more users are infected, the probability that a larger number of users do not find the malicious packet decreases exponentially.

Lemma 8.1 (Cooperative protection): When nodes cooperate to detect and remove corrupted packets, the cost a malicious user can cause by inserting one corrupted packet is constant on average.

Proof: From Eq. 1 we have:

$$\begin{aligned} E\,(\text{Cost}) &< \sum_{t=1}^{\infty}\mathcal{Q}(t)(1-p)^{\mathcal{Q}(t-1)} \\ &= \sum_{t=1}^{\infty}(\mathcal{Q}(t-1)+\mathcal{P}(t))(1-p)^{\mathcal{Q}(t-1)} \\ &= \sum_{t=1}^{\infty}\mathcal{Q}(t-1)\cdot(1-p)^{\mathcal{Q}(t-1)} \\ &\quad + \sum_{t=1}^{\infty}\mathcal{P}(t)\cdot(1-p)^{\mathcal{Q}(t-1)} \end{aligned}$$

Let's assume that the population of infected nodes does not increase arbitrarily fast, i.e. no faster than exponential. This translates to $\mathcal{P}(t) < c\mathcal{Q}(t-1)$, for a constant c. Thus, we need to show that the sum $\sum_{t=1}^{\infty}\mathcal{Q}(t-1)\cdot(1-p)^{\mathcal{Q}(t-1)}$ converges to a constant. By the definition of $\mathcal{Q}(t) \geq t$, since $\mathcal{Q}(t) = \sum_{\tau=1}^{t}\mathcal{P}(\tau)$ and $\mathcal{P}(t) \geq 1$. For $p > 0$, the terms in the summation decrease exponentially, and the summation and the expected cost converge to a constant. ■

The result of Lemma 8.1 is qualitative. It says that the damage created by a single corrupted block is constant, but it does not give any indication of the expected number of corrupted packets (and, thus, wasted network resources) that will be generated.

To get some more quantitative results we model a mesh cooperative system where each node has exactly $k+1$ neighbors chosen randomly among the total set of nodes. With network coding, when a node receives a corrupted block, then all the following blocks generated from the same node will be corrupted. Thus, the impacted node starts transmitting corrupted encodings to the rest of its k neighbors (one of the neighbors is already infected) one at a time. We assume that none of its k neighbors has a corrupted packet (which assumes a very large population of users) and that after exactly k rounds all of them will have a corrupted block. This process continues until at least one user checks its blocks and discovers the bad blocks. In this model, the population of infected users can be described with the following set of equations:

$$\mathcal{P}_i(t+1) = \begin{cases} \sum_{j=0}^{k-1}\mathcal{P}_j(t) & i = 0 \\ \mathcal{P}_{i-1}(t) & 1 \leq i \leq k-1 \\ \mathcal{P}_{k-1}(t) + \mathcal{P}_k(t) & i = k \end{cases} \tag{2}$$

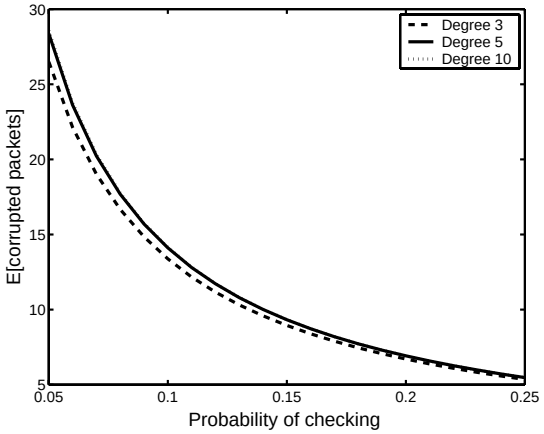

Fig. 5. Expected number of corrupted packets when using network coding and cooperative security as a function of the probability of checking and for three different values of k, the degree of each node.

where $\mathcal{P}_i(t)$ is the number of nodes that have infected i of their neighbors by time t, with $i = 0, \ldots, k$, and $\mathcal{P}_0(1) = 1$ and $\mathcal{P}_i(1) = 0\ \forall i \neq 0$. The total infected population at time t is $\mathcal{P}(t) = \sum_{i=0}^{k} \mathcal{P}_i(t)$.

Using Eq. (2) and (1) we can find the expected number of bad blocks generated by a single corrupted block introduced by a malicious user. The results are given in Fig. 5. Observe that the expected number of bad blocks is slightly higher than the inverse of the probability of checking $(1/p)$, and that this approximation is more accurate as we increase the probability of checking. This result is expected since if the probability of checking is p, then on the average $1/p$ transmissions will take place before a node checks its blocks. This result is also optimal, in the sense that if the nodes cannot check with frequency higher than p, then on the average $1/p$ corrupted blocks will propagate unchecked. Note that changing the number of neighbors does not have a big impact in the efficiency of the system since the probability of finding a malicious block depends mostly on the number of nodes rather than on the topology of the system. Similar results exist for other models of propagation, for example assuming that the distribution takes place in a k-ary tree.

b) The case of non-cooperative protection: Assume now that users do not cooperate. Then, the only way that the corrupted block gets removed from the network is that all infected nodes simultaneously decide to check their content. First, we show that the number of infected nodes is increasing with time, under the assumption that the number of nodes is infinite. (Similar results exist for finite populations of users.)

Lemma 8.2 (Non-cooperative protection): For an infinite population of users that do not cooperate the expected number of infected users is $\mathcal{P}(t+1) = \mathcal{P}(t) \cdot (1-p) \cdot (1+\gamma)$, where γ is the expected number of (not infected) nodes, a node with corrupted blocks infects per round.

Proof: Assume that at time t there are $\mathcal{P}(t)$ nodes and that i of them check and remove their corrupted blocks. The probability of this event is $\binom{\mathcal{P}(t)}{i} \cdot p^i \cdot (1-p)^{\mathcal{P}(t)-i}$. The remaining infected nodes $\mathcal{P}(t) - i$ will send corrupted blocks to $\gamma(\mathcal{P}(t) - i)$ not yet infected nodes. Thus, the expected population of infected nodes at time $t+1$ is:

$$\mathcal{P}(t+1) = \sum_{i=0}^{\mathcal{P}(t)} \binom{\mathcal{P}(t)}{i} p^i (1-p)^{\mathcal{P}(t)-i} \left(\mathcal{P}(t) - i\right)(1+\gamma)$$

which solves to $\mathcal{P}(t+1) = \mathcal{P}(t) \cdot (1-p) \cdot (1+\gamma)$. The expected population of infected nodes will increase when $(1-p)(1+\gamma) > 1$. If the probability of checking $p = 0.1$, then a value of $\gamma > 1/9$ results in an increasing population of infected nodes. ■

An increasing number of nodes with corrupted blocks, results in an increasing waste of resources, since the infected users will generate new corrupted blocks. Thus, the introduction to the system of a single corrupted packet will result in an infinite waste of resources.

B. Performance Comparison

In Sec. VIII-A we have assumed an infinite population of users and studied analytically the effect of introducing a single corrupted block so that we could provide some intuitive analytical results. In more realistic settings, however, we have a finite user population and many malicious users that often introduce corrupted blocks. In this section we simulate such environments and observe that the conclusions of Sec. VIII-A are still valid. Moreover, as we shall see in this section, the analytical results are rather pessimistic since a) in the presence of many attackers, a single check may discover and clean multiple attacks, and b) an infected node does not necessarily transmit corrupted blocks to an non-infected neighbor.

To this extend, we have built a simulator that allows us to study the damage that malicious users can cause in a cooperative content distribution network under different settings (e.g. network coding, no coding, or coding only at the source for non-cooperative and cooperative environments).

We start by generating the overlay topology of the users. We assume that each user has a constant number of neighbors k in the overlay and we construct a random well-connected topology in which each node has k neighbors. The population size was fixed to $500-1000$ nodes in most of our experiments, and the number of neighbors was set to 4; but other values of size and degree gave similar results. We randomly choose a portion of malicious users that generate malicious blocks to jam the system. The percentage of the malicious users varied between 5-20% in our experiments. We also varied the percentage of bad packets injected by a particular malicious user.

The rest of the nodes cooperate to distribute a file that has a large number of blocks and they use either network coding, no coding or coding at the server only. The simulator is round-based, and in each round each user can upload and download at most one block. In each round, each node with probability p verifies the validity of the blocks and, if one or more of them are corrupted, then the node removes them. If cooperative protection is used, then, nodes behave as described in Section VI to alert other infected nodes, so that they also check their content. We measure the number of

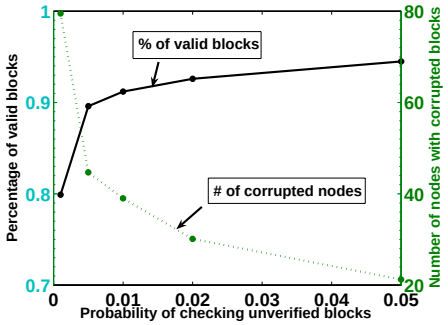

Fig. 6. Percentage of valid blocks and infected nodes for Network Coding Cooperative security as a function of the probability of checking.

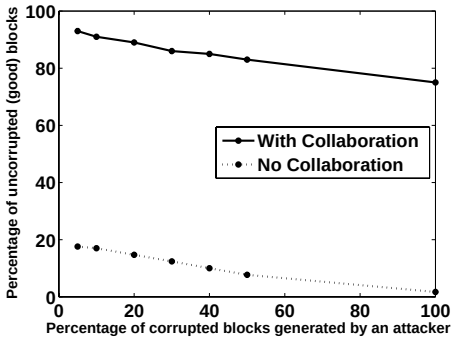

Fig. 7. Percentage of bad blocks for Network Coding with collaboration and no collaboration as a function of the percentage of corrupted blocks generated by an attacker.

transmissions of correct blocks and the number of corrupted blocks (the efficiency of the system is inversely proportional to the percentage of corrupted blocks).

Impact of the probability of checking. In Figure 6 we show the percentage of valid blocks and the number of infected nodes as a function of the probability of checking. We consider a network of 500 nodes in which 10% of them are attackers and send bad packets at a rate of 10%.

From this Figure we can see that for cooperation to be effective it requires a minimum probability of checking (e.g. 1% provides 92% efficiency). However, once passed that probability of checking, the efficiency of the system increases very slowly, hence, not justifying the extra-computational effort. The average number of corrupted nodes per round also drops fast as we increase the probability of checking.

Comparison with other schemes. In Table III we show the percentage of corrupted blocks as a function of the probability of checking for a network of 1000 nodes in which 5% of them are attackers and send a constant stream of bad packets. We study the case of network coding, and no coding (or coding only at the server), with a cooperative and a non-cooperative security system.

From this table we can first see that schemes that do not use encoding or only use source coding can use standard probabilistic batching schemes and only suffer from minor damage in the network. For instance, even if nodes check blocks with probability close to 1%, the damage in the system is still very low.

However, this is not the case for network coding since malicious packets get quickly re-encoded in the network. From Table III we can see that simple batching schemes as the ones proposed in [21], [32] fail to contain the attack. Actually, with network coding and no cooperation, the damage of the system decreases linearly with the checking probability, which requires nodes to check almost every block to have acceptable levels of efficiency.

If cooperation is added, then the performance of both network coding and no coding (or source coding) improves. Such improvement is much more significant for network coding. In fact, a cooperative scheme improves the efficiency of the system almost ten-fold for a checking probability of 2% (see Table III). Thus, with cooperation the system is able to limit the propagation of corrupted blocks. Moreover, observe that the percentage of corrupted blocks is very close to the minimum, which in this example is 5%.

Impact of the rate of attack. We next study the effort that an individual attacker needs to make to infect the network. Figure 7 shows the efficiency of the system for a network of 500 nodes where each node uses network coding and checks with 1% probability as a function of an attacker's rate (we assume 10% attackers). We first observe that the efficiency of the system largely depends on whether a cooperative scheme is in place, droping to less than 20% if this is not the case. We also observe that the efficiency of the system increases linearly as the rate of infection of a malicious node decreases, achieving 90% efficiency for an attack rate of 10%.

Colliding attackers. We now show that as the number of attackers increases, attacks overlap and the attack efficiency drops. As attackers overlap, the same blocks are infected multiple times by different attackers, and a well-behaving node is able to halt the damaged caused by multiple attackers with a single security check operation.

In Table IV we show the effect of an increasing number of attackers in a network of 1000 nodes, where attackers send a continuous stream of corrupted blocks, and the probability of checking is 5%. We see that as the number of attackers goes from 1 to 100, the mean percentage of corrupted blocks per attacker decreases from 1% to 0.19%. Thus, the attack's efficiency drastically decreases as the number of attackers increases. This is an important result. Obviously we expect that the performance of the system decreases as the percentage of attackers increases. However, as the attack grows, attackers need to put a very large effort to slightly increase the infection of the system.

C. Running Time

We now present the overall throughput that can be achieved in the system with and without our cooperative system. Our current implementation of homomorphic hash checking achieves throughputs of 128 Kbps, which is about 62 times slower than the rate of 8 Mbps at which we can encode new blocks. For an average infection rate γ of 5% of malicious nodes, nodes only need to check 1.2% of the blocks to keep the percentage of bad blocks below 13%.

TABLE III

PERCENTAGE OF BAD BLOCKS AS A FUNCTION OF THE PROBABILITY OF CHECKING.

p	Network coding with cooperation	Network coding without cooperation	No network coding with cooperation	No network coding no cooperation
0.5%	26.8%	97.8%	6.2%	16.2%
1.0%	15.5%	97.1%	5.6%	15.1%
1.5%	11.6%	96.7%	5.4%	14.4%
2.0%	9.8%	96.2%	5.3%	13.5%
3.0%	8.1%	95.2%	5.3%	12.4%
4.0%	7.8%	94.3%	5.2%	11.8%
5.0%	7.2%	93.5%	5.2%	11.3%
10.0%	6.0%	88.7%	5.2%	9.7%
20.0%	5.5%	79.3%	5.1%	7.7%

Note: 1000 nodes with 50 (5%) malicious nodes. No network coding includes both source coding and no coding at all.

TABLE IV

EFFECT OF THE NUMBER OF MALICIOUS NODES.

Number of attackers	Mean percentage of corrupted blocks		Mean number of corrupted nodes	
	Total	Per attacker	Total	Per Attacker
1	1.0	1.0	7.39	7.39
2	1.5	0.76	11.80	5.90
5	3.2	0.64	22.00	4.40
10	4.8	0.48	32.10	3.21
20	7.0	0.35	46.47	2.32
50	12.1	0.24	64.35	1.29
100	18.8	0.19	81.37	0.81

Table V summarizes the rate at which data can be checked using different schemes. We denote $MultCost(r)$ as the cost of multiplication in $\mathbb{Z}_r^*$. We also assume that most of the cost of checking homomorphic hashes is determined by the cost of making repeated exponentiations (left side of Equation 1). The first scheme corresponds to a naive homomorphic scheme that checks every block. In this case the number of corrupted blocks is limited by the number of malicious packets injected in the system, however, the throughput of the overall system is very small. The second scheme uses batching to reduce the computation overhead of the homomorphic hashes with a batch size $B = 256$ blocks. In this case, the throughput of the system is 32 Mbps, however, the infection rate is close to 100%. Finally, the cooperative scheme has a running time that depends on the probability of checking p, the infection rate γ, and the number of infected nodes per malicious packet $\mathcal{P}$. Our experimental results show that the value of $\mathcal{P}$ is usually a small constant (i.e. $\ll 10$). Given this, the cooperative approach provides a throughput higher than 10 Mbps, which allows nodes to check blocks faster than they are propagated while limiting the damage in the system to the attackers' effort.

IX. RELATED WORK

Network coding performs almost an optimal scheduling of content among nodes without the need for a global knowledge of the system. As such, network coding has recently emerged as a practical and elegant way of maximizing the throughput of P2P content distribution systems [2], [6], [34], [35]. However, little effort has been paid to discussing and understanding the security threats that arise from using network coding in large scale distributed P2P systems.

Common approaches to providing source-authentication in multicast systems often rely on either share secret keys or using asymmetric cryptography; for a taxonomy of security concerns see [43]. However, in a system where intermediate nodes in the network are allowed to re-encode content, then, the standard way of having valid senders sign their packets cryptographically does not work.

Khron et al. [21] was the first to provide a mechanism that allows for intermediate nodes to check erasure codes on-the-fly using homomorphic hashing functions [36]–[42]. Similarly, Distillation Codes use Merkle hash trees to determine which is the set of valid packets out of a large set of encoded packets with both valid and invalid packets.

However, these mechanisms are not efficient in slowing down attacks when network coding is used. Homomorphic hashing functions are too costly and often nodes can not afford to use them on every block received. Similarly, Distillation codes are based on the assumption that intermediate nodes do not re-encode the content. Another possible solution is to batch several blocks and check them all together to reduce the computational overhead [21] [32]. However, with network coding, standard batching techniques may cause particularly serious damage.

X. CONCLUSIONS

Existing, peer-to-peer content distribution networks, can use simple cryptographic primitives such as hash trees to authenticate data and prevent unverified downloads. However, these techniques do not work well with recent network coding techniques that have been proposed to improve the resilience and the throughput of such networks.

In this paper we study the security issues that arise from using network coding. We pay special attention to those attacks that try to destroy the entropy of the system or jam it completely. To improve the verification efficiency we propose a novel scheme where users not only cooperate to distribute content, but (well-behaved) users also cooperate to protect themselves against malicious users by informing affected nodes when a malicious block is found. Moreover, we present an efficient mechanism to prevent DoS attacks against bogus alert messages based on random masks and mask-based hashes.

By having a large number of nodes checking at every point in time and making them cooperate, we are able to efficiently

TABLE V

THROUGHPUT OF CHECKING FOR DIFFERENT SCHEMES.

Scheme	Running Time	Infection (%)	Throughput (Mbps)
Naive Homomorphic	$m\lambda_q MultCost(r)$	γ, (5%)	0.128
Batching Homomorphic	$m\lambda_q MultCost(r)/B$	$1-\frac{1}{B}$, (99.6%)	32.7
Cooperative Security	$m\lambda_q MultCost(r)(p+\gamma\mathcal{P})$	γ, (5%)	10.6

reduce the computation overhead at each node while efficiently limiting the damage in the system. We currently have an implementation of a P2P content distribution system based on network coding similar to the one described in this paper which supports security against entropy as well as jamming attacks. We plan on reporting our experiences with such live system in future work.

REFERENCES

[1] R. Rejaie and S. Stafford, "A Framework for Architecting Peer-to-Peer Receiver-driven Overlays", *Nossdav 04*, Ireland, June 2004.

[2] K. Jain, L. Lovasz, and P. A. Chou, "Building scalable and robust peer-to-peer overlay networks for broadcasting using network coding", *ACM Symposium on Principles of Distributed Computing*, Las Vegas, 2005.

[3] P. A. Chou, Y. Wu, and K. Jain, "Network coding for the Internet", *IEEE Communication Theory Workshop*, Italy, May 2003.

[4] *http://www.akamai.com*

[5] Ying Zhu, Baochun Li, Jiang Guo, "Multicast with Network Coding in Application-Layer Overlay Networks", *IEEE Journal on Selected Areas in Communications*, January 2004.

[6] P. A. Chou, Y. Wu, and K. Jain, "Practical network coding", *Allerton Conference on Communication, Control, and Computing, Monticello, IL*, October 2003.

[7] Zongpeng Li, Baochun Li, Dan Jiang, and Lap Chi Lau, "On Achieving Optimal End-to-End Throughput in Data Networks: Theoretical and Empirical Studies", *ECE Technical Report, University of Toronto*, February 2004.

[8] R. Ahlswede, N. Cai, S. R. Li, and R. W. Yeung, "Network Information Flow", *IEEE Transactions on Information Theory*, July 2000.

[9] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "SplitStream: High-Bandwidth Multicast in Cooperative Environments", *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[10] V. Padmanabhan, H. Wang, P. Chou, and K. Sripanidkulchai, "Distributing Streaming Media Content Using Cooperative Networking", *Proc. of NOSSDAV 2002*, May 2002.

[11] J. Byers and J. Considine, "Informed Content Delivery Across Adaptive Overlay Networks", *Proc. of ACM SIGCOMM*, August 2002.

[12] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat, "Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh", *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, 2003.

[13] Paul Francis, "Yoid: Extending the internet multicast architecture", *Unpublished paper*, April 2000.

[14] Yang-hua Chu, Sanjay G. Rao, and Hui Zhang, "A Case For End System Multicast", *Proceedings of ACM SIGMETRICS*, Santa Clara,CA, June 2000, pp 1-12.

[15] Vivek K Goyal, "Multiple Description Coding: Compression Meets the Network", *IEEE Signal Processing Magazine*, May 2001.

[16] B. Cohen, "Incentives build robustness in BitTorrent", *P2P Economics Workshop*, 2003.

[17] Rob Sherwood, Ryan Braud, Bobby Bhattacharjee, "Slurpie: A Cooperative Bulk Data Transfer Protocol", *IEEE Infocom*, March 2004

[18] P. Rodriguez, E. Biersack, "Dynamic Parallel-Access to Replicated Content in the Internet", *IEEE Transactions on Networking*, August 2002

[19] John Byers, Michael Luby, and Michael Mitzenmacher, "Accessing Multiple Mirror Sites in Parallel: Using Tornado Codes to Speed Up Downloads", *Infocom*, 1999.

[20] M. Izal, G. Urvoy-Keller, E.W. Biersack, P. Felber, A. Al Hamra, and L. Garces-Erice, "Dissecting BitTorrent: Five Months in a Torrent's Lifetime", *Passive and Active Measurements 2004*, April 2004.

[21] M. N. Krohn, M. J. Freedman, and D. Mazières, "On-the-fly Verification of rateless erasure codes for efficient content distribution" *IEEE Symposium on Security and Privacy*, 2004

[22] Dongyu Qiu, R. Srikant, "Modeling and Performance Analysis of BitTorrent-Like Peer-to-Peer Networks", *In Sigcomm 2004*.

[23] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege, "A Digital Fountain Approach to Reliable Distribution of Bulk Data", *SIGCOMM, 1998*.

[24] Petar Maymounkov and David Mazires, "Rateless Codes and Big Downloads", *IPTPS'03*, February 2003.

[25] K. Jain, M. Mahdian, and M. R. Salavatipour, "Packing Steiner Trees", *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2003.

[26] G. Robins and A. Zelikovsky, "Improved Steiner Tree Approximation in Graphs", *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2000.

[27] M. Thimm, "On The Approximability Of The Steiner Tree Problem", *Mathematical Foundations of Computer Science 2001*, Springer LNCS, 2001.

[28] T. Ho, R. Koetter, M. Medard, D. Karger, and M. Effros, "The Benefits of Coding over Routing in a Randomized Setting", *ISIT*, Yokohama, Japan, 2003.

[29] G. Pandurangan, P. Raghavan and E. Upfal, "Building Low-diameter P2P Networks", *42nd Annual Symposium on Foundations of Computer Science (FOCS01)*, pp. 492-499, 2001.

[30] M. Krohn, M. FreedMan, D. Mazieres, "On-the-Fly Verification of Rateless Erasure Codes for Efficient Content Distribution", *IEEE Symposium on Security and Privacy*, Berkeley, CA, 2004.

[31] E. Adar, B. Huberman, "Free Riding on Gnutella", *First Monday*, Available at: http://www.firstmonday.dk/issues/issue5_10/adar/, 2000.

[32] M. Bellare, J. A. Garay, and T. Rabin, "Fast batch verification for modular exponentiation and digital signatures", *Advances in Cryptology*, 1998.

[33] P. Felber, and E.W. Biersack. "Self-scaling Networks for Content Distribution", *In Proceedings of the International Workshop on Self-* Properties in Complex Information Systems (Self-*)*, Italy, 2004

[34] C. Gkantsidis, and P. Rodriguez. "Network Coding for Large Scale Content Distribution", *In IEEE/Infocom*, 2005.

[35] S. Deb, C. Choute, M. Medard, and R. Koetter. "How Good is Random Linear Coding Based Distributed Networked Storage?, *NetCod 2005*

[36] T. P. Pedersen "Non-interactive and information-theoretic secure verifiable secret sharing", *Advances in Cryptology CRYPTO*, 1991.

[37] D. Chaum, E. van Heijst, and B. Pfitzmann, Cryptographically strong undeniable signatures, unconditionally secure for the signer, *Advances in Cryptology CRYPTO*, 1991.

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

[39] N. Baric and B. Pfitzmann, "Collision-free accumulators and failstop signature schemes without trees", *Advances in Cryptology EUROCRYPT 97*, 1997.

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

[41] S. Micali, and R. Rivest, "Transitive signature schemes", *Progress in Cryptology CT RSA 2002*, 2002.

[42] R. Johnson, D. Molnar, D. Song, and D. Wagner, "Homomorphic signature schemes", *Progress in Cryptology CT RSA 2002*, 2002.

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

[44] C. Karlof, N. Sastry, Y. Li, A. Perrig, and J. Tygar, "Distillation codes and applications to DoS resistant multicast authentication", *in Proc.*

11th Network and Distributed Systems Security Symposium (NDSS), San Diego, CA, Feb. 2004.

[45] R. Merkle, “Protocols for public key cryptosystems”, *in Proc. of the IEEE Symposium on Research in Security and Privacy*, Apr. 1980.

[46] Christos Gkantsidis, Milena Mihail, Amin Saberi, “Conductance and congestion in power-law graphs” *ACM SigMetrics*, 2003.

[47] P. Maymounkov and D. Mazieres, ”Kademlia: A peer-to-peer information system based on the XOR metric”, *In Proceedings of IPTPS02*, 2002.

[48] A. Rowstron and P. Druschel, ”Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems”, *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001,