SecureStream: An Intrusion-Tolerant Protocol for Live-Streaming Dissemination

Maya Haridasan

Department of Computer Science, Cornell University

Robbert van Renesse

Department of Computer Science, Cornell University

Abstract

Peer-to-peer (P2P) dissemination systems are vulnerable to attacks that may impede nodes from receiving data in which they are interested. The same properties that lead P2P systems to be scalable and efficient also lead to security problems and lack of guarantees. Within this context, live-streaming protocols deserve special attention since their time sensitive nature makes them more susceptible to the packet loss rates induced by malicious behavior. While protocols based on dissemination trees often present obvious points of attack, more recent protocols based on pulling packets from a number of different neighbors present a better chance of standing attacks. We explore this in SecureStream, a P2P live-streaming system built to tolerate malicious behavior at the end level. SecureStream is built upon *Fireflies*, an intrusion-tolerant membership protocol, and employs a pull-based approach for streaming data. We present the main components of SecureStream and present simulation and experimental results on the Emulab testbed that demonstrate the good resilience properties of pull-based streaming in the face of attacks. This and other techniques allow our system to be tolerant to a variety of intrusions, gracefully degrading even in the presence of a large percentage of malicious peers.

Key words: Content dissemination, multicast, live-streaming, intrusion-tolerance

1 Introduction

Access to multimedia contents over the network now accounts for a large fraction of Internet traffic. This has been possible in great part because of peer-to-peer (P2P) content distribution tools, which allow the distribution of popular data to a large number of interested users. One popular style of

Preprint submitted to Elsevier *31 March 2007*

content distribution maps the problem to file sharing, where data is fully available prior to the dissemination. The main goal in file sharing is that all nodes receive the entire data within as little time as possible.

In this work, we are focused on a second scenario, P2P live-streaming, where data should be disseminated as it is generated. This style of distribution is useful to broadcast live events in close to real time and also to broadcast television over the web. In China, for example, live-streaming has become very popular, where participating peers' upload bandwidth is used to simultaneously propagate several channels to thousands of users [1].

Streaming to a large number of clients would be prohibitively expensive if the service provider should have enough bandwidth to satisfy all the clients. Several P2P multicast protocols which rely on users' upload resources have been proposed and widely studied as an appealing alternative to IP multicast, and previous work has shown that it can indeed be as efficient as IP multicast [2–5,1,6,7]. By having peers contribute to the streaming, anyone may start their own streaming session to any number of clients. Significant progress has been made, but little attention has been dedicated to the issue of security in such systems.

In this paper we target live-streaming, where malicious behavior can prevent nodes from receiving correct packets in time, and can therefore be severely disruptive. To illustrate the problem, we looked into the effects of one particular type of attack when using a single dissemination tree with varying branching factors and when using the more elegant SplitStream approach [4]. SplitStream is a robust and fair P2P system in which data is broken into several slices and each slice is propagated through a different dissemination tree.

As a measure of resilience, we compute the *continuity index* of a streaming session, which is the ratio of packets received by a peer within acceptable time. Through simulation we computed the minimum continuity index across participants for sessions with a thousand homogeneous nodes and varying ratios of malicious peers not forwarding packets.

In Figure 1, we present simulation results of the average and minimum continuity index across nodes for sessions with a thousand homogeneous nodes and various ratios of malicious peers not forwarding packets. In these experiments, attackers download data from their parents but do not forward it to their children. In the case of single trees, most certainly malicious behavior will prevent individual nodes from receiving any packet even with as low as 5% malicious members. SplitStream presents better resilience, but the damage incurred to individual nodes is still very visible.

We built SecureStream, which employs several techniques that reduce the

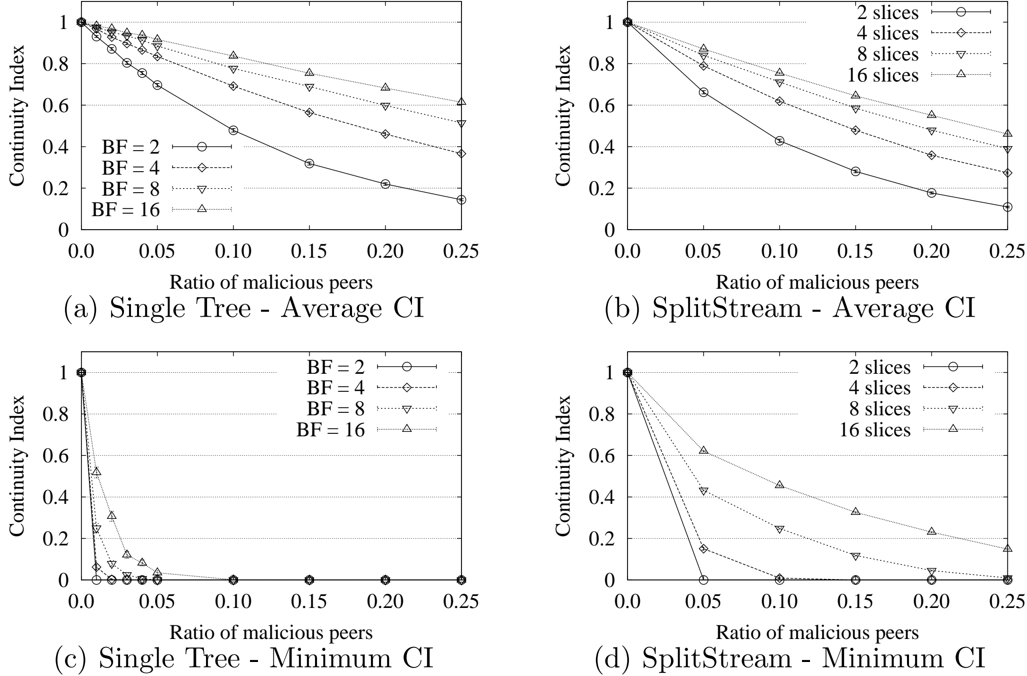


Fig. 1. Expected minimum and average continuity indices across all correct members under omission attacks.

opportunity for an attacker to compromise the quality of a streaming session, without incurring a high computational or network overhead. To repel forgery attacks, we employ an efficient packet authentication technique based on computing and distributing verification digests. To prevent attacks on the overlay structure (the membership protocol on top of which multicast systems operate), SecureStream is built upon *Fireflies*, a scalable one-hop Byzantine membership protocol [8]. *Fireflies* is a probabilistic protocol, in which members are presented with a reasonably current view of which members are live or not.

To achieve tolerance to denial-of-service attacks, SecureStream uses a pull-based packet dissemination approach, similar to the one used by the CoolStreaming [1] and Chainsaw protocols [6]. This approach is attractive because it offers participants a choice among multiple candidate packet sources. Because participants are not dependent on any particular peer and can immediately react to failures or attacks, attacks are less damaging.

Finally, we explore the use of auditing techniques that applied to SecureStream can help further alleviate the effects of malicious behavior, while incurring limited additional costs. We propose employing a variable threshold for node contribution, punishing peers who do not upload at least as much data as defined by the threshold.

This paper makes a few important contributions. We present a highly scalable

live-streaming P2P protocol that can tolerate end system attacks. We leverage previous work and present a comparison of different authentication protocols for signing and verifying packets efficiently in the context of application level multicast. We also evaluate the effects of pull based protocols in the presence of internal malicious peers. Finally, we study the potential of auditing as a mechanism for encouraging node contribution.

The rest of the paper is organized as follows. In Section 2, the system model and assumptions are presented, including a list of possible attacks to P2P streaming systems. A description of the main techniques we employed in building SecureStream is presented in Section 3. In Section 4 results of the experimental evaluation of the resilience to malicious behavior are presented and analyzed. Section 5 presents related work, and Section 6 concludes.

2 System Model

Our model of the system assumes the existence of one source, assumed non-compromised, disseminating data at a fixed rate to a set of receivers with limited buffering capacity. All nodes have similar download and upload capacities, slightly larger than the download rate. The desired behavior is that the streamed data be received within a fixed latency relative to the source's original transmission.

The term security in the context of content dissemination protocols requires further definition. According to [9] multicast systems may have different requirements: secrecy means that only multicast group members (and all of them) should be able to decipher transmitted data; authenticity means that each group member can recognize whether a message was sent by a group member and make sure that the data was not modified in any way; anonymity implies that identity of group members should be kept secret from outsiders or from other group members; non-repudiation states that receivers of data should be able to prove to third parties that the data has been transmitted; access control means that it should be possible to control the group membership; and finally, service availability means that the system should be always up.

Not all applications require secrecy and anonymity of data, hence we are not concerned with these properties. On the other hand, we believe authenticity, non-repudiation and access-control are essential. We assume that the original data is non-compromised, and therefore implicitly achieve data integrity through authenticity. Our primary focus is on guaranteed availability, namely mechanisms that prevent nodes from being isolated or severely harmed during a streaming session. We expect the system to repel external attacks and tol-

erate a limited fraction of internal Byzantine nodes, and to degrade gracefully as the fraction of Byzantine nodes increases.

SecureStream is an application-level streaming system, and only attacks to the end system hosts are addressed in this work. Attacks on the underlying network infrastructure and low-level *denial-of-service* attacks are thus beyond the scope of this work.

2.1 Byzantine Behavior

We model all forms of deviation from the original protocol as byzantine behavior. These deviations may be due to node failures, node selfishness or purely malicious intents. Attacks may originate outside the system or be internal, and attackers may compromise nodes and then work in cooperation with these faulty internal peers. One important observation is that we opted for modeling selfishness as a byzantine behavior, and to assume that most nodes typically follow the given protocol.

To the best of our knowledge, there has been no evaluation on the percentage of selfish behavior in live-streaming systems, unlike with file sharing systems, and the properties of these systems are significantly different. Since nodes are only required to upload while the streaming session is occurring, it is our belief that few nodes would opt for deviating from the proposed protocol.

The simplest form of internal attacks are those in which a single node is compromised. The extent of harm that results depends on many factors, such as the multicast protocol being used and the location of the malicious node in the overlay. These effects can be localized and minimized if the protocol in use has no single points of failure. On the other hand, vulnerable systems like those based on a single dissemination tree can be crippled if a node high in the tree is compromised.

Collusion attacks pose much more complex problems; in these, an attacker compromises a set of nodes and exploits them to perform a coordinated attack to the system, and may orchestrate the attack to confound whatever defensive mechanisms are built into the dissemination infrastructure.

For the work presented here, we make several assumptions about compromised members. They do not have sufficient computational power to break cryptographic building blocks, and cannot forge public key certificates or signatures of correct or stopped members. A classification of the types of attacks that we address in our system is presented below.

Membership attacks: The system may be attacked by compromising the

underlying overlay or membership protocol on which it runs. For example, systems that run on top of ring-based overlays are vulnerable to eclipse attacks [10], in which an attacker controls a large fraction of the neighbors of correct nodes, preventing correct overlay operation. Malicious nodes may also mimic flaky but correct members, or accuse other correct members of being down.

Forgery: In this category we include all attacks that involve fabrication and tampering of data being streamed in the system. Given time, these attacks can be easily avoided by use of a public key infrastructure. However, in the context of streaming the cost of signatures can become prohibitively high, forcing us to consider other kinds of data authentication protocols.

Denial-of-service (DoS) Attacks: Attacks in which malicious nodes overload peers with requests for packets or large amounts of duplicate packets, or other attacks that might compromise their ability to contribute to the streaming session.

Omission Attacks: Given our emphasis on low-latency data delivery, send-omission is an especially serious type of attack. By not forwarding all or part of the packets, a malicious node may disrupt overall system's availability. The main problem with this kind of attack is that a node's guilt cannot be easily proved.

3 Steps to Intrusion-Tolerant live-streaming

SecureStream employs a set of techniques to achieve resilience to the attacks previously mentioned. We use an intrusion-tolerant membership protocol to tolerate attacks to the membership layer. We also employ an efficient technique for avoiding forgery of packets by malicious peers. By employing a pull-based streaming protocol and imposing a structure to define what peers are allowed to communicate with one-another, we can avoid high-level DoS attacks and tolerate omission attacks. We also explore the potential of auditing as a tool for detecting malicious behavior. In this section, we describe these main components in further detail.

3.1 Presenting nodes with a correct view of live members

Peers in SecureStream use the membership knowledge provided by the *Fireflies* protocol to track the status of other peers. *Fireflies* is composed of three subprotocols: a pinging protocol is used to detect failures of nodes with an accuracy independent of message loss; an intrusion-tolerant gossip protocol is used for dissemination of information between correct members with probabilistic time bound Δ; and a membership protocol uses accusations and rebut-

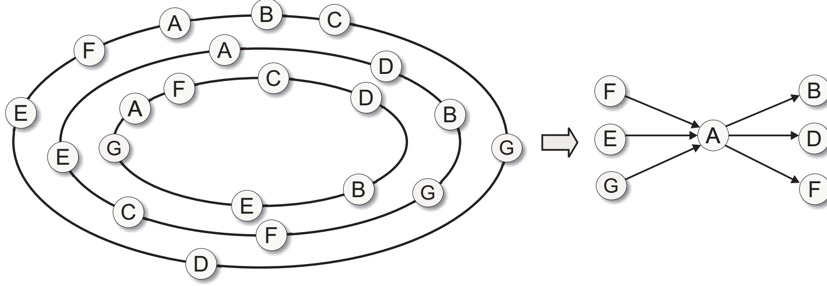


Fig. 2. In Fireflies multiple rings are used to define which peers monitor each other: A monitors B, D and F, and is monitored by E, F and G.

tals to implement the membership information that *Fireflies* provides. These components are briefly described below.

Members monitor each other for failure using an adaptive pinging protocol. Members do not use a static global timeout when waiting for the replies of ping messages, but rather estimate the probability of message loss and try to adapt to the message loss characteristics between monitor and monitoree.

Members are organized into rings, and their position on each ring depends on their identifier. These rings determine which nodes monitor, and are allowed to accuse, which other nodes (Figure 2). On each ring, each member m_i monitors the lowest ranked successor m_j that it believes to be live, and if it detects a failed node, it issues an *accusation* for that node.

When an accusation for a member m_i is received by a member m_j, m_j waits a time period of length 2Δ, and then removes m_i from its view if the accusation is valid. This time period is established so that an accused member may issue a new *note* (a *rebuttal*) to an *accusation* against itself. In order to avoid malicious nodes from abusively accusing its correct neighbors in the rings, nodes may invalidate up to t rings, implying that accusations issued by its neighbors on those rings will not be accepted as valid by any correct member. All notes and accusations are signed, and a certification authority is responsible for issuing private/public key pairs and public key certificates.

The dissemination of information such as accusations and rebuttals is performed using a robust gossip protocol. Each member periodically picks a random member from its view to exchange state information. The multiple ring structure induces a gossip mesh resilient to malicious attacks.

3.2 Ensuring Integrity of Data

One second important aspect which needs to be satisfied is that the data being distributed is correct. Several authentication protocols have been proposed for

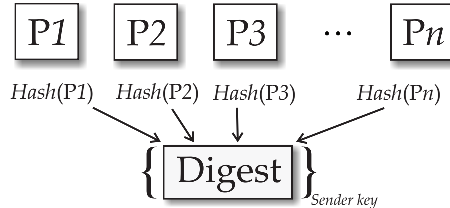


Fig. 3. In the linear digests' approach, packets' hashes are computed and combined into a single digest packet, which is then signed by the sender.

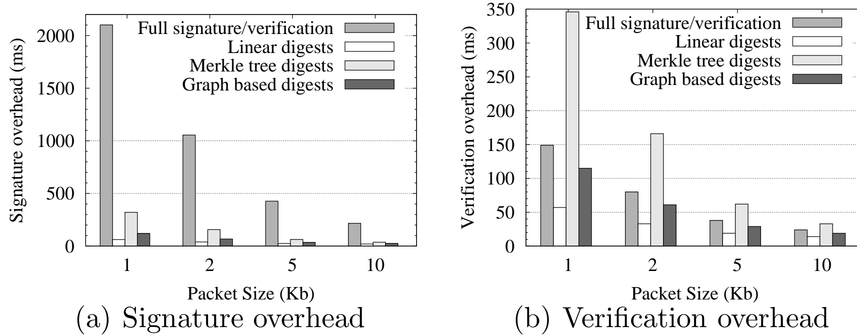
the general multicast paradigm, originally intended for IP Multicast. The standard point-to-point mechanism of appending a message authentication code (MAC) computed using a shared key does not meet the security requirements of a multicast session. If receivers and sender share the same key, any receiver would be able to forge messages. On the other hand, signing every packet using a traditional asymmetric cryptographic protocol induces high overhead, and is therefore not feasible.

Signing a packet consists of computing the hash of the contents of the packet using a secure hash function, and then signing the hashed value using the sender's private key. Variations in the packet size do not significantly contribute to the costs since computing the hash of packets is a cheap operation compared to the signature/verification operation. The choice of key size to be used is directly related to how crucial it is that the key be secret for a long time, and it is often recommended that keys of size 2048 or larger be used.

To avoid signing and verifying every packet, we group the hashes of n packets into a special message, and have it signed by the source (we call this approach *linear digests*)(Figure 3). The signed message needs to be sent to the receivers prior to the dissemination of data that it corresponds to. This implies in buffering of content on the source prior to the dissemination of data. The advantage is that this approach incurs the minimal network overhead of one hash per packet, while amortizing the cost of a single signature/verification operation over n packets.

Other approaches have been proposed to address the high costs of authenticating packets in a flow [11–15]. Wong and Lam [11] propose that the source compute the hashes of a limited number n of consecutive packets in the stream, and use them as leaves in a Merkle Tree where each internal node consists of the hash of its children. Each packet is verifiable upon receipt, since it is appended with the signed root node and the hashes of all needed interior nodes in the path from the root to itself in the Merkle Tree.

In graph-based authentication [13,16,17,14], the source only signs one packet, and the following packets in the stream are linked to it through hash chains that allow them to be verifiable. To tolerate packet loss, a graph is used instead of a single chain. Packets are represented by vertices in the graph, and a



(a) Signature overhead (b) Verification overhead

Fig. 4. Computational overheads per second when transmitting 300 Kb/s using varying packet sizes.

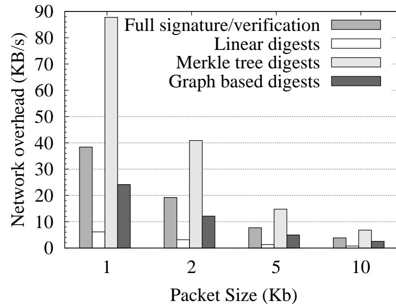


Fig. 5. Overheads per second when transmitting 300 Kb/s using varying packet sizes.

directed edge between nodes that represent packets P_i and P_j indicates that packet P_j contains the hash of packet P_i. A packet corresponding to a node can be authenticated if there is a path of already verified packets between the node and the source node of the graph.

The computational costs at the source and receivers are presented in Figures 4(a) and 4(b) and the network overheads for different authentication approaches when streaming 300 Kb/s are presented in Figure 5. We compared 4 techniques: signing and verifying every packet, linear digests, Merkle tree digests and a simple scheme of graph-based authentication. The code used for the evaluation was written in Python and executed on a Linux-based Pentium III 850 Mhz with 256 Mb RAM. Linear digests yield the lowest computational costs. Although the Merkle Tree approach is appealing due to its immediate verifiability, its network overhead is the highest, since one signature and a few hashes need to be appended to each packet in the flow.

When the packet sizes are large, which reduces the rate of packets per second, the computational costs of the three latter techniques are not significantly different. We therefore used linear digests since it minimizes network overhead and is the simplest technique. Our experience indicated that when using pull based streaming, keeping the rate of packets per second larger or equal to 30 yields good results, and reducing it further affects the quality of the streaming.

3.3 Allowing nodes to recover from malicious neighbors

We employ a pull-based approach to disseminate packets, following ideas used in the Chainsaw protocol [6]. The same rings used in *Fireflies* are used to determine a fixed set of neighbors with which each peer can exchange packets. This imposed mesh structure and the use of authenticated channels between neighbors allows the system to avoid high-level DoS attacks.

Initially, the source sends notifications to its neighbors as soon as it has available packets to disseminate. These notifications are small messages used only to inform neighbors of availability of packets. Each neighbor requests missing packets according to some pre-specified policy, to avoid overloading the source. As peers receive packets, they propagate notifications to their neighbors, and so packets get disseminated along the mesh. This pull-based approach to acquisition of packets yields a highly resilient multicast, since failure or misbehavior of one neighbor does not impede a peer from fetching packets from other neighbors. The predetermined set of neighbors for each peer also makes it hard for malicious peers to attack individual peers, since attackers lack a deterministic means of acquiring control of all of its neighbors.

Each member stores packets and forwards them to other peers while the packet is within its *availability window*. It also maintains an *interest window*, smaller than the availability window, which represents the set of packets in which the peer is currently interested. Different policies can be employed by peers about what packets to pick from each of its neighbors, and the choice of the appropriate policy is crucial to achieving best overall performance. Random selection of neighbors is usually a good candidate, leading to fair load balancing.

There is a predefined limit l on the number of outstanding requests to any neighbor. This policy not only improves the flow of packets in the absence of malicious behavior, but also makes it harder for malicious peers to overrequest packets from their neighbors. Peers maintain a queue of non-satisfied requests for packets, and if more than l requests by the same neighbor are present in the queue at any time, only the l most recent ones are maintained.

The protocol is simple and yet highly resilient to failures and attacks. The overhead incurred by notifications is not significant if large packets are used, and the protocol avoids receipt of duplicate packets. Since it is completely decentralized, the protocol does not present any single points of failure, another important consideration when building an intrusion-tolerant streaming protocol.

To ensure non-repudiation, peers may only forward packets once they have verified its authenticity. If peers are allowed to forward packets optimistically before ensuring that the packet has not been tampered with, it becomes in-

feasible to later identify the peer responsible for the tampering. This can be explored by malicious nodes, who may overload the system with incorrect packets without being accountable for them. In the linear digest approach, the packet that contains the digest is critical to the verifiability of packets, and therefore should be received by all nodes. Furthermore, it should ideally be received prior to other packets for which it contains hashes, so that they can be immediately verified.

In our streaming protocol, simply treating the digest packet as a regular packet would not yield the desired results. We proposed the following optimization to ensure immediate verifiability: each peer, when requesting a packet, uses a special bit in the request messages that indicates whether the digest packet for the current session has been received or not. Since digest packets are small, they can be appended to the packet sent in reply to the request. This would ensure that all packets are immediately verifiable, incurring a small overhead caused by duplicate digests.

3.4 Punishing Malicious Nodes

Despite the resilience of pull-based streaming to malicious behavior, we can provide further guarantees to correct peers by auditing their behavior. In this Section, we explore a simple auditing approach: to ensure that all nodes in the system contribute more than a particular specified threshold. Violations to this invariant may lead system nodes who contribute less than a particular threshold to suffer some type of punishment, such as being expelled from the system. Independent of the punishment, implementing an auditing component requires caution, and in this subsection we present some of the techniques we employ to achieve this goal.

As part of the auditing approach, each peer should group packets it receives from each neighbor every δ seconds. At the end of every interval, each peer generates and sends one signed receipt for all packets received from each of its neighbors during that interval, and collects receipts received from them. Peers are encouraged to forward receipts to their suppliers to guarantee that future requests for packets continue to be satisfied.

Auditing may be performed by dedicated external auditors, whose role is solely to identify misbehaving nodes. We propose a decentralized approach, which combines local auditors, executing at the participating peers, and global auditors, who react to violations reported by the local peers. A local auditor has two main roles. First, it acts as a representative of its local node, querying it for the set of packets it received and the set of receipts collected (packets it sent) over any particular time interval. The auditor publishes this information

to an assigned subset of its neighboring nodes, from whom other auditors may obtain it. This level of indirection is used to guarantee that each node provides the same information to all auditors.

The second role consists of periodically auditing information about the nodes with whom their local node exchanges packets. For instance, if node A exchanges packets with nodes B, C and D in the live-streaming protocol, node A's auditor monitors information regarding these three nodes. This involves ensuring that: (1) the amount of data sent by these nodes satisfies the minimum threshold; and (2) the set of packets they claim to have received from node A corresponds to the set of packets A claims to have sent to them.

In our hybrid model of auditing, global auditors only respond and act upon violations flagged by the local auditors. In order to avoid delayed detection, local auditing works continuously within small groups of nodes. We argue that a variable threshold yields better results than a static one, leaving it to the global auditors to decide what this value should be. Therefore, besides acting on information provided by local auditors, global auditors also constantly sample the amount of packets sent and received by randomly chosen individual nodes, and use this information to decide what the threshold should be at any point in time.

4 Evaluation

We originally evaluated the resilience of pull-based streaming in the presence of attacks through simulation. We also implemented SecureStream using Python, and we validated the simulation results by running experiments with the real system on the Emulab testbed [18]. Emulab is a network testbed containing hundreds of nodes, in which real applications may be executed and evaluated. It allows arbitrary network topologies to be specified, leading to a controllable and repeatable environment.

4.1 Simulation

We built an event-driven simulator and simulated 200 node networks with 50ms inter-node latency. It would be possible to simulate and present results for networks with larger numbers of nodes, but a set of experiments on increasing numbers of nodes revealed that the behavior remains the same for networks as large as 5000 nodes. We opted for a smaller size but repeated each experiment 100 times to obtain better confidence in our results.

The target streaming rate in the experiments was fixed to 300 Kb/s, and packets of 10 Kb were used. Higher streaming rates yielded similar results as long as the packet size is accordingly increased to maintain a rate of 30 packets/s. Each streaming session lasted for 200 seconds. In the basic setting, the seed's upload capacity was fixed to twice the streaming rate while other peers had a fixed maximum upload capacity of 1.2 times the streaming rate. These values are used as our baseline since they are the lowest upload rates at the seed and non-seed nodes respectively that lead to good throughput when the system is not under attack.

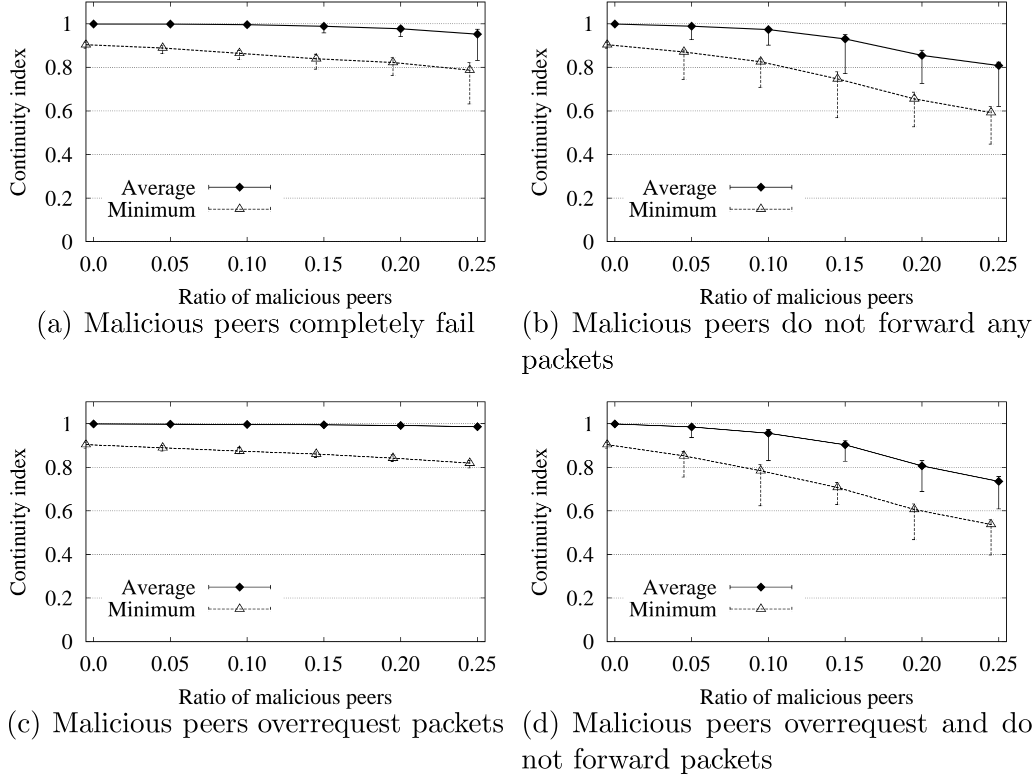
For each streaming session we computed the average and minimum download and upload rates across all correct members. We repeated each experiment 100 times, and we present the median and 95 percentile intervals across these repetitions.

We considered four types of malicious behavior. In the first type of attack malicious peers act as failed, neither requesting nor satisfying requests. In attack 2 they request packets but do not forward any packets. In attack 3 they overrequest packets from their neighbors, requesting as many distinct packets as possible from every neighbor. Finally, in attack 4 they overrequest packets and do not forward packets. The fourth type of attack is the most disruptive type and therefore the most likely, while the other three are considered mainly for comparison purposes.

Figure 6 presents results for the basic setting under each of the attack types. We are interested in minimizing the overall damage to the streaming session. Damage is quantified by the impact on average download rates to healthy nodes, and the minimum download rate for any single healthy node.

As would be expected, the results show that peer failure does not significantly affect the download rates since peers can still request packets from other correct neighbors (Figure 6(a)). Since malicious peers do not request packets in this mode, they do not disrupt the total overall upload capacity. Even though upload rates are limited, overrequesting attacks are also not significantly disruptive, due to the random policy used by peers when satisfying neighbors' requests for packets and the upper limit on the number of outstanding requests by any neighbor (Figure 6(c)).

Figures 6(b) and 6(d) show that attacks in which peers consume packets from their neighbors, but do not forward packets, inflict the most harm. There are two main reasons for this vulnerability. First, since peers upload at a maximum rate of 1.2 times the streaming rate, the overall upload capacity of the system gets compromised from peers consuming and not contributing to the system. Second, malicious nodes neighboring the seed might impede some packets from ever being received by any other peer other than itself.



(a) Malicious peers completely fail (b) Malicious peers do not forward any packets

(c) Malicious peers overrequest packets (d) Malicious peers overrequest and do not forward packets

Fig. 6. Resilience under different types of Byzantine behavior and varying ratios of attackers

The latter effect causes the 95 percentile interval bars to be wide: there is a lot of variation depending on the number of compromised peers near the seed. To make this point clear, in Figure 7 we show the percentage of packets received by increasing numbers of peers during sample streaming sessions with varying ratios of Byzantine peers. The metric to focus on here is the fraction of packets only received by one peer, which is an indicator of malicious nodes neighboring the seed. Packets received only by malicious peers at the first hop will never be disseminated in the system. To confirm this hypothesis, we executed the same set of experiments and restricted the malicious attackers to being located at least 2 hops away from the seed. The obtained medians were very close to the medians obtained in the previous experiments. The main difference was that the percentile intervals were significantly reduced when the seed had no immediate malicious neighbor, which is an important result since the intervals are significant in the original experiments with attacks 3 and 4.

To improve the resilience, we can vary parameters to improve the overall upload capacity of the system, or to avoid situations in which malicious peers can isolate certain packets. First, we considered the upload capacity of the members. In Figure 8(a) we varied the value from 1.0 to 2.0 times the streaming rate and verified the improvements to resilience under attack 4. This graph presents the average and minimum download rates when the system has 25%

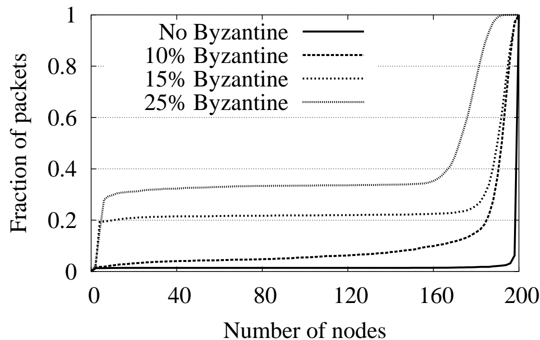


Fig. 7. CDF: Fraction of packets received by given number of nodes

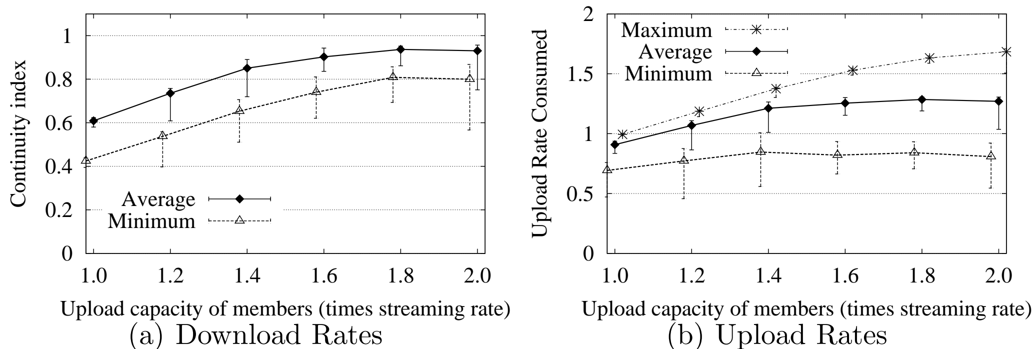
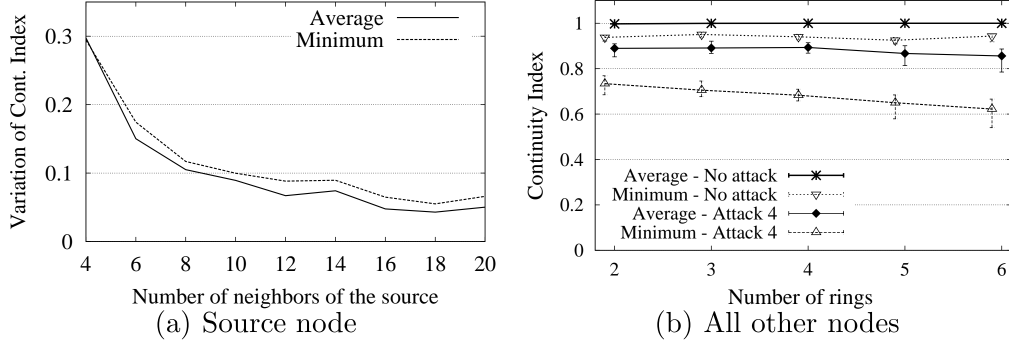


Fig. 8. Download and upload rates across nodes when maximum upload capacity is varied

of Byzantine members. The results show that the higher the upload capacity at non-seed peers the more resilient the system becomes. From Figure 8, which presents the minimum, average and maximum upload rates of members, we can see that as a consequence of increasing the upload capacity of peers the system becomes more unfair, with an increased difference between the maximum upload rate and minimum upload rate across peers. For the next few experiments we fixed the upload capacity of non-seed members to 1.4 times the streaming rate.

To improve the packet loss ratio at the first hop from the seed, we varied the upload capacity of the seed from 1.0 all the way to 6.0 times the streaming rate. Our results indicated that this naive approach to increasing the upload rate at the seed does not significantly affect the resilience of the system. We also observed that the number of neighbors of the seed is a more significant parameter than the upload capacity of the seed. We fixed the ratio of malicious nodes at 25%, the upload rate at non-seed nodes to 1.4 times the streaming rate and at the seed to 4.0 times the streaming rate, and varied the seed's number of neighbors from 4 to 20. The median slightly improves as the number of neighbors is increased, but more important, the percentile intervals are significantly reduced. In Figure 9(a) we present the absolute sizes of the 95 percentile intervals varying with the number of neighbors of the seed. The results show that a larger number of neighbors at the seed is desirable. This



(a) Source node (b) All other nodes

Fig. 9. Sensitivity to number of neighbors

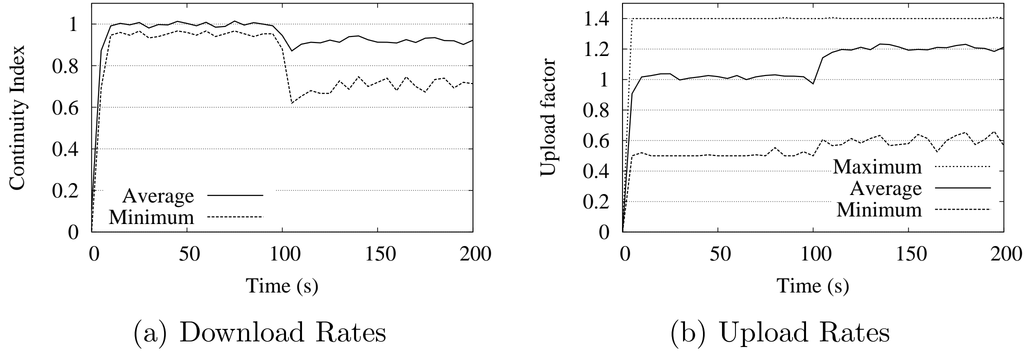
happens because with a higher number of neighbors the percentage of malicious neighbors of the seed tends to be closer to 25% across runs, and therefore there is less variation in the ratio of packets that are contained at the first hop from the seed.

Finally, to study the influence of the number of neighbors for each non-seed peer in the system, we evaluated the resilience with a varying number of rings used to define neighbors. The upload capacities at the seed and non-seed members were fixed to 4.0 and 1.4 times the streaming rate, respectively, and the seed had 16 neighbors. In Figure 9 we present the performance of the system using between 4 and 12 neighbors per node, both under no attacks and under attacks of type 4. The results surprisingly show that the use of larger numbers of neighbors does not improve resilience of the system, and even reduces when the system is under attack. Even though larger numbers of neighbors would lead to better connectivity between correct members, it also presents malicious members with more potential to overrequest packets and unbalance the system.

4.2 Emulab Testbed

In order to validate our simulation results, we ran experiments on a 200 node LAN on the Emulab testbed using our Python implementation of the Secure-Stream system. We performed extensive experiments under various parameter configurations, observing that the tendencies observed were similar to those verified through simulation.

To illustrate the behavior of the real system in execution, we present results of a sample streaming session in which 25% of the nodes are malicious, overrequesting packets and not forwarding them to neighbors. During the first 100 seconds all nodes act correctly, after which the malicious nodes start overrequesting and not forwarding packets. We fixed the upload capacity of the seed and non-seed members to 4.0 and 1.4 times the streaming rate respectively,



(a) Download Rates

(b) Upload Rates

Fig. 10. Sample streaming session on Emulab

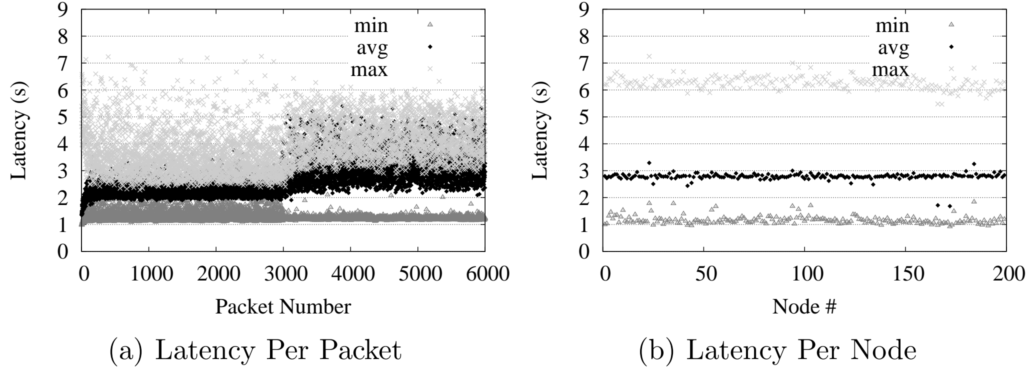
and the number of neighbors of the seed and non-seed members to 12 and 8 respectively.

Figure 10(a) presents the minimum and average continuity indices of correct peers throughout the sample session. Around the hundredth second, the average and minimum continuity indices decrease with the insertion of malicious peers. The minimum, average and maximum upload factors across all correct peers is presented in Figure 10(b). At the point when malicious nodes are inserted, the upload factors across correct peers increases to compensate for the malicious peers consuming the scarce resources from the system.

We also observed the effect of the system in the latency of packets. In Figure 11(a), a slight increase in the overall average and maximum delays per packet in the presence of attackers may be observed. Furthermore, an interesting behavior can be observed in Figure 11(b), which presents the minimum, average and maximum packet delays for each node in the system, relative to the time of origin of the packet at the source.

Unlike our initial suspicion, all nodes presented similar packet delays over the streaming session. This indicates that being close to the source does not imply in receiving packets faster than other nodes, since not all packets will be requested from the source, because of the limit in number of outstanding requests. To verify if this behavior might vary when the system scales to larger numbers of nodes, we simulated networks with up to five thousand nodes. The maximum latency used for bigger networks needs to be increased, but the average latency per node is still similar.

We also looked into the number of hops taken by packets before reaching all nodes. For the same streaming session, we registered the number of hops taken by each packet before reaching each node. In Figure 12 we present the CDF of the average and maximum number of hops taken by packets. This graph shows that for our sample session with 200 nodes, the average number of hops mostly varies between 3 and 5, while the maximum number of hops taken by each packet varies between 8 and 22. This large variation in the maximum



(a) Latency Per Packet (b) Latency Per Node

Fig. 11. Latency of packets on Emulab

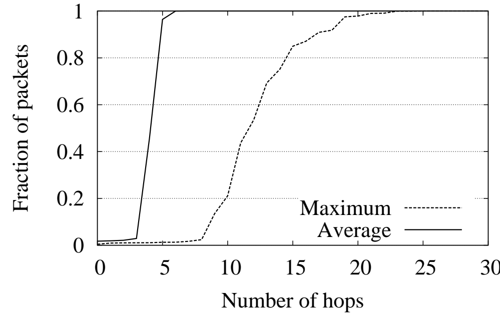


Fig. 12. CDF of average and maximum number of hops taken by packets

number of hops is also verified by the large variation in maximum latency observed in Figure 11(a).

4.3 Auditing

We also explored the potential of employing auditing in our simulations. Auditing ensures that nodes contribute more than a particular threshold factor t of upload capacity. A threshold of 0.5 during a 300 Kb/s streaming session, for instance, implies that nodes uploading less than 150 Kb/s will be removed from the system. To evaluate the effect of applying such thresholds both in the absence and presence of freeloading nodes (nodes that voluntarily contribute less than what is expected from them), we simulated audited sessions with 1000 nodes.

Figure 13 presents a detailed set of results on applying different thresholds to different freeloading profiles. The ratio of freeloading nodes was fixed to 30%, and their contribution factor (ratio relative to the streaming rate) is varied between 0, 0.25, 0.50, and 0.75 (0 meaning they do not contribute at all). We also consider a final profile named *mix*, where freeloading nodes have different contribution factors, uniformly distributed among 0, 0.25, 0.50 and 0.75.

Columns are clustered based on the threshold used to punish nodes (t). Within

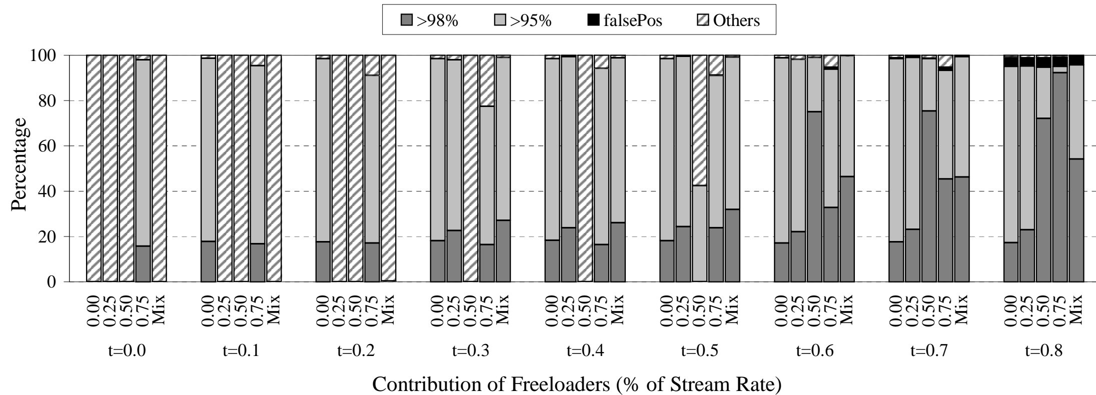


Fig. 13. Quality of streaming, measured by the percentage of correct nodes that receive over 95% and 98% packets, and percentage of nodes unfairly punished by the auditing system (falsePos). All experiments contain 30% freeloading nodes. Each cluster corresponds to a different threshold t being applied by the auditor. Within each cluster we considered 5 rates of contribution by freeloaders: 0, 0.25, 0.5, 0.75, and a mix of these values. Correct nodes contribute at a rate of 1.1.

each column we present the percentage of correct nodes that have an average upload factor of more than 98%, between 95% and 98%, false positives and others. False positives are correct nodes who get incorrectly punished by the auditing system. Notice that a threshold of 0 is equivalent to a system without auditing.

To help understand the graph, let us consider, for instance, the set of bars when $t = 0.4$. A threshold of 0.4 will detect and remove all freeloaders with an upload factor of 0 or 0.25. This may be confirmed in the first two bars, which indicate that the streaming quality is good, with almost all nodes receiving more than 95% of the data. The third bar, in which freeloaders have an upload factor of 0.5 presents unsatisfactory results, with no node receiving over 95% of the data. This was expected, since a threshold of 0.4 is not able to detect freeloaders that contribute with a factor of 0.5.

In the fourth bar, even though freeloaders do not get detected, they also do not disrupt the system significantly, since they contribute at a factor of 0.75, which is close to the ideal factor of 1.0. This is confirmed by the fact that even when there is no auditing (threshold is 0), the quality of the stream is satisfactory . The same observation holds for the *mix* configuration.

Two metrics are important when deciding the right threshold to apply: the quality of the streaming, captured by the percentage of nodes receiving over 95% of data; and the ratio of false accusations, which should be ideally null. From Figure 13 it is reasonable to assume that opting for a threshold such as $t = 0.6$ is the best approach, since it provides satisfactory streaming quality under the 5 different configurations. However, the goal of minimizing the ratio of false positives motivates the use of a dynamic threshold value, adjusted

by global auditors based on sampling the current stream quality and upload factors of nodes across the system. One possibility consists in maintaining a null threshold (t = 0) while the actual download rates across the network are satisfactory, that is, not punishing nodes unless the performance of the session is compromised. As the system starts to degrade, global auditors may slowly increase the threshold until the performance improves again.

5 Related Work

Recent work on peer-to-peer streaming systems has focused on improving fairness among peers and resilience to churn, and have not addressed behavior in the presence of malicious peers. Splitstream [4] breaks the data into stripes and disseminates each stripe through a different dissemination tree. Ideally, each peer is an internal node in only one these trees, and therefore the system as a whole is fair. Figures 1(b) and 1(d) present SplitStream's resilience to omission attacks. Bullet [5] is another protocol which attempts to improve fairness by breaking the stream into packets and sending them to peers through different dissemination paths. Packets are pushed down a tree to certain peers and then exchanged between peers through random connections.

The pull-based style of streaming used in our system was previously used in CoolStreaming [1] and Chainsaw [6]. Coolstreaming breaks the data into packets and peers organized into a mesh request packets from their neighbors using a scheduling algorithm to identify the best sources of packets. Chainsaw uses a simpler policy for requesting packets from neighbors, randomly fetching packets from neighbors with available packets respecting only a limit on the number of outstanding requests. Chainsaw presents smaller delays for the receipt of packets compared to the Coolstreaming protocol.

Omission attacks are often characterized as rational behavior and there has been a lot of work regarding incentives for peer-to-peer systems. Most work on incentives has focused in file sharing systems such as Bittorrent [19], which present significantly different properties, and cannot be directly transferred to streaming protocols. Some more recent work has focused on rational behavior in live-streaming systems.

Ngan et al. [20] consider fairness issues in the context of tree-based peer-to-peer streaming protocols. The authors present mechanisms that can distinguish peers according to their level of cooperation to the system. One of their techniques involves the reconstruction of trees as a way of punishing freeloading nodes. Most of their mechanisms require peers to keep track of their parents' and children's behavior.

PULSE [21] is a P2P live-streaming system that tries to reward nodes that contribute resources and discourage peers from contributing an insufficient amount of resources. The main idea consists in using a pull-based dissemination protocol and moving nodes that contribute more closer to the source, therefore having a smaller lag for packets received. The system makes a few assumptions which could be compromised by malicious nodes present in the system, such as requiring that nodes have some knowledge of other nodes in the system. Also, the system is only evaluated in heterogeneous settings, showing that nodes with higher upload capacity have a smaller latency compared to less favored nodes. Results in a homogeneous setting are not presented.

In [22], the use of incentives is explored as a way of avoiding the presence of selfish nodes in the Chainsaw protocol [6]. This work argues why some naive approaches to enforcing incentives do not work, similar to the analysis presented in our work, and propose and evaluate the use of a technique that relies on preferential uploading to neighbors. Nodes that contribute more are more prone to receiving data back, but in a not so fixed manner as a tit-for-tat approach. Only preliminary results are presented, and malicious behavior is not the focus of the work.

Even though incentives encourage nodes to contribute and avoid nodes from acting selfishly, they do not extend the effect to nodes who are in the system with malicious intentions.

Drum[23] targets DoS attacks on gossip-based multicast protocols, eliminating vulnerabilities to such attacks. The main idea in Drum is to have half of the links of each peer be picked by the peer itself, and half be picked by other peers. That way, even if only malicious peers connect to a peer, the peer can still get correct data from the peers that it picks. The authors showed that the approach works well for multicast protocols which do not have time delays, but have not studied its performance for multicast systems where a high throughput of packets is desired and the upload capacities are limited.

BAR Gossip [24] is a live-streaming approach that tolerates the existence of selfish and malicious nodes. Time is divided into rounds, in which each peer communicates with another peer selected using a pseudo-random function. In each round, peers exchange their current history containing the identifiers of all the current data, as basis for the next exchanges. Nodes also perform a phase of *optimistic push*, forwarding useful updates to another pseudo-randomly picked peer with no guarantee of useful return. The approach requires that the broadcasting seed has full knowledge of all members in the system and always unicasts each update to 5% of the nodes, a limitation on scalability.

6 Conclusions

We presented the design and evaluation of SecureStream, a P2P live-streaming protocol tailored to handle byzantine attacks. We described the main components of SecureStream and the main techniques employed to resist against DoS, forgery, membership and omission attacks. Furthermore, we considered the benefits of employing an auditing system to avoid the damage incurred by freeloading behavior of nodes. We evaluated our system through simulation and emulation. Our results indicate that SecureStream tolerates a limited percentage of malicious nodes in the system, and that with the aid of an auditing component, it is able to provide satisfactory quality in the face of even larger attacks.

References

[1] X. Zhang, J. Liu, B. Li, T.-S. P. Yum, CoolStreaming/DONet: A Data-Driven Overlay Network for Efficient Live Media Streaming, in: Proceedings of the 2005 Conference on Computer Communications, Miami, FL, 2005.

[2] Y.-H. Chu, S. G. Rao, H. Zhang, A Case for End System Multicast, in: Proceedings of ACM Sigmetrics, Santa Clara, CA, 2000.

[3] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, J. O. Jr., Overcast: Reliable Multicasting with an Overlay Network, in: Proceedings of the 4th Symposium on Operating Systems Design and Implementation, San Diego, CA, 2000.

[4] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, A. Singh, SplitStream: High-Bandwidth Multicast in Cooperative Environments, in: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, Bolton Landing, NY, 2003.

[5] D. Kostić, A. Rodriguez, J. Albrecht, A. Vahdat, Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh, in: Proceedings of the 19th Symposium on Operating Systems Principles, Bolton Landing, NY, 2003.

[6] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, A. E. Mohr, Chainsaw: Eliminating Trees from Overlay Multicast, in: Proceedings of the 4th International Workshop on Peer-to-Peer Systems, Ithaca, NY, 2005.

[7] V. Venkataraman, P. Francis, J. Calandrino, Chunkyspread: Multitree Unstructured Peer to Peer Multicast, in: Proceedings of the 5th International Workshop on Peer-to-Peer Systems, Santa Barbara, CA, 2006.

[8] R. van Renesse, H. Johansen, A. Allavena, Fireflies: Scalable Support for Intrusion-Tolerant Network Overlays, in: Proceedings of the 1st ACM EuroSys, Leuven, Belgium, 2006.

[9] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, B. Pinkas, Multicast Security: A Taxonomy and Some Efficient Constructions, in: Proceedings of IEEE INFOCOMM, Orlando, FL, 1999.

[10] A. Singh, M. Castro, A. Rowstron, P. Druschel, Defending against Eclipse Attacks on Overlay Networks, in: Proceedings of the 11th ACM SIGOPS European Workshop, Leuven, Belgium, 2004.

[11] C. K. Wong, S. S. Lam, Digital Signatures for Flows and Multicasts, IEEE/ACM Transactions on Networking IEEE Communications Society 7.

[12] R. Gennaro, P. Rohatgi, How to sign digital streams, in: Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology, London, UK, 1997.

[13] S. Miner, J. Staddon, Graph-Based Authentication of Digital Streams, in: Proceedings of the 2001 IEEE Symposium on Security and Privacy, Washington, DC, 2001.

[14] D. Song, J. D. Tygar, D. Zuckerman, Expander Graphs for Digital Stream Authentication and Robust Overlay Networks, in: Proceedings of the 2002 IEEE Symposium on Security and Privacy, Washington, DC, 2002.

[15] A. Perrig, R. Canetti, D. Tygar, D. Song, The TESLA Broadcast Authentication Protocol, Cryptobytes 5 (2).

[16] A. Perrig, R. Canetti, D. X. Song, J. D. Tygar, Efficient and Secure Source Authentication for Multicast, in: Proceedings of the Network and Distributed System Security Symposium, The Internet Society, San Diego, CA, 2001.

[17] A. Perrig, R. Canetti, J. D. Tygar, D. X. Song, Efficient Authentication and Signing of Multicast Streams over Lossy Channels, in: IEEE Symposium on Security and Privacy, Berkeley, CA, 2000.

[18] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, A. Joglekar, An Integrated Experimental Environment for Distributed Systems and Networks, in: Proceedings of the Fifth Symposium on Operating Systems Design and Implementation, Boston, MA, 2002.

[19] B. Cohen, Incentives Build Robustness in BitTorrent, in: 1st Workshop on the Economics of Peer-to-Peer Systems, Berkeley, CA, 2003.

[20] T.-W. J. Ngan, D. S. Wallach, P. Druschel, Incentives-Compatible Peer-to-Peer Multicast, in: Second Workshop on the Economics of Peer-to-Peer Computing, Cambridge, MA, 2004.

[21] F. Pianese, J. Keller, E. W. Biersack, PULSE, a Flexible P2P Live Streaming System, in: Proceedings of the Ninth IEEE Global Internet Workshop, Barcelona, Spain, 2006.

[22] V. Pai, A. E. Mohr, Improving Robustness of Peer-to-Peer Streaming with Incentives, in: Proceedings of the 1st Workshop on the Economics of Networked Systems, Ann Arbor, MI, 2006.

[23] G. Badishi, I. Keidar, A. Sasson, Exposing and Eliminating Vulnerabilities to Denial of Service Attacks in Secure Gossip-Based Multicast, in: International Conference on Dependable Systems and Networks, Philadelphia, PA, 2004.

[24] H. C. Li, A. Clement, E. L. Wong, J. Napper, I. Roy, L. Alvisi, M. Dahlin, BAR Gossip, in: Proc. of the 7th Symposium on Operating Systems Design and Implementation (OSDI), Seattle, WA, 2006.