Secure, Customizable, Many-to-One Communication

Kenneth L. Calvert, James Griffioen, Billy Mullins, Leon Poutievski, Amit Sehgal*

Laboratory for Advanced Networking, University of Kentucky, Lexington, KY

Abstract. Concast is a customizable many-to-one network-layer communication service. Although programmable services like concast can improve the efficiency of group applications, accompanying security concerns must be addressed before such services are likely to be deployed. The problem of securing such services is interesting because conventional end-to-end security mechanisms are not applicable when messages are processed inside the network, and also because of the potential for interaction among the various policies involved. In this paper we describe our implementation of a secure concast service, which leverages existing network-level security mechanisms (IPsec) to provide secure distribution of program code (merge specifications) as well as authentication of participating nodes. We describe the various policies supported, how they interact, and how our approach provides security against various attacks.

1 Introduction

The design of the Internet protocols has produced a remarkably flexible, robust, and scalable system. Perhaps nowhere is the end-to-end design principle more evident than in the area of security, where the best services and solutions are universally considered to be those that are closest to the application. Over time, however, a number of network services have appeared that involve, in one way or another, processing that occurs in the shared infrastructure, *away* from the end systems on which the applications reside. Many of these services depend on the ability to look into the packet beyond the information needed for traditional forwarding (i.e. the packet header), into the packet payload. In some cases, this processing is performed on the application's behalf *during* forwarding [1–6].

The problem of securing applications that rely on this type of processing is interesting because the conventional end-to-end security solutions preclude—and indeed, are intended to prevent—inspection and modification that occurs apart from the endpoints, and thus are incompatible with such applications. In addition, reliance on the infrastructure to perform processing on behalf of the application implies the existence of multiple policies that need to be enforced.

The *concast* service is a good example of a service that performs processing on the applications behalf during forwarding. Concast is a many-to-one communication

* Work supported in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF under agreement number F30602-99-1-0514, and by the National Science Foundation under grants EIA-0101242 and ANI-9977292.

service that can be viewed as a companion service to multicast (i.e., the inverse of one-to-many communication). In concast, multiple senders transmit data packets toward a single receiver; the eventual result is that a single packet, containing the combined (merged) data from the multiple senders, is delivered to the receiver. Because the nature of the merging operation varies with the application, concast allows end systems to define the merge processing that is applied at internal network nodes. The benefit of concast is in reducing the limiting factor on the scalability: from the total number of senders to the "branching factor" at any node in the "tree" formed by the paths packets follow from senders to receiver.

In this paper we consider the problem of securing scalable infrastructure-based services, in particular the problem of creating a secure concast service. We outline a general set of security requirements for such services, and identify the relevant policies and trust relationships involved. We then describe a new security approach based on the idea that the control plane can be secured using conventional point-to-point security techniques for authentication, confidentiality, and integrity. Given a secure control plane, the responsibility for end-to-end security can then be distributed among the participating nodes. We describe the application of our approach to implementing a secure concast service. We report performance measurements taken from our prototype implementation of the secure concast service.

2 Security Requirements

We assume a network environment in which network services are offered to users as a business proposition by *service providers*. We believe that a customizable service will only be deployed if it offers some benefit to the service provider. We assume this benefit takes the form of money paid to the provider in return for access to the enhanced service. Thus our first security requirement is:

> *Only authorized users can take advantage of the customizable service.*

We assume that users will pay for a service only if they are assured of receiving some benefit from it. In the case of concast, the main benefit to the user is *scalability through anonymity*: by moving application-specific processing into the network infrastructure, the service hides the details of where the data is coming from and how much processing is occurring. To put it another way: placing application-specific processing in the infrastructure hides scale and complexity from the users. This leads to an additional requirement:

> *The scale and complexity of the processing should not be exposed at any single point.*

As a consequence, the user must rely on the network to carry out processing according to user-supplied specifications. On one level, this is no different than any other network service. However, in terms of security there is a profound difference between relying on the network to *forward* data, as opposed to examining and possibly *modifying* it. In the former case, end-to-end security mechanisms exist that can provide assurance that (under standard assumptions) user data is not disclosed or tampered with. In the

latter case, the users not only have to trust the network to carry out the specified processing, but also to protect the confidentiality and integrity of the application's data. That is, the user/application has to rely on the network infrastructure to enforce its *security policies*. This brings us to the third security requirement:

> *Integrity and confidentiality of application data are protected according to user-supplied policies.*

In other words, each instance of the service has a user-supplied policy that specifies which entities are authorized to participate in that instance.

This requirement is nontrivial for two reasons. First, because the infrastructure is a key participant in the enhanced service, the application policy needs to cover not only users, but also components of the infrastructure (nodes). In other words, each participant must be able to identify nodes that are *not* trusted to carry out processing on its behalf, and the system must take steps to prevent such nodes from participating in providing service to that user. Second, and more importantly, the service is designed so that the set of participating nodes grows incrementally, hop-by-hop toward participating users. Participants are *only* aware of other participants (either users or infrastructure nodes) that are up to one hop away; this is a fundamental characteristic that is required for scalability and indeed, for practical deployment. As a consequence, users cannot themselves ensure that only trusted nodes participate in the service; they must rely on the infrastructure to enforce their policies on participation.

Our approach to satisfying the last two requirements is to state an invariant that is to be maintained at all times by the service:

> *All participating nodes are trusted by the user to enforce user policies regarding (i) processing, confidentiality and integrity of user data; and (ii) which nodes are trusted to participate.*

This can be viewed as (an explicit form of) *transitive trust*. Transitivity of trust relationships in some form seems to be an unavoidable requirement for *scalable* services that rely on third parties for key functionality.

3 Securing a Programmable Service

The first step in securing a programmable service is establishing trust relationships between the participating entities (senders, receivers, and network nodes).

Trust relationships can be represented as the set of principals (nodes) that are allowed to perform certain actions (e.g., join the concast group, receive the merge specification, or be given an encryption key). We say a *policy* defines the set of nodes that can perform a certain action. For example, a concast receiver will define the list of sender nodes that are allowed to join the group (called the *join policy*). At the same time, each local node in the provider's network will define the list of end systems that are allowed to use the concast service (e.g., have paid for the service). Clearly, both policies must be met before a sender is allowed to join a concast group.

The most important policy is the one that defines the nodes that can be trusted to enforce the policies of others. We believe this This type of transitive trust is critical for

network-level services where processing occurs hop-by-hop. Because the user's data does not remain encrypted end-to-end, intermediate nodes that handle the user's data must enforce the user's policies on the user's behalf. If a node cannot be trusted to enforce the user's policies, that node cannot be allowed to participate in the service. For example, a concast receiver must rely on routers in the network to enforce the receiver's join policy. If unauthorized senders were allowed to send data along the concast flow and the membership check did not occur until the merged packet reached the receiver, it would be too late. The damage (corruption of authorized sender data) would already have occurred at intermediate nodes in the network.

The key to achieving a scalable yet secure service is the ability to incrementally add nodes to the service such that the invariant is not violated, and to incorporate each added node's policy into the session policy. To initiate a secure service, the user's policies must be propagated, hop-by-hop through the network, checking the validity of each node along the path before adding it to the flow.

Note that this description assumes that policies are themselves propagated securely. At each hop along the propagation path, the adjacent nodes must authenticate one another and verify policy compliance before proceeding. Once authenticity and authorization have been established, the policies can be sent over a confidential channel. Because the trust relationships are established hop-by-hop, existing point-to-point security techniques can be used. In particular, protocols such as IPsec can be used both to perform the authentication check and to create the encrypted tunnel over which policies can be sent.

Once a path of trusted network hops has been established, that path can be used for control plane messages. In particular, control messages that enable service-specific processing at each trusted node can be sent along the path. Given a secure (programmable) control plane, end systems can then control security in the data plane, by installing modules that offer as much or as little security as desired. In other words, by supporting a secure, authenticated, hop-by-hop signaling protocol in the control plane, applications can implement end-to-end security in the data plane, thereby maintaining the end-to-end principle.

In the next section, we present a specific approach for implementing a secure control plane, and show how it can be applied to the concast service. The approach is novel in the sense that it leverages existing point-to-point secure communication protocols (i.e., IPsec) to create a secure path and distribute policies and user-specified processing modules. Given this basic infrastructure, end systems then define and control security in the data plane by programming the service appropriately.

3.1 The Concast Service

Before we describe how a secure concast service can be implemented using our approach, we need to take a moment and briefly review the basic (non-secure) concast service. Additional details of the concast service can be found in our earlier papers [7] and [1].

Concast is a many-to-one communication service that provides the symmetric inverse of multicast: a group of senders belonging to a *concast flow* transmit messages

that are *merged* by the network en route to a common receiver R. Like multicast, concast provides a scalable abstraction: an arbitrary number of *group member*s (senders) are treated as a single entity by R. A concast flow is identified by its receiver R and a group identifier G; senders "join" the flow before they begin sending.

The packets delivered to R on a concast flow are derived from the packets sent by the group members according to a *merge specification (MS)* supplied by the receiving application. The concast service allows a limited amount of network programmability, where the desired processing semantics are defined within the framework of a merge specification. The merge specification defines (1) how datagrams delivered to the receiver are derived from datagrams transmitted by different senders (2) the timing of datagram forwarding and delivery; and (3) which datagrams are combined with each other (e.g. only packets containing the same sequence number are merged with each other). The merge specification is supplied by the receiver at flow creation time (e.g. in the form of bytecodes for a collection of Java classes conforming to a certain type specification), and is executed by a merge daemon (*Merged*) at each network node.

Concast *merge specification* deployment is accomplished via the *Concast Signaling Protocol (CSP)*, implemented using a receiver-side CSP daemon (*RCSPd*) and a server-side CSP daemon (*SCSPd*). The CSP protocol creates the flow and establishes concast-related state, called the *flow state block (FSB)*, in network nodes (i.e. at all concast-capable nodes on the paths from group members to the receiver.) The *flow state block* records the *merge specification* describing how packets are to be merged, and an *upstream neighbor list (UNL)* that records the next concast-capable nodes "upstream" (towards the senders) for this flow. The UNL is maintained using soft-state techniques similar to RSVP [8].

Figure 1 shows the secure version of the CSP protocol, but the basic idea is the same as the original CSP protocol. First, the receiver initiates the flows (step 0,1). The senders then attempt to join the flow by *Join Flow Requests (JFR)* messages toward the receiver which are intercepted at intermediate nodes and directed to local CSP daemons. and propagated toward the receiver as *Request for Merge Spec (RMS)* messages (steps 2-8). The merge specification is then "pulled" from the receiver towards the senders (steps 9-18).

3.2 Securing Concast

Because the receiver is responsible for initiating the concast flow, the receiver should also be responsible for defining the flow's membership (i.e., join) policy. As we saw earlier, the policy must propagate through the network toward the senders so that routers can decide whether a sender is allowed to join or not. To maintain scalability, the concast receiver is not required to know (in fact never learns) the identity, or the location, of the senders. Obviously the join policy cannot be pushed into the network toward the senders until the location of the senders is known.

Because senders must identify themselves before the policies can be sent out, the secure version of the CSP protocol begins just like the original CSP protocol (see Figure 1). A new sender issues a join request message that propagates (in the clear) to the receiver (steps 2-8). At this point the path from the sender to the receiver is known and

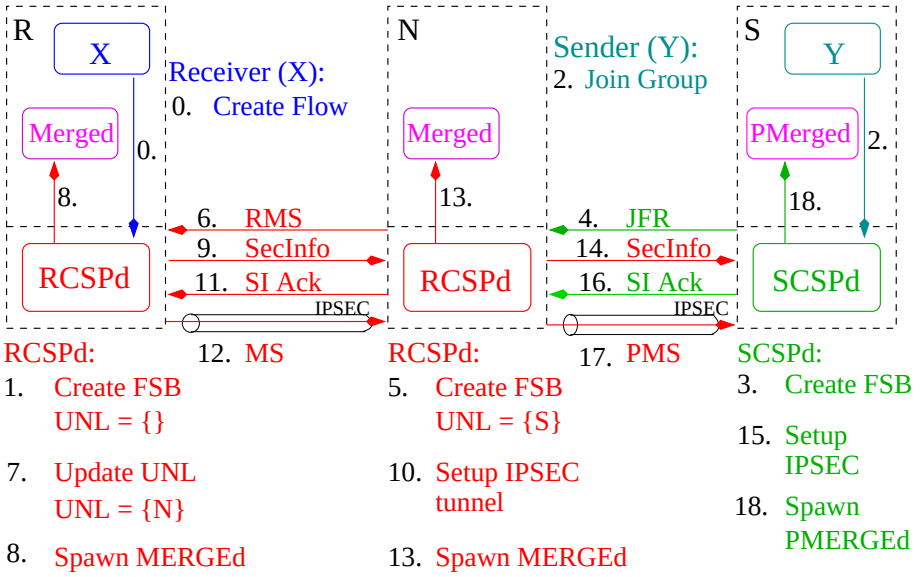

Fig. 1. The Secure Concast Signaling Protocol.

the user's join policy can be "pulled" toward the sender. This is accomplished by creating a set of secure tunnels back to the sender (steps 9-18). The secure path is created hop-by-hop, each time authenticating the next hop, verifying its admissibility according to the policy, and then passing the merge specification (including the relevant policies) across the secure tunnel (e.g., steps 9-12).

Because the merge specification is sent across a secure control channel and executes on trusted nodes, the responsibility for end-to-end data path security can be placed in the hands of the end systems. To achieve this objective, the concast merge specification itself must contain the code that controls decrypting, processing, and then re-encrypting the data packet before forwarding it on. Because the control channel is secure, the decryption and encryption key can be distributed along with the merge specification.

3.3 Merge Framework Modifications

In addition to securing the CSP protocol (i.e., securing the control plane), changes were also needed in the merging framework, in the form of "hooks" to support user-defined encryption/decryption in the data plane.

First, we enhanced the merge specification type/message to include a user-defined encryption function and decryption function as well as the secret keys to be used for encryption, decryption and authentication. These may be actually byte codes, or they may be pointers to predefined encryption and decryption functions we added into the merge framework (MergeD). As part of the encryption specification, the framework allows the user to specify whether a MAC (message authentication code) should be

include in the encrypted message. If so, the MAC will be checked when the packet is decrypted to verify its integrity.

The second change to the framework creates different forms of the merge daemon (MergeD) to be deployed at senders, merging nodes, and the receiver. Merge daemons executing on sender nodes receive packets over a local socket. Because these incoming packets are unencrypted (the local environment is trusted), the decryption function does not need to be invoked; only encryption is performed on outgoing packets. On receiver nodes the situation is reversed: incoming packets need to be decrypted, but outgoing packets go straight to the receiver application and do not need to be encrypted. On intermediate nodes, all incoming packets are decrypted and all outoing packets are encrypted (as long as merging is occurring, i.e. there is more than one upstream neighbor—otherwise, the packets are simply forwarded). Because we trust sender nodes only to transmit data, not merge packets, the signalling protocol transfers only a *partial* merge specification to the sender, containing an encryption function and the secret key (that is, neither the merge routine nor the decryption key is passed).

4 Secure Concast Signaling Protocol

This section describes the Secure Concast Signaling Protocol, which is based on the original Concast Signaling Protocol [1]. Together with IPsec, Secure CSP provides a foundation for the secure concast service. We begin by defining notational conventions, data types, and cryptographic primitives used. Next we describe the protocol messages and their contents. Finally, we give a high-level operational description of the (normal) process of setting up a concast flow.

4.1 Basic Types and Cryptographic Primitives

Our protocol uses the following types:

- **appident**: Identifier of an application-level principal, i.e. a participant in the concast flow (receiver or sender). E.g., if X.509 certificates are used, this could be an OSI Relative Distinguished Name (RDN).
- **nodeident**: Identifier of a network-level principal, i.e. a node. We use IP addresses as network identifiers.
- **flowspec**: A pair (R, G) identifying a concast flow, where R is the receiver's IP address (a **nodeident**) and G is the group identifier.
- **mergespec**: A collection of data and function definitions that defines the merge processing to be carried out by intermediate nodes, and that conforms to the requirements of the concast merging framework.
- **pmergespec**: A partial or "thinned" **mergespec**, containing only the security-related portions of the merge specification. End systems receive partial mergespecs because they need to do security-related processing but may not be trusted to apply policies or perform merging.
- **policy**: A specification of a set of principals that are authorized in some way. We consider a policy to be a predicate on identifiers (**appident**s or **nodeident**s) and

credentials; if the predicate has the value "true" for a given identifier and credential, it means that (i) the identified principal is authorized, and (ii) the given credential is an acceptable witness for evaluating authenticity of information to be provided by the principal.
- **signature**: A digital signature, essentially a cryptographic digest of message data encrypted with some principal's private key, computed and formatted according to accepted cryptographic standards (e.g. SHA-1 [9] and PKCS #1 [10]). The notation $\{h(a|b|c)\}_k$ denotes the result of concatenating messages or fields a, b and c and signing the digest (created using a well-known cryptographic algorithm such as SHA-1) of the resulting bit string with private key k. Unless otherwise specified, **signature** fields in messages cover the entire contents of the message preceding the field.
- **cert**: A public-key certificate, which binds an identifier (of type **appident** or **nodeident**) to a public key.
- **ipsecinfo**: A structure containing IPsec information of a host needed by another host to create an IPsec tunnel to the former host.
- **timestamp**: A timestamp.
- **ccasthdr**: the first field of every secure CSP message. Indicates the version of the protocol and the type of the message.

The notation *verify*(m, a, c) denotes the result of verifying the authenticity and integrity of (some part of) a message m using signature a and certificate c. This function returns true if digesting the information in m results in a value consistent with that obtained by decrypting a with the public key contained in c. For brevity, we sometimes abuse notation by indicating that the entire message m is being verified even though the authenticator covers only a portion of it.

The notation $p(u, c)$ denotes the result of applying policy p to identifier u with credential c. The value "true" means that u, presenting credential c, is authorized. The notation *time-check*(t) denotes the result of verifying that a timestamp t is within some δ of the current time as known locally. We assume that δ is configured appropriately at every node for the degree of clock synchronization achievable in the network. (As usual when timestamps are used to ensure freshness, if δ is too small the protocol may fail between nodes whose clocks are not well-synchronized; setting δ too large increases the window of vulnerablility to replay attack.)

4.2 Policies and Principals

As described earler, the signaling protocol makes use of various policies. Per-flow policies are supplied by the receiver, and specify the principals—nodes and applications—that are allowed to participate in the flow. Per-node policies are supplied by service providers (ISPs), and specify the nodes that are allowed to perform various functions in a flow. Per-node policies are only applied to **nodeident**s.

The supported policies include:

- *fp.j*: per-flow join policy. Specifies application entities (**appident**s) authorized to join the flow. This policy is specified by the concast receiver along with the merge specification.

– *fp.u*: per-flow upstream node policy. Specifies nodes (**nodeident**s) that are authorized to participate in the flow either as host of an application-level sender or as a merging node. This policy is specified by the concast receiver along with the merge specification.
– *np.r*: per-node receiver policy. Specifies nodes (**nodeident**s) that are authorized to be the terminal points of concast flows. This implies that the node is authorized to supply merge specifications. This policy would typically characterize nodes that either have had a fee paid on their behalf, or are part of some trusted nonlocal domain.
– *np.d*: per-node downstream policy. Specifies nodes (**nodeident**s) that are authorized to relay a merge specification from a downstream receiver.
– *np.s*: per-node sender policy. Specifies the set of nodes (**nodeident**s) authorized to be the source of requests to join a concast flow. Again, typically characterizes the set of nodes in this domain that have paid for service, and nodes trusted by virtue of the other domain to which they belong.
– *np.u*: per-node upstream policy. Specifies the set of nodes authorized to be upstream of this node in a flow. Note that such nodes are trusted not only to handle (merge) user data, but also to apply this node's policies.

The protocol description involves the following principals and their associated information: X is the receiver (application), which has private key k_X and certificate C_X; it is running on node R, which has private key k_R and certificate C_R. Y is a sender (application), which has private key k_Y and certificate C_Y. Y is running on node S, which has k_S and C_S. Finally, N is a merging node with private key k_N and certificate C_N.

4.3 Protocol Messages

Message contents are given in terms of the structured types shown in Figure 2, which in turn use the basic types defined above. Note that the CREATEREQ structure contains two signatures; the first covers the MERGETOKEN, while the second covers the same data except that **mergespec** is replaced by the subset of its information that constitutes a **pmergespec**. Also, the PCREATEREQ structure contains only the fields of a CREATEREQ that are relevant to the reduced mergespec, i.e. the subset of `mt` that constitutes a reduced mergespec, the `pMTSig`, and the `userCert`; given a valid CREATEREQ, a PCREATEREQ can be derived from it.

JOINREQ	
flowspec	`flowID`;
appident	`user`;
signature	`userSig`;
cert	`userCert`;

MERGETOKEN	
flowspec	`flowID`;
mergespec	`ms`;
policy	`PFUpstreamP`;
policy	`PFJoinP`;
appident	`user`;

CREATEREQ	
MERGETOKEN	`mt`;
signature	`MTSig`;
signature	`pMTSig`;
cert	`userCert`;

Fig. 2. Structures used in concast messages

The contents of the protocol messages are shown in Figure 3.

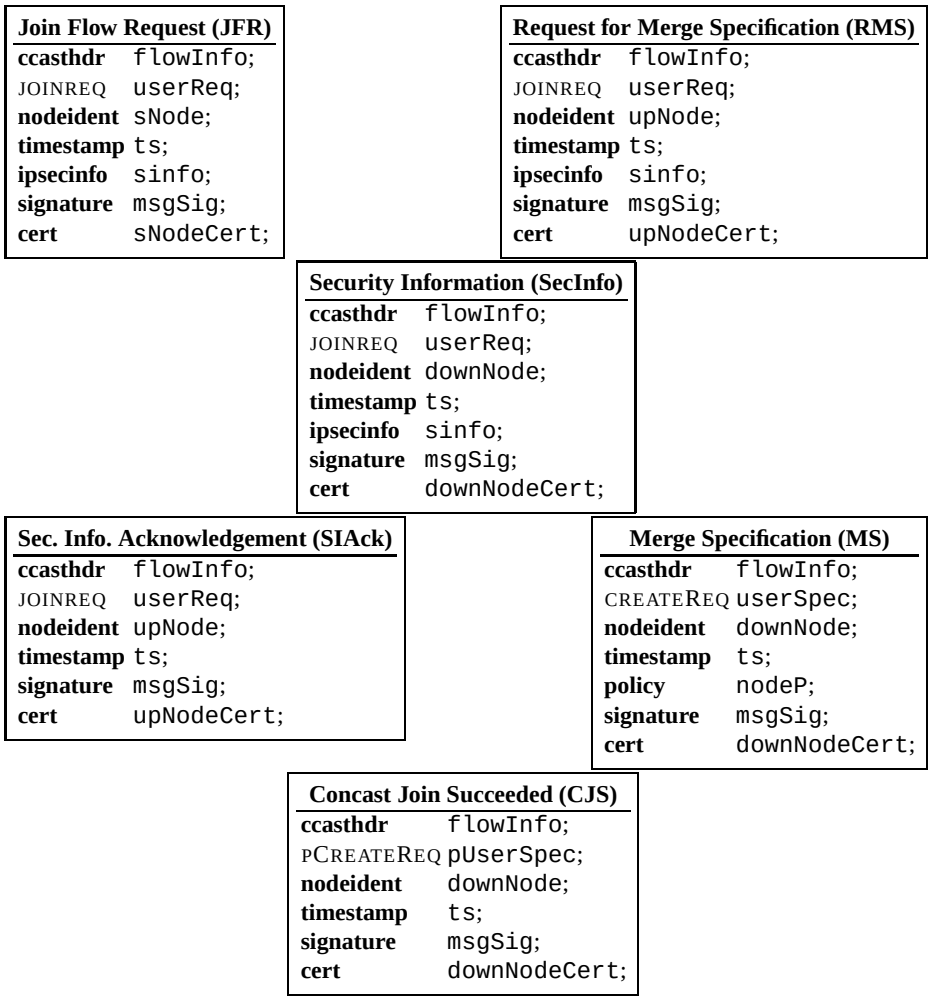

Fig. 3. Secure CSP Messages

4.4 Protocol Operation

With the help of Figure 1 we describe the normal sequence of steps for a secure concast flow establishment. In the interest of clarity we omit steps related to error processing, and assume that the flow in question is not currently present on any node involved.

Step 0: To create a flow (R, G), the receiver application X generates the secure merge specification (`ms`) and per-flow policies (`PFUpstreamP` and `PFJoinP`), formats

the requisite information as a MERGETOKEN, and generates a signature (`MTSig`) using its private key k_X. It also generates a signature (`pMTSig`) for the partial MERGETOKEN (the MERGETOKEN minus `ms`). X finally bundles the MERGETOKEN, the signatures `MTSig` and `pMTSig`, and its certificate `userCert`=C_X into a CREATEREQ and hands it over to the local CSP module.

Step 1: Upon receiving the CREATEREQ cr, the CSP at R verifies the signatures[1] cr.`MTSig` and cr.`pMTSig` using the public key in the certificate cr.`userCert`; that the principal of certificate cr.`userCert` matches identity cr.`user`; and that cr.`userCert` is a valid certificate generated by a trusted certificate authority. If the verification succeeds then the CSP creates the local flow state for the flow (R, G) and returns a success indication to X.

Step 2: To join the flow (R, G), the sending application Y creates a JOINREQ by including its identity `user`=Y, certificate `userCert`=C_Y, and a signature `userSig` generated by signing the request using its private key k_Y. The JOINREQ is then passed to the local CSP.

Step 3,4: Upon receiving JOINREQ jr from Y, the CSP at S verifies (i) the join request signature jr.`userSig` using the public key in certificate jr.`userCert`, (ii) that the principal of certificate jr.`userCert` matches application identifier jr.`user` and (iii) that jr.`userCert` is a valid certificate. The CSP at S next checks (i) if Y is allowed by local policy to act as a concast sender, and (ii) if R is an acceptable concast receiver node according to local policy, i.e. that *S:np.r*$(R, \perp)$ is true[2]. If so, a flow state block is created for the flow (R, G) and its state is marked "pending". A JFR message containing the user's join request `userReq`, the current timestamp `ts`, S's identifier `sNode`=S and certificate `sNodeCert`=C_S, IPsec information `sinfo` to connect to S and a signature `msgSig` obtained by signing the JFR message using k_S is generated and forwarded toward R.

Step 5,6: Upon intercepting a JFR message jm on its way to R, the CSP at N first verifies the signatures jm.`userReq.userSig` and jm.`msgSig` to ensure the authenticity and integrity of the user request and the JFR message respectively. It also checks the validity of the timestamp jm.`ts`. Next, the CSP verifies that *N:np.r*$(R, \perp)$, and *N:np.u*$(jm$.`sNode`$, m$.`sNodeCert`$)$ are all true. If so, it creates a temporary flow state block for the flow (R, G), adds the pair $(m$.`sNode`$, m$.`sNodeCert`$)$ to the upstream neighbor list, and marks the flow "pending". It also constructs an RMS message containing the user's join request `userReq`, a fresh timestamp `ts`, N's identifier `upNode`=$N$and certificate `upNodeCert`=C_N, IPsec information `sinfo` needed to connect to node N and a signature `msgSig` obtained by signing the RMS message using k_N. It forwards the resulting RMS message toward R.
(This process will be repeated at each concast-capable node along the path to R: the node intercepts the RMS message, validates the signatures, checks that the message source and R are acceptable to its local node policies, and then constructs and

[1] While the channel between the receiver application and the local CSP is probably trusted, this verification is a good idea because other nodes are going to perform it. If there is a problem, it is better to detect it locally. (Similarly for the JOINREQ passed by Y.)

[2] Note that this check should "tentatively succeed" at this stage without a certificate for R. The purpose is to prevent wasted effort in case R is unacceptable regardless of what credentials are presented. It will be repeated later with R's actual credentials when they are available

forwards toward R a signed RMS message containing the original JOINREQ and its own identifier and certificate. For brevity, we assume here that N is the last concast-capable node on the path toward R.)

Step 7,8: Upon receiving an RMS message rm, the CSP at R, the destination node, verifies the signatures rm.`userReq`.`userSig`and rm.`msgSig`. It also checks that *time-check*(rm.`ts`) ∧
fp.j(rm.`req`.`user`, rm.`req`.`userCert`) ∧
fp.u(rm.`sNode`, rm.`sNodeCert`) ∧
R*:np.u*(rm.`sNode`, rm.`sNodeCert`) are true, i.e. the flow policy admits the joining sender Y and both flow and node policies admit the upstream neighbor who sent the message. If so, then it spawns the Merge daemon for the flow; if not, it sends a signed error message upstream, indicating that the connection failed for policy reasons.

Step 9: Before the CSP can send the merge specification to the upstream node it must create an IPsec tunnel to the upstream node. To do this the CSP first sets up all necessary IPsec connection information using rm.`sinfo` at its own end. It then creates a SECINFO (Security Information) message that contains the `userReq`=rm.`req`, a fresh timestamp `ts`, R's identifier `downNode` and certificate `downNodeCert`, R's IPsec information `sinfo` and a signature obtined by signing SECINFO with k_R. It then sends the SECINFO message to the upstream node.

Step 10,11: Upon receiving the SECINFO message sm, the CSP at the upstream node N checks that the flow identifier sm.`userReq`.`flowID` refers to a legitimate pending flow, and verifies (i) the signatures sm.`userReq`.`userSig` and sm.`msgSig`, (ii) that certificate sm.`downNodeCert` is valid. Next the CSP checks if *time-check*(sm.`ts`) is true. It then applies its local downstream node policy, i.e. checks the truth of the predicate N*:np.d*(sm.`downNode`, sm.`downNodeCert`). If true, it sets up its local IPsec connection files using sm.`sinfo` and establishes a security association with sm.`downNode`. Upon successful creation of the IPsec tunnel the CSP creates a SIACK message that includes the `userReq`(sm.`userReq`), the node's identity `upNode`(N) and certificate `upNodeCert`(C_N), a timestamp `ts` and a signature `msgSig` obtained by signing the SIACK message with k_S. CSP then sends the SIACK message downstream toward R.

Step 12: When the tunnel is established, the CSP at R adds the pair $\langle N, C_N \rangle$ to the flow's upstream neighbor list (UNL) and then constructs a MERGESPEC (Merge Specification) message containing the flow's create request `userSpec`, a fresh timestamp `ts`, its identity `downNode`=R and certificate `downNodeCert`=C_R. R also adds its upstream policy *np.u* to `nodeP` in the MERGESPEC message, signs the message with k_R and sends it to N.

Step 13,14: Upon receiving an MS message mm (through the tunnel), the upstream node N verifies the three signatures mm.`userSpec`.`MTSig`, mm.`userSpec`.`pMTSig` and mm.`msgSig`, checks the timestamp mm.`ts` (allowing for travel and processing time to get to the receiver node and back), unpacks and installs the merge speci-

fication and policies, and then performs the following steps for each node q (with certificate C_q) in the flow's upstream neighbor list.[3]

1. Verify that q is acceptable according to the node upstream policy received in the merge specification: $mm.\texttt{nodeP}(q, C_q)$.
2. Verify that q is acceptable according to the flow's upstream neighbor policy: $fp.u(q, C_q)$.

If the UNL is nonempty after this step, spawn a MERGEd and send the MERGEd the updated upstream neighbor list. (Note that this step happens once for all upstream neighbors at intermediate and receiver nodes. At senders, however, for technical reasons a separate MERGEd is spawned for each sending application program.)

Step 14,15,16: (These steps are similar to 9–11.) N checks whether an IPsec tunnel to q already exists. If not, it sets up IPsec to establish a tunnel, and constructs, signs and sends to q a SECINFO message. SECINFO contains the original JOINREQ for the flow, its identity `downNode`=$N$and certificate `downNodeCert`=C_N, and IPsec information `sinfo`to enable establishing a tunnel. The upstream node prepares its end for the creation of an IPsec tunnel and if successful sends a SIACK message to the downstream node.

Step 17: The downstream node after receiving the SIACK message from S sends the merge specification to the upstream node. But since the upstream node S was added after the receipt of a JFR message and not an RMS message, a partial merge specification instead of a full merge specification is sent upstream. N thus creates a `pms` message that includes the original `userSpec`=mm.`userSpec`, the partial merge specification **pmergespec**, a timestamp `ts`, N's identity `downNode`=N and certificate `downNodeCert`=C_N, and a signature `msgSig` obtained by signing the `pms` message using k_N. The CSP then sends the `pms` message upstream to S.

Step 18: Upon receiving a PMS message pm, the CSP at the sender node S verifies (i) the signatures pm.`userSpec.pMTSig`and pm.`msgSig`, (ii) the timestamp pm.`ts`, and (iii) the certificate pm.`downNodeCert`. If the verification is successful S spawns a partial merge daemon and notifies Y that the join operation has completed, and data transfer can begin.

5 Security Analysis

The *Security Architecture for Active Networks* [11] enumerates the various attacks that can be mounted against an active network framework. Given this threat model, we briefly describe how our secure concast service fares under these various attack scenarios.

Attacks resulting in usurpation: *Theft of service* attacks are prevented by concast's authentication mechanisms. As described earlier, the concast service is based on well-defined trust relationships that must be verified before any node, sender or intermediate merge node, will be added to the flow. Because the flow is established

[3] Note that at this point it has already been established that the originating user satisfies *fp.j*, and that the downstream node satisfies N*:np.d*, the local downstream node policy.

hop-by-hop, each node's authenticity and integrity can be verified individually and compared against the receiver's and provider's security policies before being included in the flow. As a result, only nodes with the proper certification are allowed to access the service.

Attacks resulting in unauthorized disclosure: Outside of breaking into a host or router, packet snooping is the most common technique for obtaining access to content. In secure concast, all data traffic is encrypted. Secure up- and downstream channels (IPsec tunnels) are created and merge specifications, including encryption keys, are transported via these secure channels. Data packets are encrypted and decrypted hop-by-hop using these keys. Thus, so long as no trusted node is compromised, no confidential data is disclosed.

Attacks resulting in deception: Secure concast prevents *masquerading by spoofing* attacks via two methods. First, all control messages are sent over IPsec tunnels whose endpoints have been authenticated. The only exception are the initial JFR and RMS messages which are transmitted in the clear. However, these message carry a digital signature that can help identify spoofed addresses. Even if these messages are not identified as spoofed messages, they are simply used to trigger the initiation of fully authenticated IPsec tunnel where their identity will be checked. Second, all data packets are encrypted and carry a message authentication code. Packets can be spoofed, but without the correct encryption key, the merge daemon will discard them. At best, such packets result in a denial of service attack (see below).

Replay attacks are another form of deception. Because all control packets are carried over the IPsec tunnel, replay attacks are automatically detected by IPsec. Only the initial JFR and RMS travel outside the tunnel. Both carry an authenticated timestamp that is used to detect packets that are outside the acceptable delivery time window. Packets replayed during the window while the tunnel exists are harmless because the operations are idempotent. Regarding the data channel, all packets are encrypted and can carry a sequence number to detect duplicates if the user desires. *Substitution attacks*, which represent another form of deception, are prevented via the use of cryptographic integrity checks. All packets are digitally signed to guarantee the packets integrity.

Attacks resulting in disruption/Denial of Service: These types of attacks present the biggest problem for the secure concast service. Although secure concast prevents some of the attacks, there are several different attacks that could be launched to consume packet processing cycles at network nodes, the receiver, or senders.

An example of a disruption attack that secure concast prevents is the *join circumvention attack*. In this case a malicious node ignores the join process and simply sends data to a merge daemon for merging. Because the attacker does not know the key, the MAC check fails, so the merge daemon does not merge the packet into the stream, thereby preventing disruption of the stream with bogus data. However, the time spent processing the packet still represents a DoS attack that is difficult to prevent.

DoS attacks can also be mounted via false requests. Every time a bogus join request is received, the network nodes expends resources trying to setup the IPsec tunnel, only to find that the sender is not responding.

6 Performance evaluation

Fig. 4. Concast video application containing four merged streams.

In order to measure the costs of our secured concast service, we used a concast video-merging application[12]. Some video applications require the ability to receive video feeds from multiple sources simultaneously; examples include distance learning and video monitoring/surveillance. The objective is to receive the best possible video quality from all sources. For our concast video merging application, a concast session is established that transcodes the incoming streams into lower-quality streams, thereby reducing the network bandwidth requirements. The idea is to replace uncontrolled loss due to congestion with *controlled* loss due to transcoding. To support this type of application, we designed a simple merge function that scales the incoming video stream by down-sampling the pixels that comprise each frame of the video, and combining all incoming streams into a single outgoing stream. In other words, each network link should carry no more than one video steam. To achieve this, the merge specification keeps track of the number of incoming video streams and the number of original video streams encoded in each incoming stream. It then assigns a region of each outgoing frame to each incoming video stream and down-samples the stream appropriately to fit in the assigned region. The assignment of streams to regions takes into account the relative sizes of the (possibly already down-sampled) incoming streams. As new streams "join" the concast session, the existing images are adjusted to make room for the stream. Each composite stream carries information about how many original streams it contains and how they have last been combined so that each node can determine how to combine its incoming streams. This ensures that even if an unbalanced merge tree was built by all the concast senders, the final video stream delivered to the concast receiver will have a roughly proportional display area for each of the constituent video streams.

Our test topology is shown in Figure 5. We used four video senders, each transmitting an uncompressed black-and-white 320x240 video stream at a given frame rate. At each merging hop in the network the frames were merged into a single sub-sampled

image. Figure 4 is a resulting frame captured at the receiver node. We ran with both a 'NULL' cipher specification as well as using AES (128-bit) encryption with a SHA1 message digest at rates of 4, 6, and 8 frames per second (fps). In each case, we measured the total system and user level CPU utilization. The data was taken from merge node

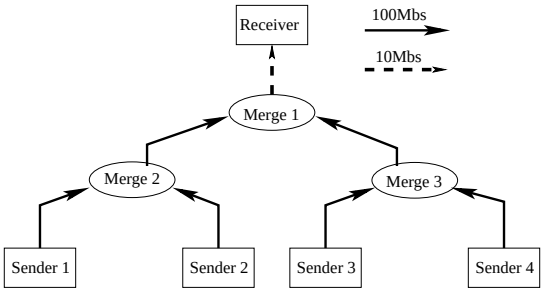

Fig. 5. Experimental Network - Secure Concast Video Merging

3 in the experimental topology (Figure 5). All nodes in the network were 1.5Ghz Pentiums with 128MB of RAM, resulting in similar results for the remaining merge nodes. Our concast merge specification was written in Java and runs in a user-level JVM, which accounts for the majority of the load. The results of our experiments are presented in Figure 6. In the graphs, load with AES encryption is on the left, with "null" encryption (i.e. encrypt and decrypt are no-ops) is on the right. In each case, encryption/decryption of the video streams imposed an overhead of roughly 20 percent. As can be seen from the results for AES/SHA1 with 8 fps, the merge nodes were running at maximum CPU utilization (system + user). When trying to go beyond 8 fps the nodes were overloaded which resulted in packet loss. The initial high load in each case is measurement taken during Java Virtual Machine startup (i.e. when spawning the Merge Daemon).

Our results demonstrate that the presented security mechanism (in particular, per packet data security) is feasible and can be implemented with a reasonable overhead. Note that our implementation of the secure merge specification has been done in Java only with the intention to show feasibility and not optimality.

7 Related Work

The DARPA Active Network community has defined an architecture for an active nodes [13] that comprises a NodeOS and one or more Execution Environments (EE). A security architecture has been proposed for the architectural framework, with particular attention paid to capsule-based EEs (i.e. those that expect code to be included in each packet). An important observation by the authors of that architecture is that some part of the active packet is dynamic (changes at intermediate hops) and the rest of it is static. Digital signatures are used to provide end-to-end authentication and integrity protection to the static part of the packet. HMAC-SHA-1 integrity protection is used between two

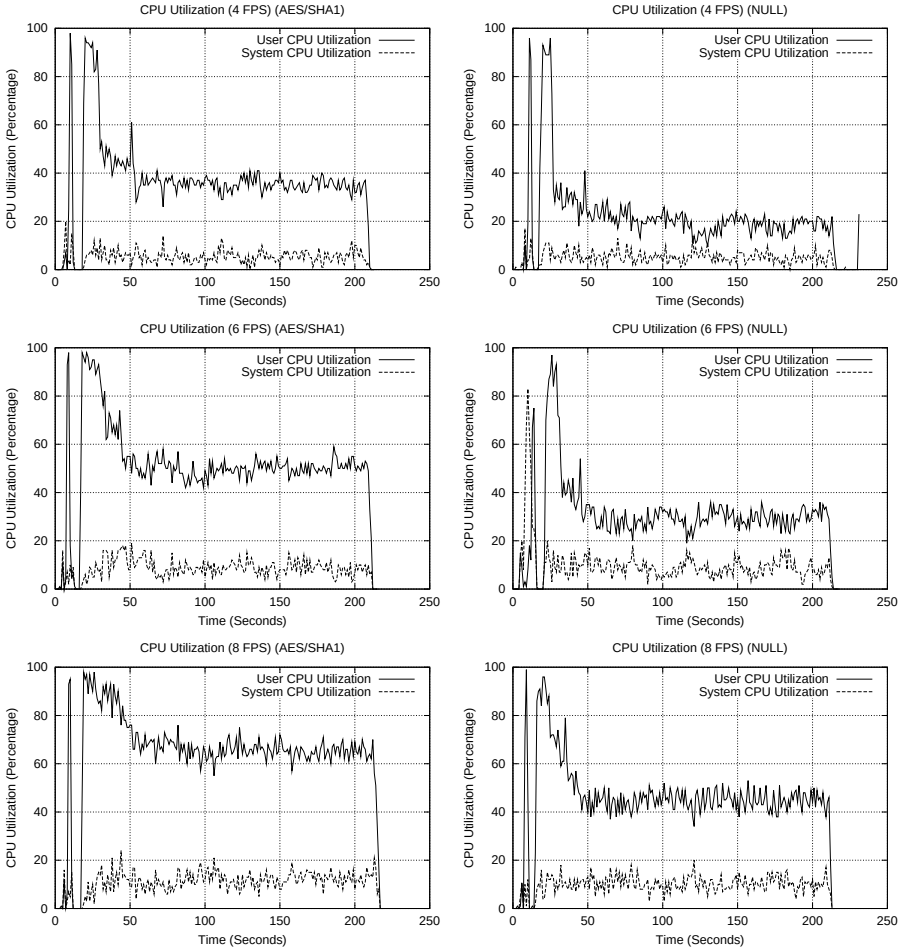

Fig. 6. Secure Merge Processing Overhead

neighboring nodes to provide integrity protection. Certificates are stored in DNS CERT records and every packet carries references to the appropriate certificates. Authorization to execute code is based on the Java 2 security architecture with modifications to support multiple policies, a feature often needed in active networks.

Like the AN security architecture, SANTS [11] differs from the approach described in this paper mainly in its focus on a capsule-based processing model. Every packet is singly responsible from its own authentication, integrity and authorization. In our approach, however, packets belong to a flow, and all packets of a flow pass through the same sequence of nodes. Initial efforts are required to authorize all the members belonging to the flow. But once achieved, the problem of confidentiality and integrity is simplified due to the use of a shared secret by all members of the flow.

The Switchware project [14] included one of the first attempts to deal with security in active networks. Like concast, the Switchware architecture allowed for flow-based programmability. The Secure Active Networks Environment (SANE) [15] also allowed an end-system control over the nodes participating in its flow, by setting up nested tunnels hop-by-hop. Unlike our approach, however, the SANE approach exposes the identity of every node being programmed to the originating end system. While this avoids the need for transitive trust, it limits scalability.

A framework to provide hop-by-hop security in an active networking environment for unicast and multicast applications was proposed by Krishnaswamy et.al [16]. Their approach makes use of a centralized Keying Server (KSV) which provides an interface to accept a secure topology in the form of “links” or “groups” for unicast and multicast respectively. Every node that is a part of the secure topology sets up an IKE SA with the KSV. The KSV uses this SA to securely convey to the each “node” in the topology all information that it needs to reliably setup a security association with its peer(s). They use Linux IPChains to enforce a policy governing which packets can be accepted into the node, and they use the DNS service to retrieve the public keys associated with nodes. However, in their approach all flows seem to share the same hop-by-hop channel for security. Also, there does not seem to be a concept of node-level or flow-level policies that would enable nodes to control the membership in the flow.

8 Conclusion

A number of security challenges are associated with active networking applications that process data on a per hop basis. Some of the requirements of such applications are secure distribution of the processing code and shared secrets, authentication and authorization of the members, confidentiality and integrity of application data. Standard end-to-end mechanisms cannot be used to solve these problems.

In this paper we have attempted to solve the security challenges specific to concast, a many-to-one communication service. A fundamental feature of our solution is the use of IPsec which provides us confidentiality, authentication and integrity on a point-to-point link. We combine IPsec with a rich set of policies and this lets us identify legitimate members of a flow, define trust relationships among the various members, and outline the type of protection required by each node and the service as a whole. The availability of a secure control plane helps us provide a platform to applications

to securely distribute shared secrets and thereby achieve confidentiality and integrity of application data.

References

1. K. Calvert, J. Griffioen, B. Mullins, A. Sehgal, and S. Wen, "Concast: Design and implementaion of an active network service," *IEEE Journal on Selected Areas in Communications (2001)*, pp. 19(3):426–437, March 2001.
2. S. Kasera, S. Bhattacharyya, M. Keaton, D. Kiwior, J. Kurose, D. Towsley, and S. Zabele, "Scalable Fair Reliable Multicast Using Active Services," *IEEE Network Magazine*, February 2000.
3. I. Kouvelas, V. Hardman, and J. Crowcroft, "Network Adaptive Continuous-Media Applications Through Self Organised Transcoding," in *the Proceedings of the Network and Operating Systems Support for Digital Audio and Video Conference (NOSSDAV 98)*, July 1998.
4. E. Amir, W. McCanne, and H. Zhang, "An application level video gateway," in *ACM Multimedia '95*, 1995.
5. D. Wetherall, J. Guttag, and D. L. Tennenhouse, "ANTS: A toolkit for building and dynamically deploying network protocols," in *IEEE OPENARCH'98*, San Francisco, CA, April 1998.
6. S. Merugu, S. Bhattacharjee, Y. Chae, M. Sanders, K. Calvert, and E. Zegura, "Bowman and CANEs: Implementation of an active network," in *Proceedings of the 37th annual Allerton Conference on Communication, Control, and Computing, Monticello, Illinois*.
7. K. Calvert, J. Griffioen, A. Sehgal, and S. Wen, "Concast: Design and implementation of a new network service," in *Proceedings of 1999 International Conference on Network Protocols, Toronto, Ontario*, November 1999.
8. Bob Braden, Lixia Zhang, Steve Berson, and Shai Herzog Sugih Jamin, "Resource ReSerVation Protocol (RSVP)," September 1997, RFC 2205.
9. D. Eastlake and P. Jones, "US Secure Hash Algorithm 1 (SHA1)," September 2001, RFC 3174.
10. B. Kaliski and J. Staddon, "PKCS #1: RSA Cryptography Specifications. Version 2.0," October 1998, RFC 2437.
11. S. Murphy, E. Lewis, R. Puga, R. Watson, and R. Yee, "Strong security for active networks," in *The Fourth IEEE Conference on Open Architectures and Network Programming*, April 2001.
12. K. Calvert, J. Griffioen, B. Mullins, S. Natarajan, L. Poutievski, A. Sehgal, and S. Wen, "Leveraging emerging network services to scale multimedia applications," *Software—Practice and Experience (SPE)*, vol. 33, no. 14, pp. 1377–1397, November 2003.
13. AN Architecture Working Group, "Architectural framework for active networks—version 1.0," July 1999.
14. D. Alexander, W. Arbaugh, M. Hicks, P. Kakkar, A. Keromytis, J. Moore, C. Gunder, S. Nettles, and J. Smith, "The switchware active network architecture," *IEEE Network*, May 1998.
15. D. Alexander, W. Arbaugh, A. Keromytis, and J. Smith, "Safety and security of programmable network infrastructures," *IEEE Communications Magazine, Special issue on Programmable Networks*, 1998.
16. S. Krishnaswamy, J. Evans, and G. Minden, "A prototype framework for providing hop-by-hop security in an experimentally deployed active network," in *DANCE: Darpa Active Networks Conference and Exposition*, 2002.