An Architecture for Security Policy Enforcement

Patrick McDaniel
AT&T Labs – Research
pdmcdan@research.att.com

Atul Prakash
University of Michigan
aprakash@eecs.umich.edu

Abstract

Recent advances in policy specification and evaluation have increased the usage of general-purpose policy frameworks. However, because these frameworks typically defer enforcement, the quality with which policy is realized is subject to the correctness of each domain-specific implementation. This paper considers the requirements and machinery of an architecture supporting general-purpose policy enforcement. The tangible result of this investigation, the Antigone 2.0 enforcement framework adopts a broad definition of policy. Antigone policies encompass context sensitive session provisioning and access control. Antigone enforces policies meeting this definition through the run-time composition, configuration, and regulation of security services. We present the Antigone 2.0 architecture, and demonstrate enforcement through several non-trivial policies. A profile of policy enforcement performance is developed, and key architectural enhancements identified.

1 Introduction

Distributed systems have historically addressed security requirements through the integration of services providing fixed policies. Recent systems have sought to provide more flexible security through the integration of policy specification and evaluation frameworks [1, 2, 3]. These frameworks provide a means of expressing and evaluating policy within rigorously defined languages. However, policy enforcement is typically left to the supported applications and services. Hence, the quality with which policy is realized is subject not only to the correctness of specification and evaluation, but also to the interpretation and implementation of policy by each domain-specific application.

This paper considers the requirements and design of a flexible policy enforcement framework. The tangible result of this investigation is the Antigone policy enforcement architecture. Antigone enforces *provisioning* policy through the run-time composition and configuration of communication and security service mechanisms. The subsequent evaluation of fine-grained *access control* policy is deferred to a representation-specific policy engine and enforced by the configured mechanisms. Hence, in separating policy evaluation from enforcement, Antigone can support a policy representation appropriate for the target environment. We consider the requirements of flexible policy enforcement, and identify approaches addressing key performance and design issues. A profile of policy enforcement performance is developed, and key architectural and policy construction issues identified.

Our previous work considered the design and use of static provisioning policies for multi-party communication, policy representation, and algorithms for policy determination [4, 5]. In this paper, we continue our investigation of policy by considering the requirements and techniques of general-purpose policy enforcement. These aggressive goals required the previous Antigone architecture be discarded. The new framework (Antigone 2.0, or simply *Antigone* throughout) introduces support for formal policy languages encompassing both provisioning and access control policy, integrates augmented mechanism control services, and considers a number enforcement optimizations. Moreover, the many new services and policies provided a framework under which the costs of group and peer security policy could be

scrutinized.

We seek to unify the services provided by existing policy frameworks. Authentication [2, 6] and trust management systems [1, 7, 8, 9] largely view security policy as statements of acceptable authorization and access control. Policy is specified within a well-defined and often rigorously evaluated framework. However, provisioning and enforcement is largely outside the scope of these systems. Conversely, in *policy based networking* [10], a policy defines generalized rules for the configuration of network resources. Typically used in network management, these systems define how resources present in a network are configured, and in the presence of changing environments, reconfigured.

Antigone builds upon the body of component and event based communication services [11, 12, 13, 14]. However, the restrictions placed on the organization, interfaces, and state maintenance of components made their direct application to policy enforcement difficult. We consider these issues in detail in Section 7.

The remainder of this paper is organized as follows. The following section identifies the requirements and goals of this work. Section 3 briefly describes the means by which a session policy appropriate for a given environment is identified. Section 4 presents the Antigone architecture. Section 5 details the construction of several optimizations addressing performance and flexibility requirements. Section 6 evaluates the enforcement costs of Antigone under several real world policies. Section 7 considers a number of architectural alternatives. Section 8 concludes.

2 Requirements and Goals

The scope of policy in existing security infrastructures has historically been defined by the policy schema [15]. Hence, user and environmental needs are addressed inasmuch as they are anticipated by system architects. However, such policies fail to capture the requirements of evolving applications and services. Recent efforts have sought to define more flexible policy specifications [1, 6, 16], but as yet, have not focused on approaches for general-purpose enforcement. This paper defines an architecture that does not enforce a given schema, but provides an infrastructure that interprets and enforces a broad class of policies. In so doing, we attempt to identify the issues and machinery of general-purpose policy enforcement.

Before any discussion of enforcement can begin, we must develop an understanding of the requirements that must be placed on the policies themselves. Centrally, any policy must be *enforceable*. While we develop a more formal definition in [5], the intuition behind enforceability states that the policy must lead to a representative and functional system configuration. A policy is representative if it identifies and configures a set of services meeting session goals (e.g., confidentiality), and it faithfully reflects the trust embodied by the environment (e.g., authorization). A policy is functional if it provides sufficient services for the session to make progress. The evaluation of these properties is the central goal of the policy determination infrastructure, and is a prerequisite of policy enforcement.

Realizing the semantics of policy is the central goal of an enforcement architecture. Hence, in addressing a broad definition of policy, Antigone has the following goals:

- *Security* - the semantics of policy must be faithfully and correctly reflected in system behavior.
- *Flexible policy enforcement* - the architecture must allow the integration of a wide range of security services and access control models.
- *Run-time provisioning* - the services required to enforce policy must be (and often can only be) identified and configured at run-time.
- *Run-time authorization* - the trust, means, and form of authentication must be evaluated within the run-time context.
- *Efficient enforcement* - overheads associated with enforcement must not significantly hamper application performance.

This paper discusses how these goals are achieved in Antigone. We begin with a brief dis-

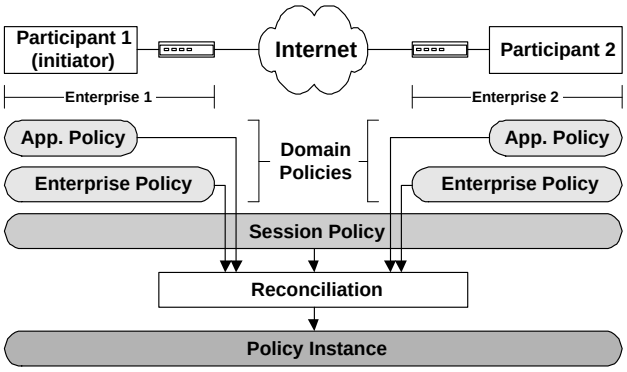

Figure 1: *Policy construction* - A session-specific policy instance is created by an initiator through the reconciliation algorithm. The instance is subsequently used to provision and regulate the session.

cussion of the means by which an enforceable policy is identified through policy determination.

3 Policy Determination

Determination is the process whereby a session policy is developed from the requirements stated by interested parties. The Ismene language and associated determination algorithms [5, 17] are used throughout to motivate a discussion of enforcement. However, the use of Ismene is an artifact of the evolution of Antigone, rather than a requirement for its use. Other policy languages (e.g., SPS [18], KeyNote [7]) may be integrated with Antigone as necessary and desirable (see Section 4.3). This is consistent with the goals of this work; the sources, representation, and determination of policy should be largely orthogonal to enforcement. The remainder of this section summarizes the means and semantics of policy determination.

Antigone policies define the security-relevant properties, parameters, and facilities used to support a session. Thus, a policy states how security directs behavior, the entities allowed to participate, and the mechanisms used to achieve security objectives. This broad definition extends much of existing policy; dependencies between authorization, access control, data protection, key management, and other facets of a communication can be represented within a unifying policy. Moreover, requirements frequently differ from session to session, depending on the nature of the session and the environment in which it is conducted. Note that the enforcement infrastructure is limited by the scope of the policies themselves. For example, fine-grain access control can only be enforced only where the policy defines it.

In Ismene, the policy *instance* enforced at runtime is the result of the reconciliation of session and domain policies. A session policy acts as a template that identifies the (potentially many) ways a session can be constructed. Each participant submits a set of *domain policies* identifying the requirements and restrictions that must be addressed by the session. Depicted in Figure 1, an initiator[1] constructs a policy instance compliant with each domain and the session policy through the reconciliation algorithm. To simplify, reconciliation arrives at the instance by refining the session policy as directed by requirements stated in domain policies.

An instance is comprised of policies for session provisioning and access control. Session provisioning identifies software modules, called *mechanisms*, used to enforce policy. Associated with a mechanism is a set of zero or more *configuration* parameters used to further specify its operation. The access control policy defined in the instance is represented as sets of access control rules. Each rule associates a protected *action* with a conjunction of positive conditionals. Access control rules are consulted when an action is undertaken, and access granted where all conditions are satisfied.

While a reconciliation algorithm may be able to identify an instance satisfying the session and domain policies, it makes no guarantees that the instance is enforceable. The *analysis* algorithm determines whether the provisioning of a session adheres to a set of assertions that express correctness constraints on a policy instance. The assertions relevant to a session are efficiently tested against the instance prior to the session initialization. For example, a common constraint placed on a policy instance requires that some authentica-

[1] Often a session participant, an initiator is the policy decision point performing reconciliation [19].

```
% Provisioning policy
provision : ::
        config(ESP(tunnel,3des,hmac-md5)),
        config(IKE(preshared,grp2,3des,hmac-md5,3600)),
        config(KA(60));

% Authorization/access control policy
accept_packet : credential(&key,$key.id=$sk) :: accept;

init_session : credential(&key,$key.key=$presharedkey),
               timeofday(0900,1700) :: accept;
```

Figure 2: IPsec/Ismene Policy Instance - an instance defines the provisioning and access control policies enforced at run-time.

tion mechanism be provisioned. Such a constraint is expressed by assertion. Any instance failing to satisfy this assertion is rejected.

Figure 2 illustrates a simplified policy instance appropriate for an IPsec [20] session. The provisioning policy states that three mechanisms, ESP, IKE, and KA, must be used to implement the session. The ESP mechanism is further configured to implement tunnel-mode, triple-DES, and MD5 HMACS. The IKE configuration states that preshared keys be used, identifies a set of cryptographic algorithms, and instructs IKE to refresh the key session once per hour. KA augments the IPsec service by introducing crash failure detection. This service periodically transmits a keep-alive message (every 60 seconds), and detects when other participants fail to do so.

The access control policy for the `accept_packet` action states that any properly formed packet transformed using the session key should be accepted. The `credential()` conditional tests whether a relevant credential has been supplied. In this case, the implied knowledge of the session key $\$sk$ is sufficient to authorize the packet. The second access control rule, `init_session` defines when a session should be accepted. In this case, the `timeofday` conditional is consulted at the point at which a particular session is initialized. If the time of day is between 9:00am and 5:00pm and the requester proves knowledge of the preshared key $\$presharedkey$, the session is accepted.

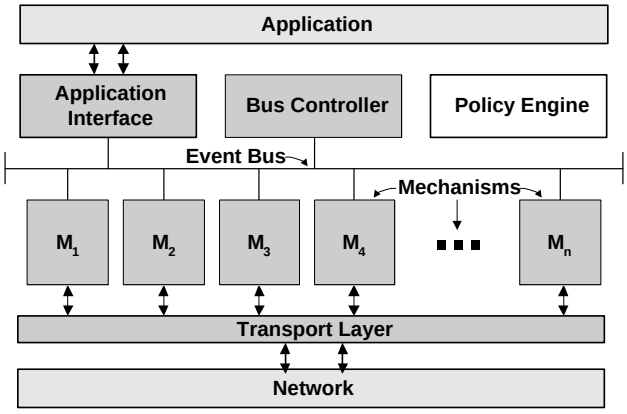

Figure 3: Architecture - the mechanisms, interfaces, and policy engine coordinate to enforce run-time determined policy.

4 Architecture

This section presents the motivation, design, and operation of Antigone. Depicted in Figure 3, the Antigone architecture consists of a collection of software components (mechanisms), a policy decision point (policy engine), a bus controller (event controller), and application and network interfaces (application and transport).

Antigone is a single-threaded component architecture[2]. Communication between the infrastructure and software components is implemented by events. Applications built on Antigone transfer control to the Antigone through socket-oriented calls (e.g., `send()`, `recv()`, `select()`). Application action (e.g., send) is translated into events and delivered to all mechanisms. Policy is enforced by the mechanism reaction to and creation of events. Hence, cascading events direct the progress of the session, and ultimately the application.

Antigone *mechanisms* are software components implementing policy. The mechanisms used to implement the session are defined at run-time by the policy instance. While typically implementing se-

[2]The decision to implement Antigone as a single thread greatly simplified component management (e.g., state maintenance), and allowed our efforts to be focused on issues of policy enforcement. We are currently in the initial phases of implementing Antigone as a multi-threaded, multi-processor architecture.

curity services (e.g., authentication, key management), other session-oriented functions can be implemented via mechanisms (e.g., auditing, failure detection and recovery, QoS). Section 4.4 provides an overview of the design and use of Antigone mechanisms.

The *policy engine* acts as the policy decision point for Antigone. Enforcement is defined in two distinct phases; provisioning and access control. The policy engine provisions the session by identifying and configuring the set of mechanisms at session initialization. Subsequent action is regulated by the policy engine through the evaluation of access control policy. The policy engine does not process or emit events; the policy engine evaluates policy as directed by the mechanisms (i.e., via upcall). The design and use of the policy engine is presented in Section 4.3.

Motivated by multiprocessor architectures, the *event bus* directs virtual or real broadcast delivery of events between the application interface and mechanisms. Events *posted* to the bus controller are delivered in FIFO order to all mechanisms and the application interface. We consider this design in Section 4.2.

State is shared in Antigone through the *attribute set*. Similar to the KeyNote action environment [7], the attribute set maintains a table of typed attributes. Attributes are defined by {name, type, value} tuples. Mechanisms and the application interface are free to access, add, modify, or remove attributes from the attribute set. Attributes are defined over basic data types (e.g., strings, integers), identities (e.g., unique identifier), and credentials (e.g., keys, certificates). For example, the local identity, session addressing information, and configured preshared keys (credentials) are stored in the attribute set.

The *application interface* arbitrates communication between the application and Antigone through a simple message oriented API. While an application need only use simple message interfaces, advanced calls are provided to extract and manipulate Antigone specific state. The *transport layer* provides a single communication abstraction supporting varying network environments (i.e., single interface for TCP, UDP, multicast, and simplified ad-hoc network [21]). For brevity, we omit further details of the application interface and transport layers except where relevant to policy enforcement.

4.1 Policy Enforcement Illustrated

This section briefly motivates the design of Antigone by illustrating the enforcement of data security, failure detection, and access control policies defined by the policy instance presented in Section 3. For this example, we assume that the session has been initialized (provisioned), and that a session key has been negotiated by the IKE mechanism (i.e., an SA has been established). As its operation is not relevant to the present discussion, we omit further mention of IKE. The following text and Figure 4 describe transmission of a single application message, (where the letters a, b, c and d correspond to the labeled figures):

a) The application transmits data over the session via the `sendMessage` API call. The call is translated into an `EVT_SEND_MSG` event (SE) by the application interface, which is posted to the bus controller. The application data (Dat) is encapsulated by the send event.

b) The bus controller delivers the send event to all mechanisms (via virtual broadcast). In response, the ESP mechanism appeals to the policy engine for an access decision of the *send* action. All relevant state (e.g., current session key, bytes to transmit, etc.) is passed to the policy engine, and used to as input to the evaluation of the *send* access control policy. Because transmission is predicated solely on knowledge of the session key (credential), the policy engine accepts the action.

c) ESP selects a data transform appropriate for the configured policy (i.e., 3des, hmac-md5). The data is transformed and headers and HMACs attached. The transformed buffer is then sent to the other session participants via the transport layer. An `EVT_SENT_MSG` (ST) event containing the sent buffer is posted to the bus controller following the transmission.

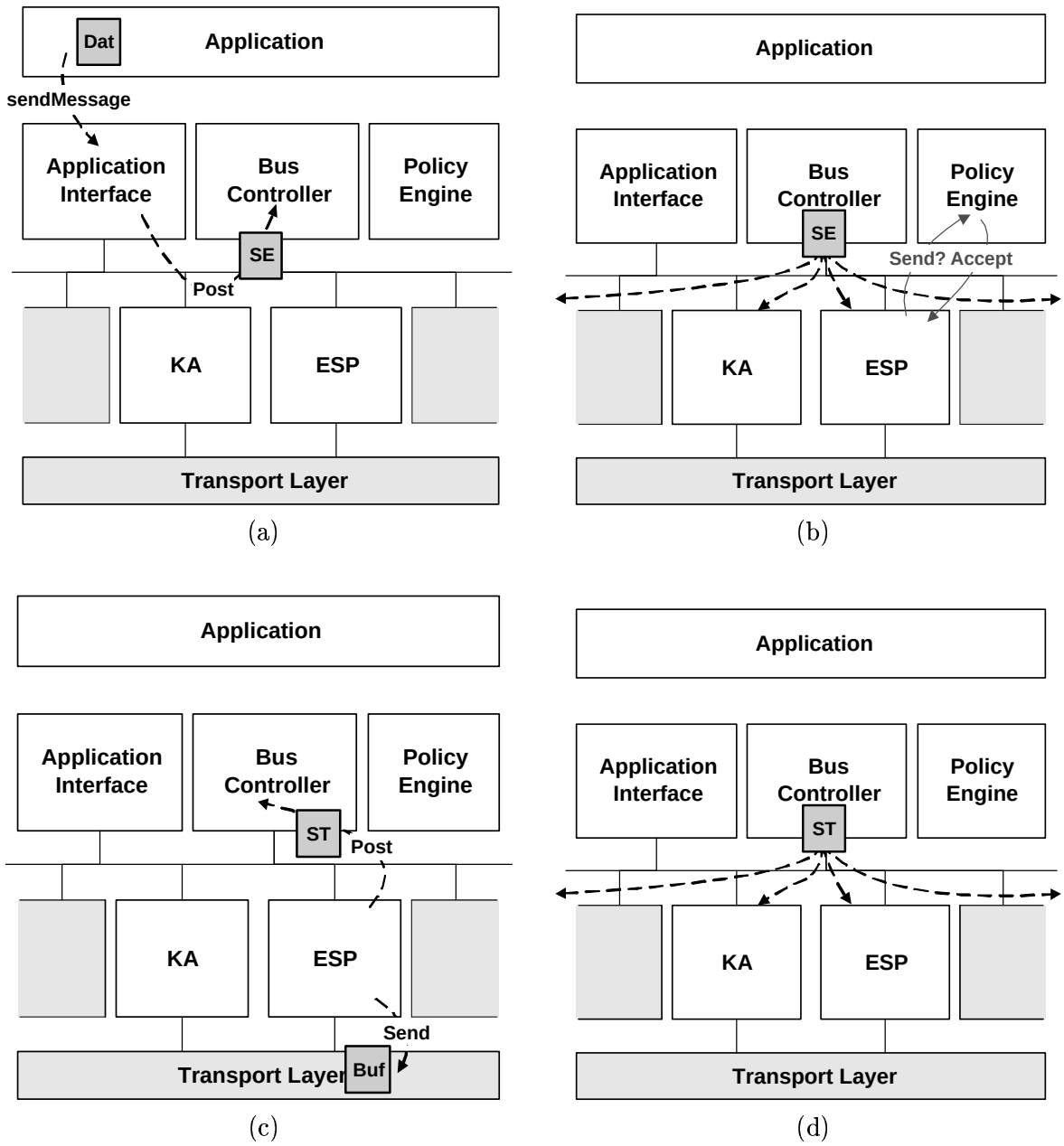

Figure 4: Policy Enforcement Illustrated - an application `sendMessage` API call is translated into a *send event* posted to the bus controller (a). The reception of the event by the ESP mechanism triggers the evaluation of the access control policy via upcall (b), and ultimately to the transmission of transformed data (c). The transmission triggers further event generation and processing (d).

d) The sent event is posted to all mechanisms. The KA failure detection mechanism, using the transmission indicated by the `EVT_SENT_MSG` event as an implicit keep-alive, resets an internal keep-alive transmission timer.

Note that other policies may result in different behavior. Such is the promise of policy driven behavior; requirements for content protection, failure detection and recovery, and other session behaviors are defined by policy. The use of common interfaces (e.g., events) allows the flexible compo-

sition and configuration of those implementations necessary to address session requirements.

4.2 Event Bus

Inspired by multiprocessor architectures, the event bus is a broadcast service. Hence, all events are delivered to every mechanism and the application interface. Events received on the bus are processed in accordance with each module's purpose and configuration. Events are acknowledged by each implementation with an indicator identifying whether the event was processed or ignored. Events that are ignored by all modules are logged.

Event delivery is modeled as being simultaneous. The event bus guarantees that *a)* events are delivered in FIFO order and, *b)* an event will be delivered to all mechanisms and the application interface before any other event is broadcast. The event bus provides no guarantees on the ordering of mechanisms to which the event is delivered. One advantage of this design is its use in multiprocessor systems; mechanisms executing on separate processors can use the processor bus to receive and handle events simultaneously. In this way, Antigone can optimize enforcement costs by committing processors to high throughput mechanisms (e.g., data handler in a video-on-demand server).

The authors of the Polylith system [22] have noted that broadcast delivery increases event granularity. Events requiring ordering constraints must be decomposed and posted to enforce the ordering. For example, if the send event defined in Section 4.1 was guaranteed to be delivered to the data handler prior to failure detection mechanism, the failure mechanism could simply reset the keep-alive timer only where the transmission was successful (and hence avoid the creation of the *sent* event). In general, however, our experience in policy enforcement shows that such constraints are few [4]. However, due-diligence must be expended in analyzing the use of events across mechanisms. The tangible result of this analysis is a set of compatibility and requirement assertions used by the policy analysis algorithm.

4.3 Policy Engine

All interpretation of policy occurs within the language-dependent policy engine. However, the Antigone enforcement infrastructure need not be aware of the mechanics of policy evaluation; all policy decisions are deferred to the policy engine. This section describes the operation of the Ismene policy engine. Policy engines supporting other languages will differ in operation not because of the mechanics of evaluation, but by the scope and semantics of the supported policies. We are currently investigating the integration of policy engines supporting a range of policy languages (e.g., SPS, KeyNote [7], GSAKMP [23]).

Initially, as directed by the policy instance, the policy engine provisions the mechanism layer by initializing and configuring the appropriate software mechanisms. The provisioning policy is not consulted after initialization. We describe how the instance is distributed to session participants in Section 4.6.

The policy engine enforces access control policy over the lifetime of the session. Each mechanism is cognizant of the actions to be protected by policy (i.e., hard-coded in implementation). For example, an IKE mechanism consults the policy engine when a participant attempts to initiate a session. The rules associated with the `init_session` action are evaluated, and access granted where the relevant conditions are satisfied.

Antigone currently defines a range of basic actions protected by policy through the available mechanism implementations. However, mechanisms are free to define new protected actions. Policies must define access control rules for each protected action. It is incumbent on the policy engine to decide what to do when an action is undertaken for which no access control policy is defined. For example, Ismene implements a closed-world policy in which all such actions are denied.

Mechanisms supply information describing the context under which a particular action is attempted when appealing to the policy engine for an access control decision. The mechanism constructs an action set (which is frequently a subset of the attribute set) of relevant information. This set primarily consists of the rights-proving creden-

tials, but may also contain environmental data (e.g., current processor load). The mechanism must decide on the appropriate set of attributes to provide to the policy engine. For example, acceptance of an incoming packet encrypted under a session key implies knowledge of the session key. Hence, the session key can be used as credential when assessing acceptance. We enumerate and discuss the set of actions supported by the current implementation in [5].

4.4 Mechanisms

An Antigone *mechanism* defines a basic service required by the session. Unlike traditional protocol objects in component protocol systems [12, 24], mechanisms are not vertically or hierarchically layered (e.g., X-kernel [11]). Note that this does not mandate that mechanisms implement monolithic or course-grained components. Each mechanism embodies an independent state machine, which itself may be layered. For example, the layered Cactus membership service [25] can be integrated within Antigone as a single mechanism.

Each mechanism is identified by its type and implementation. Antigone currently supports six mechanism types; authentication, membership management, key management, data handling, failure detection and recovery, and debugging. A mechanism implementation defines the specific service provided. For example, we have implemented three multiparty key management mechanisms: Key-Encrypting-Key [26], Authenticated Group Key Management [5], and Logical Key Hierarchy [27]. These categories are not exhaustive; new types (e.g., congestion control) or implementations (e.g., One-Way Function Tree key management [28]) can be introduced as new services are needed.

Internally, session operation is modeled in Antigone as *signals*. Each signal indicates that some relevant state change has occurred. Policy is enforced through the observation, generation, and processing of signals. Antigone defines event, timer expiration, and message signals. The interfaces used to create and deliver signals are presented in Figure 5.

Events signal internal state changes. An event

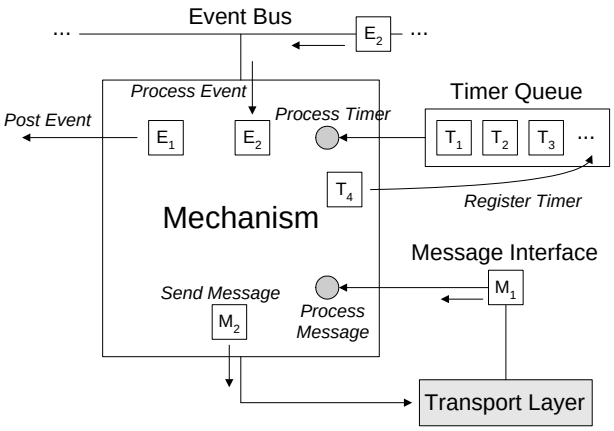

Figure 5: Mechanism Signal Interfaces - Policy is enforced through creation and processing of *events*, *timers*, and *messages*. Events are posted to and received via the event bus. Signals are processed and emitted through the various post, register, send, and processing interfaces.

is defined by its type and data. For example, send events are created in response to an application calling the sendMessage API. This event signals that the application desires to transmit content. The send event has an `EVT_SEND_MSG` type and its data is (a pointer to) the content. Note that mechanisms are free to define new events as needed. This is useful where sets of cooperating mechanisms need to communicate implementation-specific state changes.

A *timer expiration* indicates that a previously defined interval has expired. Timers may be global or mechanism-specific; all mechanisms are notified at the expiration of a global timer, and the registering mechanism is notified of the expiration of a specific timer. Similar to events, a timer is defined by its type and data. For example, the expiration of the keep-alive transmission timer discussed in Section 4.1 signals that a keep-alive should be sent. The data identifies context-specific information needed to process the timer expiration (e.g., keep-alive sequence number).

Messages are created upon reception of data from the transport service. Messages are specific to (must be marshaled/processed by) a mechanism. Every message is defined by a mechanism identifier, an implementation identifier,

and a message type identifier. For example, the header {FDETECT_MECH, KA_MECH, KA_KALIVE} header identifies a failure detection, keep-alive implementation, keep-alive message. This information is used to route incoming messages to the appropriate mechanism for processing.

We have used the mechanism interfaces to implement large number of mechanisms supporting a wide range of peer and multiparty services. The following subsections illustrate the use of these interfaces by delving into the details of two important mechanisms: a data-handler and an authentication mechanism.

4.5 Data Handling

The Antigone data handling (ADH) mechanism implements content security guarantees on application messages through the application of cryptographic transforms. Current, the Antigone ADH supports *confidentiality*, *integrity*, *authenticity*, and *sender authenticity*. ADH is configured to provide zero or more of these properties. A data transform is defined for each unique combination of properties.

ADH handles outgoing application transmission as described in Section 4.1. Upon reception of a message, ADH applies the reverse transform and evaluates the *send* action via the policy engine (using the transform keys as credentials in the evaluation process). If a positive result is returned, an EVT_DAT_RECV event encapsulating the recovered plaintext is posted to the event bus. Note that a received message may require (e.g., is encrypted by) a session key that the local member has not yet received. In this case, recovery is initiated by the posting of an EVT_KDST_DRP event. If available, key management and failure recovery mechanisms use this event to initiate recovery (acquisition of a new session key). Note that there is sufficient context within every received message to determine the transform and cryptographic algorithms under which it is transmitted (header information). This allows the transform, and indirectly the policy, under which a message is transmitted to be determined and applied on a per-message basis.

ADH does not directly participate in the acquisition of the session keys. This highlights a dependency between data handling and other security services; key management mechanisms must be provisioned to negotiate keys appropriate for the ADH content policy. For example, a key management mechanism that negotiates a 56-bit DES key is incompatible with an ADH policy that requires the use of 128-bit AES. Such dependencies are expressed through assertion and enforced by the analysis algorithm.

The flexibility of ADH has allowed us to investigate the enforcement of many content policies. For example, we have implemented several group source authentication mechanisms (e.g., packet-signing, stream signatures [29]), and integrated DES, Blowfish, RC4, AES, SHA-1, and MD5 with all content policies. We summarize an evaluation of the costs associated with content policy enforcement in Section 6.

4.6 Authentication

An authentication mechanism initializes a session by performing mutual authentication, a key exchange, and policy instance distribution. Antigone sessions are established between an initializer and one or more *requestors*. Note that it is assumed that a policy instance is established prior to session initialization[3]. Antigone currently supports three authentication mechanisms; a null authentication mechanism (which exchanges keys and policy in the clear), an OpenSSL based mechanism [30], and a Kerberos mechanism [31]. The following text describes the operation of the OpenSSL based authentication mechanism (OAM) from the perspective of a requestor. However, independent of the means of authentication, the operation of each of these mechanisms is largely similar.

Initially, the requestor evaluates a *local policy* to arrive at a default policy instance. This instance defines the provisioning and access control policy used to initialize the requestor environment, and is discarded when the session-defining policy instance is acquired (see below). The policy engine

[3]We describe a more flexible model where policy is determined during session initialization in [5].

creates an authentication mechanism specified by the local policy instance when the application is initialized.

The authentication protocol begins when an `EVT_AUTH_REQ` event is posted by the application interface. In response, OAM performs the SSL handshake (establishing a mutually authenticated secure channel using certificate and addressing information stated in the local policy), and receives a public key certificate for the initiator. The certificate is translated into an Antigone credential, and provided to the policy engine for evaluation of the *session_auth* action[4]. If the action is accepted, the authentication mechanism obtains the policy instance and a long-term *pair key* over the SSL-secured channel. Note that the pair key is not used to secure session content, but is used by key management services to negotiate and replace session keys. The SSL connection is closed, and an `EVT_POL_RCVD` and `EVT_AUTH_COM` events are posted.

Upon reception of the `EVT_POL_RCVD`, the application interface destroys the configured mechanisms, discards the local policy instance, and passes the received instance to the policy engine. The policy engine uses the received instance to create and configure session-implementing mechanisms. Once complete, the `EVT_AUTH_COM` signals that the session is ready to begin. This often leads to the initiation of key management protocols.

A number of error conditions can arise during authentication. For example, a policy configured retry timer is registered when the authentication process is initialized. Any exchange not completing prior to expiration is retried and a retry count incremented. If a configured retry count is exceeded, a fatal error is generated and the session is aborted. Similarly, any denial of a *session_auth* action fatally errors the authentication process, and ultimately the session.

[4]The validity of the certificate (e.g., certificate path construction, signature validation, and assessment of revocation information) is assessed during the evaluation of the *session_auth* policy.

5 Optimizing Policy

This section briefly introduces architectural enhancements aimed at improving the performance and usability of Antigone. For brevity, we omit a number of other architectural optimizations (e.g., slab-allocation [32]).

5.1 Policy Evaluation Cache

Where supported by policy, the enforcement of fine-grained access control policy can incur significant overheads. For example, the costs of enforcing Ismene per-message transmission/reception access control (e.g., *send* action policy) in high-throughput applications can be prohibitive. However, because of the way such policies are specified, most evaluation can be amortized. Hence, we introduce a two-level cache that stores the results of rule and condition evaluation.

The *condition evaluation* cache stores the result of each condition evaluation (e.g., *credential()*, *timeofday()*). In addition to a Boolean result, the evaluation process identifies the period over which the result is valid. This validity period may be *transient*, *timed*, or *invariant*. Transient results should be considered valid for only the current rule evaluation. Timed results explicitly identify the period during which the result should be considered valid (e.g., until 4:30pm). Invariant results are considered valid for the lifetime of the session. The cache is consulted during rule evaluation, and timed cache entries evicted when the associated validity period expires.

The *rule evaluation* cache stores the relevant context under which an action was considered (e.g., evaluation credentials and conditions). Entries in the cache are considered valid for the minimum of the reported condition evaluations. Hence, any participant testing the same conditions and credentials (as would be the case in frequently undertaken actions) avoids repetition of potentially complex and costly rule evaluation by accessing cached results.

5.2 Generalized Message Handling

By definition, a flexible policy enforcement architecture must implement a large number of protocols, messages, and data transforms. However, correctly implementing these features requires the careful construction of marshaling code. The *Generalized Message Handling* (GMH) service is designed to address the difficulties of protocol development. GMH uses message specifications and system state to marshal data. Message specifications are interpreted at run time, and the appropriate encryption, hashing, encapsulation, padding, byte ordering, byte alignment, and buffer allocation and resizing are handled by the supporting infrastructure.

While we found that other marshaling compilers (e.g., RPC [33], CORBA [34]) provided excellent facilities for the construction of plaintext messages, they provided limited support for complex security transforms. Moreover, because message specifications are typically interpreted at compile-time, it was difficult to support protocols with run-time specified behavior (e.g., run-time determined message formats). This feature was required by many multi-party key management and source authentication protocols.

We illustrate the use of GMH through the following (tunnel mode) ESP transform:

$$msgDef = \text{“}\texttt{H[LLE[DDDDcc]]}\text{”}$$

Each character in the message specification represents a field (data) or encapsulation operation (e.g., encryption). The latter field types identify the scope of operations using bracket symbols. In the above definition, the character `L` represents a long integer (SPI, sequence number), `D` represents variable-length data (IP/TCP headers, payload, padding), and `c` represents a byte field (pad length, next header). The symbols `H[...]` and `E[...]` signify HMAC and encryption operations. Mechanisms associate data, keys, and cryptographic algorithms with each field at run-time. GMH marshaling code is called, and a message buffer is created, transformed per the specification, and returned to the calling mechanism.

Upon reception of a message, GMH reverses the marshaling process. However, GMH may not initially have sufficient context to unmarshal all the data. In the above example, GMH does not know *a priori* which key was used to calculate the HMAC (i.e., `H[...]`). GHM recovers as much data as possible and appeals to the calling mechanism for guidance (through an upcall). The mechanism uses the previously unmarshaled fields to determine the appropriate keys and algorithms (e.g., mapping unmarshaled SPI to SA). The keys and algorithms are returned to GMH, and the process continues (possibly recursively) until all fields are unmarshaled.

6 Performance Evaluation

This section investigates the performance of the Antigone 2.0 architecture by profiling enforcement overhead (microbenchmarks) and characterizing communication throughput and latency (macrobenchmarks). The current implementation of Antigone consists of 58,000 lines of C++ code in 133 classes (approximately 10% of which was retained from the original Antigone architecture), and has been used as the basis for several non-trivial applications. All source code and documentation for Antigone, the Ismene policy language, and applications are freely available from the Antigone website [35].

All experiments were conducted on an isolated 100 Mbit Ethernet LAN between two unloaded 750 megahertz IBM Netfinity servers. Each server has 256 megabytes of RAM, a 16-gigabyte disk, and runs the Redhat 7.1 distribution of the Linux kernel 2.2.14-5.

6.1 Microbenchmarks

The first series of experiments sought to characterize the functional costs of policy enforcement in Antigone. A test application was instrumented to classify the overheads incurred by the transmission of a single message into *event processing*, *marshaling*, *I/O*, *access control*, and *buffer management and queuing*. All measurements were obtained from the x86 hardware clock and averaged over 100 trials. The results of these experiments are presented in Table 1.

Our experiments show that almost 50% of re-

Operation	recv		send	
	usec	%	*usec*	%
Event Processing	56.35	49%	37.44	39%
Marshaling	33.35	29%	25.92	27%
I/O	10.35	9%	19.2	20%
Access Control	8.05	7%	6.72	7%
Buf/Queue Mgmt.	6.9	6%	6.72	7%
Total	115	100%	96	100%

Table 1: Microbenchmarks - measured overhead of a single application transmission.

ceive overhead (and 40% of send overhead) can be attributed to event processing. This is the fundamental cost of an event architecture; processing costs are often dominated by the event creation, delivery, and destruction.

Note that the difference between the total `send` and `recv` costs can be attributed to additional receive processing requirements; e.g., recursive unmarshaling, additional data copies. Our experiments also reported a similar, but inverse, asymmetry between send and receive I/O. The `send()` system call takes approximately twice as long to complete as `recv()`. The difference can be attributed to our measurement technique; received packets are asynchronously serviced by an interrupt handler upon arrival. Hence, much of the kernel processing was completed prior to the `recv()` system call, and thus not included in the measurements.

About 30% of the overhead is consumed by marshaling. GMH interpretation of message template structures and context processing up-calls is less efficient than hard-coded protocol implementations. However, as GMH has not as yet been fully optimized, we are optimistic that these costs can be reduced.

These experiments also demonstrate that the cost of fine-grained access control enforcement can be mitigated by caching. In these tests, the "send" action was regulated on a single unconditional access control rule (e.g., authorized by the session key). Hence, the "send" action policy was evaluated only on the first send/receive, not consulted thereafter (e.g., invariant result served by rule evaluation cache). Note that these results serve as a lower bound; other policies may require more complex or frequent evaluation.

6.2 Macrobenchmarks

The second series of experiments profile enforcement costs by measuring the maximum burst rate and average round trip time (RTT) under a range of security policies. Note that because the latency measurements calculate the total round trip time, the results represent four traversals of the protocol stack.

The *direct* experiment establishes a performance baseline using a non-Antigone application implementing Berkeley socket communication. The *null* policy specifies no cryptographic transforms be applied to transmitted data (i.e., data is sent in the clear). The *integrity* policy is enforced through SHA-1 based HMACs. The *confidentiality* policies encrypt data using the identified algorithm. Appropriate only for multiparty communication, the *integrity, confidentiality, and source authentication* policy specifies SHA-1 HMACS, Blowfish encryption, and 1024-bit RSA stream signatures. A variant of Gennaro-Rohatgi On-line signatures [29], the stream signature mechanism chains-forward signatures by including a hash of each succeeding packet from an initial signed packet. A new stream signature is generated once every 100 msec or when 20 packets are queued for transmission.

As presented in Figure 6, throughput in Antigone is largely driven by the strength of the enforced data handling policy. While the testbed environment (direct) is capable of transmitting up to 9 MBytes/Second, Antigone is limited to just under 8 (null). This 11% reduction can be attributed to the overheads described in the preceding section.

Integrity and confidentiality policies exhibit similar throughout. It is interesting that a confidentially policy using the slower Blowfish algorithm only marginally reduces throughput over a similar policy using RC4[5]. Because the cryptographic algorithms are significantly faster than

[5]The throughput RC4 and Blowfish were benchmarked in the test environment at 51.17 and 24.30 MB/sec, respectively.

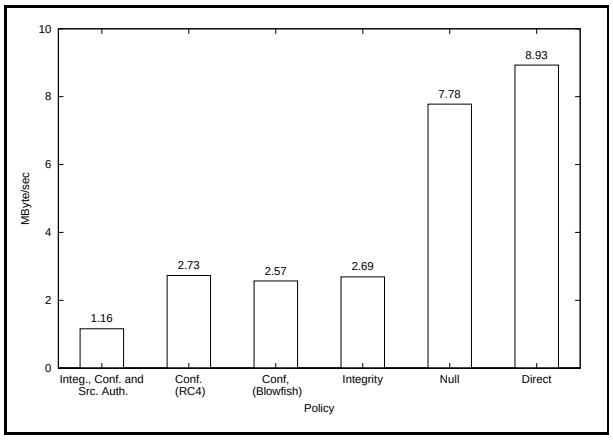

Figure 6: Throughput - maximum throughput under diverse data handling policies.

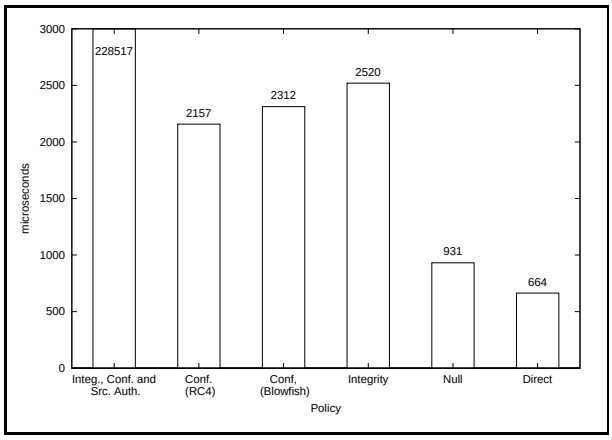

Figure 7: Latency - single message RTT under diverse data handling policies.

the network, throughput is limited by marshaling. This further highlights the need for optimization of the GMH service.

The integrity, confidentiality, and source authentication policy demonstrates the canonical strong multiparty data handling policy. Our experiments show that high data rates can be achieved through the application of stream signatures. Hence, the costs of strong data handling policies do not necessarily prohibit their use in high-throughput applications (e.g., conferencing).

Presented in Figure 7, the latencies associated with the experimental policies mirror throughput. The null and direct (differing by 10%), confidentiality and integrity policies (differing by at most 4%) exhibit similar latencies. Note that the latency of integrity, confidentially, and source authentication policy is dominated by a data forwarding timer used by the stream signature transform. This timer delays the packet transmission by 100 milliseconds in each direction, and hence, significantly affected the RTTs.

7 Alternative Architectures

While many aspects of the Antigone architecture are present in previous works, the unique requirements of policy enforcement made the direct use of existing component frameworks inappropriate. Centrally, the need to compose re-configurable and fine-grained components at run-time dictated the development of infrastructure not present in extant systems.

A number of recent works have investigated the construction of flexible and efficient distributed systems from components [11, 12, 36, 24]. Components conforming to uniform interfaces are composed in different ways to address application requirements. Hence, new requirements can be quickly addressed by altering the composition of underlying components. This approach has been successfully extended to security [37, 38, 39], where services and protocols addressing a specific set of security requirements are built from components. These works significantly constrain system organization; largely motivated by protocol stack designs, components are organized into vertical or hierarchical message processing pipelines. Hence, these frameworks are suitable for the creation of tightly coupled protocol state machines. Antigone, in contrast, composes loosely coupled services. Each mechanism transmits messages, processes timers, and monitors state independently of other services. Hence, the traditional model of layered services (e.g., TCP/IP, Cactus) is not often suited to the service composition offered by Antigone. Moreover, the interfaces over which state is communicated in traditional protocol component systems are typically restricted to connection management and data handling infor-

mation. Note that while these architectures are not well suited to Antigone, they may be useful in creating flexible implementations of individual mechanisms.

Typically used in the construction of complex distributed systems in heterogeneous environments, configuration programming frameworks specify component interfaces through a language-agnostic module interconnection language (MIL) [13, 22]. Developers construct distributed systems from MIL component interconnection specifications. The framework translates and routes all communication between the components defined by the developer. As these systems are designed to support communication between largely autonomous and distributed components, shared state is explicitly forbidden. In contrast, the mechanisms of Antigone are required to share a significant amount of state (e.g., keys, timers, attributes, etc.). Hence, the loose coupling and translation overheads often make these frameworks inappropriate for end-host policy enforcement.

Software buses have traditionally been used to construct distributed object architectures [14, 40, 34, 41, 42]. Components in these frameworks are typically used to define interfaces to database, compute, or user-interface services. Communication between components is handled via standardized marshaling interfaces. Hence, tool-kits of diverse components can be used to flexibly construct distributed systems. Components in these systems represent course-grained and possibly distributed services. Hence, the overheads associated with inter-component communication (i.e., marshaling and inter-process communication) are in conflict with the needs of high-performance policy enforcement.

8 Conclusions

This paper has presented an investigation of the requirements and machinery of general-purpose policy enforcement. The new Antigone 2.0 architecture adopts an expansive definition of policy encompassing both *provisioning* and *access control*. Provisioning policies direct the run-time composition of communication and security services. Subsequent action within the session is regulated through the enforcement of fine-grained and conditional access control policy. Note that Antigone does not mandate a policy representation. Hence, Antigone can be tailored to enforce precisely those aspects of session security within the scope of a policy specification.

Antigone is an event-based component system. Software mechanisms are composed as directed by policy. Relevant state changes are communicated via events broadcast over the event bus. Event interfaces allow the flexible composition of mechanisms required by provisioning policy. Antigone separates policy evaluation from enforcement. Hence, mechanisms are free from the complexities of assessing the context in which policy is enforced. We identified several optimizations addressing performance and development costs. Our performance experiments demonstrate that Antigone can support high throughput (8 megabyte/sec), low latency ($<$ 1 milliseconds) communication under real-world policies.

Antigone has been used to construct a number of non-trivial applications. For example, the AMirD multi-party file-system replication service efficiently synchronizes digital content among many hosts. The considerable resource requirements of file-system replication have allowed us to profile the performance of a number of interesting policies. Other efforts have investigated the transparent integration of Antigone with existing peer and group applications. These and other ongoing works will aid us in developing a deeper understanding of the applicability of Antigone, and ultimately of general-purpose policy enforcement.

References

[1] M. Blaze, J. Feigenbaum, and Jack Lacy. Decentralized Trust Management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173, November 1996. Los Alamitos.

[2] M Thompson, W. Johnson, S. Mudumbai, G. Hoo, K. Jackson, and A. Essiari.

Certificate-based Access Control for Widely Distributed Resources. In *Proceedings of 8th USENIX UNIX Security Symposium*, pages 215–227. USENIX Association, August 1999. Washington D. C.

[3] Sotiris Ioannidis, Angelos D. Keromytis, Steve Bellovin, and Jonathan M. Smith. Implementing a Distributed Firewall. In *Proceedings of Computer and Communications Security (CCS) 2000*, pages 190–199, 2000. Athens, Greece.

[4] P. McDaniel, A. Prakash, and P. Honeyman. Antigone: A Flexible Framework for Secure Group Communication. In *Proceedings of the 8th USENIX Security Symposium*, pages 99–114, August 1999.

[5] P. McDaniel and A. Prakash. Methods and Limitations of Security Policy Reconciliation. In *2002 IEEE Symposium on Security and Privacy*. IEEE, MAY 2002. Oakland, California, (to appear).

[6] T. Ryutov and C. Neuman. Representation and Evaluation of Security Policies for Distributed System Services. In *Proceedings of DARPA Information Survivability Conference and Exposition*, pages 172–183, Hilton Head, South Carolina, January 2000. DARPA.

[7] M. Blaze, J. Feignbaum, J. Ioannidis, and A. Keromytis. The KeyNote Trust Management System - Version 2. *Internet Engineering Task Force*, September 1999. RFC 2704.

[8] Y. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. REFEREE: Trust Management for Web Applications. In *Proceedings of Financial Cryptography '98*, volume 1465, pages 254–274, Anguilla, British West Indies, February 1998.

[9] Trevor Jim. SD3: A Trust Management System with Certified Evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, May 2001.

[10] M. Stevens, W. Weiss, H. Mahon, B. Moore, J. Strassner, G. Waters, A. Westerinen, and J. Wheeler. Policy Framework (*Draft*). *Internet Engineering Task Force*, September 1999. `draft-ietf-policy-framework-00.txt`.

[11] N.C. Hutchinson and L.L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1994.

[12] D. Schmidt, D. Fox, and T. Sudya. Adaptive: A Dynmaically Assembled Protocol Transformation, Integration, and eValuation Environment. *Journal of Concurrency: Practice and Experience*, 5(4):269–286, June 1993.

[13] J. Kramer. Configuration Programming - A Framework for the Development of Distributable Systems. In *Proceedings of IEEE International Conference on Computer Systems and Software Engineering (CompEuro 90)*, pages 374–384, May 1990. Tel-Aviv, Israel.

[14] H. Schulzrinne. Dynamic Configuration of Conferencing Applications using Pattern-Matching Multicast. In *Proceedings of 5th International Workshop on Net. and O.S. Support for Digital Audio and Video*, pages 231–242, 1995. Durham, New Hampshire.

[15] Jamie Jason, Lee Rafalow, and Eric Vyncke. IPsec Configuration Policy Model (*Draft*). *Internet Engineering Task Force*, November 2001. `draft-ietf-ipsp-config-policy-model-04.txt`.

[16] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen. Policy Core Information Model – Version 1 Specification. *Internet Engineering Task Force*, February 2001. RFC 3060.

[17] P. McDaniel. *Policy Management in Secure Group Communication.* PhD thesis, Univeristy of Michigan, Ann Arbor, MI, August 2001.

[18] J. Zao, L. Sanchez, M. Condell, C. Lynn, M. Fredette, P. Helinek, P. Krishnan, A. Jackson, D. Mankins, M. Shepard, and S. Kent. Domain Based Internet Security Policy Management. In *Proceedings of DARPA Information Survuvability Conference and Exposition*, pages 41–53, Hilton Head, South Carolina, January 2000. DARPA.

[19] D. Durham, J. Boyle, R. Cohen, S. Herzog, R. Rajan, and A. Sastry. RFC 2748, The COPS (Common Open Policy Service) Protocol. *Internet Engineering Task Force*, January 2000.

[20] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. *Internet Engineering Task Force*, November 1998. RFC 2401.

[21] J. Janotti, D. Gifford, K. Johnson, Kaashoek M, and O'Toole J. Overcast: Reliable Mutlicasting with and Overlay Network. In *4th USENIX Symposium on Operating System Design and Implementation (OSDI 2000)*, page (*to appear*). USENIX, October 2000. Santa Clara,CA.

[22] James M. Purtilo. The POLYLITH software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, January 1994.

[23] H. Harney, A Colegrove, E. Harder, U. Meth, and R. Fleischer. Group Secure Association Key Management Protocol (*Draft*). *Internet Engineering Task Force*, June 2000. `draft-harney-sparta-gsakmp-sec-02.txt`.

[24] Nina T. Bhatti, Matti A. Hiltunen, Richard D. Schlichting, and Wanda Chiu. Coyote: A System for Constructing Fine-Grain Configurable Communication Services. *ACM Transactions on Computer Systems*, 16(4):321–366, November 1998.

[25] M. Hiltunen and R. Schlichting. A Configurable Membership Service. *IEEE Transactions on Computers*, 47(5):573–586, May 1998.

[26] H. Harney and C. Muckenhirn. Group Key Management Protocol (GKMP) Specification. *Internet Engineering Task Force*, July 1997. RFC 2093.

[27] C. K. Wong, M. Gouda, and S. S. Lam. Secure Group Communication Using Key Graphs. In *Proceedings of ACM SIGCOMM '98*, pages 68–79. ACM, September 1998.

[28] D. McGrew and A. Sherman. Key Establishment in Large Dynamic Groups Using One-Way Function Trees. Technical Report TIS Report No. 0755, TIS Labs at Network Associates, Inc., May 1998. Glenwood, MD.

[29] R. Gennaro and P. Rohatgi. How to Sign Digital Streams. In *Proceedings of CRYPTO 97*, pages 180–197, August 1997. Santa Barbara, CA.

[30] The OpenSSL Group. OpenSLL, May 2000. `http://http://www.openssl.org/`.

[31] B. C. Neuman and T. Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications*, pages 33–38, September 1994.

[32] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98, 1994.

[33] Inc.Unix Man Page Sun Microsystem. rpcgen - An RPC Protocol Compiler.

[34] Steve Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 14(2), February 1994.

[35] Software Systems Research Lab, University of Michigan. Antigone Homepage, January 2002. `http://antigone.eecs.umich.edu/`.

[36] H. Orman, S. O'Malley, R. Schroeppel, and D. Schwartz. Paving the Road to Network Security or the Value of Small Cobblestones. In *Proceedings of the 1994 Internet Society Symposium on Network and Distributed System Security*, February 1994.

[37] M. Moriconi, X. Qian, R. A. Riemenschneider, and L. Gong. Secure Software Architectures. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 84–93, May 1997.

[38] P. Nikander and Arto Karila. A Java Beans Component Architecture for Cryptographic Protocols. In *Proceedings of 7th USENIX UNIX Security Symposium*, pages 107–121. USENIX Association, January 1998. San Antonio, Texas.

[39] M. Hiltunen, S. Jaiprakash, R. Schlichting, and Carlos Ugarte. Fine-Grain Configurability for Secure Communication. Technical Report TR00-05, Department of Computer Science, University of Arizona, June 2000.

[40] R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley and Sons, First edition, 1997. New York, NY.

[41] Jim Waldo. The Jini Architecture for Network-centric Computing. *Communications of the ACM*, 3(4):76–82, July 1999.

[42] Joerg Ott, Dirk Kutscher, and Colin Perkins. The Message Bus: A Platform for Component-based Conferencing Applications. In *Proceedings of CBG2000: The CSCW2000 workshop on Component-based Groupware*, December 2000. Philadelphia, PA.