Security Policy Enforcement in the Antigone System

Patrick McDaniel (*Contact Author*)
SIIS Laboratory
Pennsylvania State University
mcdaniel@cse.psu.edu

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

Abstract

Works in communication security policy have recently focused on general-purpose policy languages and evaluation algorithms. However, because the supporting frameworks often defer enforcement, the correctness of a realization of these policies in software is limited by the quality of domain-specific implementations. This paper introduces the Antigone communication security policy enforcement framework. The Antigone framework fills the gap between representations and enforcement by implementing and integrating the diverse security services needed by policy. Policy is enforced by the run-time composition, configuration, and regulation of security services. We present the Antigone architecture, and demonstrate non-trivial applications and policies. A profile of policy enforcement performance is developed, and key architectural enhancements identified. We conclude by considering the advantages and disadvantages of a broad range of software architectures appropriate for policy enforcement.

Keywords: Security Policy, Cryptographic Protocols, Distributed Systems, Component Systems, Software Architecture

1 Introduction

A security policy is *enforced* when its semantics are realized by software behavior. Enforcement can be as simple as the dropping of a packet by a firewall, or as complex as the execution of a leader election protocol in a secure group. How an application or service enforces policy has a direct affect on the security and efficiency of its operation.

Communication security policies have historically been crafted for the specific systems they support [60, 25]. The architects of these systems explicitly define the range of security behaviors desired. Therefore, security is addressed only inasmuch as the architects foresee the needs of its future users. Recent efforts within the security and policy communities have investigated general-purpose representations that vastly increase the scope of policy [7, 45, 11, 41], and hence address the needs of a much larger constituency. While these efforts have achieved many of the stated goals, they have not yet considered general-purpose policy enforcement.

This paper introduces the Antigone general-purpose communication security policy enforcement architecture. Antigone fills the gap between general-purpose representations and enforcement by defining a framework in which the diverse services required by policy can be easily implemented and integrated. We consider the number of systemic challenges presented by such an architecture, and show how arbitrary session-oriented policy representations can be enforced. The architecture of Antigone and its integration with several representative security services is detailed. The performance of Antigone is measured and discussed. We show that the per-message overheads are measured in the tens of micro-seconds, and the burst-rate bandwidth of network communication is reduced by less than 15%. The flexibility of the architecture is demonstrated through several applications and policies. We conclude by considering the trade-offs of other possible architectural designs.

Antigone does not implement policy-enforcing software, but provides APIs and an associated framework for its definition and use. Built upon these APIs, Antigone *mechanisms* are software components that implement security services. For example, one can implement the IPsec Encapsulating Security Payload (ESP) [24] data transform in an Antigone mechanism. Note that such services often require additional infrastructure to function. ESP is dependent on the cryptographic keys identified by a key management service. Furthermore, to be general, the implementation of ESP must inter-operate with arbitrary services supporting the key management function (e.g., Kerberos [37], IKE [15]). Antigone provides the framework in which these services can be implemented and their operation flexibly coordinated.

The term *security policy* has been used to represent many aspects of computer security [35, 59, 48, 7, 5, 3, 30]. This work considers the session-oriented provisioning and authorization policies associated with communication in distributed systems [29]. A provisioning policy defines the services and configurations used to implement security (e.g., cryptographic algorithms, data transforms). An authorization policy defines the rights of the participating parties. Antigone policies govern sessions. Antigone sessions are defined as continuous communication between two or more end-points. Antigone supports peer, multiparty, or broadcast communication protocols, and does not rely on any particular transport media (e.g., UDP, TCP, multicast). Coordinating the enforcement of the communication security policy over time and across end-points is the central purpose of Antigone.

Antigone is the first general-purpose provisioning and authorization policy enforcement framework. A central contribution of this work is the demonstration that policy representation can be separated from enforcement. Antigone's novel component architecture abstracts the specifics of security service operation from policy. Policy users can adopt new policies without cognizance of its affect on the enforcement infrastructure. This facility introduces a number of challenges. How the natural tensions between generality, simplicity and performance are weighed was a major factor in the design of Antigone, and its resolution a key contribution. We demonstrate the efficacy of our approach through the design and use of diverse applications, and show that Antigone-based policy services can address a wide range of environmental requirements.

The remainder of this paper considers how Antigone addresses the needs of general-purpose policy enforcement. We begin in the next section by considering the goals and requirements of Antigone.

2 Requirements and Goals

The design of Antigone is driven by a range of security and systems-oriented goals and requirements. We have identified the following goals for Antigone:

- *Security* - the semantics of policy must be correctly reflected in system behavior.
- *Flexibility* - the architecture must embrace a wide range of security services and authorization models.
- *Efficiency* - the overheads associated with the policy enforcement framework must not limit the application's ability to properly function.

- *Simplicity* - the exported interfaces must be only as complex is as necessary for the facilities they implement.

These goals present a number of challenges, and ultimately dictate a set of design requirements. Synthesized from these goals, the design of the Antigone architecture was built upon the following central design requirements.

Antigone must be *policy agnostic*. Any mandate of a particular policy representation would limit the environments in which Antigone can be used. Hence, it is important that the enforcement architecture is not dependent on any aspect of the policy language or its associated evaluation algorithms. Antigone is currently restricted to session oriented communication security policies. Exploration of other policy domains is left to future work.

The architecture must *run-time configured and governed*. The services required to enforce policy must be (and often can only be) identified and configured at run-time. Hence, the architecture must construct the correct security enforcement software based on run-time requirements. Moreover, the trust, means, and form of authorization must be evaluated within the run-time context. That is, the architecture must enforce run-time specified authorization policy.

Antigone must support the *easy development or integration and use* of security services. There are two important facets to this requirement. First, the architecture must provide easy to use interfaces for the rapid development or integration of security services. Second, the composition of security services is not always straightforward; how state dependencies are managed will directly affect both the efficiency and security of the solution. For example, data transform (e.g., payload encryption) services are often dependent on the state (e.g., keying material) of key management services. The architecture must support the efficient state sharing between service.

It is interesting to note that there is often tension between these goals and requirements, e.g., simplicity vs. efficiency, flexibility vs. easy service integration, etc. We discuss the affect these tensions played on the Antigone design in the following sections. First, however, we delve into the details of the definition and use of security policy within Antigone in the next section.

3 Policy

Antigone currently enforces communication security policy [30]. These policies define both session *provisioning* and *authorization*. Provisioning policies specify the services and parameters used to

implement communication security, and encompass any aspect of the security-relevant configuration of the session [29]. For example, the Secure Shell (SSH) [60] defines and enforces provisioning policy over remote user login sessions. Through local host configuration or command line parameters, each end-point defines the required methods of authentication (e.g., password, public key), the message transforms (e.g., SSH version 1, version 2), and the cryptographic algorithms (e.g., DES, 3DES) used to support the remote login.

Authorization policy states the actions governed entities are authorized to perform [4, 48, 49].[1] In Antigone, the authorization policy states explicitly the set of actions a session participant is permitted to perform. SSH specifies and enforces a simple, but illustrative, authorization policy by enumerating a set of acceptable public key credentials. Any entity that is successfully authenticated using these credentials is authorized to login to the system. The existence of the credential in the list acts as an implicit authorization policy.

Note that Antigone authorization policy supports much finer grained access control than that supported by SSH. Where necessary, session policy can stipulate control over fine-grained operations (e.g., data transmission, key agreement protocol initiation). Fine-grained access control is particularly useful in group sessions. For example, group authorities may wish to exert control over the session (e.g., dictate keys, control membership, eject misbehaving entities). Fine-grained access control is used extensively to control broadcast replication groups in the AMirD system (see Section 6.1).

A security policy is *evaluated* toward some policy decision. The chief difference between provisioning and authorization policy is the result of this decision making process. Provisioning policies define configurations and authorization policies determine whether some action should be allowed or denied. A policy is *enforced* when that decision is realized in behavior. This work focuses on how to perform enforcement flexibly, securely, and efficiently. We defer issues of evaluation to the other works in policy specification.

Antigone provisioning and authorization policies define the security-relevant properties, parameters, and facilities used to support a session. Policies state how security directs behavior, the entities allowed to participate, and the mechanisms used to achieve security objectives. This novel use of policy embraces dependencies between authentication, authorization, data protection, key management, and other facets of communication. The way Antigone manages these dependencies is detailed throughout and is a key contribution of this work.

[1] Authorization policy has been used synonymously with the term access control policy. We choose to use the former except where common use mandates otherwise (e.g., to refer to fine-grained access control).

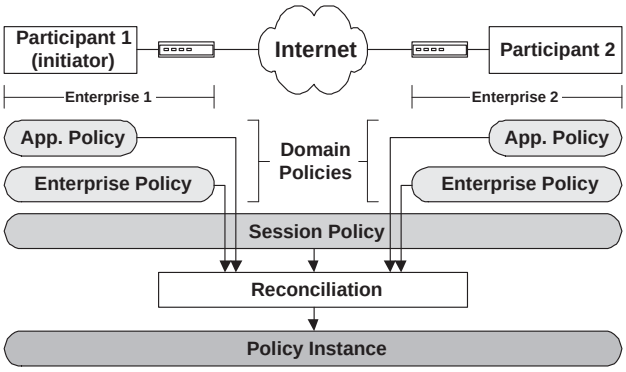

Figure 1: *Policy construction* - A session-specific policy instance is created by an initiator through the reconciliation algorithm. The instance is subsequently used to provision and regulate the session.

3.1 Policy Determination

Policy is developed in Antigone from the stated requirements of interested parities through a policy determination process. The result of determination is the concrete *policy instance* used to implement a session. Described below, Antigone currently uses the Ismene policy language and associated determination algorithms [30, 29]. Ismene is used throughout to motivate a discussion of enforcement, but is not required. Other policy languages (e.g., SPS [61], KeyNote [8]) may be integrated with Antigone as is necessary and desirable (see Section 4.3).

In Ismene, the policy instance enforced at run-time 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[2] 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.

Antigone session provisioning identifies the 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 authorization policy defined in the instance is represented as sets of rules. Each rule associates a protected *action* with a conjunction of positive conditionals. Authorization rules are consulted when an action is attempted, and the action is permitted where all conditions are satisfied.

[2]Often a session participant, an initiator is the policy decision point performing reconciliation [11].

```
% 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 authorization policies enforced at run-time.

Figure 2 illustrates a simplified policy instance appropriate for an IPsec [25] session. The provisioning policy states that three mechanisms, ESP (data transform), IKE (key and session management), and KA (failure detection), 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 IPsec security association key 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 authorization 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 use of the session key $\$sk$ is sufficient proof of authenticity, and hence the packet should be accepted. The second authorization rule, `init_session` defines when a session should be accepted. 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.

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).

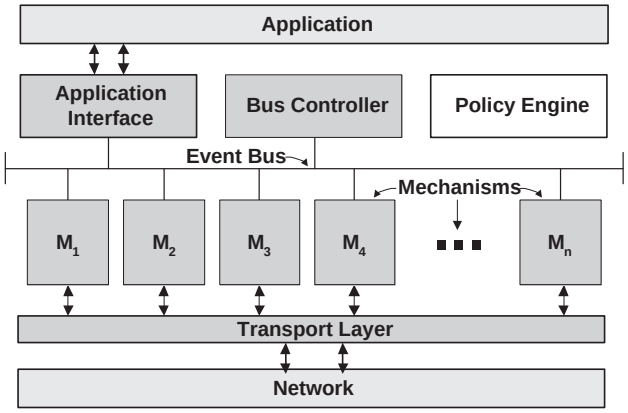

Figure 3: Architecture - the mechanisms, interfaces, and policy engine coordinate to enforce run-time determined policy.

Antigone is a single-threaded component architecture[3]. Communication between the infrastructure and software components is implemented by events. Applications transfer control to Antigone through socket-oriented calls (e.g., `send()`, `recv()`, `select()`). Application actions (e.g., send) are translated into events and delivered to all mechanisms. Policy is enforced by the mechanism reaction to and creation of events. Cascading events direct the progress of the session, and ultimately the application.

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. The decision to use an event architecture was primarily motivated by the need for flexibility. Mechanisms can implement complex interfaces through simple APIs by annotating events with custom data structures. This approach enables the service composition needed by diverse application policies.

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 security 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. The provisioning policy is enforced by provisioning the session. The policy engine directs provisioning by identifying and config-

[3]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.

uring the set of required mechanisms at session initialization. The policy engine indirectly regulates subsequent action within the session (e.g., connection initiation, message transmission) by evaluating authorization policy. Mechanisms appeal to the policy engine for authorization policy decisions via up-call. The design and use of the policy engine is presented in Section 4.3.

Persistent state is shared in Antigone through the *attribute set*. Similar to the KeyNote action environment [8], 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., a single interface for TCP, UDP, multicast, and simplified ad-hoc networks [23]). 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 authorization policies defined by the policy instance presented in Section 3.1. For this example, we assume that the session has been initialized (provisioned), and that an IPsec security association (SA) containing the IPsec configuration and session key 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

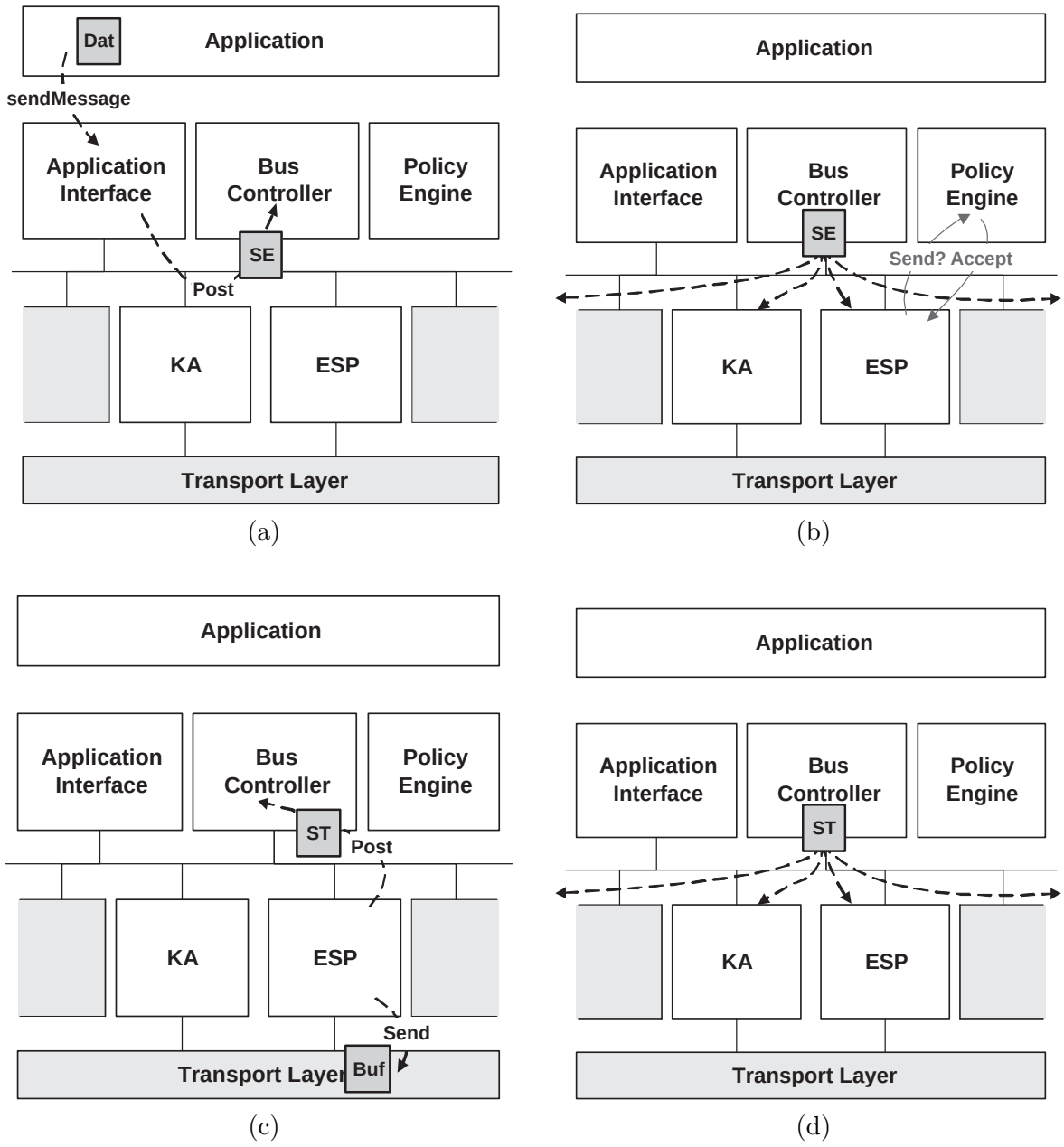

Figure 4: Policy Enforcement Illustrated - an application `sendMessage` API call is translated into a *send event* posted to the bus controller (a). The reception of the event by the ESP mechanism triggers the evaluation of the authorization policy via upcall (b), and ultimately to the transmission of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

relevant state (e.g., current session key, bytes to transmit, etc.) is passed to the policy engine, and used as input to the evaluation of the *send* authorization 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.

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

4.2 Event Bus

All events are delivered to every mechanism and the application interface via the event bus. 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., the data handler in a video-on-demand server).

The authors of the Polylith system [42] 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 [31]. However, due diligence must be expended in analyzing dependencies and incompatibilities in the use of events across mechanisms.

4.3 Policy Engine

The policy engine makes Antigone policy agnostic. The policy engine does not implement evaluation for a fixed policy representation, but provides a policy evaluation API [30]. All interpretation of policy occurs within an externally defined language-dependent implementation that adheres to the policy engine API. The Antigone enforcement infrastructure need not be aware of the mechanics of policy evaluation. All policy decisions are deferred to the policy engine through the API. Antigone currently uses the Ismene language to implement the policy engine. Policy engines built on 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 [8], GSAKMP [16]).

As directed by the run-time determined policy instance, the policy engine initially provisions the mechanism layer by the appropriate software mechanisms identified at run-time. 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 authorization policy over the lifetime of the session. Each mechanisms identifies the set of actions to be protected by policy in its 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 is granted where the relevant conditions are satisfied.

Mechanisms are free to define new protected actions as needed. However, governing policies must also define authorization rules for these actions. It is incumbent on the policy engine to decide what to do when an action is undertaken for which no authorization policy is defined. For example, Ismene implements a closed-world policy in which all such actions are denied.

Mechanisms supply information describing the context in which a particular action is attempted when appealing to the policy engine for an authorization 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 credentials, 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 a credential when assessing acceptance. We enumerate and discuss the set of actions supported by the current implementation in [30].

4.4 Mechanisms

An Antigone *mechanism* defines a basic service required by the session. Unlike traditional protocol objects in component protocol systems [50, 6], mechanisms are not vertically or hierarchically layered (e.g., X-kernel [20]). Note that this does not mandate that mechanisms implement monolithic or coarse-grained components. Each mechanism embodies an independent state machine, which itself may be layered. For example, the layered Cactus membership service [19] can be integrated within Antigone as a single mechanism.

A 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 [17], Authenticated Group Key Management [30], and Logical Key Hierarchy [58]. These categories are not exhaustive; new types (e.g., congestion control) or implementations (e.g., One-Way Function Tree key management [34]) 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 are notifications of internal state changes. An event 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

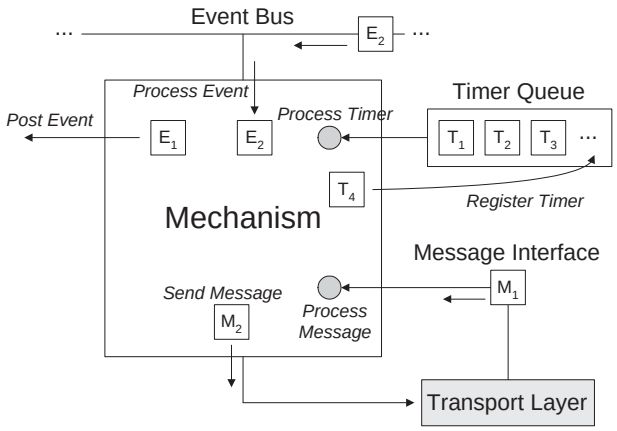

Figure 5: Mechanism Signal Interfaces - Policy is enforced through creation and processing of *events*, *timers*, and *messages*. Events are posted to and received via the event bus. Signals are processed and emitted through the various post, register, send, and processing interfaces.

or mechanism-specific; all mechanisms are notified at the expiration of a global timer, and a 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 Antigone 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. The ADH mechanism 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 the header of every received message to determine the transform and cryptographic algorithms under which it is transmitted. 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 can only be resolved by the governing policy. How a policy representation identifies and resolves such dependencies is a real and non-trivial problem. In Ismene, dependencies (and incompatibilities) are annotated by policy assertions that are validated at run-time.

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 [13]), 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 7.

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[4]. Antigone currently supports three authentication mechanisms: a null authentication mechanism (that exchanges keys and policy in the clear), an OpenSSL based mechanism [14], and a Kerberos mechanism [37]. 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 authorization policy used to initialize the requestor environment, and is discarded when the session-defining policy instance is acquired (see below). The policy engine 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[5]. If the action is accepted by the authorization policy, the mechanism obtains the policy instance and a long-term *pair key* over the SSL-secured channel. The pair key is a symmetric key shared by the local requester and the initializer, and is used to create a secure channel between these two entities. 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 terminates session.

[4]We describe a more flexible model where policy is determined during session initialization in [30].

[5]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 [9]).

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 authorization policy (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 [54], CORBA [55]) provided excellent facilities for the construction of plain-text 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:

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

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[...]`). GMH 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 Applications

The value of Antigone is determined by the kinds of applications and policies it supports. This section sketches several complex applications that illustrate the range and flexibility of Antigone. Because past development has primarily focused on multiparty systems, we restrict the following to group applications. The policies described below were developed for the specified environments. As is the purpose of Antigone, other policies addressing other security requirements may be defined as

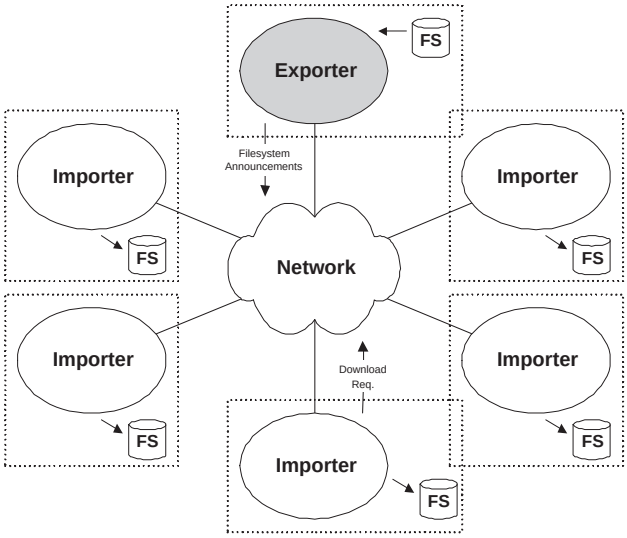

Figure 6: *AMirD* - Built on Antigone, the AMirD application securely and efficiently replicates filesystem content to importers over a broadcast media.

is desirable.

6.1 AMirD

The Antigone MIRroring Daemon (AMirD) replicates filesystem content to groups of receiver agents over a broadcast media [33]. The broadcast media allows large groups to quickly receive content and has obvious applications to, for example, patch management and website mirroring. AMirD replication operates in two phases: consistency control and content replication. The consistency control phase identifies the state of the replicated filesystems. The replication phase distributes file content. The control phase is governed by a single policy instance. Because files have different sensitivity levels and access restrictions, a policy is determined and a new group established for each file distribution in the replication phase.

Illustrated in Figure 6, an AMirD session consists of a collection of one or more exporters and one or more importers. Each exporter periodically broadcasts the state of the exported filesystem (i.e., file and directory names, modification dates, MD5 hashes of files). Importers request updates for all files that are out of date or missing from its local mirror. Subgroups are established for each file update (up to maximum number of simultaneous subgroups) based on a policy and file content transmitted.

The policies appropriate for AMirD are as diverse as the environments in which it can be used. We have extensively studied the AMirD policies for four example environments [33]. These environments

represent very different views of security. Some policy requirements were shared by all the AMirD environments. For example, all environments required some way to securely manage and communicate group membership. Conversely, each environment required a different authentication mechanism. A secure group membership mechanism was implemented. This mechanism uses the identities identified by a separate authentication mechanism to drive the secure group membership protocols.

We briefly discuss the policy enforced by AMirD in two of the studied environments. Because the environment is largely trusted, the *local LAN environment* required few protections. Adequate enforcement of local user identification and optionally confidentiality of the file content was deemed sufficient. This required the development of a new UNIX authentication mechanism and used a simple data handler. In essence, the LAN environment needed a few lightweight security services.

The *mobile user environment* represents the canonical road warrior scenario. A local user mirrors data from a remote service over an untrusted network. This environment requires strong authentication (certificates) and authorization, as well as integrity of the control and file content. The environment further required content sender authenticity (to combat content forgery by compromised mobile users). However, because the costs of signing each packet were prohibitive, we use stream signatures to the data handler (See Section 4.5).

The mobile user scenario also required dynamic policy enforcement. Any file (or filesystem) could be marked as instantaneously sensitive in response to some heightened threat level. The infrastructure would react by altering the data handler configuration of related sessions by adding a confidentiality guarantee. The data handler would then encrypt all application layer traffic passing between the end-points, in addition to applying the other integrity and sender authenticity transformations.

AMirD demonstrates the flexibility of Antigone. We implemented and tested all environments using the same compiled code. Because new policy requirements can addressed without modifying the application, developers can largely defer security issues to policy.

6.2 Legacy Systems Security

It is often difficult to retrofit security into legacy systems. For example, we found that adding security to the `vic` and `vat` conferencing tools required major modification to the architecture and code [1]. The `Socket_s` library addresses this difficulty by providing a socket API to an Antigone multicast group [32]. Applications use this library by replacing existing socket calls (e.g., *socket*, *bind*, etc.) with similarly named `Socket_s` calls (e.g., *socket_s*, *bind_s*, etc.). The functions redirect operations associated with multicast traffic to Antigone, and non-multicast traffic to the standard *C* library

implementations.

`Socket_s` acts as a "bump in stack" by inserting Antigone between the application and the standard network interfaces. The Antigone implementation executes in a separate thread and establishes a socket with the parent application. Hence, Antigone acts a transparent proxy for all multicast traffic, and enforces security policy over data sent through the multicast channel.

The policies appropriate for `Socket_s` are simply those needed by secure multicast [28, 10]. Key management and data handling are the essential policy requirements in this domain. Because of the wide possible range of supported applications and environments, we implemented several key management strategies. For example, our Logical Key Hierarchy [57] mechanism offers a highly efficient approach for group key management. Because it was already highly flexible, we used the existing data handler.

6.3 Reliable Multicast

Reliable and secure multicast systems have been studied for some time [43, 2]. The security services supported by these systems are largely fixed for a set of target environments. Conversely, the Reliable Transport Layer (RTL) [32] provides reliable (FIFO) delivery of application traffic under a policy-defined Antigone security service. RTL uses a combination of Forward Error Correction (FEC) and the approach used in the Scalable Reliable Multicast (SRM) protocol [12] to detect and recover from lost packets.

RTL inserts two protocol layers between the application and Antigone. The FEC layer transmits $p + q$ forward-error correcting packets [44], where p is the number of original packets and q is the desired redundancy. Receivers receiving any p of the packets can recover the original data. The SRM layer uses a local recovery to increase the reliability of the group communication. Members detecting lost packets use TTL-limited and randomly delayed broadcasts to request retransmissions.

RTL simply increases the security on multicast groups, and hence simple solutions for authorization, key management, and data handling are often sufficient. However, the RTL service itself requires policy; an RTL application can use FEC, SRM, or both. The selection of protocols and their configuration (e.g., amount of redundancy, retransmit request strategy) are directly specified in the policy instance. RTL probes the Antigone policy instance for the appropriate configuration during initialization, and appeals to the application for direction where a configuration is not specified.

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%
Authorization	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.

7 Performance Evaluation

This section investigates the performance of the Antigone architecture by profiling enforcement overhead (microbenchmarks) and characterizing communication throughput and latency (macrobenchmarks). The current implementation, version 2.0, represents a complete redesign of the original system [31]. Antigone 2.0 consists of 58,000 lines of C++ code in 133 classes (approximately 10% of which was retained from the original Antigone architecture). Source code and documentation for Antigone, the Ismene policy language, and applications are freely available from the Antigone website [53].

The experiments described in this section 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.

7.1 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*, *authorization*, and *buffer management and queuing*. All costs not specific to Antigone were removed from the test measurements (e.g., encryption). 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 receive 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

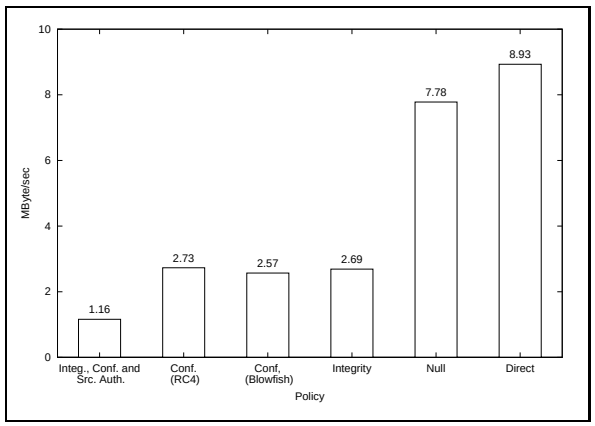

Figure 7: Throughput - maximum throughput under diverse data handling policies.

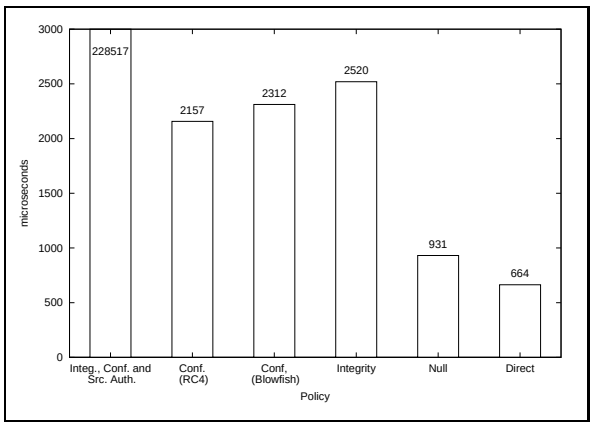

Figure 8: Latency - single message RTT under diverse data handling policies.

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 further 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 authorization 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.

7.2 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 [13], 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 7, 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[6]. Because the cryptographic algorithms are significantly faster than 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 8, 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.

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

8 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 existing systems.

A number of recent works have investigated the construction of flexible and efficient distributed systems from components [20, 50, 39, 6]. 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 [36, 38, 18], 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, is composed of 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, in traditional protocol component systems the interfaces over which state is communicated are typically restricted to connection management and data handling information. Note that while these architectures are not well suited to Antigone, they may be useful in creating flexible implementations of individual mechanisms.

Configuration programming frameworks specify component interfaces through a language-agnostic module interconnection language (MIL) [27, 42]. 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 [51, 52, 55, 56, 40]. Components in these frameworks are typically used to define interfaces to database, computing, or user-interface services. Communication between components is handled via standard-

ized marshaling interfaces. Hence, tool-kits of diverse components can be used to flexibly construct distributed systems. Components in these systems represent coarse-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.

The GAA-API system [46] provides a framework for evaluating and enforcing authorization policy. GAA-API extends the traditional notion of authorization policy enforcement in diverse applications (e.g., web server [47]) through *advanced security policies* that specify conditions that must be true prior to, during, or after, or in response to a positive or negative evaluation. These conditions can be both active (e.g., implement some operation) and stateful. As in Antigone, the management of state present in GAA-API presents challenges. For example, the introduction of stateful condition implementations requires careful policy specification to avoid non-deterministic evaluation. GAA-API differs from Antigone in purpose and scope. GAA-API serves to extend authorization policy enforcement behavior, but does not support the coordination of session provisioning. However, the service of GAA-API could be used to enhance the Antigone authorization policy enforcement (i.e., used in conjunction with a provision policy language to implement a policy engine).

The STRONGMAN [26] shares many goals with Antigone. Both support a wide range of policies over diverse applications. STRONGMAN provides facilities for the specification and automated management of security policies. However, unlike Antigone, the policy enforcement software is largely developed for each application independently [22, 21]. Hence, the focus of these two systems is quite different.

9 Conclusions

This paper has introduced the Antigone general-purpose session security policy enforcement architecture. Antigone fills the gap between the general-purpose policy representations and the often functionally limited enforcement systems that use them. This work further explores the separation of policy representation from enforcement. We remove the barriers placed on policy by its representation. Users can employ new or extended policies without requiring changes to the underlying enforcement framework. However, this approach introduces a number of complexities. How Antigone addresses the tensions between generality, performance, and simplicity is the driving force behind its design.

The Antigone architecture addresses all of its primary design requirements. Antigone is policy

representation agnostic. The policy engine interface abstracts the evaluation of policy away from the enforcement mechanisms. Mechanisms do not need to be aware of any facet of the representation, and can be used to enforce the policies of many different representations. Session security requirements often can only be determined at run-time. Antigone constructs a security infrastructure at run-time from the available security mechanisms as directed by the policy engine. Authorization policy is subsequently evaluated and enforced throughout the session lifetime. We have further shown the ease with which existing services can be integrated and new ones quickly developed. The event API allows services to quickly build state machines that interact with the infrastructure and other mechanisms.

We have demonstrated how Antigone has achieved the goals of security, flexibility, efficiency, and simplicity. AMirD and the other presented applications demonstrate how to simply and securely build on Antigone enforced policy. The polices used by those applications, and indirectly the mechanisms that those policies require, further illuminate the broad range of Antigone enforcement. The performance analysis demonstrates efficiency, and shows that the costs of using Antigone are nominal.

Much of the future of security lies in policy. The security requirements of current and future environments are incompatible with the largely fixed security models embodied in current software systems. Security policy addresses this incompatibility by allowing users and administrators, rather than developers, to state what security should be applied to the environment. Because security requirements are as diverse as the environments in which systems exist, support for flexible policy-defined security is needed. Antigone and other works like it are actively pursing these goals, and their ultimate success or failure will in some part define the security of future distributed systems.

10 Acknowledgments

We would like to thank Sharad Mittals, and Thai-Chuin Thuang with their work on the Antigone applications, and to Jim Irrer for his help building security mechanisms. We would like to also thank Peter Honeyman for his many contributions to the Antigone project, and to Kevin Butler, Trent Jaeger, Sugih Jamin, Paul Resnick, Avi Rubin and the anonymous reviewers for their many thoughtful comments.

This work is supported in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0508. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions

contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government. This work is also supported in part by the National Science Foundation under grant number 088285.

References

[1] A. Adamson, C.J. Antonelli, K.W. Coffman, P. D. McDaniel, and J. Rees. Secure Distributed Virtual Conferencing. In *Proceedings of Communications and Multimedia Security (CMS '99)*, pages 176–190, September 1999. Katholieke Universiteit, Leuven, Belgium.

[2] Yair Amir, Giuseppe Ateniese, Damian Hasse, Yongdae Kim, Cristina Nita-Rotaru, Theo Schlossnagle, John Schultz, Jonathan Stanton, and Gene Tsudik. Secure Group Communication in Asynchronous Networks with Failures: Integration and Experiments. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, pages 330–343, April 2000.

[3] Yair Bartal, Alain J. Mayer, Kobbi Nissim, and Avishai Wool. Firmato: A novel firewall management toolkit. In *IEEE Symposium on Security and Privacy*, pages 17–31, 1999.

[4] D. Bell and L. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report M74-244, MITRE Corporation, Bedford, MA, 1973.

[5] S. Bellovin. Distributed Firewalls. *;login:*, pages 39–47, 1999.

[6] 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.

[7] 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.

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

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

[10] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast Security: A Taxonomy and Efficient Constructions. In *Proceedings of IEEE Infocom 1999*, volume 2, pages 708–716. IEEE, March 1999. New York, New York.

[11] 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.

[12] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking*, pages 784–803, December 1997.

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

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

[15] D. Harkins and D. Carrel. The Internet Key Exchange. *Internet Engineering Task Force*, November 1998. RFC 2409.

[16] 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`.

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

[18] 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.

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

[20] 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.

[21] John Ioannidis, Sotiris Ioannidis, Angelos D. Keromytis, and Vassilis Prevelakis. Fileteller: Paying and Getting Paid for File Storage. In *Proceedings of Financial Cryptography 2002*. International Financial Cryptography Association (IFCA), March 2002. Bermuda.

[22] 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.

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

[24] S. Kent and R. Atkinson. IP Encapsulating Security Payload (ESP). *Internet Engineering Task Force*, November 1998. RFC 2406.

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

[26] Angelos D. Keromytis, Sotiris Ioannidis, Michael B. Greenwald, , and Jonathan M. Smith. The STRONGMAN Architecture. In *Proceedings of Third DARPA Information Survivability Conference and Exposition (DISCEX III)*. IEEE, April 2003. Washington, D.C.

[27] 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.

[28] P. Kruus. A Survey of Multicast Security Issues and Architectures. In *"Proceedings of 21st National Information Systems Security Conference"*, October 1998. Arlington, VA.

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

[30] P. McDaniel and A. Prakash. Methods and Limitations of Security Policy Reconciliation. In *2002 IEEE Symposium on Security and Privacy*, pages 73–87. IEEE, MAY 2002. Oakland, CA.

[31] 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. Washington, DC.

[32] P. McDaniel, A. Prakash, J. Irrer, S. Mittal, and T. Thuang. Flexibly Constructing Secure Groups in Antigone 2.0. In *Proceedings of DARPA Information Survivability Conference and Exposition II*, pages 55–67. IEEE, June 2001. Los Angeles, CA.

[33] P. McDaniel and Atul Prakash. Securing Distributed Applications Using a Policy-based Approach. Technical Report TD-5UDKVD, AT&T Labs - Research, Florham Park, NJ, December 2003.

[34] 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.

[35] J. McLean. The Specification and Modeling of Computer Security. *IEEE Computer*, 23(1):9–16, January 1990.

[36] 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.

[37] B. C. Neuman and T. Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications*, 32(9):33–38, September 1994.

[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] 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.

[40] 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.

[41] G. Patz, M. Condell, R. Krishnan, and L. Sanchez. Multidimensional Security Policy Management for Dynamic Coalitions. In *Proceedings of Network and Distributed Systems Security 2001.* Internet Society, February 2001. San Diego, CA, (*to appear*).

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

[43] M. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *Proceedings of 2nd ACM Conference on Computer and Communications Security*, pages 68–80. ACM, November 1994.

[44] L. Rizzo. Effective Erasure Codes for Reliable Computer Communication Protocols. *ACM Computer Communications Review*, 27(2):24–36, April 1997.

[45] 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.

[46] Tatyana Ryutov and Clifford Neuman. The Specification and Enforcement of Advanced Security Policies. In *Proceedings of the Conference on Policies for Distributed Systems and Networks (POLICY 2002)*. ACM, June 2002.

[47] Tatyana Ryutov, Clifford Neuman, Dongho Kim, and Li Zhou. Integrated Access Control and Intrusion Detection for Web Servers. *IEEE Transactions on Parallel and Distributed Systems*, 14(9), September 2003.

[48] Ravi S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, 1993.

[49] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 20(2):38–47, 1996.

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

[51] 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.

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

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

[54] Inc.Unix Man Page Sun Microsystems. rpcgen - An RPC Protocol Compiler, 1988.

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

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

[57] Debby M. Wallner, Eric J. Harder, and Ryan C. Agee. Key Management for Multicast: Issues and Architectures. *Internet Engineering Task Force*, June 1999. RFC 2627.

[58] 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.

[59] T. Woo and S. Lam. Authorization in Distributed Systems; A New Approach. *Journal of Computer Security*, 2(2-3):107–136, 1993.

[60] Tatu Ylonen. SSH - Secure Login Connections Over the Internet. In *Proceedings of 6th USENIX UNIX Security Symposium*, pages 37–42. USENIX Association, June 1996. San Jose, CA.

[61] 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.