A Flexible 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

Significant progress has been made on the design of security policy representations for complex communication systems. A significant problem however remains – how to design software architectures that enforce ever-changing security policy requirements efficiently. This research summary describes the security policy enforcement architecture of the Antigone 2.0 the group communication system. The architecture is designed to be flexible: new security mechanism modules are added as needed to support emerging policy requirements. Such mechanisms regulate the processing of system and network events as directed by the policy and enforce fine-grained control over sensitive data. A software bus is used coordinate the delivery of these events to mechanisms within each process. We summarize an analysis of the performance of the architecture and show that the overheads are modest for typical environments.

1 Introduction

Policy languages are increasingly being used in different contexts as a vehicle for representing authorization and access control [1, 2, 3, 4, 5, 6] and peer session security [7] requirements. These works focus primarily on policy specification, rather than enforcement, leaving the problem of determining how best to enforce the specified policy up to applications or infrastructure. In contrast, this summary examines the complementary problem of *flexible enforcement* – how to enforce security policy efficiently through a modular composition of appropriate security mechanisms. The primary domain of this work is group communication systems that need to enforce wide-ranging and fluid security policies, for example, as might occur in missions involving dynamically formed coalitions.

The tangible result of our investigation of flexible policy enforcement is the Antigone the group communication system (or just *Antigone* throughout). Antigone implements multi-party communication security policy. The supported *session policies* define the security-relevant properties, parameters, and facilities used to support a group session. Thus, a session policy states how security directs behavior, the entities allowed to participate, and the mechanisms used to achieve security objectives. This broad definition extends many policy approaches: dependencies between authorization, access control, data protection, key management, and other facets of communication are represented in a unifying policy.

As part of the Antigone project, we have explored more general issues of multi-party policy determination and representation in depth in the Ismene system [8]. Centrally, our work in Ismene identified the theoretical limits of policy reconciliation and compliance checking. Our central result showed that reconciliation (and compliance evaluation) in unrestricted policy representations is intractable. In that same work, we identified heuristic algorithms that perform reconciliation in restricted policy languages. Throughout, we defer issues of policy determination to that work.

The Antigone architecture is informed not only by studies in security policy, but in, among many works, the construction of flexible software systems and policy driven applications. In particular, component systems [9, 10] exhibit a characteristic highly advantageous to policy enforcement: the ability to construct complex implementation from service specifications. However, the restrictions placed on the organization, interfaces, and state maintenance of components made the direct application of existing architectures (e.g.,

*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.

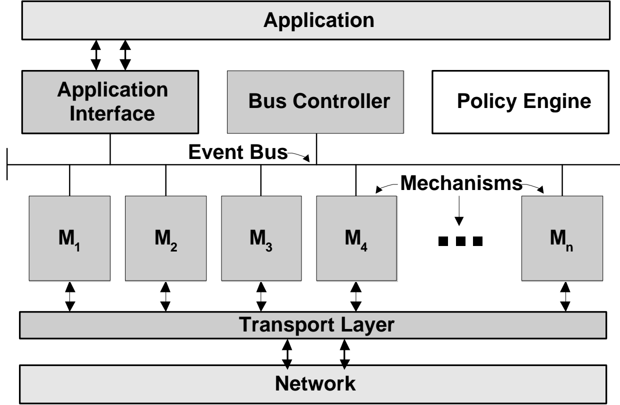

Figure 1: Antigone Enforcement - the mechanisms, interfaces, and policy engine coordinate to enforce session policy.

the X-kernel) to policy enforcement difficult. While building upon these works, Antigone necessarily allows a tighter coupling of events and session state than present in existing frameworks.

Built on the KeyNote [11] trust management system, the STRONGMAN system is used to construct low level policies from high level specifications [12]. This allows policy issuers to be concerned with the meaning of policy, while deferring the complexities to the STRONGMAN environment. This separation of meaning from enforcement allows new environments to be deployed without affecting policy. STRONGMAN does not provide a general-purpose enforcement infrastructure, and is used to support policies in fixed domains (e.g., firewalls [12]).

Security Policy System (SPS) [7] defines a framework for the specification and reconciliation of security policies. SPS focuses on the correct distribution and evaluation of IPsec policies [13]. In SPS, policy databases warehouse and distribute specifications to policy clients and servers. Policy servers coordinate the interpretation, negotiation, and enforcement of SA policies. However, because SPS governs a fixed enforcement infrastructure, their application to more general environments is area of future work.

The lessons learned from these works, as well as investigations of representations and applications were used to guide the design of Antigone [14]. The remainder of this summary briefly describes the design of Antigone is its subsequent evaluation.

2 Antigone

This section presents a brief overview of the Antigone architecture, illustrates enforcement by example in the following subsection, and concludes in Section 2.2 by identifying several key architectural optimizations. Depicted in Figure 1, the Antigone framework 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 event-based component architecture. Applications built on Antigone transfer control to the framework through socket-oriented calls (e.g., `send()`, `recv()`, `select()`). The application actions signaled by API calls are translated into events and delivered to all mechanisms. Policy is enforced by the mechanism processing of these events. Cascading events direct the progress of the session, and ultimately the application.

A *mechanism* is a policy implementing software component. The mechanisms used to implement the session are identified at run-time by the session policy. Each mechanism is implements some security (e.g., authentication, key management) or other functional aspect of the session (e.g., auditing, failure detection and recovery). Events are ordered and broadcast to mechanisms and the application interface via the *event bus*. The *policy engine* acts is a policy decision point. Enforcement is defined in two distinct phases; provisioning and authorization. A session is provisioned by configuring the set of mechanisms as defined by the session policy. Subsequent action is regulated by the policy engine through the evaluation of authorization policy (i.e., fine-grained access control). Note that Antigone is policy representation-agnostic: any representation able to correctly and completely specify provisioning and authorization policy can be used (e.g., Ismene).

Session state is maintained in the *attribute set* (not shown in figure). Similar to the KeyNote action environment [11], the attribute set is a table of typed attributes. Attributes are defined by {name, type, value} tuples representing a base value (e.g., strings, integers), an identity (e.g., unique identifier), or a credential (e.g., keys, certificates). These attributes are used by mechanisms to interpret events, and by the policy engine to evaluate policy.

The *application interface* arbitrates communication between the application and Antigone by implementing the socket-oriented API calls. 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). For brevity, we omit further details of the application interface and transport layers.

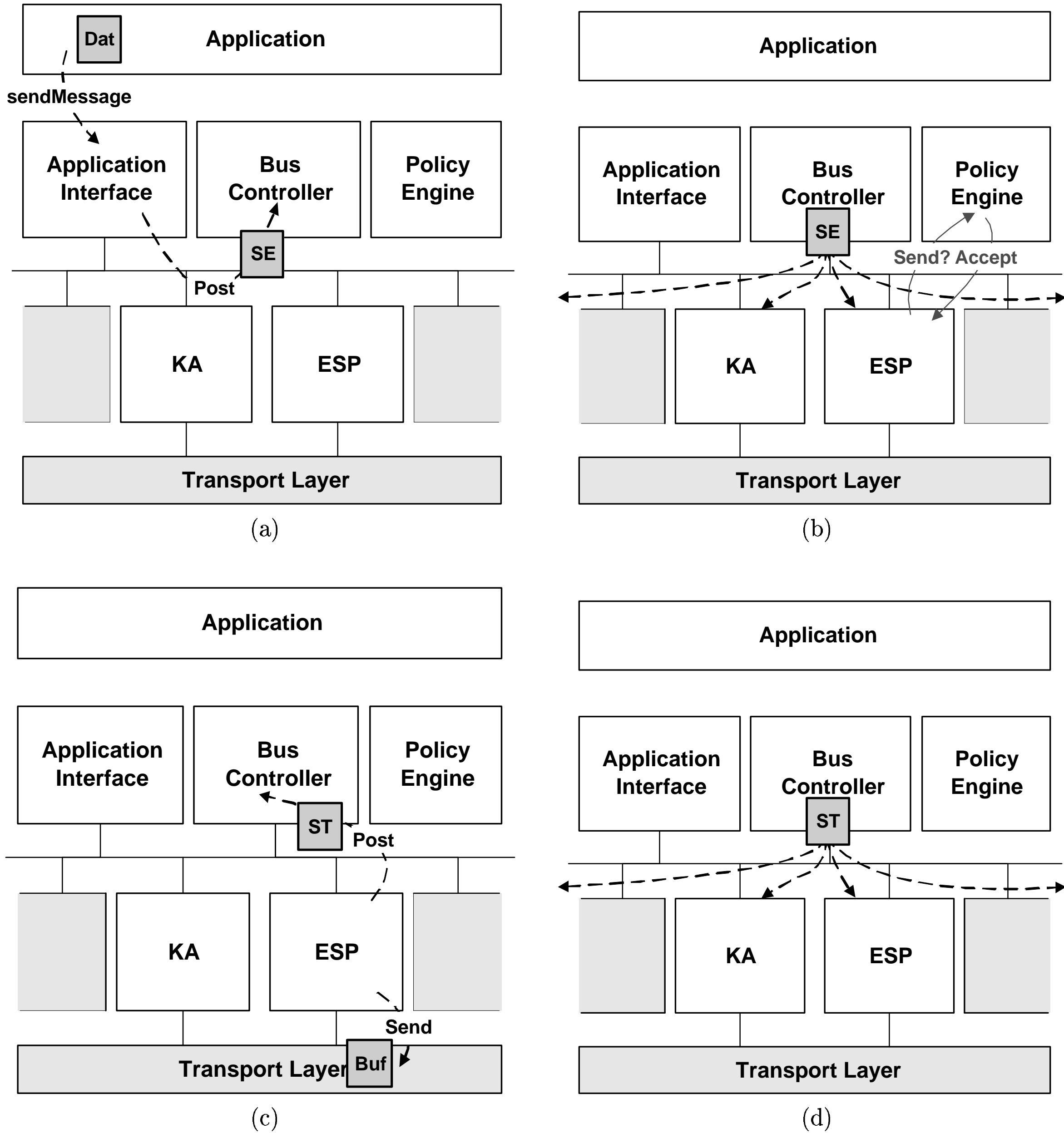

Figure 2: Policy Enforcement Illustrated - an application `sendMessage` API call is translated into a *send event* posted to the bus controller (a). The reception of the event by the ESP mechanism triggers the evaluation of the access control policy via upcall (b), and ultimately to the transmission of transformed data (c). The transmission triggers further event generation and processing (d).

2.1 Policy Enforcement Illustrated

The following example illustrates Antigone through the enforcement of an IPsec session policy [13]. For ease of exposition, we illustrate a peer IPsec session. Note that a multi-party session would operate similarly, e.g., with an MESP mechanism used instead of ESP, etc. We assume that the session has been initialized (provisioned), and that a session key has been negotiated by an IKE mechanism (i.e., an SA has been established). The following text and Figure 2 describes policy enforcement during the transmission of a application message (where the letters a, b, c and d correspond to the labeled figures):

a) An application initiates a message transmission through 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.

c) ESP selects a data transform dictated by policy (i.e., 3des, hmac-md5), and transforms the data (e.g., encrypts). The transformed buffer is sent to the other session participants via the transport layer. An `EVT_SENT_MSG` (ST) event containing the buffer is posted to the bus controller following 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 of those implementations necessary to address session requirements.

2.2 Optimizing Enforcement

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.

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 policy 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 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 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 policy evaluation by accessing cached results.

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 addresses the complexities of protocol development by providing simple string oriented specifications. Message specifications are interpreted (and possibly constructed) 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, CORBA) 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 and other complex transform features were required by multi-party key management and source authentication protocols, and thus mandated additional services.

3 Performance Evaluation

This section summarizes an investigation of the performance of the Antigone architecture [14]. This study profiled low-level enforcement costs and characterized communication throughput and latency. All experi-

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.

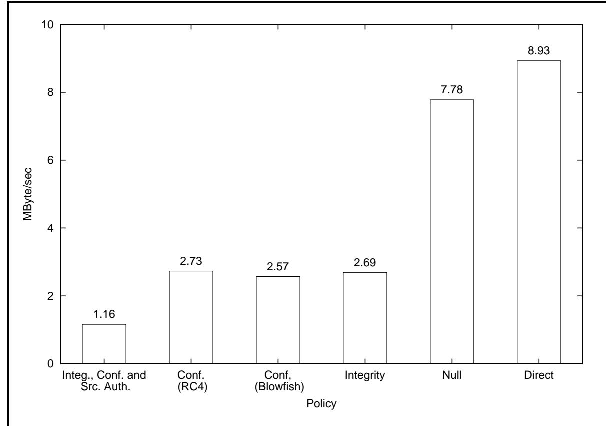

Figure 3: Throughput - maximum throughput under diverse data handling policies.

ments 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 ran RedHat Linux 7.1, kernel 2.2.14-5.

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 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 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. Due to interrupt processing, the `send()` system call takes approximately twice as long as `recv()`.

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 caching can mitigate the cost of fine-grained access control. 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).

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. 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 multi-party communication, the *integrity, confidentiality, and source authentication* policy specifies SHA-1 HMACS, Blowfish encryption, and 1024-bit RSA stream signatures.

As presented in Figure 3, 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. As the cryptographic algorithms are significantly faster than the network, throughput is limited by marshaling. The integrity, confidentiality, and source authentication policy demonstrates the canonical strong multi-party data handling policy. Our experiments show that high data rates can be achieved through the application of stream signatures.

Not shown, the latencies associated with the exper-

imental 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/source authentication policy is dominated by a data-forwarding timer used by the stream signature transform.

4 Conclusions

While technologies supporting the representation and manipulation of security policy have seen significant advances in recent years, issues of policy enforcement has not yet received due consideration. The Antigone 2.0 system begins to address this concern by providing a modular software architecture that efficiently enforces wide range of security policy requirements. Antigone is designed to be flexible – new security mechanisms can be freely added as needed to support new policy requirements for a given session. The mechanisms communicate with each other and the policy engine via a software event bus. A key advantage of this architecture is that mechanisms are not required to be layered as in component-based communication systems, thus considerably simplifying mechanism interaction. Our performance analysis shows that the overheads of the Antigone architecture are modest for a range of communication policies, and that the costs of fine-grained access control need not be excessive.

Antigone currently 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:

`http://antigone.eecs.umich.edu/`

References

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

[2] 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.

[3] L. Cholvy and F. Cuppens. Analyzing Consistancy of Security Policies. In *1997 IEEE Symposium on Security and Privacy*, pages 103–112. IEEE, May 1997. Oakland, CA.

[4] 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, February 1998.

[5] T. Woo and S. Lam. Designing a Distributed Authorization Service. In *Proceedings of INFOCOM '98*, San Francisco, March 1998. IEEE.

[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. DARPA, January 2000.

[7] 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. DARPA, January 2000.

[8] 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).

[9] 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.

[10] 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.

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

[12] 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.

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

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