A Paradigm For User-Defined Security Policies

Winfried E. Kühnhauser
German National Research Center For Information Technology (GMD)
Schloss Birlinghoven, D-53754 Sankt Augustin, Germany
e-mail: winfried.kuehnhauser@gmd.de

Abstract

One of today’s major challenges in computer security is the ever-increasing multitude of individual, application-specific security requirements. As a positive consequence, a wide variety of security policies has been developed, each policy reflecting the specific needs of individual applications. As a negative consequence, the integration of the multitude of policies into today’s system platforms made the limitations of traditional architectural foundations of secure computer systems quite obvious.

Many of the traditional architectural foundations originally aimed at supporting only a single access control policy within a single trusted system environment. This paper discusses a new paradigm to support user-defined security policies in a distributed multi-policy system. The paradigm preserves the successful properties of the traditional architectural foundations while additionally providing strong concepts for user-defined security policies. Among these concepts are policy separation, encapsulation, persistency, cooperation, and reusability. We illustrate the application of our approach in a DCE environment.

Keywords: Security, Confidentiality, Integrity, Access Control, Reference Monitor, Application-Dependent User-Defined Security Policies, Multi-Policy Environments.

1 Introduction

One of the key foundations of a secure computer system is the security policy. A security policy is a set of laws, rules, and practices that regulate how an organization manages, protects, and distributes sensitive information [Dep83]. The economical, social, and technical progress in the last decade has confronted us with higher and more individual security requirements on our information processing systems. Security policies are now emerging that support the needs of individual organizations and applications, with the Bell/LaPadula model (providing confidentiality) being only the first in an ever-increasing multitude [BL76, Bib77, CW87, BN89, San92]. As a consequence, the development of application dependent security policies and their integration into distributed systems has become a major challenge in computer security.

Several security policies in modern applications address a wide variety of aspects of system security, such as access control, authentication, availability and auditing. This variety requires concepts in the underlying system platform that go beyond the limits of traditional architectural foundations of secure computer systems. One of the most prominent examples is the reference monitor concept defined in the Trusted Computer System Evaluation Criteria (TCSEC) [Dep83]. The reference monitor is an access control concept defining properties of an abstract machine that enforces an access control policy. Any reference monitor implementation must

(a) mediate all accesses of subjects to objects within a system,
(b) be tamperproof,
(c) be small and well-structured to allow for a formal analysis.

While reference monitors have been quite successful within their intended scope, their role in supporting a wide range of application-specific policies is limited. The wide variety of today’s security policies requires concepts that go beyond granting or denying access in a subject/object communication (property (a)): a security policy that protects the integrity of communication in a distributed system must be able to enforce the use of cryptographic mechanisms, and a security policy that requires the authentication of its subjects and objects must be able to enforce an appropriate authentication mechanism. Second, the strong encapsulation to achieve
property (b) makes it conceptually difficult to integrate new policies into the reference monitor, which is particularly restrictive as security policies still evolve. Policy evolution requires a manageable way to integrate new or improved policies into the trusted computing base. With respect to property (c), in large distributed systems with a multitude of security policies it is technically and politically impossible to integrate all security policies into a common reference monitor. A concept is needed that reflects the separation of the reference monitor into the multitude of user-defined policies and a policy-neutral reference monitor basis. To support property (c), a policy encapsulation concept is required that provides fault isolation and boundaries for separate verification.

This paper discusses the custodian paradigm, an object model for implementing user-defined security policies in distributed multi-policy environments\(^1\). In the custodian object model, policies are represented by encapsulated collections of algorithms and data. A policy is imposed on an application by associating the corresponding custodian with the application.

In contrast to conventional list-based or capability-based security mechanisms, a policy representation by a collection of algorithms and data provides much more flexibility in the types of security policies that can be implemented. Examples for such policies come from many areas. In hospital information systems, several access decisions depend on the notion of time: nurses usually have different rights for accessing patient records during day and night hours. In a finance consulting company or a bank, the separation-of-duty policy is often applied when issuing a report or transferring large values. An example in [TNT92] describes a scenario where students in a university course are given access to a sample solution of their homework only after they have submitted their own solution; this is an example for access decisions depending on the state of other objects within a system. All these examples are difficult to implement using current, non-algorithmic access control mechanisms.

The policy object model provided by custodians is part of a framework for cost-effective development of security policies [KvKO95]. The framework encourages reuse of policy specifications and implementations, it simplifies policy implementation and system integration, and it enables policy cooperation in multi-policy systems. Within the framework, policies are implemented in the custodian paradigm. The framework is currently being implemented in the Distributed Computing Environment (DCE) of the Open Software Foundation.

\(^1\)The paper will use a technical interpretation of the term "security policy" and apply it to a collection of software modules that implement and enforce an abstract policy within a computer system.

\(^2\)This view corresponds to the "multiple interconnected systems view" in the Trusted Network Interpretation (TNI) of the TCSEC.
policy) accessing the documents and a hierarchy of confidentiality among the documents (the objects of the policy). The hierarchies are represented by labels associated with each subject and object. The single rule of our policy is that a subject is granted access to a document if the subject’s label is at least as high as the document’s.

A security policy is precisely represented by a security model that is the foundation of a policy implementation. On the model level, the single rule of the policy is easily expressed by an operation comparing the labels of subject and object. A complete specification of our example is given by the following small algebraic model consisting of types, operations, and axioms. The types in our example are the set \( E \) of application entities (subjects and objects), the set \( \text{BOOL} = \{ \text{true}, \text{false} \} \), and the set \( N \) of natural numbers (the labels). The operations are a less-than-or-equal-to function “\( \leq \)”, a function returning the label of an entity, and a \( \text{ReadCheck} \) function with its semantics defined by the single axiom.

\[
\text{policy MLS}
\]

\[
\text{types} \quad E, N, \text{BOOL}
\]

\[
\text{operations} \quad \leq : \quad N \times N \rightarrow \text{BOOL}
\]

\[
\text{label} : \quad E \rightarrow N
\]

\[
\text{ReadCheck} : \quad E \times E \rightarrow \text{BOOL}
\]

\[
\text{axioms} \quad \text{ReadCheck}(s, o) = \leq (\text{label}(s), \text{label}(o))
\]

On the implementation level, the single rule of the model is traditionally implemented by an access control matrix, describing for each subject the access rights it holds with respect to each object. The access control matrix is initially defined based on the subject/object labels. The legality of a subject/object access can then be decided by looking up the object’s entry in the subject’s row. This works as long as the number of entities and the assignment of labels to entities is static. Whenever a new object is created, the rights of all subjects with respect to the new object have to be computed based on their labelling, and a corresponding new matrix row has to be created; new subjects result in similar actions. When documents are reclassified or the labelling of a subject changes, the corresponding column respectively row of the matrix has to be updated. Our simple policy rule thus results in several matrix handling operations that have nothing to do with the policy per se and that make fine-grain policy specification and verification more difficult and expensive.

A solution to this problem is based on the observation that several well-known security policies (as well as our own small example) are specified using algebraic techniques [BL76, Bib77, TW89, BN89]. Algebraic specifications are closely related to the notion of abstract data types (ADTs) [Hoa72, CW85]. This affinity strongly suggests the implementation of security policies as abstract data types: encapsulated collections of data and algorithms.

An ADT-based implementation of the same policy makes the entire access control matrix together with the updating mechanisms obsolete by simply providing a procedural implementation as defined by the axiom: a comparison of the labels of the subject and object implemented by a single line of code. The much simpler implementation together with the affinity of the implementation to the formal algebraic specification closes the semantical gap between model and implementation, simplifies the verification and is much more efficient in its dynamic behavior.

Many security policies have a memory. In our example, the access decisions are based upon a classification of subjects and objects (modelled by the \text{label} operation). This classification is a long-term state of the policy that must be long-term stored together with the algorithms that implement the policy. Another example is the homework policy from the introduction: the policy must keep track of the students who have delivered their homework and who have looked at the sample solution. As a conclusion from this subsection, custodians implement security policies as persistent abstract data types.

### 2.2 Separation

The goal of separation is a rigorous isolation to gain integrity or confidentiality or other properties such as self-protection [AHIJ92]. While the TCSEC explicitly mentions the protection of the integrity of a security policy as one of the reference monitor principles, maintaining the confidentiality of the policy is similarly vital: the state information within a policy (the labels of subjects and objects in the above example, conflict relationships between competing enterprises in the Chinese Wall policy [BN89, San92], keys for issuing certificates in a X.509 certification authority [CCI88]) is itself confidential information that must be stored in a way that guarantees its confidentiality and integrity in the same way as is guaranteed for the security policy algorithms. Our paradigm thus explicitly must safeguard both, policy confidentiality and integrity.

To provide separation, a fundamental property of the paradigm thus will be policy encapsulation, which is well in tune with the approach to implement security policies by ADTs.
2.3 Metapolicies

The multiple interconnected systems view in the TNI covers large and complex computer networks connecting several independent organizations. Typical for this environment is the large amount of coexisting security domains and the need for cooperation between the domain’s policies. Cooperation within such a network requires the definition of the relationships between the involved security policies, a concept often referred to as a metapolicy [Hos92].

Figure 1: A metapolicy controlling security policies controlling entities

Metapolicies are policies about policies. They create a framework for the complex coexistence of several security policies, containing rules for interfacing, cooperation and conflict resolution. Such a framework basically needs two principles: policy separation and policy communication [MS88].

Policy conflicts generally occur at the perimeters of policy domains when domains overlap or when entities belonging to different domains interact. As an example, consider an estate agency with a multilevel security (MLS) policy governing its document handling systems. The agency uses the services of a finance consulting company. The consultant company is British, so it is legally obliged to submit its own document handling systems to the Chinese Wall policy [BN89, San92]. A market analyst from the consulting company thus (as a subject) is submitted to Chinese Wall, while any document of the estate agency (as an object) is submitted to MLS.

Whenever an analyst from the Chinese Wall domain accesses a document within the MLS domain of the estate agency, this access will not run smoothly. Neither is the subject within the MLS domain nor is the object within the Chinese Wall domain. In this situation, a metapolicy kicks in and handles policy conflicts and cooperation based on rules that have been agreed upon by both, the estate agency and the finance consultants.

The cooperation between metapolicy and security policies requires a policy communication mechanism. For example, the metapolicy may contain a rule assigning labels to well-known subjects of the consulting company and thus allow the MLS policy to proceed. To that end, the metapolicy must be able to call an interface operation of the MLS policy to provide the label of a subject.

On the model level, we provide a policy interface by a public section. To model the above metapolicy interaction, our MLS algebra is extended by an interface description providing an operation SetLabel used by a metapolicy to define the label of a subject outside the MLS domain.

\begin{itemize}
    \item \textbf{policy MLS public}
    \item \textbf{SetLabel}
    \item \textbf{types}
        \begin{itemize}
            \item $\mathcal{E}, \mathcal{N}, BOOL$
        \end{itemize}
    \item \textbf{operations}
        \begin{itemize}
            \item \textbf{SetLabel} : $\mathcal{E} \rightarrow \mathcal{N}$
            \item \textbf{ReadCheck} : $\mathcal{E} \times \mathcal{E} \rightarrow BOOL$
            \item \textbf{ReadCheck} : $\mathcal{E} \rightarrow \mathcal{N}$
        \end{itemize}
    \item \textbf{axioms}
        \begin{itemize}
            \item \textbf{ReadCheck}(s, o) $\iff$ $\leq (\text{label}(s), \text{label}(o))$
            \item \textbf{SetLabel}(s) $\in l$ $\Rightarrow$ $\text{label}(s) = l$
        \end{itemize}
\end{itemize}

Concluding, policy communication is essential for policy cooperation in multipolicy environments. Again this requirement is well in tune with the ADT approach that provides access to an ADT instance via a collection of well-defined interface operations.
2.4 Security Policy Reusability

From an economic point of view, the amount of effort invested in software development makes it generally desirable to reuse software in a variety of applications. Security policies, their formal models and their implementations are renowned to be particularly expensive: completeness and correctness properties in many cases have to be formally proved. It is thus highly desirable that a security policy, once developed, coded, and verified, is used for more than a single application.

Encouraging policy reuse is one of the goals of the policy engineering framework discussed in [KvKO95]. Policy reuse is based on the one hand on a strict separation of security policy and application, and on the other hand on a language to compare policies and compose new policies out of existing ones.

Policy composition is based on subtyping and inheritance. The preceding sections argued that data abstraction is an appropriate representation paradigm for security policies. Combining data abstraction and the notion of types for policies makes policies first-class objects so that they can be manipulated within a formal language. Combining data abstraction and types with subtyping and inheritance allows the specification of abstraction as specialization relations among policies. Inheritance is a type (policy) composition mechanisms for reuse of properties in the definition of new types (policies) and avoids the redefinition and reimplementation of existing components. Subtypes can be uniformly manipulated as belonging to their supertype.

Data abstraction, types and inheritance are the foundation of a formal language for composing security policies by operators such as "$=$", "$<$", "$>"", "$+$", "$\land$", "$\lor$". The algebraic base is currently developed, first steps can be found in [Bry95].

Concluding, security policy reuse requires a foundation for comparing policies and for reasoning about the semantics of policy composition. The reuse of policy implementations then requires the notion of policy types, together with a scheme for subtyping and inheritance.

2.5 The Reference Monitor Principles

The principles of the reference monitor have long since proven to be a sound foundation for access control policy implementation. Tamperproofness, access mediation and verifiability are thus mandatory properties for any paradigm to support user-defined policies.

The abstract data type approach that has been argued for up to now has already two properties of the reference monitor. Policy encapsulation has already been identified as a mechanism to achieve policy confidentiality and integrity, which includes policy tamperproofness. Types and subtypes used for policy reuse also allow for separate modular verification of policy components and help in limiting complexity.

For providing the property of total access mediation for a user-defined policy, the underlying policy-neutral reference monitor must maintain an association between an application and its governing security policy. Based on the reference monitor’s own property of total communication control within a system, this association is used to detour any communication of an application to its associated custodian. The binding mechanism thus expands the total communication control property of the reference monitor to user-defined policies.

3 The Custodian Paradigm

Custodians implement an object model to support user-defined security policies in distributed systems. In the object model, policies are represented by collections of algorithms and data. Custodians constitute a protective shell for a policy, thus providing policy tamperproofness. A method-call communication paradigm provides a well-defined policy interface to support policy cooperation in multi-policy environments. Custodians are persistent to implement policy memory. Custodians are typed to support policy reuse via subtyping and composition.

A policy is imposed on an application entity by associating the corresponding custodian with the application entity. All entities associated with the same custodian make up the policy’s domain. In this sense, a policy becomes an attribute of every entity in the policy’s domain.

Entities with a policy attribute are no longer free in their communication. Every communication of such entities (incoming as well as outgoing) is rerouted to the associated custodian and results in a call to a policy operation. The association of policy operations with entity operations depends on the granularity of the policy: an access control policy may provide a single operation (such as $\text{CheckReadAccess(Subject, Object)}$). On the other hand, the homework policy in the next subsection is an example for a fine-grained policy defining a single policy operation for every operation of a submitted entity. Additionally, a security policy may define operations for cooperating with metapolicies (see section 2.3).

To enforce a security policy, the custodian paradigm has minimum infrastructure requirements to an underlying system platform. It requires
• an encapsulation scheme with a clear perspective how to build entities from collections of algorithms and data

• a communication mechanism with a clear perspective how to build a method-call communication paradigm

• a clear perspective how to add persistency to entities

• a clear perspective how attributes of an entity's individual environment are defined and used to achieve communication interception.

The precise separation, persistency and communication models are defined "lazily" when the paradigm is brought into a concrete environment. While this may result in restrictions caused by an inappropriate infrastructure, we gain an easy integration of the concept into a wide variety of platforms. On every platform, custodians are ordinary system entities containing the security policy for their associated objects. In the BiriX Security Architecture, custodians are ordinary user-defined abstract data types [HKK93]. On a Unix platform, custodians are ordinary processes (with policy memory implemented by Unix files), and in the OSF Distributed Computing Environment custodians are implemented as DCE servers.

3.1 An Example

To give a practical overview on our approach, we will illustrate the development steps of the homework delivery policy briefly described in the introduction. We will give an informal description of the policy and a formal specification. We will provide a custodian implementing the formal model, and finally discuss the way the custodian properties are implemented within a DCE environment on a Unix platform.

The formal specification is written in the algebraic language of our above examples; the custodian is written in C++.

3.1.1 Homework Policy Specification

Our scenario is a university course where students have to deliver their homework by putting a document into a directory. The directory also contains a sample solution. Life would be easy for the students if access to the directory were not governed by two rules:

1. the sample solution can be accessed only after a student has delivered the homework document 
2. after having accessed the sample solution, further delivery of homework documents is denied.

The following algebraic model implements the policy. The semantics of the model is defined by equational axioms using "=" as equality sign. "=": TypeItem × TypeItem → BOOL denotes type-correct testing for equality, and "⇒" denotes a conditioned axiom BOOL⇒(expr1 == expr2), meaning if the boolean expression is true, then the axiom applies. ""|=" in the interface description denotes by which operation of the algebra an interface operation is modelled.

```
policy Homework;
public  
  PutHomework(s) ⇒ DeliverRequest(s,true) 
  GetSample(s) ⇒ SampleRequest(s,true) 
```

```
types
  S,BOOL
```

```
operations
  DeliverRequest : S×BOOL → BOOL
  SampleRequest : S×BOOL → BOOL
  Delivered : S → BOOL
  HasSample : S → BOOL
```

```
variables
  b : BOOL
```

```
axioms
(1)  DeliverRequest(s,b) ⇒ ((HasSample(s) = false) == true)
(2)  DeliverRequest(s,DeliverRequest(s,b)) ⇒ ((Delivered(s) = true) == true)
(3)  SampleRequest(s,b) ⇒ ((Delivered(s) = true) == true)
(4)  SampleRequest(s,SampleRequest(s,b)) ⇒ ((HasSample(s) = true) == true)
```

The operations PutHomework and GetSample define the policy interface. The interface operations are implemented by the operations DeliverRequest and SampleRequest. The semantics of DeliverRequest and SampleRequest are defined by the equational axioms. Axiom (1) guarantees rule one. Axiom (2) models the setting of a mark in the boolean array Delivered that subject s has delivered its homework: if DeliverRequest has been executed at least once before by s, Delivered(s) has the value true. The boolean parameter is a syntactic placeholder for the recursive operation call and has no semantical meaning.

This algebraic specification precisely defines the semantics of the homework policy and is the foundation for the following implementation by a C++ custodian.
3.1.2 The Homework Policy C++ Custodian

An implementation of our homework example within the DCE scenario consists of a homework server managing a directory with the students’ homeworks and the sample solution, and the homework policy custodian associated with the server. To deliver a student’s homework and to require the sample solution, the server defines two operations PutHomework and GetSample. Students access the server via ordinary DCE clients.

![Diagram of Homework Custodian, Homework Server, Sample Solution, Student’s Homeworks, and Student Clients]

Figure 3: The homework example in a DCE scenario

As a side remark, this example cannot be implemented with the ordinary Unix access control mechanisms: reading of a file (the sample solution) and changing of write access permissions to a directory (to deny further homework deliveries) cannot be made atomic without a transaction mechanism such as implicit to a DCE server.

The following C++ class is the essence of the DCE custodian implementation of the above specification. Minor details (such as binding handles) have been omitted.

class HomeworkCustodian:
    public DCEMetaCustodian
    {
        public:
            BOOL PutHomework(Subject s)
            { return(DeliverRequest(s)); }
        
            BOOL GetSample(Subject s)
            { return(SampleRequest(s)); }
        
        private:
            BOOL Delivered[SubjectRange];
            BOOL HasSample[SubjectRange];
        
        BOOL DeliverRequest(Subject s)
        { if (HasSample[s]==false)
            return(Delivered[s]=true);
            return(false);
        }
        
        BOOL SampleRequest(Subject s)
        { if (Delivered[s]==true)
            return(HasSample[s]=true);
            return(false);
        }
    }

It strikes the eye that the semantic gap between the formal model and the C++ custodian is extremely small. The abstract data type approach makes auxiliary mechanisms such as access control matrices obsolete and allows the direct implementation of the axioms of the formal model by procedures of the implementation language. The persistency property makes auxiliary persistency mechanisms such as files obsolete and allows the direct implementation of the model’s persistent state variables.

3.1.3 The DCE Custodian Implementation

Custodians are implemented as DCE servers. A DCE server is a program with a well-defined interface that handles requests from clients. The server paradigm provides two of the four infrastructure requirements that custodians impose on an underlying system platform: encapsulation and a well-defined communication paradigm. Encapsulation is based on address space separation; communication between clients and servers is by a remote procedure call scheme only.

**Persistency**

Persistency of memory of a DCE server has to be implemented explicitly by the persistency mechanisms of the underlying operating system. We implemented custodian persistency in a separate class (DCEMetaCustodian) that is a superclass to all DCE custodians. The implementation uses Unix files: whenever a server starts, it reads the file containing its state information; when the server shuts down, it writes its current state back to the file. Persistency of DCE custodians thus is insecure, clumsy, error-prone, and inefficient; it stresses the importance of a clear and efficient infrastructure concept for implementing persistent ADTs.

---

3This of course is an idealized view; clients and servers in a Unix DCE implementation are still Unix programs and are free to use all the Unix system calls...
Communication Control

We adopt the idealized view that all communication within a DCE client/server application system is by DCE remote procedure calls only. The technique to intercept communication then is a hook within the DCE RPC implementation and a RPC call to the associated custodian.

The standard DCE runtime system maps application level RPC's to messages. The messages are handled by the underlying operating system via communication protocols bound to sockets.

For example the homework policy was designed in a way that a single rule controls the execution of every server operation, and the policy interface is written to exactly match the homework server interface. We thus have to link matching server and custodian operations so that whenever a server operation is called, the corresponding custodian operation is called first, to check the access right of the caller. To achieve this, a hook for the custodian is infiltrated into the normal RPC flow. In figure 4 we observe three alternatives for placing this hook:

1. embedding the hook into the operating system's socket implementation
2. embedding the hook into the DCE runtime system
3. embedding the hook into the application server.

The difference in the alternatives lies in the resulting sizes of the trusted computing base (TCB). It is a sound principle to keep the size of the trusted computing base small: the smaller its size, the more easy it is to validate the confidence in the TCB's correctness.

Placing the hook within the operating system results in the smallest TCB size: no server cooperation is needed to enforce that the custodian is always called. On the other hand, this solution requires that the operating system's socket implementation is modified.

A hook within the DCE runtime system makes DCE custodians independent from the underlying system platform and is also easier to implement. However, the server now is part of the TCB; knowing the internals of the runtime system, a malicious server can always circumvent the custodian call. The major disadvantage of this solution is the (sometimes drastic) increase of the TCB's size.

Alternative (3) finally puts the custodian call within the application itself. This alternative requires modification of the server's source code and needs the largest amount of server cooperation, while on the other hand it is quite comfortable with the standard DCE runtime system and the standard operating system.

Apart from these alternatives, implementing total access mediation within a DCE environment still has several weaknesses. Both, the client and the server may still circumvent a security policy by directly using sockets, pipes, or the file system for establishing a communication outside the RPC scheme. Additionally, the persistency implementation via Unix files is prone to integrity and confidentiality attacks to the policy itself. Considering these inherent weaknesses of a Unix/DCE platform, no new dimension of insecurity is introduced by choosing alternative (3): any effort to implement the RPC hook outside the server would have given us only a false impression of a small trusted computing base. While alternative (3) definitely is impractical for commercial applications, it is a flexible foundation for experimenting with various implementations of hooks.

3.2 Performance

The total access mediation property requires that a custodian is called whenever the associated server communicates via RPCs. Every server operation thus results in an additional custodian operation. The time needed for the detour is the time needed for checking whether a custodian is hooked plus the time needed for calling the custodian plus the time the custodian needs for the policy execution. While the hook time can be neglected...
(<0.1μsec on a 50 MHz 486 machine), the time for calling the custodian depends on the physical closeness of custodian and server (local/nonlocal RPC) and the physical encapsulation of the custodian (bound to server in same address space vs. separate address space).

The straight-forward implementation of a custodian as an independent DCE server with a separate address space provides the highest protection for the policy and also is the most expensive. Regular RPCs are used for calling the policy operations, and thus the cost of a policy operation call is the RPC time plus the policy execution time.

Several improvements of this basic scheme apply. Depending on the usage pattern of the server, hooks with caches can eliminate a part of the actual custodian calls. Depending on the trustworthiness of the server, a custodian may share the server's address space to reduce the RPC overhead to that of a simple procedure call. Finally, inline-compiled policy operations result in zero overhead.

4 Related Work

The origin of custodians go back to the BirliX Security Architecture [HKK93] which uses custodians as access control instances.

In terms of the Muse Object Architecture [YTM+91], custodians can are objects that define the environment of other objects with respect to security. A custodian thus constitutes that part of an object's _meta space_ that defines an object's reigning security policy.

A security mechanism using algorithms for access control decision can also be found in [BP88] and [TNT92]. Theimer uses access control programs within capabilities for fine-grained delegation control, while the watchdogs of Bershad more generally allow the modification of the semantics of Unix file system operations. A more detailed comparison of the custodian paradigm with access control programs can be found in [Kühl95].

5 Conclusions

The paper argues that data abstraction, types and inheritance are well suited concepts for representing user-defined security policies in a multipolicy environment. Together with the traditional reference monitor principle of total access mediation, a wide variety of policies can be supported.

The custodian paradigm is an object model for security policies based on these principles. We successfully have used custodians for implementing a wide variety of access control policies as well as for implementing simple traditional security mechanisms. We currently are developing a complex security policy for a hospital information system and will use custodians for its implementation.

Custodians have been implemented within the OSF Distributed Computing Environment. Although our current implementation does work well in friendly environments with trustworthy servers, considerable efforts still must be made by platforms like DCE to reduce the size of the trusted computing base. An ideal platform for implementing user-defined security policies would provide

- a clear perspective how to use data abstraction
- a type scheme supporting inheritance
- a well-defined communication paradigm
- a clear perspective how to implement persistency
- a clear perspective how entity attributes are defined and exploited.

6 Acknowledgement

The author would like to thank the systems security group at GMD, including Ciarán Bryce, Wolfgang Lux, Matthias Reinholz and Michael von Kopp Ostrowski, for innumerable valuable discussions.

References


