The BirliX Security Architecture

Hermann Härtig, Oliver Kowalski and Winfried Kühnhauser
GMD (German National Research Center for Computer Science)
D 5205 Sankt Augustin 1, Germany
email: hermann.haertig@gmd.de

Abstract

A security architecture is a complete framework that allows to enforce coexisting application and environment specific security policies by applying security mechanisms in a consistent manner. It is argued that such a security architecture can be derived smoothly from an object-oriented architecture such as BirliX.

BirliX applications run on an abstract machine that provides types and instances. Applications are clusters of instances that invoke methods on each other. The BirliX Security Architecture provides means to control the invocations (access control) and to control the enforcement of the abstract machine (infrastructure).

Both, access control and infrastructure, are controlled by users respectively systems administrators to support application dependent security policies.

1 Overview

There exists a significant amount of work invested in assurance of implementations of specific mandatory — Bell La Padula [BL76] based — security policies. There is also a rich set of cryptography-based mechanisms such as file encryption, authentication protocols and secure channels, which became practically feasible through technological advances such as encryption hardware and smart-cards.

The BirliX [HKLR92] Security Architecture work is focused in another direction: it provides a framework

- to define and enforce security policies that take into account specific requirements of applications and environments, and
- to incorporate various, exchangeable security mechanisms and technologies.

The security architecture makes use of the object-oriented system architecture of BirliX based systems:

- To formulate *access control* as needed by user applications, i.e. by directly controlling the invocation of methods instead of controlling e.g. read/write operations, to which method invocation is broken down in conventional systems. Experience shows that some awkward and failure critical schemes that are found in other systems can be avoided.
- To formulate *infrastructure* requirements, i.e. requirements that are needed to enforce the object-oriented abstract machine.

Both, access control and infrastructure, can be controlled to adapt to specific user or environmental requirements.

BirliX applications run on an abstract machine (provided by an operating system kernel) that provides types and instances. Applications are clusters of instances that invoke operations on each other.

These invocations can be controlled in two ways:

- *Access control lists (ACLs)* of an instance
  specify which methods of the instance may be invoked by other instances.
- *Subject restriction lists (SRLs)* of an instance
  specify which other instances the instance may invoke operations on.

Thus, method invocations are controlled in a fine grained manner according to an application specific semantic without the need to reduce access control to poor read/write semantics.

Access control relies on the proper enforcement of the BirliX abstract machine. To do that, the infrastructure has to provide authentic messages, encapsulated instances and types. Encapsulation of types and instances can not be enforced against a user who has physical access to systems (i.e. total control over long term storage of data representations). Thus, encapsulation is based on trust in persons managing and/or owning a system. Trust is expressed in terms of confidence domains describing the set of systems that are considered trustworthy from the point of view of an object. To provide confidence domains authentication of systems and persons is needed. In BirliX authentication relies on a secure booting process.

The separation of access control and infrastructure provides for a clear structure to insert security mechanisms in a way to support application dependent requirements. The naming scheme is crucial to control access control and infrastructure.

The security architecture described herein uses state of the art technology and provides a framework for new mechanisms.

2 Security and Object Orientation

System architectures that provide an object-oriented programming paradigm are an apt basis for designing and building secure systems:

- Objects interact by one and only one well-defined communication paradigm, the invocation of methods. This provides a natural hook for the integration of security mechanisms and policies.
- Object interfaces are much richer than the simple *read/write*-interface of files. While in general this allows a more problem-oriented expression of semantics, it also allows the controlling of access rights in a very fine and problem-oriented way.
- Security in general has the reputation of being expensive. Thus, a security architecture must be adaptable to the scale of needed security, and must cause performance costs only where higher security is needed and people are willing to pay for it. The ability of attaching expensive security mechanisms and policies to individual objects is well supported by inheritance, both statically (defined globally for all objects of the same *type* (or *class*)) or dynamically

(defined when a single instance is created). Multiple inheritance can be used to attach several security mechanisms defined in class libraries (e.g. ACLs and SRLs, see below), whereas multilevel inheritance can be used to built hierarchies of cooperating security policies.

The BirliX Security Architecture is a framework for the integration of application systems, security policies, and security mechanisms. The basic architectural components are:

- An object-oriented model where application systems are sets of objects interacting via method invocation; components of the architecture enforce an abstract machine that provides the object model.
- Security mechanisms that control object interaction via controlling method invocation.
- Security policies that contain application-specific security goals, rules, environment conditions. The enforcement of security policies rely on the enforcement of the object-oriented model and the security mechanisms that control object interaction.

3 The BirliX Object Model

The BirliX operating system kernel has an object-oriented paradigm; it supports *types* (or *classes*), *instances* created from types, and instance communication via *method invocation*. In many ways, BirliX Types are similar to Eden Types [ABLN85] or Smalltalk classes.

Types are descriptions consisting of an interface (the *methods*) and the code of the methods. *Instances* are incarnations of types. Each instance has a unique identifier; the mapping of high-level symbolic names to unique identifiers is done by name servers that also provide location information. Instances can be active on their own; they may contain *threads* that run in parallel to each other within the same instance.

Instances are also persistent, they can be checkpointed and recovered, and they may migrate; for details see [HKLR92].

In the following example, a workstation maintains a calendar that keeps tracks of dates and issues a reminding mail when a date becomes hot.

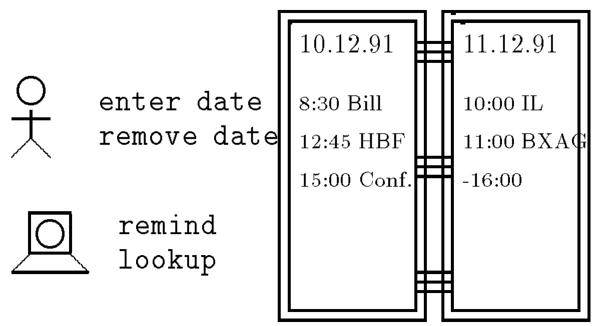

Figure 1: The calendar example

A straight-forward way for modeling the calendar example in an object-oriented programming language is to define a type with methods to enter and remove dates, and an interface for a reminder process that may lookup the calendar contents.

This is a well-established model within the scope of programming languages. Unfortunately, this model is far too simple for objects in the scope of operating systems. Here, we have to deal with distributed applications with remote method invocation, *real* encapsulation (in contrast to compile-time checking), protection, persistence, or fault tolerance. Operating systems thus have to provide a much richer object model.

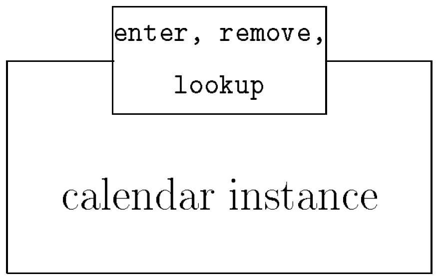

Figure 2: A calendar in an object-oriented language

The BirliX Kernel provides its object model by embedding each language-level object in a separate frame called the BirliX *Object Infrastructure*.

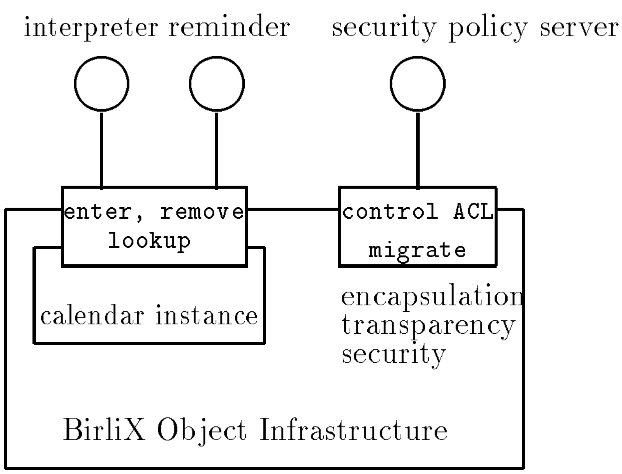

Figure 3: A calendar in the BirliX object model

The object infrastructure provides for each instance the type-independent object model properties like encapsulation and transparency of method invocation. Additionally, basic fault tolerance and security mechanisms are attached to the instance as methods inherited from a system-defined library of primary types.

Two mechanisms to mention are migration and replication. Migration is a generalization of file cacheing and process migration, as found in some conventional systems. Since migration and replication cause long-term data storage on another node, both are security relevant.

4 ACLs and SRLs

The BirliX protection system offers two security mechanisms realizing two basic views in an object management system, namely the *object* and the *subject* view. The object view defines for a given object which subjects may call which methods of the object. In contrast, the subject view defines for a given subject to which objects the subject may have access. A protection domain is defined in terms of both views.

The object view is supported in commonly used operating systems (e.g. Unix) by means of access control lists (ACLs). Most known ACL implementations only support entry types related to users or groups of users, and a simple read/write-right semantic. This isn't sufficient for various applications, therefore several work-arounds are used. Examples for this are the use of *pseudo*-users (e.g. for server-processes in Unix), new interpretations of access rights (e.g. sticky-bit for directories in Unix) or new security mechanisms (e.g. the xhost-mechanism of the X-window system).

Orienting these ACLs towards the BirliX architecture, better means are achieved to cope with these requirements. This results in offering an enhanced ACL-mechanism. Both entry types and rights are extended to meet security requirements. Besides users and groups, the following additional entry types exists:

- *Instance* entries.
  They refer to exactly one instance. This allows right definitions for single instances, e.g. long-living servers.

- *Type* entries.
  They refer to groups of instances all being of the same type. This allows right definitions on the basis of types, e.g. mail-spoolers.
- *Custodian* entries.
  They refer to objects which have to be asked for whether a subject should have access or not. One example is a time-dependent custodian. During working hours it allows access, outside the working hours, access is denied. Another example for a custodian is the *group*, which simply is a collection of the above mentioned ACL entries. Because custodian entries can be members of a group, this permits the definition of group hierarchies.

Another enhancement of the BirliX ACLs is the possibility to *permit* or *refuse method calls* for each of the above entries, in contrast to the read/write permissions. The contents of an ACL and the behavior of a custodian defines the protection policy in terms of the object view.

The subject view is supported in capability-based systems by means of capability lists (c-lists). The BirliX Security Architecture supports both, the subject and the object view. The subject view is supported because in distributed systems the object views cannot be changed in a reliable way. The subject view enforces access restrictions on subjects. The mechanism is called subject restriction list (SRL). SRLs are used to control the outgoing method calls of a subject. If access to an object should not be granted, the result is the rejection of the call.

Both ACLs and SRLs can only be manipulated by calling the appropriate method calls of the primary type. Therefore they are subject to access control. It is possible to enforce mandatory access control policies, because even the object itself can access its ACL and SRL only via method invocation and it is up to the policy who may have access to the appropriate methods.

In various policies, the usage of roles is recommended. Roles are supported in BirliX by using instances as intermediaries.

The fine-grained protection domains may be used to enforce a strict least privilege policy. As an example, the emulation of the Unix-system on top of BirliX makes use of these features to discard the setuid/setgid-mechanism, which is a prominent source for security risks. For a detailed description of the ACL/SRL mechanism see [KH90].

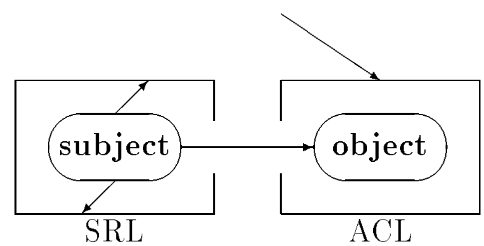

Figure 4: ACLs and SRLs

Entries in ACLs and SRLs refer to instances and types by their unique identifier. While this results in an efficient implementation, two problems have to be discussed. The first one is the necessary administration overhead if a new version of a type is created. In BirliX this results in a new unique identifier for the type, therefore the ACLs of a possibly large number of objects have to be modified. A solution for this problem is the usage of group entries in ACLs, meaning that only the group has to be updated.

The second problem is the presentation of ACL and SRL entries to human users. Users are not interested in unique identifiers, but in names that are readable and interpretable by humans. This results in the need of a mapping function for unique identifiers to symbolic names, which requires the usage of trustworthy name-servers, which is discussed in chapter 7.

5 Confidence Domains

In a non-distributed environment the enforcement of SRLs and ACLs relies on the proper implementation of the BirliX kernel. In a distributed environment without centralized administration and security control, confidence is needed:

- about the *identity of nodes*, where the identity of a node consists of the identity of the hardware and the identity of the operating system running on it.
- about the *trustworthiness* of the owner respectively operator of the node, whether he circumvents software implementing access control, e.g. by directly accessing long term data representations of objects on disk.

In general, confidence is needed about the correct enforcement of the abstract machine.

However, nobody assumes that the abstract machine can be enforced globally against all sorts of attacks, especially if long term storage is concerned. Therefore, means are required to let people express their personal confidence in the enforcement of the abstract machine. Certainly, the degree of confidence to other nodes and their operators depends on the objects in question.

Thus *confidence domains* denote for an object, with respect to which set of nodes there is confidence about the enforcement of the abstract machine with their access control mechanisms.

Requirements

We will discuss three examples for confidence domains: the laptop/server scenario, the labstation/server scenario, and a common document development scenario. We use them to show how confidence domains are used and derive implementation properties of confidence domains.

The Laptop/Server Scenario

A laptop is a self-contained, mobile computer system that has one owner. The laptop sometimes is connected via some untrustworthy network to a pool of servers, e.g. to move documents or read electronic mail. That connection will usually have to rely on arbitrary untrustworthy routers. The laptop contains data that the owner considers to be very valuable, e.g. a personal notebook. The owner does not consider even the servers to be trustworthy enough to allow the notebook to migrate to the servers or even to communicate with other objects on the servers directly.

Using confidence domains, that scenario can be described easily. The confidence domain of the notebook consists of the laptop only. Obviously, the confidence domain of all objects residing on a node contain at least that node. Less critical objects have confidence domains, containing the laptop plus a set of server nodes.

The implementation of confidence domains has to provide a reliable way of authenticating these servers, without relying on the trustworthiness of the intermediate routers.

The Labstation/Server Scenario

A labstation is a stationary computer system which is utilized on demand by many persons. Examples are nodes of a workstation pool or measurement/control computers in a chemistry analysis laboratory. The labstation may have special devices that directly control experiments. It is operated by everyone who needs its services, therefore it is not considered to be very trustworthy. The labstation is connected to a pool of servers. A user may collect critical data using an object called the *collector*. The collector resides on a pool of servers and is assumed to trust the servers' operators and the operating system. The collector has to be downloaded (migrated) to the labstation to be used. The owner of the object does not trust all the labstations connected to the servers but has to trust one labstation while he operates and controls it.

This scenario can easily be described in terms of confidence domains. The confidence domain of the *collector* consists of the servers only. Then, to enable migration of the object to the labstation, the confidence domain of the object is extended by the labstation. Once the experiment is completed, the object is migrated back to the servers and its confidence domain is shrunk down to the pool of servers again.

A correct implementation enables the user, when he starts using the labstation, to make sure, that the abstract machine with its access control is enforced by the labstation and its connection to the servers. A disconnection and *logout* of the workstation is not allowed before all objects are migrated back to the server and the confidence domain is shrunk. A disconnection because of some fault cannot be prevented, but before *logout* the *collector* has to be removed from the labstation.

Common Document Development Scenario

A document (e.g. a contract) is developed in a stepwise manner. The document object resides on a server that is under control of a public notary. The object representing the contract can be manipulated only using a well-defined set of operations that are designed to keep track of all changes during the development process. The document is developed under supervision of a notary, who has allowed two (or more) partners, Bill and Joe, to make changes to the document. Bill and Joe operate from their own node, but their nodes are not considered to be trustworthy to let the contract migrate to them.

That scenario can be described in a straight forward way. The contract has two confidence domains, one containing Bill's node and the notaries server, the other containing Joe's and the notaries server.

That leads to an interesting property of confidence domains. If an object has two confidence domains, as in the example above, two interpretations are possible:

- The *transitive* interpretation
  It allows the object to migrate respectively communicate within each of the confidence domains.
- The *non-transitive* interpretation
  It allows migration respectively communication only in the intersection of its confidence domains.

It can be seen from the example, that migration (that involves long term storage) requires the non-transitive interpretation while communication requires the transitive interpretation. Therefore, without restricting the generality of usage for practical matters, the non-transitive interpretation is chosen for migration and the transitive interpretation is chosen for communication in the BirliX architecture.

A slightly different scenario may arise, if Bill's and Joe's nodes are not trustworthy for direct communication either. Then, the contract's confidence domain just contains the notaries node. To enable access to contract for Bill and Joe two additional manipulator objects are inserted as intermediates for Bill and Joe. The intermediate objects' confidence domain contain the notaries station and Bill's respectively Joe's node.

Summary of Requirements

These examples illustrate the requirements for confidence domains and their implementation. They are summarized below:

- Users are able to authenticate nodes.
- Nodes are able to authenticate other nodes.
- Nodes are able to authenticate users.
- Authentication is possible in presence of untrustworthy intermediates.
- Confidence domains are defined with respect to objects.
- Confidence domains may change often during the lifetime of their objects.
- Confidence domains are not transitive.

The remainder defines confidence domains more precisely.

Definition

A confidence domain is a BirliX object. It represents a set of nodes. On each node, there exist two standard confidence domains:

- *MyNode*
  containing just the node itself.
- *AllNodes*
  containing all nodes.

Each BirliX type instance has a *Confidence Domain List* (CDL), that contains one or more confidence domains (i.e. via their unique identifiers). The intersection of the sets of nodes as defined by the CDL entries denotes which nodes are considered to have an infrastructure that can be trusted when interpreting the ACLs and SRLs. An object may communicate and migrate within that set of nodes.

Confidence domains provide operations to add or to remove a node. They can be added to respectively removed from the CDL of an object. A confidence domain is called *active*, if at least one object with that confidence domain is active and communicating with an object of another node. A time interval is specified for each confidence domain denoting the frequency after which the nodes in active confidence domain are checked for their authenticity.

As its initial value, an object inherits its CDL from the instance that created it. Confidence domains have ACLs and CDLs that control the operations AddNode, RemoveNode.

6 Secure Booting and Mutual Authentication

Prerequisite for the establishment of a confidence domain is the authentication of the nodes of the confidence domain. Hereby, the authentication of nodes has to include the authentication of the *hardware* and the *operating system*. That provides a basis for authenticating types and instances.

This chapter describes a method for mutual authentication of nodes, that is based on secure booting and certificates for the hardware and operating system identity issued by the hardware and operating system manufacturers. Some assumptions on hardware are made, that are not fulfilled by current architectures but can be provided using current chip technology. The protocol has first been published in [Gro91].

Three steps are distinguished in the process, namely *preparation*, *booting* and *mutual authentication*.

Preparation

A manufacturer's association certifies the public keys of the hardware and operating system manufacturers. The certificates are delivered to the customers of the manufacturers.

The manufacturer of the hardware central unit (CU) stores the central unit identifier (CU-ID) into some internal memory. Upon its first startup, the device generates itself a key pair, stores the private key internally and delivers the public key. The manufacturer issues a certificate and delivers it to its customers.

The manufacturer of the operating system creates an identifier for the operating system and signs it together with the operating system.

Booting

Booting consists of five steps:

- The operating system code is loaded.
- The signature of the operating system is verified. To do that, the public key of the operating system is needed.
- A new key pair (the *node key*) is generated for use by the (hardware, operating system) pair, that has been generated by the current booting process.
- A boot certificate is generated, containing the identifier and the public key of the operating system as well as the public key of the operating system manufacturer, that has been used to check the authenticity of the booted operating system. The certificate is signed by the central unit using its private key.
- The operating system is started.

Mutual Authentication

During operation, a node (i.e. its central unit and operating system) can be authenticated, if the public keys of the manufacturers (central unit and operating system) are known. This is done in two steps:

- send a nonce, and
- receive:
  nonce signed by node key
  the boot certificate
  the CU certificate.

The CU certificate delivers the authenticity of CU key. The CU is used to check the boot certificate. The boot certificate delivers the key used to check the identity of the operating system, and OS-Id and the node key. The node key is used to check the signature of the nonce.

Implementation Requirements

For the process described above, we have assumed that a number of requirements are guaranteed for the hardware and software components involved. These requirements are:

- A central unit can be built in such a way, that it can protect a key for the lifetime of the CU.
- An operating system can protect a key for the time between booting and shut-down. A (forced) shut-down does not reveal the keys nor any other data under supervision of the operating system .

Although there is currently (to our knowledge) no computer fulfilling these requirements, we believe that such hardware can be built.

System Authentication by Users

A user can authenticate a station using the protocol described above, if we assume to have a smart-card, that is able to execute the authentication protocol. The physical integrity of the smart-card must be obvious for the user. If the smart-card contains the mentioned identifiers and certificates for a node, the node can be authenticated without relying on name-servers and/or trustworthy routers.

7 Symbolic Names

Human users can directly deal with only a relatively small number of unique identifiers, keys, or certificates that are generally stored on smart-cards or printed on business cards. When identifying objects, in most cases symbolic names are used. Symbolic object names are provided by the naming system that uses concatenated name servers to provide services for *locating* and *identifying* objects.

Unless we take denial of service into account, the locating services of the naming system are not critical for system security if the identity of the object is known. The authentication procedure described in the previous section provides authentication of nodes, and nodes provide authentic messages.

However, the identification services of the naming system are critical for system security for a number of points mentioned in previous sections of the paper:

- to establish confidence domains, the identification and authentication of objects (e.g. nodes) must be reliable
- ACLs, SRLs, and CDLs contain unique object identifications
- establishment of certification hierarchies

When a confidence domain is created, the identifiers of its systems are obtained from the naming system by mapping the symbolic system names to system identifiers. When a confidence domain is established, authentication of the systems belonging to a domain is based on reliable knowledge of the systems' identifiers.

For setting and checking ACL (SRL, CDL) contents, the naming system must provide reliable mapping and reverse mapping of object identifications to symbolic names. A malicious name server could betray a user and map a user-level name to a wrong object identifier, resulting in rights granted to objects other than intended by the user.

Virtually all security policies depend on the identities of the communicating parties being reliably known. When a user refers to an object by its symbolic name, concatenated name servers map the name to an object identifier. As mapping a symbolic name may involve several name servers, the mapping process may involve a name server outside the confidence domain, or it may end up with the object itself being located outside. Within a confidence domain, the name servers are trusted to provide a reliable mapping of user-level object names to system-level unique object identifiers, and the operating systems are trusted to correctly authenticate objects. Outside the confidence domain, authentication is based on an X.509 - style authentication framework that establishes trust based on public keys, certification chains, and certification authorities [CCI88].

X.509 claims that the naming system (the *directory* in X.509 terms) is the natural place to hold authentication information, because the name servers hold information required to establish a communication before it actually takes place. Thus the BirliX security architecture provides trusted name servers for an X.509 - style authentication framework.

Using that scheme and a smart-card, the authentication of a personal workstation is possible from any machine in the world, if either the smart-card itself knows the object identifier of the workstation or the smart-card at least knows the object identifier of a trusted name server that will provide the workstation's identifier.

8 Extensions

Two extensions are discussed that illustrate the extensibility of the security architecture, namely shared control and prevention of denial of service attacks.

Shared Control

Integrity and confidentiality of objects rely especially on the integrity and confidentiality of their long term storage data representations. A number of techniques have been applied to file servers to distribute that risk among several nodes respectively their operators, namely cryptographic threshold protocols and information scattering (used e.g. in the Delta-IV project [RF91]).

These techniques form another example for a possible usage/extension of the BirliX infrastructure. We assume, that hardware and operating systems can be built, that prevent inspection or modification of main memory under circumvention of operating systems. Then the infrastructure of the BirliX abstract machine can intercept pagefaults to collect scattered pieces of a page of a long term storage representation from a number of nodes and it can intercept pagewrites to store scattered pieces of a page of a long term storage representation to a number of nodes.

Denial of Service

Successful denial of service attacks deny accesses that should be allowed. We observe various types of denial of service attacks:

- Exhaustion of resources
  To prevent others from using a service is much older than information technology. The worm attack [ER89] can also be considered as an attack of that kind.
- Blocking of resources
  There are many obvious and subtile means to block access to resources. Whenever an component of a system important for delivery of requests or providing a service is under control of an attacker, he is able to intercept those requests. The Ethernet is a classical example for such a hardware system. To keep a lock for a long time is another example.

Gligor and others [YG88, Mil92] argue that denial of service can be prevented if cooperating subjects agree to limitations of their usage of resources (so called *user agreements*).

User agreements form another example for a natural extension of the BirliX architecture. SRLs are means to express limitations of subjects. Currently these limitations are expressed in terms of names of the objects that can be accessed. We are fairly convinced, these limitations can be extended to express fractions and time of resources. We assume that services can refuse accesses. The infrastructure has to be extended to relay self-imposed limitations as contained in SRLs to the servers, that in turn refuse overextended usage by clients. In that case, confidence domains have to be extended to all routers, because otherwise resources that are needed to provide services are out of control of the architecture.

9 Related Work

There exists a significant amount of cryptography-based mechanisms such as authentication protocols, which are supported by technological advances such as encryption hardware and smart-cards. The most notable works are Kerberos [SNS88] and X.509 [CCI88]. While the usefulness of these mechanism is out of question, they cover only the aspect of authentication in distributed environments.

A security architecture as defined here is the Digital *Distributed System Security Architecture* (DSSA) [GGKL89, LABW91]. In contrast to the BirliX Security Architecture, the DSSA is designed for a heterogeneous environment, i.e. it is supposed to be applied to the full range of Digital's operating systems.
A major architectural difference between DSSA and BirliX is that in DSSA all is reduced to the enforcement of access control in a secure way. Therefore, the major extension to all supported systems is the delegation. One can speak on behalf of somebody else, and this is checked in ACLs. BirliX separation of access control and infrastructure is open to address new additional problems and solutions, such as shared control and user agreements. We are fairly convinced, that delegation could be added to the BirliX infrastructure, and that confidence domains can be formulated in terms of delegation. We feel as a major achievement of DSSA that it provides a framework of terms and algorithms to enforce access control, while BirliX is open to more extensions with less terms to describe the BirliX framework.
DSSA uses symbolic names wherever objects are named with respect to security. This is motivated by the non existence of general low level identifier mechanisms in the various system architectures to be supported. The BirliX Security Architecture uses BirliX' unique identifiers for this purpose. The usage of symbolic names avoids the problems of administring versions of objects, whereas the usage of unique identifiers is far more efficient, but means to deal with versions are needed.

The operating system L3 [LBB+91] supports a mechanism called *Chiefs and Clans* [Lie92], which generalizes the SRL and custodian mechanism by using controllable message redirection facility, which can be efficiently implemented. It is in discussion to use this mechanism in BirliX too.

As a representative of systems incorporating a specific security policy TMACH is to mention. TMACH is an effort to produce a version of the MACH operating system which should be validated to B3 according to the DoD classification. The emphasis of the project is on assurance of the implementation of a specific security policy (Bell/LaPadula mandatory access control), while the BirliX security architecture is built as a framework to incorporate application specific policies. Some points may be interesting to mention: the security policy is enforced by the kernel (i.e. kernel objects tasks are labeled) TMACH has added a type server, and a full reconstruction is considered now to build a B3 version of the kernel.

10 Conclusions

A security architecture has been described as a framework to integrate various security mechanisms to support application specific security policies. It has been shown, that — in an object-oriented system architecture — such a framework can naturally be build by separating infrastructure to enforce the object-oriented paradigm from the user defined access control. User defined access control — consisting of SRLs and ACLs — allows an implementation of the principle of least privilege. Confidence domains — based on secure booting — allows the description of those areas where the paradigm is trustworthily enforced. It has been sketched how access control can be extended to also allow prevention of denial of service attacks.

11 Acknowledgement

This work was supported by the German Ministry of Research and Technology in the REMO (REference MOdel for secure systems) project under contract number 01 IS 202 D. REMO is a joint venture of the European Institute for System Security at the University of Karlsruhe (E.I.S.S.), the German National Research Center for Computer Science (GMD), the Industrieanlagen Betriebsgesellschaft (IABG), Siemens AG, and Telematic Services mbH (TELES).

References

[ABLN85] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The eden system: A technical review. *Transactions On Software Engineering*, 11(1):43–59, January 1985.

[BL76] D. Elliot Bell and Leonard J. LaPadula. Secure computer systems: Unified exposition and multics interpretation. Technical Report 2997, MITRE, March 1976.

[CCI88] CCITT. *Recommendation X.509: The Directory-Authentication Framework*, 1988.

[ER89] Mark W. Eichin and Jon A. Rochlis. With microscope and tweezers: An analysis of the internet virus of november 1988. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 326–343. IEEE, May 1989.

[GGKL89] Morrie Gasser, Andy Goldstein, Charlie Kaufman, and Butler Lampson. The digital distributed system security architecture. In *12th National Computer Security Conference*, pages 305–319. NIST/NCSC, October 1989.

[Gro91] Michael Gross. Vertrauenswürdiges booten als grundlage authentischer basissysteme. In *Verlässliche Informationssysteme (VIS'91)*. GI, 1991.

[HKLR92] H. Härtig, W.E. Kühnhauser, W. Lux, and W. Reck. Operating system(s) on top of persistent object systems – the birlix approach. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, volume I, pages 790–799. ACM, January 1992.

[KH90] Oliver C. Kowalski and Hermann Härtig. Protection in the birlix operating system. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 160–166. IEEE, May 1990.

[LABW91] Butler Lampson, Martin Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theorie and practice. In *Proceedings of the 13th ACM Symposium on Operating Systems Priciples*, pages 165–182, 1991.

[LBB+91] J. Liedtke, U. Bartling, U. Beyer, D. Heinrichs, R. Ruland, and G. Szalay. Two years of experience with a mikro-kernel based os. *ACM Operating Systems Review*, 25(2):51–62, April 1991.

[Lie92] Jochen Liedtke. Clans & chiefs. In *Proceedings of the 12th GI Fachtagung*. GI, 1992.

[Mil92] Jonathan K. Millen. A resource allocation model for denial of service. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 137–147. IEEE, May 1992.

[RF91] Brian Randell and Jean-Charles Fabre. Fault and intrusion in object-oriented systems. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*, pages 180–184. IEEE Computer Society Press, Oct 1991.

[SNS88] J.G. Steiner, B.B. Neumann, and J.I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 Usenix Conference*, pages 191–201, February 1988.

[YG88] Che-Fn Yu and Virgil D. Gligor. A formal specification and verification method for the prevention of denial of service. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 187–202. IEEE, April 1988.