In *Proc. Hawai'i International Conference on System Sciences: Emerging Technologies Track.* Vol III, pp. 316–307. Jan. 1998.

The Architecture of Secure Systems

Jim Alves-Foss
Laboratory for Applied Logic
Department of Computer Science
University of Idaho
jimaf@cs.uidaho.edu

Abstract

Secure system design, verification and validation is often a daunting task, involving the merger of various protection mechanisms in conjunction with system security policy and configurations. This paper presents a generic approach to secure system development that can be readily applied to a wide range of secure systems. Use of this approach, based on separability, will greatly simplify the developer's overall design, verification and validation effort.

1 Introduction

In this paper we discuss a generic approach to the design, verification and validation of secure systems. This approach, based on Rushby's separability model [7, 8], provides a standard methodology that can be used by system designers and verifiers in the implementation of a wide range of secure systems. This approach will assist in greatly simplifying the system design, verification and validation effort by providing a proven template from which to build the system. It is the ease and security of this approach that will provide the greatest benefit to users of the approach. It is important to understand that this approach will not work in all cases, but works in a wide range of common situations, and in those situations it is beneficial.

The material in this paper progresses as follows. Section 2 presents the general system model and discusses how it can be applied to new and existing systems. Section 3 presents a formal model of the high-level system specification and discusses the verification and validation of the system. Section 4 then presents an exemplary secure system design using the technique presented in this paper.

2 System Model

In the early 1980's, John Rushby presented the *separability* concept for secure system design and implementation [7, 8]. The basic idea behind this concept is to model the behavior of a secure system as if it were a physically distributed system. In a distributed system, we have well-defined lines of communication between the individual machines. If we can develop a *separation kernel* for a secure system such that it provides an execution environment similar to that of a distributed system, then we can simplify system design, development, verification and validation.

A separation kernel provides mechanisms for the existence of several virtual machines on one hardware platform. Communication between the virtual machines is limited to well-defined communication paths controlled by the kernel. There exist no other mechanisms for inter-machine communication. In [7, 8], Rushby presents a set of criteria necessary for proof of separability. In [1] we proved that a system modeled on the separation kernel concept satisfies the *restrictiveness* security policy [3, 4]. Given that these criteria are met, and the proof mentioned above, we can be assured that any system based on the separation kernel model is secure. A more general system can be built from a combination of separation kernels and truly distributed components.

In the remainder of this section we discuss a specific approach to satisfying the requirements of secure distributed systems from both the design and the verification and validation point of view. A system designed using this approach will be provably secure, and will not fall victim to the poor design practices found in ad hoc solutions.

2.1 Secure Distributed System Design

A distributed system consists of a collection of separate components connected through a well defined communication medium. Without the medium, each component is completely isolated from the others and unable to share information or resources with them. Such a disjoint system is, by default, secure. This is the basis of the concept of separability. If each subject in the com-

puter is relegated to a separate execution domain, then at a high level of abstraction, the model of computation is precisely that of a distributed system.

Thus, when designing a secure system, we can view the system as a collection of distributed components. This may be the result of truly distributed components on a network and/or multi-compartment single components. If a single component is to handle multiple security regions, then it will need to provide a separation kernel for those regions.

With any combination of multi-level components and distributed components, we can design a system such as that depicted in Figure **??**. The regions depicted in the figure demonstrate separate security regions with a strict information flow-control policy between regions. Each region is labeled with a specific set of security labels, and the system security policy is used to specify authorized information flow between regions. A region with one label is considered *single-level*, while a region with many levels is *multi-level*. We denote the communication path between regions as a shared network, although in an actual implementations this can be a LAN, WAN, internal communication buffer or any combination of these. The only constraint we have on the shared network is that is be a secure network. This means that either all components attached to the network implement the separability policy, or that they are unable to send or receive messages interpretable by those implementing the separability policy. This can be implemented via a secure kernel for internal communication and cryptography for any communication over an external network.

2.2 Secure Distributed System Verification and Validation

The verification and validation of system security involves several issues, each of which is still an active area of research and thus can only be partially addressed in this paper. The composition of choices from these research areas drives the overall system verification and validation effort.

- *Policy.* A secure system can only be designed if there exists a well-defined system security policy. This policy can be formally or informally stated, but it must specify the permissible and forbidden actions of the system. For the purposes of this paper, we limit those actions to communication or information-flow between regions. The main questions that must be answered are:

    1. Do we mandate specific information flow restrictions between certain regions or is that left to the discretion of the users?
    2. How do we map specific users into security regions? A user may have several job functions each requiring access to different security regions.

- *Formal Security Model.* Given the specified policy, it is essential to define a formal security model that satisfies the policy and which can be used in the verification and validation of the specification and implementation of the system. There are several formal policies that have been discussed in the literature, the ones most applicable to the design approach of this paper are *separability* [5] and *restrictiveness* [3, 4]. Both of these policies focus on the concept of information flow. Specifically, they model how processes in different regions should not interfere with each other's operations. One benefit of both separability and restrictiveness is that they are composable security properties. In other words, a system design developed from the connection of components that satisfy these properties is also guaranteed to satisfy the same property. This is not necessarily true for all formal security models.

- *Verification and Validation.* Now that we have a policy and formal security model, it is essential to verify that the system satisfies the model. In general this involves multiple levels of verification: showing that the system specification and design satisfy the policy and showing that the implementation satisfies the design. For example, in [1] we developed a high-level specification of a secure system based on an early version of the approach discussed in this paper. We then proved that the specification satisfied the *restrictiveness* security model. Although formal verification proofs may seem excessive to some, the approaches discussed in this paper apply to systematic testing and validation as well.

3 Formalism

This section presents an approach to the formal specification, verification and validation of secure systems modeled on the approach outlined in Section 2. In general we follow a standard top-down divide and conquer approach to system development. We decompose components of the system into constituent components until we reach a clear security boundary. For single-level components, beyond this boundary we are no longer faced with security verification and validation but rather

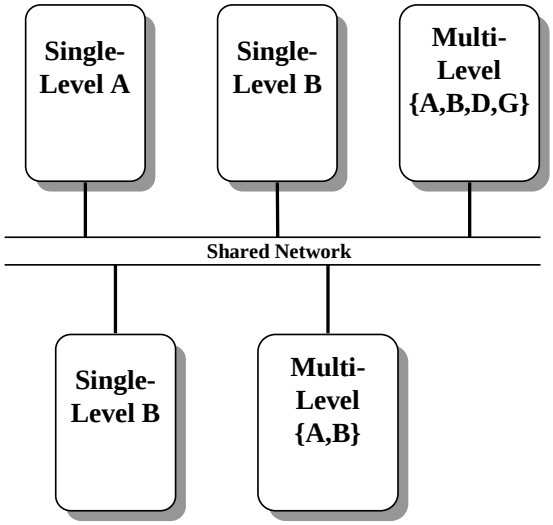

Figure 1: Simple Secure System Layout

with more traditional concerns such as performance and functionality. The approach we use decouples the these traditional concerns from security concerns, thus we can simplify our efforts by looking at only the security boundaries and the true security-critical components of the system.

Each execution environment is classified as either a single-level process or as a multi-level process. Although we claim that the existence of the separation kernel should permit the implementation of multi-level execution environments as a collection of single-level virtual machines, we provide the ability to model independent multi-level execution environments to permit flexibility in the model. A single-level process processes information within a single security compartment. All information received, transmitted or manipulated by this process is categorized within the same security compartment. Such a device can be implemented using standard off-the-shelf components and software engineering technologies. It is assumed that the operation within the component is untrusted with respect to security management. A multi-level process handles information within a set of security compartments and guarantees the information flow and access control restrictions between those compartments. Such a component needs to be designed and analyzed carefully and should have some level of certification that it satisfies the multi-level security constraints of the component.

Our distributed system can be created using instances of these processing elements. Our design, verification and validation efforts should ensure that these elements do not communicate or otherwise interfere with each other. Given this assurance we know that security is maintained between these elements. Now we need to add a communication facility to enable these independent elements to share information. The communication facility must ensure the security and integrity of the information passing between processing elements. To do this, we must explicitly state what constraints we are putting on the communicating elements and the network. The following sections detail the major generic components we use in secure system specification.

3.1 Specification of Components

All specifications in this paper are written in LOTOS [6], a formal specification language of the ISO. In general, all LOTOS specifications consist of a collection of process specifications parameterized by *gates* and values. These specifications can be combined, invoked and recursively defined. Communication and synchronization between processes occurs through events at the gates. An event occurs at a gate when all processes using that gate are ready to proceed. An event at a gate is denoted by the name of the gate. For example if we have a gate, g, the term `g` denotes some event at g, `g!v` denotes that the specific data value, `v`, is to occur at g while `g?x:Type` denotes that some value of type Type is to occur at g, and that this value is stored in the local variable x. The symbol `[]` denotes alternation,

```
PROCESS SingleLevelSecureProcess
          [networkIn, networkOut]
           (id: ProcessId,
            label : SecLabel) : noexit :=
  (
   (* send data messages *)
   (choice x: DMessage []
      [(label IsLabelOf x) and (id IsSourceOf x)] ->
       networkOut ! x ;
       exit
   )
  [] (* Receive data messages for this processor *)
    networkIn ? x: DMessage
      [(id IsDestinationOf x) and
       (Label(x) IsValidReceiptFor label)];
          exit
  ) >>
  SingleLevelSecureProcess
     [networkIn, networkOut]
     (id,label)
ENDPROC
```

Figure 2: Secure Single-Level Process Interface Specification

or choice of actions and `;` denotes sequential ordering. In addition, processes p1 and p2 can proceed in parallel completely synchronized with events at gates g1 and g2, denoted by `p1 |[g1,g2]| p2`; completely synchronized with events at all gates, denoted by `p1 || p2`; or unsynchronized and thus independent, denoted by `p1 ||| p2`. In addition, actions may be guarded, denoted by `[guard expression] -> action` and phases of execution may by sequentially composed, denoted by `phase1 >> phase2` .

In addition to the process specification notation, LOTOS has a useful data abstraction specification style. Due to space limitations, we are unable to show any of the data type specification in this paper. A copy of this paper with an appendix showing the full LOTOS specification is available from the author.

3.2 System Components

In this section we present the specification of the secure processes (both single and multi-level) and the secure network that combines them. These three components are used as the basis of the all of our system specifications. To generalize the composition as much as possible, we have specified network input and output events over separate synchronization gates. A network that has input and output on the same port can invoke this device with the same gate name for each parameter.

Secure Single-Level Process. Figure 2 defines the external interface for a single-level secure process. In this case, the process is connected to two external gates defining the network, and is parameterized by a unique process identifier and a security label; specific implementations may provide additional networks. This interface specifies that the process may perform one of two events:

- The event of sending a message. The only constraints on the message sent are that the source address in the message corresponds to the sending process and that the security label of the message is that of the sending process. This is denoted in LOTOS using the `networkOut !x` notation. The `choice` operation and the expression in brackets preceding this operation denote a parameterized composition of message transmission. The notation states that the process may send any data message, x, as long as the security label and source address of x are correct.
- The event of receiving a message. The constraints on this event are that the message was destined for this process, and that the security label of the incoming message is consistent with the security policy and the label of the recipient process. This is checked via the `IsValidReceipt` function, which may be a simple equality check, or otherwise a dominance check. This is denoted in LOTOS using the `networkIn ?x: DMessage`. This process will accept any of these messages provided they satisfy the expression in the brackets following the input command.

In sending a message, we are only concerned about labeling the message from the process appropriately. However, when receiving a message we must check that the message is destined for this process and that the security policy permits messages to this process from the source (based on the security labels of the message and the destination). Note that a simple implementation of such a security policy would be equality of security labels; otherwise there is the additional burden of showing that the `IsValidReceipt` function satisfies the security policy, and that the policy is sound. The use of the equality test maps to the separability security model while the dominates relationship maps to the restrictiveness security model.

Secure Network. The preceding specification is sufficient for single-level process interface specification, but still requires a definition of the network interface for interprocess communication. We define the network as a

```
PROCESS SecureReliableNetwork
          [networkIn, networkOut]
          (messages : Queue):noexit:=

  (* send data message to processor *)
   [not(IsEmpty(messages))] ->
      networkOut ! Head(messages)  ;
      SecureReliableNetwork
         [networkIn, NetworkOut]
         (Tail(messages))
 [] (* Receive data messages from a processor *)
   networkIn ? x: DMessage;
   SecureReliableNetwork[networkIn, networkOut]
                        (Enqueue(x,messages))
ENDPROC
```

Figure 3: Secure Reliable Network Interface Specification

simple queue that does **not** permit modification of messages. We may implement such a network in terms of a an internal communication mechanism inside a single machine operating system or as a local network and a collection of trusted network interfaces. These interfaces would be responsible for checking the validity of messages and assuring that all messages passed on to the processes are labeled correctly. Such an interface can be a stand alone hardware device, an implementation of the interface embedded on the network card or low-level network software. In any case, the algorithms and protocols used by these interfaces must be sufficient to provide the level of security desired. The specification in Figure 3 provides an abstract specification of such a secure network. Notice that this specification provides reliable in-order transmission of data. This interface specifies that the process may perform one of two events:

- The event of sending a message. The only constraint on the message is that it is the first pending message on the network queue.
- The event of receiving a message. There are no constraints on this event, but rather it just places the incoming message on the end of the queue.

Secure Multi-Level Process. The only remaining base component for our specification is a multi-level process interface. These processes are needed to model devices that must handle information from multiple security compartments concurrently, where there is no clear-cut separation between all of the compartments, or where the multi-level certification was performed independently. Examples of such components are multi-level file servers, databases, multi-level document processes, etc. As with the single-level interface, this interface specifies that the process may perform one of two events:

```
PROCESS MultiLevelSecureProcess
          [networkIn, networkOut]
          (id: ProcessId,
           labelSet : SecLabelSet) : noexit :=

  (
   (* send data messages *)
   (choice x: DMessage []
     [(Label(x) ISIN labelSet) and
      IsSourceOf(id,x)] ->
       networkOut ! x ;
       exit
   )
  [] (* Receive data messages for this processor *)
    networkIn ? x: DMessage
      [IsDestinationOf(id,x) and
       IsValidSetReceipt(Label(x),labelSet)];
          exit
  ) >>
  MultiLevelSecureProcess[networkIn, networkOut]
                         (id,labelSet)

ENDPROC
```

Figure 4: Secure Multi-Level Process Interface Specification

- The event of sending a message. The only constraints on the message sent are that the source address in the message corresponds to the sending process and that the security label of the message is in the set of valid labels of the sending process.
- The event of receiving a message. The constraints on this event are that the message was destined for this process, and that the security label of the incoming message is consistent with the security policy and the set of labels of the recipient process.

3.3 Steps to Specifying and Building a Secure System

This section discusses how we can take the given concept of separability and the components we have presented, and model a system using these methods. Given this approach, a system designer can model a secure system in a manner that readily makes apparent the portions of the system that needs to be secured. Note that the specification requires that all entities be specified as processes, whether they be physical devices, processes, threads or data objects. This simplifies the specification process and does not tie us to any particular implementation. Readers may note that this is similar to the

object-oriented paradigm and can be naturally mapped to object-oriented implementations.

The first step in the system design is to determine a top-level interface for your system and the security policy that it will maintain. We recommend formalizing the policy using a security model such as *restrictiveness* [3, 4] or *separability* [5]. Following this, we can iterate the following steps for each level of abstraction in the specification and design.

1. Isolate the processes (main components or objects) of the system. At the level of abstraction discussed in this paper, we do not specify details of the implementation of these components, but rather treat them all as abstract processes.
2. For each process, specify the security label (or labels) associated with that process. If labels can not be assigned at this time, define the method by which labels are determined and specify the set of valid labels.
3. For each process, assign an appropriate interface (multi-level or single-level). A multi-level interface is required if and only if the object will communicate with other processes at multiple security levels, or will be responsible for managing data at multiple security levels.
4. Define the network (or networks) that interconnect these processes in terms of communication paths; understanding that a specification of separate networks forces a separate network implementation.
5. Define the composite system by connecting all processes to their appropriate networks.

Given that the networks are secure and reliable, and that the components satisfy the given security policy, the composite system will be secure. This is true only if the security policies are composable [4]; restrictiveness and separability are both composable. Our previous work [1, 2] proves that the type of components discussed here are secure and that the composition maintains restrictiveness; similar work is underway for separability. Given these existing proofs, the verification and validation of the composite system is nearly automatic.

Now all we have to do is continue to iterate the process until the level of abstraction is detailed enough to permit system implementation. Note that no further security consideration is necessary below the single-level process abstraction beyond appropriate message labeling. As for multi-level processes, if they can be implemented using single-level processes we can directly use this method to ensure the security of the multi-level process (an example of this type of decomposition appears in [2]); otherwise other techniques must be used.

```
PROCESS LoginProcess
          [networkIn, networkOut,
           filesysIn, filesysOut,
           databaseIn, databaseOut]
          (IdSet : ProcessIdSet) : noexit :=

 (* Perform some internal event to authenticate user *)
 (*  generate a new process, id and label *)

  i;
  (choice id:ProcessId,
         label: SecLabel []
  [not(id IsIn IdSet)] ->
    (SingleLevelProcess[networkIn,networkOut,
                        filesysIn,filesysOut,
                        databaseIn,databaseOut]
                        (id, label)
     |||
     LoginProcess[networkIn,networkOut,
                  filesysIn,filesysOut,
                  databaseIn,databaseOut]
            (Insert(id,IdSet))
    )
  )
ENDPROC
```

Figure 5: Login Process Interface Specification

4 Exemplary System

In this section we present a simple exemplary secure system and demonstrate how to specify it using the techniques outlined in the previous section. The system consists of multiple user processes running on a single stand alone system with interprocess communication, a login process and two shared resources. The first resource is the file system, the second a shared database, both of which are considered multi-level. The login process enables users to login at a single level and generates new single-level user processes for each login. Communication between processes exists over a shared network; communication with the shared resources exists over a separate shared network for each resource. The external interface for such a system is trivial, and thus is not shown here. The security policy we follow is based on *restrictiveness*; which requires that processes with security labels that are not permitted to communicate under the security policy will not interfere with each other's operations.

4.1 Exemplary System Specification

We define each of the networks as shown in Figure 3, where the gates for the networks will be dependent on

```
PROCESS SingleLevelSecureProcess1
           [networkIn, networkOut,
            filesysIn, filesysOut,
            databaseIn, databaseOut]
           (id: ProcessId,
            label : SecLabel) : noexit :=
 (
   (* send data messages on network *)
   (choice x: DMessage []
      [IsLabelOf(label,x) and IsSourceOf(id,x)] ->
       networkOut ! x ;
       exit
   )
  [] (* Receive data messages from network *)
    networkIn ? x: DMessage
       [IsDestinationOf(id,x) and
        IsValidReceipt(Label(x),label)];
         exit
  [] (* send messages to file server *)
   (choice x: DMessage []
    [IsLabelOf(label,x) and IsSourceOf(id,x)] ->
       filesysOut ! x;
       exit
   )
  [] (* Receive data messages from file server *)
    filesysIn ? x: DMessage
       [IsDestinationOf(id,x) and
        IsValidReceipt(Label(x),label)];
           exit
  [] (* send data messages to database *)
   (choice x: DMessage []
    [IsLabelOf(label,x) and IsSourceOf(id,x)] ->
       databaseOut ! x;
       exit
   )
  [] (* Receive data messages from database *)
    databaseIn ? x: DMessage
       [IsDestinationOf(id,x) and
        IsValidReceipt(Label(x),label)];
          exit
 ) >>
  SingleLevelSecureProcess1[networkIn, networkOut,
                            filesysIn, filesysOut,
                            databaseIn, databaseOut]
              (id,label)
ENDPROC
```

Figure 6: Example Secure Single-Level Process Interface Specification

the device. This results in our needing 3 pairs of gates for the system instead of just one. The use of these pairs is shown in Figures 5-7.

We define the login process as seen in Figure 5. In this process we wait for a user to perform some valid login sequence, which we leave unspecified and denote with `i`, then fire off one instance of a single level process with an unused identifier and continue to run the login process with the identifier added to the valid identifier set. Note that we use the `|||` notation to denote full parallel composition of the single level process with the rest of the system. Since this is a special multi-level process, we need to verify its implementation separately. However, all new single-level processes generated from it can be verified independently.

Each single-level process is defined following the format of Figure 2, except that we need interface information for each network, the details are shown in Figure 6. Note that this example uses the same abstract network message data type and security checking mechanisms for each network. If this is not desired, we could make the change here. Notice that this is still in the form of the secure single level processor given earlier, and thus can easily be shown to satisfy our security policy.

Each of the file system and database resources can be specified as multi-level components as in Figure 4. Following this approach, we require that these components have unique system-wide identifiers and that the range of security labels for the devices is well defined.

The final composite system, specified in Figure 6, consists of a simple parallel composition of the login process, the networks, file system and database. All single-level user processes are dynamically added. Notice that externally we only provide a network access, the database and file system networks are *hidden*. This prevents processes outside of the composite system from accessing these internal shared resources.

4.2 Further Detailed Specification

The preceding specification provides an example of a high level specification of a system. However, we are interested in a secure development of the multi-level components as well as the secure single-level components. In this section we present an approach to developing a secure single-level component from a standard untrusted single-level component and a special interface unit, and an approach to specifying the secure multi-level file system. Proofs of the security of these components are similar to the proofs shown in [1, 2].

```
PROCESS Composite_System
          [networkIn, networkOut] : noexit :=

 Hide filesysIn, filesysOut,
      databaseIn, databaseOut in

   LoginProcess[networkIn,networkOut,
                filesysIn,filesysOut,
                databaseIn,databaseOut]
          ({} OF ProcessIdSet)
  ||
   (
     FileSystem[filesysIn, filesysOut]
        (EmptyFileSys, FileSysId, FileSysLabelSet)
   |||
     Database[databaseIn, databaseOut]
        (EmptyStack, DatabaseId, DatabaseLabelSet)
   |||
     Network[networkOut, networkIn]
   |||
     FileSystemNetwork[filesysOut, filesysIn]
   |||
     DatabaseNetwork[databaseOut, databaseIn]
   )
ENDPROC
```

Figure 7: Composite System Specification

```
PROCESS SingleLevelProcess
          [networkIn, networkOut,
           filesysIn, filesysOut,
           databaseIn, databaseOut]: noexit :=

  (
    (* send data messages on a network *)
    ( choice gate in
        [networkOut, filesysOut, databaseOut] []
          choice x: DMessage[]
            gate ! x;
            exit
    )
  [] (* Receive data messages on a network *)
    ( choice gate in
       [networkIn, filesysIn, databaseIn] []
         gate ? x: DMessage;
         exit
    )
  ) >>
  SingleLevelProcess[networkIn, networkOut,
            filesysIn, filesysOut,
            databaseIn, databaseOut]
ENDPROC
```

Figure 8: Untrusted Single-Level Process Interface Specification

A Secure Single-Level Component. The important features of a secure single-level component are how it processes messages it is sending or receiving. To trust such a component, it must appropriately label outgoing messages and correctly filter incoming messages. All other concerns about a single-level system are operational and not based on the security of the system. In other words, we can specify the secure single-level process as one consisting of a trusted interface unit and an untrusted operational component. The trusted interface unit ensures that all communication between the component and the network(s) are properly labeled and filtered.

Figure 8 shows the LOTOS specification of an untrusted single-level process. This process sends and receives messages as does the trusted single-level process, but no constraints are placed on the messages sent or received. Figure 9 shows the specification of a trusted interface unit that sits between the untrusted single-level process and the network. This unit ensure that all messages are appropriately labeled and filtered according to our security policy. Figure 10 provides the specification for the composite system. Here all communication gates from the untrusted component are explicitly routed through trusted interface units, effectively isolating the untrusted component while ensuring the incoming and outgoing messages satisfy the security property.

The notation $\mid [net1In, \ldots] \mid$ is used to force synchronization of the single-level process and the trusted interface unit. Any communication over these gates must occur with agreement from both components. Such decomposition greatly reduces system verification efforts since the only security relevant component is the trusted interface unit. Proof of the security of this type of component is straight-forward [1].

Secure Multi-Level Database. In this section we define a true multi-level device; although we have made the operation of the device simple for the sake of clarity. The device is a simple database process that receives *publish* and *acquire* requests from the connected network. Associated with each request is an object identifier, data for publish requests, and the source id and security label. For a publish request the database simply adds it to its records. For an acquire request it searches through its list of previous requests for a matching publish with the same identifier as the acquire message. It will return the first one that has a security label that satisfies the security policy with respect to a requested minimum security label and the acquire message security label. If a match is found, an appropriate response is sent, otherwise a default response is sent.

```
PROCESS TrustedInterfaceUnit
           [processIn, processOut,
            networkIn, networkOut]
           (id: ProcessId,
            label : SecLabel) : noexit :=

 (
  (* pass-on data messages from process *)
    processIn ? x : DMessage;
    networkOut ! (SetId(SetLabel(x, label),id));
    exit
  [] (* Receive data messages *)
    networkIn ? x: DMessage
    [IsDestinationOf(id,x) and
     IsValidReceipt(Label(x),label)];
       processOut ! x;
       exit
 ) >>
  TrustedInterfaceUnit[processIn, processOut,
                       networkIn, networkOut]
               (id, label)
ENDPROC
```

Figure 9: Trusted Interface Unit Specification

```
PROCESS SingleLevelSecureComposition
           [networkIn, networkOut,
            filesysIn, filesysOut,
            databaseIn, databaseOut]
           (id: ProcessId,
            label : SecLabel) : noexit :=

  (* define internal gates between single-level *)
  (*  process and interface units *)

   Hide net1In, net1Out, net2In, net2Out,
        net3In, net3Out in

   SingleLevelProcess[net1In, net1Out,
          net2In, net2Out, net3In, net3Out]
   |[net1In,net1Out,net2In,net2Out,net3In,net3Out]|
   (TrustedInterfaceUnit[net1Out,net1In,
          networkIn,networkOut](id,label)
    |||
    TrustedInterfaceUnit[net2Out,net2In,
          filesysIn,filesysOut](id,label)
    |||
    TrustedInterfaceUnit[net3Out,net3In,
          databaseIn,databaseOut](id,label)
   )
ENDPROC
```

Figure 10: Composition of Trusted Interface Units and Untrusted Process

```
PROCESS Database
           [gateIn, gateOut]
           (dataStack : Stack,
            id: ProcessId,
            labelSet : SecLabelSet) : noexit :=

  (* receive publish request *)
     gateIn ? x : DMessage
     [IsDestinationOf(id,x) and
      IsValidSetReceipt(Label(x),labelSet) and
      IsPublishRequest(x)];
         Database[gateIn, gateOut]
           (Push (PublishData(x), dataStack),
            id, labelSet)

  (* receive acquire request *)
  [] gateIn ? x : DMessage
     [IsDestinationOf(id, x) and
      IsValidSetReceipt(Label(x),labelSet) and
      IsAcquireRequest(x)];
         gateOut ! GetValidMatch(x, dataStack);
         Database[gateIn, gateOut]
           (dataStack, id, labelSet)
ENDPROC
```

Figure 11: Secure Database Process Interface Specification

The LOTOS specification of the database is given in Figure 11. The security of this process depends on the security of the functions `GetValidMatch`, which ensures that only the most recent valid publish message is seen, and `ResponseFor` which generates a response from the match. Note that this response needs to include the id and security label of the requesting message. The current state of the system is represented as a stack of publish requests that is passed as a parameter to the database after every event is processed. It must be verified that this process does not violate the information flow control security policy of the system when computing these and other auxiliary functions. Complete testing of this specification with the multi-level process will show that this is an instantiation of the multi-level interface given in Section 3.1.

An alternate version of the multi-level database requires that requests and responses are of the same level. Hence, the `GetValidMatch` function would search for a previous publish request with the same security label. Given such functionality, the database becomes a collection of single-level databases connected via a shared network. The input to this shared network would be a simple switch that would route all messages to the appropriate single-level server based on the security label. A simple approach to implementing this in LOTOS would be to create internal identifiers for each security label and associated server and have the switch trans-

late between the generic database id and the internal id. Diagrammatically, this appears as in Figure 12. The actual specification, verification and validation of this subsystem follow the same approach outlined above for the full system.

5 Conclusion

This paper discussed the concept of separability and how it can be used in the design and implementation of secure systems. Although originally presented over a decade ago, this concept has strong applicability in the design of modern systems, and can naturally be applied to object-oriented systems as well as more conventional designs. The approach taken in this paper demonstrates how we can clearly specify the security of a system in terms of the separability model. Using the formal specification approach, we can readily see the security boundaries of the system and where appropriate security measures must be implemented and security testing must occur.

As discussed in this paper, this approach can be used to specify secure applications such as databases, network services, secure networked or distributed systems, or secure operating systems. It is this wide range applicability of this approach that makes it so attractive for system design. The only portions of the system that must be verified for security purposes are any true multilevel components, the labeling and filtering portions of the single-level components, the network, and the security policy. This greatly reduces the verification and validation effort seen by many system developers who are often unsure what portions of the system and how much must be validated for security.

If the network is a virtual network implemented internally within a machine or process, we have to ensure that no other interprocess communication can occur. This problem was originally discussed and solutions presented by Rushby in his original papers [7, 8].

References

[1] J. Alves-Foss. *Mechanical Verification of Secure Distributed System Specifications.* PhD thesis, Department of Computer Science, University of California, Davis, 1991.

[2] J. Alves-Foss. Specifying trusted distributed system components. *Journal of Computing and Information*, 2(1):238–257, 1996.

[3] D. McCullough. Specifications for multi-level security and a hook-up property. In *Proc. IEEE Symposium on Security and Privacy*, pages 161–166, 1987.

[4] D. McCullough. Noninterference and the composability of security properties. In *Proc. IEEE Symposium on Security and Privacy*, pages 177–187, 1988.

[5] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 79–93, 1994.

[6] Information processing systems Open Systems Interconnection. *LOTOS – A formal decription technique based on the temportal ordering of observational behaviour.* International Organization for Standardization, International Standard 8807-02-15 edition, 1989.

[7] J.M Rushby. Design and verification of secure systems. In *Proc. ACM Symposium on Operating System Principles*, volume 15, pages 12–21, 1981.

[8] J.M. Rushby. Proof of separability: A verification technique for a class of security kernels. *Proc. International Symposium on Programming, Lecture Notes in Computer Science*, 137:352–367, 1982.

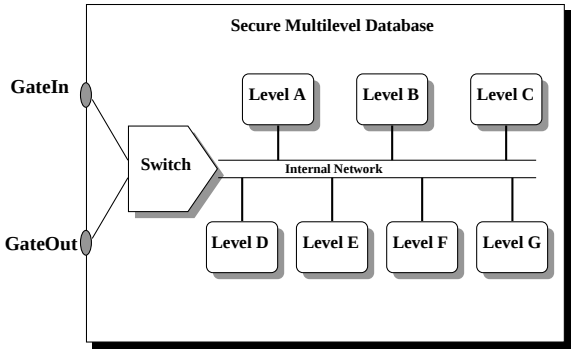

Figure 12: Secure Database Implementation