The Verification of Secure Distributed Systems

Jim Alves-Foss and Karl Levitt

Division of Computer Science
University of California, Davis
Davis, Ca 95616

Abstract

In this paper we discuss the verification of secure distributed systems. We discuss design issues related to secure distributed systems, particularly with respect to verification, and survey a few existing systems to see how they address these issues. We present specification and verification techniques that are applicable to secure distributed systems, emphasizing an approach to achieving a verified design through the hook-up of verified components. We conclude with discussing how this approach can be applied to a secure distributed system design.
Keywords: Distributed Systems, Computer Security, Verification.

1 Introduction

Informally, a system is secure if the information it stores is protected against release or modification by unauthorized users. The Multilevel Security Policy (MLS) as described in the Orange Book [1] associates security levels with users and objects, and requires that a user gets to see the contents of objects at his or lower levels; that is information can flow to higher levels but never lower levels. A more abstract model of security that avoids the need to consider objects has been formulated by Goguen and Meseguer [9, 10], Feiertag, Levitt and Robinson [8], and McCullough [16, 17] In these models the information a user observes is to be dependent on the actions of users at his level or lower. That is, the actions of higher level users cannot be observed by lower level users.

The burden of security falls on the operating system, although appropriate hardware support can minimize the impact of security features on system performance. In simple terms an operating system that satisfies the MLS policy (or other policies) must enforce access control: processes from having access to objects in accordance with the security policy. In addition, the operating system itself must not be a channel for the communication of information not in accordance with the security policy. Such unwanted information flow can potentially occur through objects managed by the operating system and to which more than one users has access; the term covert channel is often used in referring to such objects.

There have been attempts to develop systems that implement the MLS policy, most for single host/multiple user systems such as mainframes or shared workstations.

In recent years there has been increasing interest in distributed systems and, naturally, in secure distributed systems. For our purposes in this paper a distributed system consists of hosts (which could be workstations, mainframes, personal computers), servers (which could be repositories for objects accessible to multiple hosts, such as files, directories, names, passwords, etc.), and a network through which the hosts and servers communicate with each other. Security is perhaps more important for distributed systems in that such systems are likely to have many hosts and many users with different authorizations. The security policy for a distributed system is identical to the (abstract) version of MLS: A user is not to observe the actions of users except those at its level or lower. A user can be associated with a host or might be an eavesdropper on the network – passive or active.

Proposals for secure distributed systems vary according the services offered by the system. In the simplest case, each host can support a single user or, slightly more generally, users operating at the same level. In this

CH2961-1/91/0000/0177$01.00 © 1991 IEEE

case, Rushby and Randell [23], the burden of assuring security can fall on the network, which can mediate all communication between hosts to assure only those intended to communicate with each other do so. Indeed, the fact that the users are permitted to communicate only through a few well defined interfaces makes the attainment of security for this simple (albeit useful) distributed system easier than for a multiuser mainframe. More complex distributed systems would include multilevel file servers – also see Rushby and Randell.

A more general distributed system would support multilevel hosts. More complex distributed systems permit the sharing of memory or of hosts across the system; hosts could be shared through process migration. For such systems the attainment of security requires secure hosts in addition to secure interhost services.

The Orange Book defines a rating for a secure system according to the system's services in support of security and the extent of the certification of the system with respect to a security policy. The highest ratings, A1 and beyond-A1, are granted to systems that have been formally verified to satisfied the policy. A1 certification requires that the system design be verified, while beyond-A1 is more stringent in that it requires the verification of the system's implementation. For a single-host system the design is considered to be specifications of the functional behavior of each service provided by the system, e.g., system calls and ordinary instructions accessible to user processes. A few systems have been verified according to the A1 criterion. Although yet to be attempted, an interface for the processes running on any distributed system could be specified and verified.

Once the interface specification of a system has been verified, it remains to verify the implementation. Of course, a system cannot be considered to be verified unless the executable code is verified. However, many errors that could render a system insecure can be eliminated through verification of design decisions that can be formulated in stages of development well before code is produced. One approach to staging the development and verification of a distributed system is to consider it to be the interconnection of large self-contained components, each of which has a specification. McCullough has developed a methodology for the verification of systems at this level, which he calls hookup verification; the security policy is called restrictiveness. Through the methodology the hookup of a set of components is concluded to be secure provided each of the components (represented by its specification) is verified to satisfy restrictiveness and their hookup is shown to satisfy particular properties,

This paper extends, mechanizes, and applies the McCullough methodology. Our extension involves writing specifications for a class of generic building blocks, the class including filters of various kinds. Next we define a large class of components that are instantiations of these filters and show that the instantiations satisfy restrictiveness. These components include queues, transformers, multiplexors, de-multiplexors, and switches. A secure distributed system can be configured using these components as building blocks, these blocks providing the links through which untrusted components communicate. To demonstrate the utility of the methodology, we show how a set of our components can be connected to produce a verified Rushby-Randell distributed system design.

Section 2 of this paper presents a brief overview of three distributed systems for which security was the main design goal. Section 3 discusses two approaches to modeling distributed systems. Section 4 presents the restrictiveness security property. Section 5 presents description of generic component classes and the verification of the security property applied to them. Finally section 6 presents the Rushby Randell distributed secure system and discusses how one would verify it.

2 Design of Secure Systems

Current research in designing secure distributed systems focus on several key issues with the primary issue being security. System designs revolve around the security policy while providing a large amount of functionality. Another issue is integrity, where system designs attempt to ensure data integrity throughout the system. Additional issues include resource management, performance and user interface support.

This section discusses three examples of secure distributed operating systems designs the Alpha Kernel, Rushby-Randell system and the SDOS system to demonstrate various approaches to the issues involved in this type of system. Although none of these systems is a complete general purpose secure distributed operating system each of these systems has incorporated security as a major component of the total system design. Each system takes a different approach, and supplies different functionality, yet they share the ultimate

goal of developing a verified secure distributed system.

2.1 Alpha Kernel

The Alpha Kernel [20, 15] is a distributed operating system kernel for a real-time command and control system being developed at Concurrent Computer Corporation.

This kernel is meant to support a system that provides services for a single real-time command and control application. There is no concept of multiple users or a diversity of processes. The single application executes as a set of threads running through several objects (possible concurrently). These threads have real-time constraints on their performance that the kernel must be able to support.

Security is addressed using the "Single Trusted System View" of the *Trusted Network Interpretation*[2]. Interconnection between nodes is considered part of the system. Objects are maintained in a hardware protected space and accessed through the use of capabilities.

2.2 Distributed Secure System

A cost effective and highly efficient system has been proposed by Rushby and Randell [23]. This system is composed of a network of standard UNIX systems connected thorough the "Newcastle Connection" and small trusted hardware devices.

The system is effectively a distributed multilevel system which acts, from the users vantage point, as a multilevel single host system. The main technique used in this system is that each host actually runs as a single level system, thus implicitly maintaining the mandatory access controls.

The individual hosts are connected by a system called "UNIX UNITED" [5] through a local area network. Between each host system and the network is the heart of this system the "Trustworthy Network Interface Unit" (TNIU). The TNIU unit is a hardware system that guarantees via label checking and encryption that messages sent across the network maintain the mandatory access policies. In other words, a machine classified as Top-Secret can not send any messages to a machine with a lower security clearance such as Secret.

This system supports single level hosts and connects them together to develop a distributed multilevel system. It supports whatever constituent operating systems exist on the hosts. The distributed nature of the system is limited by the inter-host communication protocol established by the TNIU. We illustrate our approach to verifying secure distributed systems by applying to the Rushby Randell system in section 6.

2.3 SDOS

The SDOS system is an experimental prototype for a B3 level secure distributed operating system [24, 6]. The system is designed to allow connection of a network of hosts with a heterogeneous hardware and software base.

The main premise behind the SDOS system is that the network meets the requirements for B3 level classification, and each host runs a secure multilevel constituent operating system (COS) which also meets the requirements for B3 level classification.

SDOS exists on the host machine as a collection of programs running as separate processes on the host COS. Using SDOS, a client process can either execute on the host COS or as a client of an SDOS process. Either way, all of the local operating system functions are performed by the COS. But, whenever a client process wants access to an SDOS object or operation it must go through the SDOS system manager process. Access to non-local object is routed through the SDOS switch, a process which allows communication between SDOS kernels on separate hosts. The COS guarantees that the local objects and operations provided by the SDOS processes are protected from use by any non-SDOS processes.

3 Specification and Verification

Once the design of a system is completed, one can begin the process of specifying and verifying the system. Several research projects have developed techniques to specify these systems and verify that they satisfy a set of constraints. These constraints usually specify how the system handles concurrent access to shared memory locations, and process execution dependencies. In this section we discuss the two common models used to specify and verify distributed systems.

3.1 Concurrent Processes

This model defines the system as a collection of concurrent processes communicating through shared memory. For this model one has to be concerned about the processes *interfering* with each other and about the *liveness* of the system. A set of security constraints will limit the information flow between processes through shared memory.

Goguen and Meseguer [9, 10] developed a technique to constrain concurrent access to shared memory. This technique relies on a set of pre and post conditions for the program statements of the process that can access shared memory. These conditions specify what values may exist in the shared memory before and after the statement executes. Another technique developed by Jones [13] specifies a set of conditions for the whole process. These conditions specify the range of values that can exist in the shared memory while the process is running, and what range of values the process can place in shared memory. These two techniques expand on current sequential program verification techniques by building on an exiting base of research. One can verify these systems by using the standard sequential techniques and showing that the additional constraints will hold in all cases.

A method developed by Lamport [21, 14] incorporates liveness properties of the system using proof lattices and temporal logic. This method can be used to develop constraints on the state of the system. Once the state of the system is constrained one can reason about the temporal properties of the system, ensuring continued execution and progress.

3.2 Distributed Processes

This model defines the system as a collection of distributed processes communicating through messages. For this model one can use standard sequential techniques to verify that each process behaves as specified and that the combination of processes also behaves as specified. A set of security constraints will limit the content of the messages between processes to control the information flow.

A common method for verifying the behavior of these systems is to specify a set of *traces* for each process. These traces are sequences of messages sent and received by the process. To constrain the information flow, one specifies the set of possible traces for each process.

Chen and Hoare presented a technique for specifying these traces requiring fields for each communication channel [11, 7]. These fields specify the history of messages on the channel, the set of messages the process is ready to send on the channel, and the set of messages the process is ready to receive on the channel. One proves properties of these systems by reasoning about the values of the fields for each process.

McCullough presented a technique in [18, 17] that uses an event-based model derived from the processes of Milner [19] and Hoare [12]. This model defines systems in terms of sequences of events where each event is either a communication event (input or output), or an internal transition event. McCullough's model of event-systems requires a 4 - tuple representation consisting of the set of events, set of inputs, set of outputs, and set of valid traces for the system. A valid trace of the system is a sequence of events that is a possible event history for the system. Any two event systems will be considered identical if there exists no difference in their behavior; that is, if the set of traces for the systems are identical.

If the two event system are running in parallel and communicate by sharing some events then the combination of these two component systems is their *hook-up*. The two component systems only share events that are an input of one component and an output of the other component.

McCullough specifies the hook-up of two event systems to create a third event system. This is used in the construction of complicated systems as seen in Figure 1. The system specification can be decomposed into a collection of simpler components that behave as the system when they are hooked-up. An advantage of this decomposition is that for some predicates (hook-up predicates) one can show that if the components of a system satisfy the predicate, then the hook-up of those components will also satisfy that predicate. This enables one to prove the predicate for a simple component specification and use the hook-up property to prove the predicate for the hook-up of the components.

For the verification of a secure distributed system one needs to specify security property that states that the systems satisfies a particular security policy. One then has to verify that property with respect to the system specification. If the property can be specified by

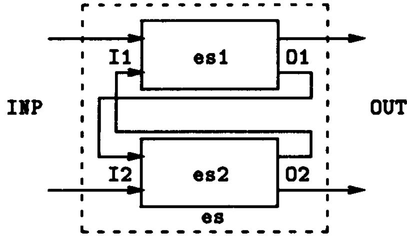

Figure 1: The hook-up of two event systems.

a hook-up predicate, the verification can be performed on simple components and then combined for the whole system.

4 A Formal Security Property

This section presents the *restrictiveness* security property defined by McCullough in [16, 17, 18]. McCullough has shown that security property can be specified by a hook-up predicate, and thus this predicate can be used to simplify verification of system specifications.

Figure 2 contains a predicate specifying McCullough's restrictiveness for event systems and an auxiliary predicate, SAME-VIEW. This auxiliary predicate is true if two traces (sequences of events) are the same when restricted to the events in a particular view (i.e. When ignoring events outside the view, the sequences are the same).

McCullough's definition assumes that all event systems are *input-total*, such that they are always ready to receive input events. The definition then states that a system is restrictive if events outside of the user's view do not affect the set of possible sequences of outputs events in the user's view.

For an event system *es* to be considered restrictive with respect to a user's view of the system *v*, we must look at all valid traces of the system. Let *a* be a valid trace of the event system, and let *b1* and *b2* be sequences of inputs of the event system such that they appear the same with respect to the view *v*. We know, due to the fact that this definition assumes the event system is input total, that *a* concatenated with either *b1* or *b2* is a valid trace of the system. Since these traces will

```
SAME_VIEW
  ⊢def ∀v t1 t2.
       SAME_VIEW v t1 t2 =
       (t1 SEQ_RESTRICT v = t2 SEQ_RESTRICT v)

IS_RESTRICTIVE_DEF
  ⊢def ∀v es.
       IS_RESTRICTIVE v es =
       (∀a b1 b2 g1.
        let EV = EVENTS es and
            TR = TRACES es and
            INP = INPUTS es and
            I_D_V = (INPUTS es) DIFF v in
         (a IN TR ∧
          SEQ_IN_SET_STAR b1 INP ∧
          SEQ_IN_SET_STAR b2 INP ∧
          SEQ_IN_SET_STAR g1 EV ∧
          ((a SEQ_CONC b1) SEQ_CONC g1) IN TR ∧
          (SAME_VIEW v b1 b2) ∧
          (g1 SEQ_RESTRICT I_D_V = NULL_SEQ) ⇒
           (∃g2.
            (SEQ_IN_SET_STAR g2 EV ∧
             ((a SEQ_CONC b2) SEQ_CONC g2) IN TR) ∧
            (SAME_VIEW v g2 g1) ∧
            (g2 SEQ_RESTRICT I_D_V) = NULL_SEQ)))
```

Figure 2: Restrictiveness as defined by McCullough.

appear the same with respect to the view *v*, the system will be secure only if the sets of possible future events, for each trace, appear the same with respect to the view *v*. Thus, if we let *g1* be a sequence of future events for the trace (a SEQ_CONC b1), the event system will be restrictive, with respect to *v*, if there exists a sequence of future events *g2* for the trace (a SEQ_CONC b2) that appears the same as *g1* with respect to the view *v*.

Using restrictiveness to verify security of a distributed system depends on the structure of the system and the ability to decompose it into simple components. One can then verify the simple components and use the hook-up property to verify the composition of those components. The proof that restrictiveness is a hook-up property has been carried out using a mechanized theorem proving system [3, 4].

5 Verifying Restrictiveness

Since restrictiveness is a hook-up property, one wants to be able to specify the system in terms of the composition of simple components, and then show that those components satisfy restrictiveness.

Although the specification of theses components may be simple, the verification of them is not necessarily simple. To simplify the verification further one can develop a set of generic building blocks that are shown to satisfy restrictiveness. Any component that is an instantiation of one of these building blocks can be easily shown to satisfy restrictiveness.

We decided to classify the system components into four classes that encompass all cases for system components of interest. Each class specifies a group of generic system components which can be modeled as filters, each processing inputs and sending outputs. Three of the classes are sequential (i.e. "first come first serve") and the last is scheduled. The components described in this below are input-total, if we expand on the work in [22] we can specify non-input total or *input limited* versions of these classes. We name these classes based on the way they relate inputs to non-inputs, on a one-to-one, one-to-many, many-to-one or many-to-many basis.

The one-to-one class consists of components that process messages on an first-come first-serve basis. Each input event is either ignored or transformed by a simple constant function and sent as an output event. The one-to-many class consists of components that process messages identically to the 1-1 class except that each input event generates a sequence of non-input events according to the function. The many-to-one class consists of components that process messages on an first-come first-serve basis. Each input event is considered either a control event or a data event. Control events do not generate any additional events, but can change the future behavior of the system. Data events are transformed based on the history of previous events, and sent as a non-input event. The many-to-many class is the catch-all. It will take input events, and process them based on the history of events. The inputs are not necessarily processed on a first come first serve basis, and the transformations are not necessarily constant.

Once we defined these classes of components we developed a generic specification for components in each class in a mechanized theorem proving system. Every component can be instantiated from the generic definition. Not only does this standardize and simplify the specification of components, it also aids in the proof of properties about the components. Since each component is a specific instance of the generic component, a proof of restrictiveness for generic component will automatically apply to the specific instance. For each of our classes we proved, for a collection of views, that the generic specification is restrictive for those views.

6 Verifying a Distributed Secure System

This section contains a description of a real system specification that was proven using the generic classes and components developed with the methodology presented above. The specification we use is based on the Distributed Secure system presented in [23].

This system consists of a collection of untrusted single level hosts, a network and a multilevel file server. The major component of this system is the Trusted Network Interface Unit (TNIU). This device is an intermediary that sits between hosts on a network and the actual network. The TNIU prevents information on the network from reaching the host if the information does not come from another host with the same security classification level, the exception being the Multi-Level file server. This server is actually a collection of single level servers and a trusted intermediary that routes the messages to the servers.

The implementation of the TNIU requires that there exist a collection of security partitions. Each host and its corresponding TNIU belong to exactly one partition.

To specify the system one needs to define the scope of the system. This includes the set of data messages, security partitions, host ids, and user view's. One also needs to define functions that determine the security classification level for host ids, system events, and data messages.

Following the methodology presented in this paper, events must be characterized as pairs. The first element of the event pair is the name of the I/O channel on which the event occurs, the other element is the message. For a network system each message must consist of at least three components. The first component is the source host ID, the second component is the destination host ID and the third component is the data message. One can define functions that will return the correct component for a given message.

Views in this system correspond to the security partitions. There is one view per partition. That view contains all events such that the classification level of the event is dominated by the view's classification level.

One determines the classification level of an event based on the classification level of the source host.

Using our methodology we have developed a high level specification of the system and shown that it satisfies restrictiveness[1]. Describing this system in our classification system is straightforward.

The TNIU consists of two components, a transformer that places the messages on the network and a filter that pulls messages off the network. The network is a broadcasting device that takes single input messages and places them on every single output port (a 1-M generator). The file server is a simple database (a M-1 programmable filter).

7 Conclusion

The burden of security falls on the operating system, although appropriate hardware support can minimize the impact of security features on system performance. In recent years there has been increasing interest in distributed systems and, naturally, in secure distributed systems. Security is perhaps more important for distributed systems in that such systems are likely to have many hosts and many users with different authorizations. The security policy for a distributed system is identical to the (abstract) version of MLS: A user is not to observe the actions of users except those at its level or lower. A user can be associated with a host or might be an eavesdropper on the network – passive or active.

This paper discussed design and verification techniques towards the goal of a verified secure distributed system. We presented a methodology based on extending, mechanizing, and applying the McCullough methodology as a technique of obtaining this goal. Our extension involves writing specifications for a class of generic building blocks, the class including filters of various kinds. We defined a large class of components that are instantiations of these filters and showed that the instantiations satisfy restrictiveness. These components include queues, transformers, multiplexors, demultiplexors, and switches. A secure distributed system can be configured using these components as building blocks, these blocks providing the links through which untrusted components communicate. To demonstrate the utility of the methodology, we showed how a set of our components can be connected to produce a verified Rushby-Randell distributed system design.

Current research is in progress to expand on these building blocks. We continue to develop and verify collections of generic components, and are developing a methodology for automating large portions of these specifications and instantiations. More complex distributed system designs are being examined with the goal of specifying and verifying them using our methodology. Eventually the methodology will be carried down to the code level implementation of specifications and used to verify secure distributed systems at a level beyond-A1.

[1] Due to the limitations of the security model we are using, we do not include any of the encryption requirements in our specification. We are currently looking into ways to expand the security model to handle encryption.

References

[1] *Department of Defense Trusted Computer System Evaluation Criteria.* Department Of Defense Computer Security Center, August 1983.

[2] *National Computer Security Center Trusted Network Interpretation.* National Computer Security Center, July 1987.

[3] J. Alves-Foss and K. Levitt. *A Model of Event Systems in Higher Order Logic: Sequence and Event System Theories.* Technical Report CSE-90-45, Division of Computer Science, University of California, Davis, November 1990.

[4] J. Alves-Foss and K. Levitt. *A Security Property in Higher Order Logic: Restrictiveness and Hook-Up Theories.* Technical Report CSE-90-46, Division of Computer Science, University of California, Davis, December 1990.

[5] D.R. Brownbridge, L.F. Marshal, and B. Randell. The Newcastle Connection, or UNIXes of the world unite! *Software — Practice and Experience,* 12:1147–1162, December 1982.

[6] T.A. Casey, S.T. Vinter, D.G. Weber, R. Varadarajan, and D. Rosenthal. A secure distributed operating system. In *Proc. IEEE Conference on Security and Privacy,* pages 27–38, 1988.

[7] Z.C. Chen and C.A.R. Hoare. Partial correctness of communicating sequential processes. In *Proc. International Conference on Distributed Computing,* 1981.

[8] R. J Fiertag, K. Levitt, and L. Robinson. Proving multilevel security of a system design. In *Proc. Symposium on Operating System Principles*, pages 57–95, 1977.

[9] J.A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Conference on Security and Privacy*, pages 11–20, 1982.

[10] J.A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc. IEEE Conference on Security and Privacy*, pages 75–86, 1984.

[11] C.A.R. Hoare. A calculus of total correctness for communicating processes. *Science of Computer Programming*, 1(1):49–72, 1981.

[12] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, London, 1985.

[13] C.B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions On Programming Languages And Systems*, 5(4):596–619, 1983.

[14] L. Lamport. *What Good is Temporal Logic?* Technical Report, Digital Research Systems Center, November 1982.

[15] K. Loepere, F. Reynolds, E. Jensen, and F. Lunt. Security for real-time systems. In *Proc. National Computer Security Conference*, pages 318–332, 1990.

[16] D. McCullough. *Foundations of Ulysses: The Theory of Security*. Technical Report RADC-TR-87-222, Odyssey Research Associates, Inc., July 1988.

[17] D. McCullough. Noninterference and the composability of security properties. In *Proc. IEEE Conference on Security and Privacy*, pages 177–186, 1988.

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

[19] R.A. Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

[20] J.D. Northcutt. *Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel*. Academic Press, Inc., Orlando, Florida, 1987.

[21] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions On Programming Languages And Systems*, 4(3):455–495, 1982.

[22] D. Rosenthal. Security models for priority buffering and interrupt handling. In *Proc. Computer Security Foundations Workshop*, pages 91–97, IEEE Computer Society Press, June 1990.

[23] J. Rushby and B. Randell. A distributed secure system. *IEEE Computer*, 16(7):55–67, 1983.

[24] S.T. Vinter, T.A. Casey, and K.A. Huber. *The Secure Distributed Operating System Design Project*. Technical Report, BBN Laboratories Inc., 1988.