

# Lessons Learned Using Alloy to Formally Specify MLS-PCA Trusted Security Architecture\*

Brant Hashii  
Northrop Grumman Corporation  
Integrated Systems  
hashii@nrtc.northrop.com

## ABSTRACT

In order to solve future Multi Level Security (MLS) problems, we have developed a solution based on the DARPA Polymorphous Computing Architecture (PCA). MLS-PCA uses a novel distributed process-level encryption scheme to provide high assurance separation between different security levels. High level security evaluations of the TCSEC and Common Criteria require formal specification. Further, in order to enhance our understanding of the model and to facilitate a high assurance implementation, we have formally specified the architecture in Alloy. This paper presents our efforts to produce the formal model and what we have learned from it. We found a number of errors and initiated design changes as a result of the modeling effort.

## Categories and Subject Descriptors

D.2.4 [Software/Program Verification]: Assertion checkers, Class invariants, Formal methods, Model checking, Validation; C.2.0 [General]: Security and protection

## General Terms

Security, Verification

## Keywords

Alloy, formal specification, high assurance, Multilevel Security (MLS), network security

## 1. INTRODUCTION

Formal methods are critical for high assurance security systems, primarily to provide a guarantee that trusted software can indeed be trusted. This paper will discuss our

---

\*Partially supported by DARPA through the Air Force Research Laboratories, USAF under agreement number F33615-01-C-1891, "Security/Trust as a Polymorphic Computing Constraint"

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FMSE'04, October 29, 2004, Washington, DC, USA.  
Copyright 2004 ACM 1-58113-971-3/04/0010 ...\$5.00.

experiences modeling a Multilevel Security (MLS) architecture for the Defense Advanced Research Projects Agency's (DARPA's) Polymorphous Computing Architecture (PCA) program.

PCA is a multi-processor architecture that allows a processor to morph during operations to provide the best type of processor for the job at hand. The idea is to provide the flexibility of general purpose processors and the performance of special purpose processors.

The goal of MLS-PCA is to create a high assurance security infrastructure for multi-processor distributed applications. By high assurance, we mean that the trusted aspects of the system needs to have been verified, at a high level, under a certification program, such as the DoD's Trusted Computer Security Evaluation Criteria (TCSEC) [31] or its replacement, the Common Criteria (CC) [1]. High levels of evaluation require formal models and analysis. Unfortunately, there are few systems that meet this level of assurance. Traditional MLS approaches involved creating a trusted operating system (TOS). However, that is hard to do at high assurance levels. The only systems certified under the TCSEC at the A1 level, the highest, is BLACKER [33], GEMSOS [17, 15], and the Boeing MLS LAN [18]. There have so far been no comparable, i.e. EAL 7, CC evaluations.

We solve this problem by projecting to the year 2020. By that time, Moore's Law indicates an increase in processing capability, allowing a multitude of cheap processors. This will allow us to assign one process per processor, removing the need to share resources. As a result, many of the covert channel issues associated with the sharing of resources are eliminated. In fact we require only a minimal runtime package, excluding the need for a TOS.

This paper discusses the formal modeling effort and lessons learned from it. We found the benefits of the formal specification to be three fold. First, the act of writing the specification forces one to examine the details of the design at a higher, more abstract level, than one would get from writing code. This allows the designers to focus on the issues that are important and abstract away those that are not. In addition, both the writing and analysis of formal specification allows one to encounter and deal with problems in the design phase. Eliminating problems after coding is extremely costly, mainly due to the need for regression testing. Finally, the ability to mathematically verify that specification holds to some criteria provides confidence that the design is correct.

The rest of the paper is organized as follows. The MLS security architecture is discussed in greater detail in Sec-

tion 2. We used MIT’s Alloy constraint language [11, 10, 9], examined in Section 3, to model the architecture. The formal model itself will be discussed in Section 4. Section 5 discusses how the modeling process influenced the design of the architecture. Section 6 reflects our experiences in using Alloy. Section 7 relates previous work to this project. Finally, Section 8 concludes.

## 2. OVERVIEW OF MLS-PCA ARCHITECTURE

Joint Vision 2020 [30] foresees a fully interconnected battle space among multinational partners. As communication is over a shared network, the main problem with distributed systems is security. Currently, most military avionics data is unclassified, with a very small percentage classified Secret or Top Secret. Typically, however, the aircraft is classified system-high, i.e., the classification of the highest level of data in the system. For example, if an aircraft has one data element that is Top Secret, then the entire plane is classified at Top Secret. In the net-centric future depicted by Joint Vision 2020, such an approach will not work. The multinational environment requires high assurance MLS to allow communications.

One problem with avionics is the age of the processor. There is always the desire to increase processing power by having the latest, fastest processors. However, such luxuries are usually not possible in avionics, due to the amount of time it takes to build and certify an aircraft plus the long operational lifetime. An alternative to having the latest processors, is to have more of the older ones.

Current avionic systems have hundreds of processors running a number of different functions including navigation, targeting, sensors, and communications. We foresee that by 2020, because of Moore’s Law, avionics systems will have tens of thousands of processors. This will allow us to design an architecture that assigns only one process to each processor. For example, there has been much research in the area of distributed agent systems [2, 12] that might facilitate such a design.

The primary threat is the occurrence of Trojan Horses. These Trojan Horses may run on any of the processors and attempt to leak data, either singly or collectively. Because we are targeting an aircraft’s avionics system, we assume some physical security over the network. An attacker cannot inject messages into the network without going through an EPE, but may be able to eavesdrop. The security property we wish to guarantee over such an environment is multilevel separation.

We solve this problem by taking advantage of Moore’s Law and dividing each application into distinct functions, each function operating at a single security level on a single processor. Each processor can then communicate with others through trusted inter-process communication (IPC). Encryption enforces the separation of security levels and provides integrity, authentication, confidentiality, and access control.

As shown in Figure 1, each single level Avionics Application Process (AAP) is front-ended by an Encryption Processing Element (EPE). All communication by an AAP must go through an EPE. All communication between EPEs is encrypted and authenticated. Keys are distributed to the EPEs by the Network Security Element (NSE) based on a

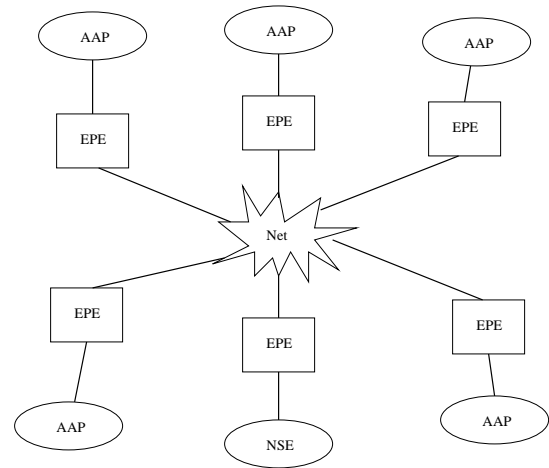


Figure 1: MLS-PCA Architecture

security policy set up by mission control. The NSE enforces both Mandatory Access Control (MAC) and Discretionary Access Control (DAC). There is a unique key for each element of the security policy. For example, there is a key for each security level and compartment in the MAC security lattice, as well as for each pair in the DAC matrix. The NSE generates a session key between two AAPs by XORing the relevant policy keys with a one-time random key. The session key is then distributed to each of the AAP’s EPE. Note that this session key must be distributed encrypted. In fact, all messages between an NSE and another AAP’s EPE must be encrypted and authenticated. This means the NSE and each EPE must already have established a session. The method for doing this is discussed in more detail in Section 5.1.

All connections between two AAPs are simplex. This allows a low level process to send information up to a high level process, but not vice versa. We also allow another type of connection, called a coalition, that consists of AAPs at a common security level and using a common key. In addition, the EPEs are also transparent to the AAPs, preventing the EPEs themselves from being used as a covert channel.

There are a number of protocols between an EPE and the NSE to perform various tasks. For instance, when an AAP, *A*, tries to send a message to another AAP, *B*, *A*’s EPE will first determine if there is already an open connection to *B*. If not, the EPE will send an open request to the NSE. The NSE will determine if permission should be granted and if so, generate a session key via XOR as described above. It will then send this session key, as well as an authentication key used to authenticate communication between the two AAPs, to each AAP’s EPE. Each EPE will acknowledge receipt of the keys. When the NSE has received both acknowledgments, it will send a synchronization message to *A*’s EPE telling it to start using the key. *A* can then transmit messages to *B* over a trusted connection protected by the session key. If *B* wishes to send messages back to *A*, *B*’s EPE must send an open request to the NSE, repeating the process in the other direction. By having each direction request and maintain a separate session key, we can allow a low level AAP to write up information to a high level AAP,

Signatures	63
Relations	64
Operations	39
Invariants	18
Predicates	38
Facts	19

Table 1: Size of Model Elements

while at the same time preventing the high level AAP to write down to the low level AAP. There are similar protocols for other operations, such as rekeying and revoking a connection, joining or leaving a coalition, and changing DAC permissions. A companion paper provides additional details on the functional model [34].

### 3. OVERVIEW OF ALLOY

Alloy was developed by Dr. Daniel Jackson at MIT for the purpose of abstract software design. This section will present an overview of Alloy.

The Alloy language is comprised of a fairly straightforward ASCII text notation. The language is based on set theory, similar to Z, with the standard set operators and quantifiers. A state is defined by sets and relationships among them. An operation will transform a state to a new state, i.e., the sets are modified. Alloy also allows the specification of invariants. The Alloy Analyzer (AA) attempts to show that no operation produces an illegal state that violates the invariants. One can then use an inductive argument to claim that if an initial state is legal and all operations produce legal states, the system cannot be in an illegal state and the specification is correct. For our application, an illegal state is a state that violates the security policy.

AA is a tool that can be used to determine if a specification is over-constrained or under-constrained. One specifies a scope size, the number of instances of a particular type of variable. If no instances are found that satisfy the constraints, then the model is over constrained. One can also specify assertions that the analyzer attempts to disprove by finding counterexamples. In this manner, one can detect if the model is under-constrained. Note that this does not create a formal proof of the model, only that the model is valid within a particular scope. The possibility exists that the analyzer might be able to find a counterexample by increasing the scope size. In our model, we were able to determine a minimal scope size, beyond which additional instances were unlikely to affect validity. For example, instantiating more than three EPEs (one for each end of the connection and one for the NSE) would not affect the correctness of establishing a connection.

### 4. OVERVIEW OF FORMAL MODEL

Table 1 gives the size of our specification in terms of the number of elements that make up the MLS-PCA Alloy model: Signatures, Relations, Operations, Invariants, Predicates, and Facts. This section will provide an overview of the formal specification with discussions and examples of each of the model elements.

#### 4.1 Signatures

The model has five basic Signatures or types: processors, processes, data, security labels, and state. The other 58 Sig-

```
sig State {
  // at most one-to-one relationship
  bound: AAP ?->? EPE,
  executingOn: Process ?->? Processor,
  // many-to-many relationship
  Inbuffer: Process -> Message,
  Outbuffer: Process -> Message,
  // stores a message while waiting for an open
  Waitbuffer: EPE -> Message,
  // stores message after decryption for
  // later processing
  Processbuffer: EPE -> Message,
  localMemory: Processor -> Data,
  members: Coalition -> AAP,
  ...
}
```

Figure 2: Excerpt from State

natures consist of subtypes of these five. There are two primary types of processes: EPEs and AAPs, with the NSE as a subtype of AAP. We also define another subtype of AAP, the SSO (System Security Officer), that has certain administrative privileges, such as changing DAC permissions and rekeying connections. There are three main types of data: messages, cryptographic keys, and audit logs. Messages are further divided into the individual message types used to implement the protocols described in Section 2. Likewise, cryptographic keys are further divided based on their use: authentication keys, encryption keys, and policy keys. This last separation was to ensure that cryptographic keys are used for the purposes for which they are intended, and, more importantly, they are not reused for another purpose [16]. If a key is reused, say in an encryption and an authentication algorithm, then a weakness in one could increase the risk of cryptanalysis on the other.

#### 4.2 Relations

The 64 Relations between Signatures consist of both dynamic Relations, defined in the state Signature, as well as static Relations defined outside of the state. Examples of the dynamic Relations include the binding of AAPs to EPEs, which processes are running on which processors, the contents of local memory and communication buffers, and current coalition membership. Figure 2 contains an excerpt from the state Signature showing these Relations.

Examples of the static Relations include message fields, both header and payload, that should not change throughout the life of the message (see Figure 3). The security labels of both processes (or principals) and data are also static Relations. We had considered making them dynamic Relations, but that would have violated the Bell-La Padula (BLP) Tranquility Principle [3, 24]. BLP is our formal security policy for the MLS-PCA model.

The figure also shows the definition of encryption as a relationship between a key, a message, and an encrypted message. The details of the relationship, i.e., the encryption algorithm such as DES or AES, is not relevant. The goal of this effort is to verify that encryption is used correctly, not that the encryption algorithm itself is adequate. Message authentication and integrity are handled in a similar manner.

```

sig Data {
  classification: securityLabels
}

sig Principal {
  // a process's clearance
  currentLevel: securityLabels
}

disj sig Message extends Data {
  // can only ever be a single process
  sender: Process,
  // can be a Process or Coalition
  receiver: Principal,
  // only the encrypted message will have a seal
  seal: option MIC,
  // keyed message integrity code, e.g. SHA1-HMAC
  auth: cryptoKey ->? MIC,
  // encrypted version of message
  encrypt: cryptoKey ->? Message
}

```

Figure 3: Description of a Message

### 4.3 Operations

The Operations describe all possible state changes. Each Operation of the NSE and EPE are modeled, as well as the AAP's send and receive operations. Many of the NSE and EPE operations reflect the protocols mentioned in Section 2. Each protocol contains a dialogue between the NSE and an EPE. Each part of the dialogue has its own Alloy Operation. For instance, the open protocol (shown in Figure 4) begins with an AAP, *A*, sending a message, *M*, to another AAP, *B*. The second Operation has the EPE taking that message, realizing that it has no key for that connection, and sending an open request to the NSE, encrypted in the session key *SA1* that is used for communicating with the NSE. The third Operation has the NSE's EPE decrypting messages and passing cleartext to the NSE. In the fourth Operation, the NSE determines if access is allowed, and either sends the session and authentication keys for *A* to talk to *B* (*SAB*) to both ends of the requested connection, or it replies with a denied access message and records an audit log. The fifth Operation has the NSE's EPE encrypting and sending the *SAB* response to *A*'s EPE and *B*'s EPE. Note that because session keys are unidirectional, another NSE-EPE key for the other direction is used by the NSE's EPE, *SA2*. There are two options for the sixth Operation. Either the AAP's EPE has the keys and decrypts the response or access is denied. Assuming access is allowed, the AAP's EPE sends an acknowledgment that the key was received. When both acknowledgments are received in Operation seven, the NSE will send a message to *A*'s EPE to start using the key. In Operation eight, *A*'s EPE will send *M*, encrypting it in the new session key, *SAB*. *B*'s EPE will now be able to receive and decrypt *M*, and finally, in Operation nine, *B* will receive *M*. An example of an operation written in Alloy is omitted due to space limitations for publication.

There are two additional Operations involved in this sequence that are not shown. One is to move messages around the network. The other decrypts and authenticates messages arriving at an EPE. There is a similar sequence of Operations for each of the other protocols mentioned in Section 2.

```

// Data is only accessible by a process if allowed.
fun invMemAccess (s:State) {
  all p:Process,d:Data | inMemory(s,p,d) =>
    canAccess(p,d)
}

// If an EPE has a key, then anything decrypted
// by that key can be read by the bound process.
fun invAllowedToUseKey (s:State) {
  all p: EPE, m1,m2:Message |
    // m1 is the plaintext, m2 the ciphertext
    encryptAndAuthenticate(s,p,m1,m2) =>
      canAccess(s.bound.p,m1)
}

// If an encryption device has a key,
// then the process is allowed
// to communicate in that direction,
// i.e. no write-downs.
// Note that NSEs are allowed to communicate
// to all processes.
fun invAllowedToCom (s:State) {
  all p1,p2:AAP - NSE | openPath(s,p1,p2) =>
    MACAllowed(s,p1,p2)
}

```

Figure 5: Invariant Examples

### 4.4 Invariants

Invariants are used to show the correctness of the model. The primary Invariants pertain to ensuring the validity of the MLS BLP policy, i.e., an AAP's security label always dominates the label of data in its memory. Figure 5 shows this Invariant written in Alloy along with some additional Invariants that are required to ensure that the policy is valid after every Operation. For example, the MLS BLP policy Invariant also requires an additional Invariant that says that an EPE only sends a message if the receiver is allowed to access it. As an EPE will only send a message if it has an appropriate key, this Invariant, in turn, requires another that says that if an EPE can decrypt and authenticate a key distribution message from the NSE, then the EPE is allowed to use that key. As the NSE can guarantee the correctness of this statement, the original Invariant can be shown to hold. In this manner, the functionality of the system can be shown to be correct.

### 4.5 Predicates and Facts

The final part of the specification includes Predicates and Facts. Predicates are boolean functions that are used to ease the specification of Operations and Invariants by providing a higher level abstraction for a set of constraints. For example, there is a Predicate that specifies both MAC and DAC constraints. There is also an encryption Predicate that defines what encryption means in the model in terms of Relations between messages and keys (see Figure 6).

Facts are additional constraints on the model, similar to Invariants, but differ in that they are true by definition. Figure 7 gives some instances, starting with a Fact used to define the relationship between security labels, and continuing with the Fact that a message's classification is derived from the message's sender.

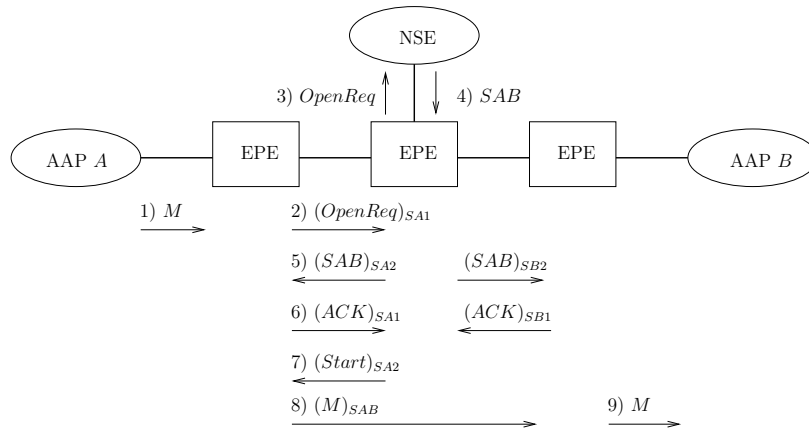


Figure 4: Open Protocol

```
// allowable access given in terms of clearance
fun canAccess(p:Principal,d:Data) {
  d.classification in p.currentLevel.dominates
}

fun MACAllowed(s:State,sndr,rcvr:Principal) {
  // receiver dominates currentLevel of sender
  sndr.currentLevel in rcvr.currentLevel.dominates
  // or the msg is to/from an NSE
  || rcvr in NSE || sndr in NSE
}

fun DACAllowed(s:State,sndr:Process,rcvr:Principal)
  // there is some entry in the DAC matrix
  { some s.DACKey[sndr][rcvr] }

// m2 is the encrypted form of m1 using key k
fun encryptedWith(m1:Message,m2:Message,k:cryptoKey)
  { m1.encrypt[k] = m2 }
```

Figure 6: Predicate Examples

```
sig securityLabels {
  dominates: set securityLabels
}
// since trichotomy is not specified,
// one can have 2 security label's
// neither of which dominates the other,
// representing different cells in the lattice
// (a security level/category pair)
fact {
  // reflexive transitive closure
  all s1: securityLabels |
    s1.*dominates in s1.dominates
  all s1,s2: securityLabels |
    s1 != s2 && s1 in s2.dominates =>
      s2 !in s1.dominates
}

fact {
  // the message's classification is the sender
  all m:Message |
    m.classification = m.sender.currentLevel
}
```

Figure 7: Fact Examples

## 5. INFLUENCE OF FORMAL METHODOLOGY ON DESIGN

The promise of formal methods is to detect errors early in the design cycle, when they are less costly to fix. This section will discuss some of the influences the formal modeling had on the design. The process of writing the formal specification resulted in early design decisions. The analysis found a number of problems.

### 5.1 Focused Design

Writing the formal specification focused the design early on important questions, e.g., how do you initialize the system? What is the initial condition, the initial secure state? The NSE needs to be able to communicate with EPEs and it needs to be able to do so securely in order to distribute keys. Since these keys should not be sent in the clear, they must themselves be encrypted. This means that the NSE and EPEs must already share a key. How is this key distributed? The key could be built into the EPE code, but then how is this code distributed and loaded in a secure manner? There must be some bootstrapping mechanism to get a key loaded into an EPE.

There are a number of options we considered. The first was to have an *ignition key* physically inserted in the NSE and each EPE. However, while this might be acceptable for the NSE, it is not acceptable for the EPEs as our model can consist of tens of thousands of EPEs. Another option was to have the key built into the hardware. The Trusted Computing Platform Alliance (TCPA) has begun work involving built-in hardware keys [28]. However, this approach suffers similar problems as there will be thousands of EPE processors. We wanted our approach to be flexible as EPEs may exist in software and/or hardware. We also wanted to map the model to existing processors. We considered using Diffie-Hellman [5] but that is susceptible to a man-in-the-middle attack. This form of attack is usually countered using certificates and public key signatures [6]. The problem with the signature approach is that it requires the EPE to have a private key, which returns us to the problem of getting a private key into thousands of EPEs.

The solution we settled on came from the realization that, while we could not load the EPE with a secret key, it could be loaded with a public key. The NSE loads an EPE/AAP

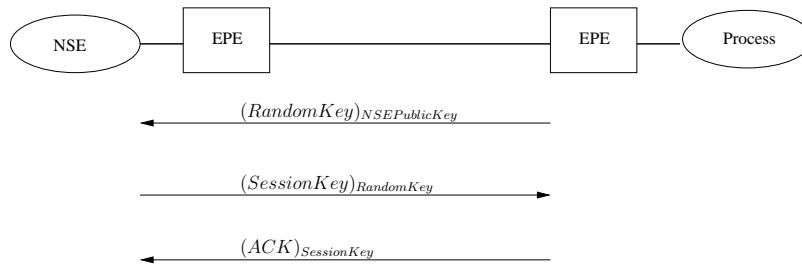


Figure 8: Hello Procedure

pair at a known address. All EPE’s have the NSE’s public key built in. Figure 8 illustrates this initialization, the Hello procedure. When the EPE is ready to begin communications with the NSE, it first generates a random key. Note that the EPE needs a source of true randomness, e.g., sensors of various physical phenomenon [29]. The EPE will then send this random key message to the NSE encrypted in the NSE’s public key. The NSE identifies the EPE by its net address and responds by sending the NSE-EPE session key message. The session key is generated via the same XOR procedure for AAP-AAP communications, *wrapped* in the EPE’s random key. The NSE is authenticated to the EPE by its use of the random key. Finally, when the EPE receives the session key, it responds with an acknowledgment encrypted in the session key.

We have shown that this protocol does not lead to a security violation given that all processes are front-ended by an EPE, as the EPE will prevent spoofing. If we expand the powers of the attacker to include the ability to insert messages directly into the network, then an attacker can steal an NSE-EPE connection by spoofing the EPE. There are two possible solutions, depending on the properties of the network being protected. If the network is small, we could use one of the rejected solutions discussed earlier, such as a manually inserted authentication key. Alternatively, we could detect an attack is occurring because after stealing the NSE-EPE connection, the EPE whose connection was stolen would no longer be able to communicate with the NSE. When the NSE receives message that it cannot decrypt, it will know that something is amiss and could take appropriate action.

## 5.2 Design Error

Another advantage of doing a formal model is the ability to examine the model for design errors, even before doing a formal analysis.

One such error was found when dealing with reboots. There are times when an avionics system needs to accommodate an inflight reboot [27]. For example, fault recovery might consist of rebooting a corrupt process. As explained later in Section 5.3, when an AAP goes down, its associated EPE must also go down. As a result, all connections previously established will be lost. We could either notify the other end of the connection, or simply re-establish the connections. We chose the latter approach. When an AAP/EPE combination comes back up, the EPE will do the initialization procedure described in Section 5.1. The NSE will then send a new set of keys for each previously opened connection involving that AAP. Note that this ap-

proach allows for the possibility of a denial of service attack by a malicious AAP against the NSE continually rebooting. However, this behavior can be detected and access from that AAP can be blocked.

On the other hand, human examination of the formal specification found a logical error pertaining to the synchronization of the new keys. There are two types of messages used to transmit keys from the NSE to an EPE: the *initial key distribution* message and a *rekey* message. Both messages require a two phase commit to ensure that both sides of the connection receive the key. Originally, these messages were erroneously treated interchangeably. However, there are cases where the distinction must be made, for example, when a *key distribution* message is immediately followed by a *rekey*. The EPEs will send acknowledgments to the NSE for both. When the NSE receives acknowledgments from both sides of the connection, the NSE will tell the EPEs to start using the key. Note that the NSE will need to be able to distinguish between acknowledgments for the *key distribution* message and the *rekey* in order to make sure that both sides get the same key.

## 5.3 Error Found Via Automated Analysis

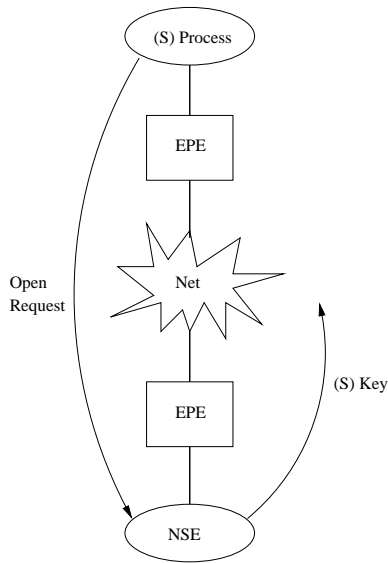
Alloy allows us to check for errors using AA. The vast majority of errors found by AA were errors in writing the spec: typos, an incomplete frame condition, and misunderstandings in learning Alloy. However, errors in the formal spec revealed errors in the functional design.

One of the earliest errors found by the Analyzer was the need for an AAP and its bound EPE to be *fate sharing*. In other words, they need to stop execution at the same time. Early versions of the specification without this feature had a problem. The spec allowed a process to die and be replaced by another AAP at a different security level, but bound to the same EPE. The analyzer found a case, illustrated in Figure 9, where a message destined for an old Secret (S) process could arrive at a new Unclassified (U) one. The problem is that messages should not be decrypted by an EPE bound to processes that are no longer running. Thus, an EPE and AAP should both go down at the same time – sharing the same fate. This has the additional advantage of flushing the EPE’s memory, clearing old session keys.

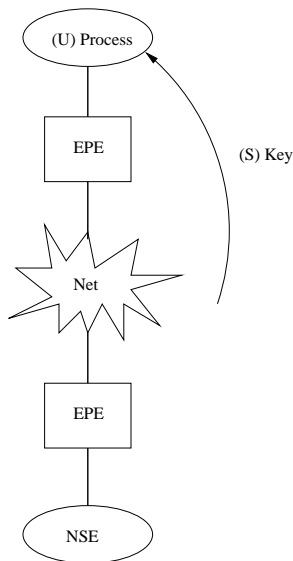
## 5.4 Design Simplifications

In our efforts to reduce the complexity and size of the specification, we discovered a number of areas where the design itself could be simplified.

One of these simplifications is the notion of a current security level and a maximum security level. BLP distinguishes



(a) A Secret bound EPE requests keys for a new connection and the NSE sends the (S) keys, but the EPE does not yet receive them



(b) The AAP reboots at Unclassified and then receives the (S) keys from the NSE, delayed by network latency

**Figure 9: The Need for Fate Sharing**

between a user's current security level and a user's maximum security level [3]. The reason is that the user might at times wish to create and release information at a lower level than the user's maximum security level. This is typically done by running a lower level process. Although it makes sense for a user to have both a current and maximum level, it makes no sense for a process to have them. Early versions of the specification contained both. However, since the MLS-PCA model only deals with the concept of processes, the maximum level was never used in any meaningful way. A process is only at one security level and does not change dynamically. If a process needs to change its security level, a new process can be created at the new level and information can be passed to it as appropriate. As a result, the notion of a maximum level could be safely discarded.

Another simplification involved the revocation operation. The original idea was that revocation would take advantage of the rekey capability. When a connection was to be revoked, one side would be rekeyed with a fake key and the other would not, thus breaking the connection. In addition to needing to manage a fake key, we also had to make sure that we did not inadvertently perform a valid rekey and re-establish the revoked connection accidentally. While writing the specification, we realized that we have another operation with somewhat similar properties called *zeroize*. It essentially causes the EPE to clear all keys, and effectively shut down. A simpler implementation of revoke is to use the zeroize mechanism, but to only zeroize keys for a particular connection, instead of zeroizing all of them. Note that here zeroize means to clear the key and stop using the key. Thus, we were able to greatly reduce the complexity of the revoke operation.

## 6. EXPERIENCES WITH ALLOY

This section discusses our experiences in using Alloy to model our architecture. We will finish by examining our experiences in creating an implementation based on the specification.

### 6.1 State Space Size

Our model is fairly large, as can be seen in Table 1 from Section 4. As a result, the greatest problem we encountered was the size of the state space. We performed our work on a 300MHz, 128 MB Sun UltraSPARC-II running Solaris 2.6, and a 2.26 GHz, 512MB IBM Pentium 4 running Windows 2000. If the state space was too large, our development machines would run out of memory, resulting in the analyzer either crashing or hanging.

The need to keep the model as small and simple as possible required simplification and abstraction. For example, we assumed reliable traffic. The abstract formal model should not specify how a network is made reliable, only that it is. This allowed us to not require acknowledgments after every message, as one would find in TCP/IP; we only needed to synchronize state between the various elements. This also allowed us to forgo timers and, as a result, an explicit time variable. Likewise, encryption is a simple relation between a plaintext message, a ciphertext message, and a key. The details of the encryption algorithm, e.g., DES or AES, do not need to be specified.

However, sometimes the need to reduce the state space resulted in specifications that were not ideal. For example, consider the case of a revoke message being sent immedi-

ately after a key distribution message. There is a possibility that the EPE receives the messages out of order, i.e., revoke, then a new key. In such a case the EPE might accept a key for a revoked path and thereby allow unauthorized communication. Thus, messages must be delivered in order. As a buffer is modeled as a set, which is unordered, this proved problematic. Ordered delivery is typically done using sequence numbers. Unfortunately, this would have increased the state space by adding a new Signature type. Likewise, Alloy provides an ordered list package that could have been used for the communication buffers. This too would have increased the state space. We eventually decided to modify the definition of the message buffers so that only key distribution messages can be sent more than one at a time. Effectively reducing the buffer size to one forces an ordering in that a message must be processed before the next is sent. This solution has the additional benefit of forcing a smaller state space. Unfortunately, it adds an unnecessary constraint to the model. However, since an Operation can only process one message at a time anyway, we felt that this constraint would not prevent finding bad states.

## 6.2 Revocation Synchronization

The model's lack of an explicit notion of time, as discussed in the previous section, can, however, result in problems. For example, the NSE and EPE need to synchronize when DAC permissions are revoked or an AAP is removed from a coalition. One of the Invariants the analyzer checks says "a communications path between AAPs is in existence only if the MAC and DAC policies allow it." The problem is that DAC permissions can change and there is a delay in communicating that change to the EPEs. As a result, there will be a period of time where a path is in existence and it is not allowed. Most operating systems do not revoke access immediately because of the complexity involved in dealing with this synchronization issue. The Multics operating system, on the other hand, did implement immediate revocation [13]. The analyzer noticed that, unlike Multics, our revocation couldn't be immediate due to the distributed nature of our architecture. However, we make an attempt to revoke as soon as possible.

As a result, the analyzer found a case where the Operation for changing permission invalidated the DAC Invariant. Normally, this sort of Invariant check is used to determine if an Operation is under-constrained. An alternative possibility, as was the case here, is that the Invariants are not valid and it is they that are over-constrained. The solution was to weaken the DAC Invariant and return to the policy that most operating systems have: only guarantee DAC is true for new paths.

## 6.3 Experiences During Implementation

After having corrected the problems discovered in Section 5, we began implementing a prototype in order to further shake out the model. The PCA paradigm assumes multiple CPUs per chip. As we did not have thousands of processors on which to test, we simulated the environment using grid computing. The implementation raised two kinds of additional design issues.

The first case resulted from omissions in the Operations. Although these omissions did not violate the security Invariants, they are omissions that need to be addressed in the running system. For example, when an AAP process re-

boots, the NSE automatically re-established the AAP's pair-wise connections, but not its coalition keys. The NSE either needs to re-establish those keys or remove the rebooted process from the coalition. We decided that since the AAP is responsible for joining coalitions, it could be responsible for re-joining a coalition after it reboots. Thus, the preferred solution is for the NSE to remove the rebooted AAP from coalition membership until a new join request is made. Another example is that the `NSESendRekey` Operation only considers pair-wise connections, not coalitions. Note that in both case, there is no security Invariant violated. The reason these problems were not detected is that we did not write Invariants to cover these cases, as we had not thought about them at the time.

The second area of design considerations are problems due to functional or operational errors. For instance, the model says that the EPE and its bound AAP should start and stop at the same time. In reality, the EPE will probably have to start first in order to facilitate the loading and execution of the AAP. Note that the *fate sharing* solution discussed above is only necessary for stopping execution, not starting. Another example is that the model treats an NSE-EPE rekey the same as any other rekey. When a rekey occurs, the new key is distributed to the EPE via the NSE-EPE session key. However, a rekey should not be encrypted in the key it is replacing. This is because if the key being replaced has been compromised, the rekey can still be done securely. Thus, the NSE-EPE session key needs to be rekeyed using a different mechanism. The solution is to reuse the bootstrap mechanism for setting up the initial session key.

Although there were a few areas where the implementation uncovered problems, the formal model provided a framework from which to analyze the problems and devise solutions. Since many of the problems were found during the design, the implementation required very little re-coding.

## 7. RELATED WORK

There has been a great deal of work formally specifying systems for security certification [31, 33]. In addition, Rushby not only examined the various types and degrees of formal methods in general use, but also applied formal methods to the development and certification of critical software [22], in particular the avionics safety certification standard DO-178B [19].

There has been additional work modeling policies, such as flow control models [25] and noninterference [7]. BLP is a special case of these policies. The main difference between BLP and flow control is that BLP is transitive and flow control can be non-transitive. This allows a flow control policy to say that two entities can only communicate through a communication channel. This is consistent with our architecture in that AAPs can only communicate through EPEs.

In noninterference policies domain  $A$  does not interfere with domain  $B$  if  $A$  cannot influence  $B$ 's output. The advantage of noninterference is that it is a statement of pure policy with no specification of mechanism. Our goal, however, is to specify a mechanism and show that it adheres to a particular security policy. Rushby showed that transitive noninterference is identical to MLS [21].

Noninterference is a key element of current avionics design. In the early 1980s, the Air Force created the Pave Pillar architecture which consists of modular components running on a distributed network on board an aircraft [14].

A recent trend in Integrated Modular Avionics (IMA) is to run multiple modules on one processor. A separation kernel, based on the concept of noninterference, is used to protect each of the modules [20]. This approach has been formally examined [8]. Di Vito applied formal methods to cooperative noninterference, a version of noninterference that allows communications channels, with the goal of showing that federated and integrated avionics architectures are equivalent [32].

A separation kernel can be thought of as a collection of virtual processors, or partitions, running on a single physical processor. The approach presented in this paper takes this a step further by putting each virtual processor on a separate physical processor. Not only does the parallelism increase performance, but the physical separation simplifies the effort of proving trust. In addition, by existing in the network layer, MLS-PCA is independent and transparent to the AAP's operating system, allowing MLS-PCA to be inserted into existing systems.

Note also that a separation kernel only provides noninterference. A separation kernel also allows the partitions to communicate through ports, but there is typically no access control policy over these ports. MLS-PCA can provide that policy. It can run on top of the separation kernel, where each partition maps to a processor in MLS-PCA's formal model.

Previous MLS formal modeling efforts involved lengthy proofs [33, 26]. Alloy's automated consistency checker avoids this, yet still has much of the benefits provided by previous system, catching logic errors.

The MLS-PCA architecture is similar to Rushby and Randell's security architecture for the Newcastle Connection [23]. The EPE serves the same function as their trustworthy network interface unit (TNIU). The main difference is that the TNIU is manually keyed for duplex communication at a single level. We envision MLS-PCA to be used with tens of thousands of processors as part of an avionics system. Manual keying in such an environment is not feasible. They provide a write-up feature via a trusted data store. This provides greater reliability at a cost of greater complexity and a loss of transparency.

## 8. CONCLUSION

We found the formal specification to have three main benefits: it forces designers to look at the details of the architecture at an early stage of development; problems are detected while the solution is still cheap; and verification provides evidence that the requirements are maintained.

Alloy provides a necessary middle ground between performing expensive, complex formal proofs and no formal analysis at all. We found Alloy fairly easy to learn and use, particular since it is based on well understood notions of sets and first-order logic. The formal modeling helped facilitate and advance our thinking on the functional model. The formal model also found some problems in the design that needed to be addressed, and forced us to simplify the design by eliminating unnecessary components.

We have implemented a prototype of the architecture using grid computing [4]. This helped us to further refine the design. Many of the problems uncovered by the implementation were either functional correctness criteria not considered during the formal specification, or operational realities of which we were not aware.

Our next step is to obtain performance data to show the feasibility of the architecture in a real application. We have chosen to parallelize an image processing application, dividing a large image into multiple smaller images distributed to multiple grid processors. Our initial tests necessarily consisted of a small number of AAPs. We need to increase this number in order to determine when the NSE becomes a bottleneck. We are also revising the design of the NSE to allow multiple NSEs for redundancy and load balancing. The goal is to demonstrate MLS-PCA as a practical method to achieve security and performance for 21st century avionics.

## 9. REFERENCES

- [1] *Common Criteria for Information Technology Security Evaluation, Version 2.1*, August 1999. Available at <http://www.radium.ncsc.mil/tpep/library/ccitse/ccitse.html>.
- [2] Earl Barr, Raju Pandey, and Michael Haungs. MAGE: A distributed programming model. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS '01)*, pages 303–312, Mesa, AZ, April 2001.
- [3] David E. Bell and Leonard J. La Padula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, ESD/AFSC, Hanscom AFB, Bedford, MA, 1975. Available at <http://csrc.nist.gov/publications/history/>.
- [4] Viktors Berstis. Fundamentals of grid computing, 2002. Available at <http://publib-b.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/redp3613.%html?Open>.
- [5] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.
- [6] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, 1992.
- [7] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, April 1982.
- [8] David Greve, Matthew Wilding, and W. Mark Vanfleet. A separation kernel formal security policy. In *Fourth International Workshop on the ACL2 Prover and Its Applications (ACL2-2003)*, Boulder, CO, July 2003. Available at <http://hokiepokie.org/docs/>.
- [9] Daniel Jackson. *Micromodels of Software: Modelling & Analysis with Alloy*. MIT Lab for Computer Science, November 2001. Available at <http://sdg.lcs.mit.edu/alloy/book.pdf>.
- [10] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *Proceedings of the ACM SIGSOFT Conference on the Foundations of Software Engineering / European Software Engineering Conference*, pages 62–73, Vienna, Austria, September 2001.
- [11] Daniel Jackson and Jeannette M. Wing. Lightweight formal methods. *IEEE Computer*, pages 21–22, April 1996.

- [12] Martha L. Kahn and Cynthia Della Torre Cicalese. The CoABS Grid, January 2002. Presented at Goddard / JPL Workshop on Radical Agent Concepts, Tysons Corner, CA. Available at <http://coabs.globalinfotek.com/public/downloads/Grid/documents/010904Ka%hnCicaleseNASA.pdf>.
- [13] Paul A. Karger. New methods for immediate revocation. In *1989 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 1989. IEEE Computer Society. Available at <http://www.multicians.org/biblio.html>.
- [14] Lockheed Sanders, Inc., Hughes Aircraft, and ISX Corporation. RASSP architecture guide rev. b, January 1995. Available at [http://www.eda.org/rassp/documents/sanders/arch\\_guide\\_b.pdf](http://www.eda.org/rassp/documents/sanders/arch_guide_b.pdf).
- [15] National Computer Security Center. Final evaluation report, Gemini Computers, Incorporated, Gemini Trusted Network Processor, version 1.01, June 1995. Available at <http://www.geminisecure.com/resource.htm>.
- [16] National Institute of Standards and Technology. Special publication 800-57 recommendation for key management part 1: General guideline, January 2003. Draft, Available at <http://csrc.nist.gov/CryptoToolkit/tkkeygmt.html>.
- [17] National Security Agency. GTNP version 1.01, network component M only, EPL entry CSC-EPL-94/008, September 1994. Available at <http://www.radium.ncsc.mil/tpep/epl/entries/CSC-EPL-94-008.html>.
- [18] National Security Agency. MLS LAN version 2.1, network component MDIA, EPL entry CSC-EPL-94/006, September 1994. Available at <http://www.radium.ncsc.mil/tpep/epl/entries/CSC-EPL-94-006.html>.
- [19] RTCA. RTCA/DO-178B: Software considerations in airborne systems and equipment certification, 1992. Committee SC-167/Eurocae WG-12.
- [20] J. M. Rushby. Design and verification of secure systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, volume 15, pages 12 – 21, Pacific Grove, CA, USA, December 1981. ACM Press. Available at <http://www.csl.sri.com/papers/sosp81/>.
- [21] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1992. Available at <http://www.csl.sri.com/papers/csl-92-2/>.
- [22] John Rushby. Formal methods and their role in the certification of critical systems. Technical Report SRI-CSL-95-1, Computer Science Laboratory, SRI International, Menlo Park, CA, March 1995. Also available as NASA Contractor Report 4673, August 1995, and to be issued as part of the FAA Digital Systems Validation Handbook <http://www.csl.sri.com/papers/csl-95-1/>.
- [23] John Rushby and Brian Randell. A distributed secure system. *IEEE Computer*, 16(7):55–67, July 1983. Available at <http://www.csl.sri.com/users/rushby/abstracts/computer83>.
- [24] Ravi S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, 1993. Available at <http://citeseer.nj.nec.com/article/sandhu93latticebased.html>.
- [25] Gerhard Schellhorn, Wolfgang Reif, Axel Schairer, Paul A. Karger, Vernon Austel, and David Toll. Verification of a formal security model for multiapplicative smart cards. In *6th European Symposium on Research in Computer Security (ESORICS)*, volume 1895 of *Lecture Notes in Computer Science*, pages 17–36, Toulouse, France, October 2000. Springer. Available at <http://citeseer.nj.nec.com/schellhorn00verification.html>.
- [26] Richard E. Smith. Cost profile of a highly assured, secure operating system. *ACM Transactions on Information and System Security*, 4(1):72–101, February 2001.
- [27] John A. Tirpak. The F-22 on the line. *Air Force Magazine*, 85(09), 2002. Available at <http://www.afa.org/magazine/Sept2002/0902raptor.html>.
- [28] Trusted Computing Platform Alliance. *Trusted Computing Platform Alliance (TCPA) Main Specification Version 1.1b*, February 2002. Available at <http://www.trustedcomputing.org/tcpaasp4/index.asp>.
- [29] K. H. Tsoi, K. H. Leung, and P. H. W. Leong. Compact FPGA-based true and pseudo random number generators. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, California, USA, 2003. Available at <http://www.cse.cuhk.edu.hk/~phwl/publications.html>.
- [30] United States Joint Chiefs of Staff. Joint vision 2020, May 2000. Available at <http://www.dtic.mil/jointvision/jvpub2.htm>.
- [31] U.S. Department of Defense. DoD trusted computer system evaluation criteria, December 1985. Available at <http://www.radium.ncsc.mil/tpep/library/rainbow/index.html>.
- [32] Ben L. Di Vito. A model of cooperative noninterference for integrated modular avionics. In *Proceedings of Dependable Computing for Critical Applications (DCCA-7)*, San Jose, CA, January 1999. Available to <http://shemesh.larc.nasa.gov/people/bld/>.
- [33] Clark Weissman. BLACKER : Security for the DDN, examples of A1 security engineering trades. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 286–292, 1992.
- [34] Clark Weissman. MLS-PCA: A high assurance security architecture for future avionics. In *Proceedings of the 19th Annual Computer Security Applications Conference*, pages 2–13, Las Vegas, Nevada, USA, December 2003. IEEE Press.