Design and Validation of a General Security Model with the Alloy Analyzer *

Charles L. Chen[1], Paul S Grisham[1], Sarfraz Khurshid[2], and Dewayne E. Perry[1]
[1]Empirical Software Engineering Laboratory
[2]Software Verification and Test Group
Department of Electrical and Computer Engineering
The University of Texas at Austin
{clchen,grisham,khurshid,perry}@ece.utexas.edu

ABSTRACT
We define secure communication to require message integrity, confidentiality, authentication and non-repudiation. This high-level definition forms the basis for many widely accepted definitions of secure communication. In order to understand how security constrains the design of our secure connectors, we have created new logical formulas that define these security properties. Our novel definitions use first-order epistemic and modal logics to precisely describe the constituent properties of secure communications. Our definitions should be applicable to describe security in the general case. We subsequently codified our logical formulas into the Alloy language and executed them using the Alloy Analyzer to validate that our models are correct. This paper presents the definition of our security model, our Alloy implementation, and the results of our validation efforts.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification—*validation*; F.3 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms
Security, Verification

Keywords
Alloy, Security, Validation, First-Order Logic, Modal Logic

*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, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

First Alloy Workshop '06 Portland, Oregon USA

1. INTRODUCTION
Non-functional requirements have a profound impact on software architecture and accommodating them often requires significant changes to the functional architecture. One critical example is security. In particular, we are interested in how the functional architecture is affected when security needs to be included at the architecture design stage.

The Perry/Wolf model of architecture defines an architecture of a system as elements, form, and rationale [15]. The elements are computational components and connectors between those components. We are interested primarily in the notion of *secure communication*, that is, how components communicate with each other securely. In this paper, we do not concern ourselves with intra-component or physical definitions of security, as well as denial of service attacks. The primary insight is that standard connectors between components can be replaced with a *Secure Connector*-type connector, which adds security properties, policies, and protocols to the standard connector's operational behavior. The idea is accepted in the WS-Security standard [14], which transparently adds security properties to Service Oriented Architecture [13] applications.

Unfortunately, a precise definition of security requirements at the architecture level has not been generally available. Security is largely understood in terms of specific kinds of vulnerabilities. While designing an architecture for security, it became obvious that we did not have a detailed enough definition of security to be able to perform rigorous analysis of our design. In order to enable the kind of analysis that would allow us to understand the security properties of our architectural design, we had to create a general definition of security, and relate that definition to our architectural elements.

We define secure communication as requiring message integrity, confidentiality, authentication and non-repudiation. This high-level definition forms the basis for many widely accepted definitions of secure communication. In order to understand how security constrains the design of our secure connectors, we have created new logical formulas that define these security properties. Our novel definitions use first-order epistemic and modal logics to precisely describe the constituent properties of secure communications. Our definitions should be applicable to describe security in the general case. We subsequently codified our logical formulas

into the Alloy language and executed them using the Alloy Analyzer to validate that our models are correct. This paper presents the definition of our security model, our Alloy implementation, and the results of our validation efforts.

Section 2 describes the epistemic logic we used to define security as well as the system model and communication assumptions we used. Section 3 presents our working security model. Section 4 gives an overview of the design of our Alloy models, as well as the results of our Alloy analysis and some discussion of the consequences of our design. Section 5 discusses the application of Alloy to other security concerns and other approaches to automated validation of security. Section 6 discusses our plans to extend this work.

2. LOGICS OF SECURITY

We considered several logical systems in order to precisely define our security model. We describe the semantics of these systems in this section, as well as a description of the system models.

2.1 BAN Logic

BAN logic is an epistemic logic used to describe beliefs in the context of secure authentication protocols [4]. In conventional descriptions of communication protocols, the source, destination and contents of each message are symbolically listed. In BAN logic, these descriptions are replaced by logical formulas which represent an idealized version of the original message. This idealized version is then annotated with assertions, which describe the beliefs of the principals (sender, receiver and other principles trusted by them) at each step of the protocol. The protocol can then be analyzed using a set of inference rules.

BAN has shortcomings which made it incomplete for use with our approach to defining security. The most significant is that BAN logic was designed primarily for reasoning only about authentication protocols. Specifically, BAN proofs assume confidentiality and integrity are assured through the use of an effective cryptographic system, which guarantees that encrypted message contents are secret and that encrypted messages can not be modified without decrypting. BAN requires the assumption that the cryptosystem is not breakable, and that a public key infrastructure with a trusted certification authority exists. Moreover, BAN does not provide a general set of invariants or constraints to define a secure protocol.

BAN is also not directly suitable for non-repudiation protocols, although some limited success has been reported recently among researchers in using BAN and BAN-like logics for proofs of non-repudiation [2, 3].

Another shortcoming is that BAN, as originally described, and as commonly used, does not have precise semantics for its operations. The semantics are descriptive rather than definitive, and as such, not entirely appropriate for use with automated reasoners without providing more precise semantics. Several attempts to more precisely define BAN semantics have been reported [1], though most of them have been subsumed by extensions to BAN which added additional expressiveness to the core logic [8].

- Axioms
  - A1. All tautologies of propositional calculus
  - A2. $(K_i\varphi \wedge K_i(\varphi \Rightarrow \psi)) \Rightarrow K_i\psi)$ (Distribution Axiom)
  - A3. $K_i\varphi \Rightarrow \varphi$ (Knowledge Axiom)
  - A4. $K_i\varphi \Rightarrow K_iK_i\varphi$ (Positive Introspection Axiom)
  - A5. $\neg K_i\varphi \Rightarrow K_i\neg K_i\varphi$ (Negative Introspection Axiom)
- Inference Rules
  - R1. From φ and $\varphi \Rightarrow \psi$ infer ψ (Modus ponens)
  - R2. From φ infer $K_i\varphi$ (Knowledge Generalization)
- Corollaries
  - A6. $\neg K_i false$ – from A1, A3, and R1
  - A7. $\varphi \Rightarrow K_i\neg K_i\neg\varphi$ – from A3 and A5

Figure 1: The Logical System S5

2.2 Knowledge Logic S5

In order to make up for the shortcomings of BAN in describing general concepts of knowledge and messaging, we decided to use a more general knowledge logic for representing our general security model. Specifically, we opted to use S5 [7], which is a sound and complete logic for describing knowledge. S5 has been previously used to describe other distributed system problems [16, 9], and is easily extendible with temporal operators and suitable for use with automated reasoning systems [17].

The basic operator in S5 is K, which is used in the predicate $K_P(\phi)$ (P knows some fact ϕ). The semantics of K derive from the possible worlds model which refers to the set of possible worlds (models) that may be true given what a process knows. In other words, if ϕ is true in all the possible worlds P considers valid, $K_P(\phi)$ holds.

The set of axioms and inference rules that define S5 are presented in Figure 1. We chose to use the full expressive power of S5 in constructing our security definitions because it is sound and complete over the class of models we wanted to consider for our security model.

2.3 Communication Model

We extended S5 to include a model of process communication similar to the one used by Chandy and Misra [5]. In this model, processes learn new facts by communicating. We allow two new formulas $Rcvd_P(\phi)$ and $Sent_P(\phi)$ to describe the fact that at some time in the past, a process P received a message containing ϕ, or sent a message containing ϕ, respectively. In addition, we allow that processes are allowed to automatically know facts which are local to themselves, so for instance, $Rcvd_P(\phi) \Rightarrow K_P(Rcvd_P(\phi))$ is an inference rule in our system.

We make the following assumptions about our communi-

cation model. We view the system as made up of some finite number of processes or components. We assume reliable broadcast messaging semantics, so that if a message is sent on a channel, all listening processes will eventually receive the message. Any processes can hear any message sent by any other process, but we do not consider $\forall P : Sent_Q \phi \Rightarrow Rcvd_P \phi$ to be interesting in the general case. We are usually only interested in the receiving process if the receiving process is the intended receiver or some malicious process attempting to break the security. We believe that these semantics accurately represent an operating environment that must be secured against attack.

3. SECURITY DEFINITIONS

We define security to mean the joint characteristics of *confidentiality*, *integrity*, *authenticity*, and *non-repudiation*. If a system can enforce these four properties at all times, then we say that a system is *secure*.

3.1 Confidentiality

Confidentiality is the concept that messages can only be read by their intended readers. No other readers may read the message. By the nature of our communication model, we allow that these other readers may *see* the message, but they may not *read*, or come to know, the contents of that message. For a cryptography-based system, this means that the non-intended readers (those readers who do not possess the decryption key), cannot deduce any or all of the plaintext message from the cryptotext.

We did not want to restrict our generalized model of security to cryptography-based systems, so we created the abstract notion of a security policy. To express a secured message M using policy i, we use the notation $[M]_i$. For a security policy i, there are a set of processes which have the capability to compose or write messages using this policy. That is, given M, these processes can produce a valid $[M]_i$. We say that these processes have *write permission* for policy i. We use the notation $K_P([]_i)$ to mean that process P possesses (or knows) write permission on policy i.

There are also a set of processes which have the capability to read messages using this policy. That is, given $[M]_i$, they can produce the original M. We say that these processes have *read permission* for policy i. We use the notation $K_P([]_i^{-1})$ to mean that process P possesses (or knows) read permission on policy i.

Returning to our definition of confidentiality, we wished to express that a given process which does not have read permission on i cannot add the contents of $[M]_i$ to its knowledge after receiving a message containing $[M]_i$. In order to express this, we were forced to use temporal operators $\bullet$ and $\circ$. $\bullet$ modifies the following formula to mean that the formula held in the prior state, that is before some action takes place. $\circ$ modifies the following formula to mean that the formula holds in the next state, that is after some action takes place.

To express confidentiality, we use the following formula:

$$Rcvd_P([M]_i) \wedge \bullet\neg K_P(M) \wedge \neg K_P([]_i^{-1}) \Rightarrow \circ\neg K_P(M)$$

This formula expresses the notion that process P received a message containing a protected message M, it did not previously know M, nor does it have read permission for the policy which secured M. As a result, P cannot learn M from the message.

3.2 Integrity

Integrity captures the notion that a message, once protected, is considered an atomic unit and cannot be modified in whole or in part by a process. To change a message in practice requires that a secured message must be read to access its protected contents, the contents changed while they are unprotected, and then re-secured using the same policy so that it appears that which message is original and which is modified message cannot be determined. In practice, integrity means that attempts to change the message without write permission destroy the integrity of the protected message, so that after reading the contents, the modification can be determined. We have added a predicate, $isValid([M]_i)$, which is true exactly when the contents of $[M]_i$ can be read (the message unsecured and M extracted) and those contents are meaningful and correct.

To express integrity, we use the following formula:

$$isValid([M]_i) \Rightarrow (\exists P \in \mathbb{P} : K_P([]_i)) \wedge Sent_P([M]_i)$$

3.3 Authenticity

Authentication is the process of identifying who said what, or specifically, which process sent a particular message. In order to accomplish this, several practical mechanisms are used in the real world. One is the notion of a shared secret, such as a personal identification number (PIN) for a bank machine. The bank compares the account number (public information) against the entered PIN (secret information) and decides that if the PIN corresponds to the account number, then the user is an authorized account holder. In public/private key encryption systems, the private key is a secret owned by the sender, and any public-key holder which reads a valid message can know who the sender was.

Our approach uses a similar technique to the public/private key encryption method, though we do not specify encryption as a security policy. Instead we use the notion of write permission to identify the sender. In our model, a set of processes may possess write permissions, so our definition only narrows the identification of the sender to a member of that set. In the case where a policy has exactly one process with write permission, our definition gives us exact authentication.

To express authenticity, we use the following formula:

$$Rcvd_R([M]_i) \wedge isValid([M]_i) \Rightarrow K_R(\exists S \in \mathbb{P} : K_S([]_i) \wedge Sent_S([M]_i))$$

3.4 Non-repudiation

Non-repudiation refers to the inability of a process to deny sending a message it has received or to deny receiving a message it has received. Non-repudiation is difficult to express in both BAN and S5 because knowledge about messages that have been sent and received are local to the sender and the receiver respectively. To express non-repudiation, we had to include the notion of a process knowing about the sending

or receiving, even though for true non-repudiation, another process does not necessarily know about the sending or the receiving. Our model works because we do not disallow the case that the process that knows about the sending or receiving process is the sender or receiver.

To express non-repudiation on the sending of a message, we use the following formula:

$K_P(Rcvd_R(M)) \Rightarrow Rcvd_R(M)$

To express non-repudiation on the receiving of a message, we use the following formula:

$K_P(Sent_S(M)) \Rightarrow Sent_S(M)$

3.5 Other Definitions

In addition to these formulas, which define the four characteristics of security, we needed to add some additional formulas to make our system useful. The most important of these formulas we call *applicability*, and expresses the idea that a process with read permission can access the contents of a protected message. Without this formula, no secured message could ever be read, which would be a perfectly secure system but a practically useless one.

Applicability of a security policy is expressed by the following formula:

$$isValid([M]_i) \wedge Rcvd_R([M]_i) \wedge K_R([]_i^{-1}) \Rightarrow K_R(M)$$

The other property we require is *blind propagation*, which expresses the idea that a process may forward or replay a message it does not have write permission for. Moreover, the formula expresses that the message was originally sent by some process which did have write permission. Combined with integrity and the rules for how knowledge is transmitted via messages our system is able to express messages that are echoed, replayed, delayed, re-transmitted, and so on.

Blind propagation of messages is expressed by the following formula:

$$Sent_S([M]_i) \wedge \neg K_S([]_i) \Rightarrow \exists P \in \mathbb{P} : K_P([]_i) \wedge Sent_P([M]_i)$$

Similarly, we can also express the idea that the propagating process initially received the message with the following formula:

$$Sent_S([M]_i) \wedge \neg K_S([]_i) \Rightarrow Rcvd_S([M]_i)$$

4. ALLOY MODELS

To validate that our definitions of security were consistent and compatible with each other, we created an Alloy model for security. This section presents some basic information about Alloy, the architecture of our Alloy security model, the rules of this model, the translation of the security definitions into Alloy, the obstacles to each of these security definitions, and our results.

4.1 Alloy Basics

The Alloy Analyzer [10] is a constraint solver for analyzing (verifying and validating) models written in Alloy. It supports two kinds of automatic analysis: simulation, in which the consistency of an invariant or operation is demonstrated by generating a state or transition; and checking, in which a consequence of the specification is tested by attempting to generate a counterexample. The analyzer achieves this by exhaustive search for conforming instances within a space bounded by user-defined limits on the cardinality of entity sets. As a result, the analyzer is sound and complete in the defined scope. The search is accomplished by the construction of a complete Boolean formula for the model space, converted to Conjunctive Normal Form (CNF), which is passed to an off-the-shelf SAT solver to find an assignment of values to the Boolean variables which satisfies the CNF.

We use Alloy by first defining a set of rules to lay the foundation of how messages and knowledge behave in our system. We then create predicates that model our security definitions and the obstacles which violate these security definitions. We test our model in a three step process. First, we run the security definitions together to ensure that they are self-consistent and mutually compatible. Second, we run the obstacles by themselves to ensure that they are self-consistent and instantiatable in a system without security. Finally, we run each security definition against the obstacle that violates it to ensure that such a system would be inconsistent.

4.2 Alloy Security Model

Our Alloy model has three main sigs: Process, Formula, and Policy. Processes are the agents in the system. Formulas are the data; this sig is further extended by generic messages, protected messages, and information about the state of the system (such as which processes sent which messages, which processes have write access to which messages, etc.). Policies define which processes have read and/or write access to protected messages.

Our initial model treated the notion of time as simply pre- and post-states of a learning action. Each process knows a set of formulas and learns a set of formulas. We chose this method because of its ease of implementation and performance efficiency. However, while this method was adequate for very abstract reasoning about security, it could not be used to reason about the steps in a protocol. Because it could only look at a single action and not a sequence of actions, we believe it is necessary to create a second version. Our second version associates a time with the formulas that a process knows. Learning is shown by a process not knowing a formula at time t_n but knowing the formula at t_{n+1}. This second version was more difficult to implement and is slower than the first for the same scope since there is more to be analyzed, but it is able to analyze sequences of events. The code for this second version can be seen in Figure 2.

4.3 Rules

To define the basic rules of our model, we constrain our model with the following facts:

`PolicyRule1` It is impossible for there to be a `CanWrite` relationship if there is no policy which supports it. A `CanWrite` relationship is an extension of formula, and it states that a particular process has write access to a particular protected message. The lack of such a relationship does

```
module security

open util/ordering[Time] as TO

sig Time{}

sig Policy{ hasRead, hasWrite: set Process }

sig Formula{ known_by: set Process->Time }

sig Process{ knows: set Formula->Time }

sig Sent extends Formula{
  sender: one Process, msg: one Msg }

sig Msg extends Formula {
  contents: one Formula, lastWriter: one Process }

sig Protected_Msg extends Msg {
  protected_by: one Policy }

sig Recvd extends Formula{
  recvr: one Process, msg: one Msg }

sig CanWrite extends Formula{
  writer: one Process, msg: one Protected_Msg }

fact PolicyRule1{
  no c: CanWrite | c.writer not in c.msg.protected_by.hasWrite }

fact MemoryRule1{
  all t: Time - TO/last() | let t' = TO/next(t) |
  all p: Process | all m: Msg |
  m->t in p.knows => m->t' in p.knows }

fact MsgRule1{ no m: Msg | m in m.contents }

fact MsgRule2{
  all record: Sent | no bad_rec: Sent |
  (record.msg = bad_rec.msg) &&
  (record.sender != bad_rec.sender) }

fact MsgRule3{
  all t: Time | all m:Msg | some r: Process |
  ( m->t in r.knows ) }

fact KnowledgeRule1{
  all t: Time | all f: Formula | all p: Process |
  p->t in f.known_by <=> f->t in p.knows }

fact KnowledgeRule2{
  all t: Time - TO/last() | all p: Process |
  all m: Msg - Protected_Msg | let t' = TO/next(t) |
  m->t in p.knows => m.contents->t' in p.knows }

fact KnowledgeRule3{
  all t: Time - TO/last() |
  all p: Process | all m: Protected_Msg |
  let t' = TO/next(t) |
  ( m->t in p.knows && HasReadAccess(p,m) )
  =>  m.contents->t' in p.knows }

fact KnowledgeRule4{
  all t: Time | all p: Process | all m: Protected_Msg |
  p = m.lastWriter => m.contents->t in p.knows }

pred HasReadAccess(p: Process, m: Protected_Msg){
  p in m.protected_by.hasRead }

pred HasWriteAccess(p: Process, m: Protected_Msg){
  p in m.protected_by.hasWrite }

pred IsSecret(f: Formula){
  all u: Msg - Protected_Msg | f != u.contents }

pred IsUnique(f: Formula){ one u: Msg | f = u.contents }

pred IsValid(m: Protected_Msg){ m = m }

pred Confidentiality(){
  all t: Time - TO/last() |
  all a: Process | all m:Protected_Msg |
  let t' = TO/next(t) | ( (m->t in a.knows) &&
  (m.contents->t not in a.knows) && (IsSecret(m.contents)) &&
  (!HasReadAccess(a, m)) ) => (m.contents->t' not in a.knows) }

pred Integrity(){
  all m:Protected_Msg | some p: Process |
  IsValid(m) => ( HasWriteAccess(p,m) && m.lastWriter = p ) }

pred Authenticity(){
  all t: Time | all m:Protected_Msg | all r: Process |
  one record: Sent | one c: CanWrite | (  IsValid(m) &&
  (m->t in r.knows) ) => (c.writer = m.lastWriter) &&
  (c.msg = m) && (c->t in r.knows) &&
  (record.sender = c.writer) &&  (record.msg = m) &&
  (record->t in r.knows) }

pred NonRepudiationReceiverSide(){
  all t: Time | all m: Msg | all p,q: Process |
  all record: Recvd | ( (record.recvr = q) && (record.msg = m) &&
  (record->t in p.knows) )  => (m->t in q.knows) }

pred NonRepudiationSenderSide(){
  all t: Time | all m: Msg | all p,q: Process |
  all record: Sent |
  (record.sender = q) && (record.msg = m) &&
  (record->t in p.knows)  => ( m.lastWriter = q ) }

pred SecureSystem(){
  Confidentiality() and Integrity() and Authenticity()
  NonRepudiationReceiverSide() and NonRepudiationSenderSide() }

pred Eavesdropping(){
  some pro:Process | some m:Protected_Msg |
  some t: (Time - TO/last()) - TO/prev(TO/last()) |
  let t' = TO/next(t) | let t'' = TO/next(t') |
  !HasReadAccess(pro,m) && (m->t in pro.knows) &&
  (m.contents->t not in pro.knows) &&
  (m.contents->t'' in pro.knows) && IsUnique(m.contents) }

pred AntiEavesdropping_System(){
  SecureSystem() and Eavesdropping() }

pred AntiEavesdropping_Confidentiality(){
  Confidentiality() and Eavesdropping() }

pred EavesdroppingWhenSentInClear(){
  some pro:Process | some m:Protected_Msg |
  some t: (Time - TO/last()) - TO/prev(TO/last()) |
  let t' = TO/next(t) | let t'' = TO/next(t') |
  !HasReadAccess(pro,m) && (m->t in pro.knows) &&
  (m.contents->t not in pro.knows) &&
  (m.contents->t'' in pro.knows) }

pred EavesdroppingWhenSentInClear_System(){
  SecureSystem() and EavesdroppingWhenSentInClear() }

pred Corruption(){
  some m: Protected_Msg | !(HasWriteAccess(m.lastWriter, m)) }

pred AntiCorruption_System(){ SecureSystem() and Corruption() }

pred AntiCorruption_Integrity(){ Integrity() and Corruption() }

pred Spoofing(){
  all t: Time | some m: Protected_Msg |
  some r: Sent | some p: Process |
  (r.sender != m.lastWriter) &&
  (r.msg = m) && (m->t in p.knows) }

pred AntiSpoofing_System(){ SecureSystem() and Spoofing() }

pred AntiSpoofing_IntegrityAlone(){ Integrity() and Spoofing() }

pred AntiSpoofing_Authenticity(){ Authenticity() and Spoofing() }

pred DeniableReception(){
  all t: Time | one p,q: Process | one m: Msg |
  one record: Recvd | record.recvr = q && record.msg = m &&
  record->t in p.knows && m->t not in q.knows }

pred AntiDeniableReception_System(){
  SecureSystem() and DeniableReception() }

pred AntiDeniableReception_NRRS(){
  NonRepudiationReceiverSide() and DeniableReception() }

pred DeniableSending(){
  all t: Time | one p,q: Process | one m: Msg | one r: Sent |
  r.sender = q && r.msg = m &&
  r->t in p.knows && q != m.lastWriter }

pred AntiDeniableSending_System(){
  SecureSystem() and DeniableSending() }

pred AntiDeniableSending_NRSS(){
  NonRepudiationSenderSide() and DeniableSending()  }
```

Figure 2: Complete Security Model in Alloy

not mean that a process cannot somehow still write to a protected message; it merely means that there is no legitimate write access.

`MemoryRule1` Information that is known is not forgotten. This guarantees that knowledge is monotonically increasing.

`MsgRule1` A message cannot contain itself. This is to prevent an unrealistic loop that could develop if the model were not constrained. This does not prevent a message from being the contents of a different message.

`MsgRule2` A message can only have one source. This is only the purported source of the message and is not necessarily the true source of the message.

`MsgRule3` All messages must be known by some process. This is to prevent trivial messages which are not known by any process.

`KnowledgeRule1` The knowledge relationship between a process and a formula is mutual. This is to guarantee the consistency of our model; it would not make sense if a process knew a formula but that formula was not known by that process.

`KnowledgeRule2` If a process knows a message, it will know its contents in the next time step. This implements learning from a message.

`KnowledgeRule3` If a process knows a protected message and has read access, it will know its contents in the next time step. This implements learning from a protected message. Note that this does not preclude the ability of a malicious process to learn the contents anyway by eavesdropping; it merely provides legitimate processes with a direct path for learning.

`KnowledgeRule4` A process can only send what it knows. This is to guarantee the consistency of our model; it would not make sense if a process sent something but did not know what it was. The passing process knows it has passed a protected message but does not know the contents of the protected message that it sent.

4.4 Security Definitions

We translated our security definitions into Alloy predicates as follows:

`Confidentiality` If a process knows a protected message at t_n, does not know its contents, its contents were never sent in the clear, and the process does not have read access to the protected message; then the process will not know the contents of the protected message at t_{n+1}.

`Integrity` If a protected message is valid, then there is some process which has write access to that protected message and is the source of that protected message.

`Authenticity` If a protected message is valid and a process knows this protected message, then this process knows that there is some process which had write access to this protected message and is the source of this protected message.

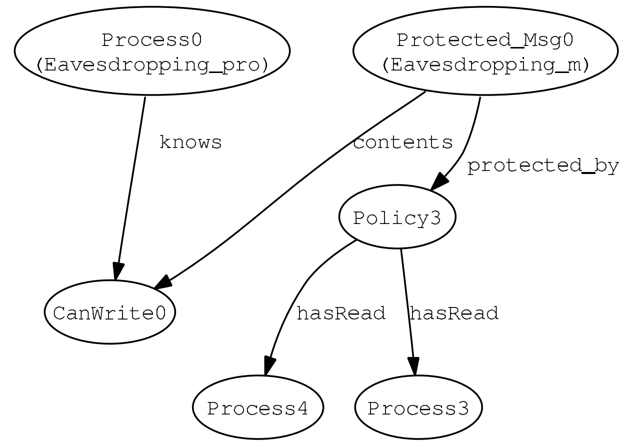

Figure 3: Eavesdropping

`NonRepudiationReceiverSide` If some process knows that a particular process received a particular message, then that particular process must know that particular message.

`NonRepudiationSenderSide` If some process knows that a particular process sent a particular message, then that particular process must be the source of that particular message.

4.5 Obstacles to Security

To test our definitions for their effectiveness against vulnerabilities, we created the following obstacles:

`Eavesdropping` Some process is able to learn the contents of a protected message even though this process did not have read access to the protected message, did not know the content beforehand, and the contents of this message were never sent in any other message (protected or not). This obstacle violates `Confidentiality`. Figure 3 shows the relevant portions of the instance that Alloy generated for this obstacle.

`Corruption` Some process which does not have write access to a protected message is the source of that protected message. This obstacle violates `Integrity`. Figure 4 shows the relevant portions of the instance that Alloy generated for this obstacle.

`Spoofing` There is some process which knows that a particular process sent a particular protected message; however, that particular process is not the source of that protected message. This obstacle violates `Authenticity`. Figure 5 shows the relevant portions of the instance that Alloy generated for this obstacle.

`DeniableReception` Process P knows that Process Q received a particular message; however, Q does not know this message. This obstacle violates `NonRepudiationReceiverSide`. Figure 6 shows

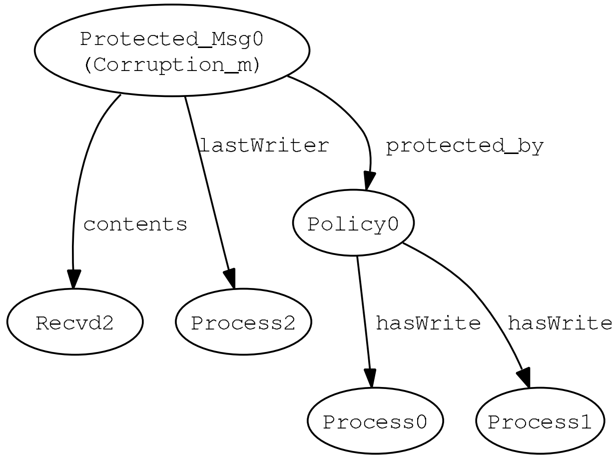

Figure 4: Corruption

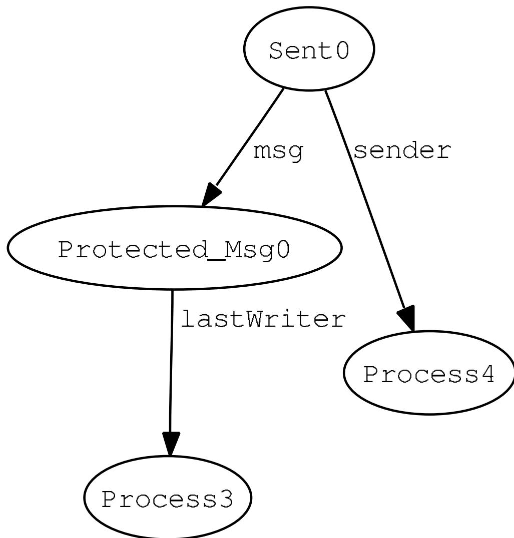

Figure 5: Spoofing

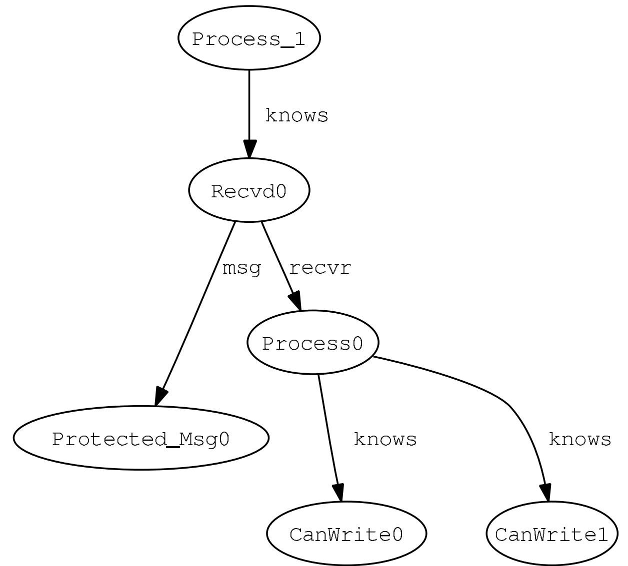

Figure 6: DeniableReception

the relevant portions of the instance that Alloy generated for this obstacle.

`DeniableSending` Process P knows that Process Q sent a particular message; however Q is not the source of that message. This obstacle violates `NonRepudiationSenderSide`. Figure 7 shows the relevant portions of the instance that Alloy generated for this obstacle.

4.6 Results

We were able to generate several instances where all of the security predicates held. This indicates that our definitions are internally consistent and compatible with one another.

For all of the obstacles, Alloy was able to generate an instance if security did not hold; however, as we expected, Alloy was unable to generate any instances where both the obstacle, and the corresponding security predicate it violated, held. We used a scope of five to give us greater confidence that our security definitions guarded against these obstacles correctly.

The Alloy model also exposed the need to be explicit about the underlying assumptions in our security model. For example, in an earlier iteration of our Eavesdropping obstacle, we did not specify that the contents of the protected message had to be unique. Thus, it would be possible for a process to send the contents of a protected message in another message. Alloy was able to generate an instance where both this version of Eavesdropping when a message is sent in the clear and `Confidentiality` both held. The process which did not have read access to the protected message was still able to learn its contents, even when `Confidentiality` held, because that process was able to learn it from an unprotected message. Figure 8 shows the relevant portions of the instance that Alloy generated for this case.

The Alloy model that we created was a good representation of our security definitions, but there were some differences.

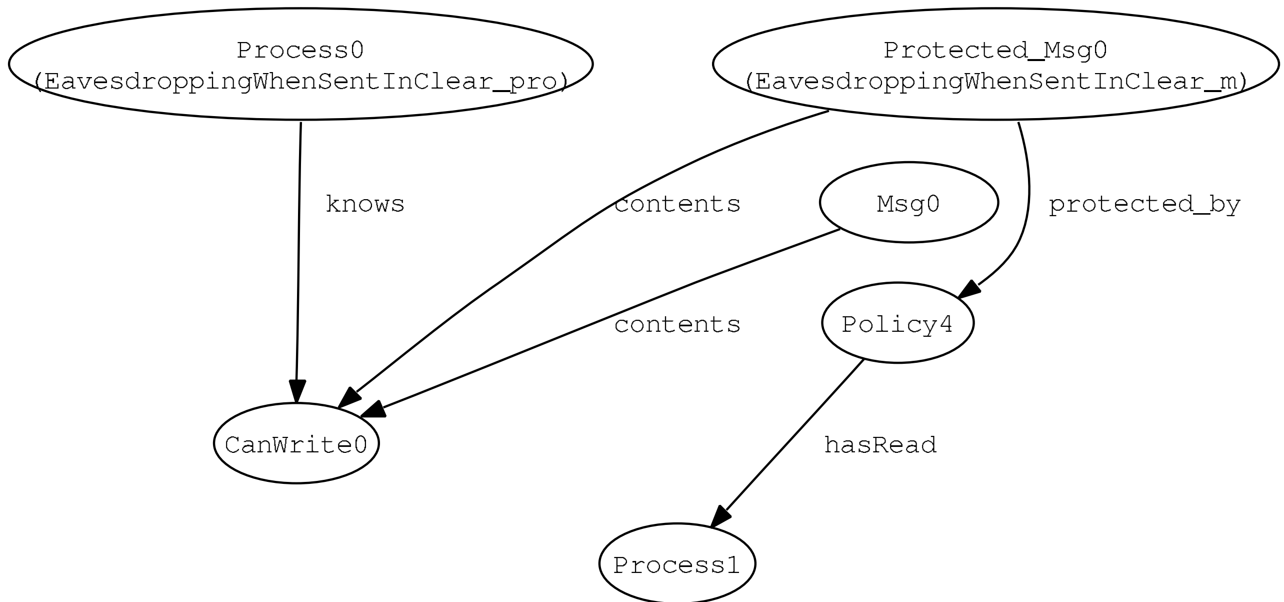

Figure 8: Confidentiality is not guaranteed if the contents of a protected message are also sent in the clear

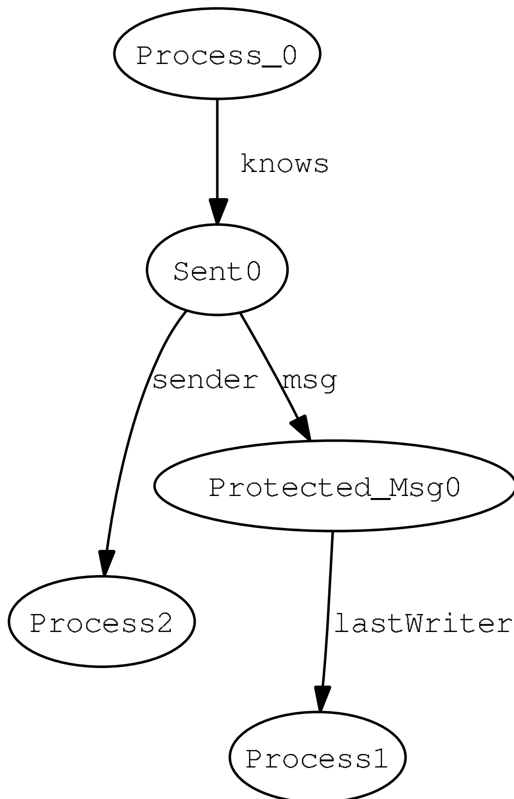

Figure 7: DeniableSending

We did not model the knowledge relationship of knowing that one knows that one knows that one knows, etc. since that would eat up the scope and not be particularly interesting. However, as a consequence, we were not able to have as much granularity in some instances. For example, there is a subtle, but real difference between our security definitions and our Alloy model in regards to the above example of eavesdropping when a message is sent in the clear. In our security definitions, while the end result is still that an eavesdropper without read access has learned the contents of the protected message, the eavesdropper will not know that he knows the contents of the protected message since he will have no means of equating the contents of the unprotected message with those of the protected message. Another difference is our usage of "knows" and "learns" to deal with the issue of temporality in Alloy by only considering the system before (the formulas that a process already "knows") and after (the formulas that a process "learns") a learning operation; in our security definitions, there is only "knows" and the system is considered as a sequence of messages.

From our experience with Alloy in this project, we have learned that using constraints to ensure that models are nontrivial can result in the need to use larger scopes. In order to prevent Alloy from generating models where there were only processes or where processes had no relations with formulas (no knowledge, no learning), we created a "KeepInteresting" fact to constrain Alloy to generate only nontrivial models. We discovered that for one of our earlier checks against confidentiality where we expected a counterexample, Alloy was unable to find a counterexample until the scope was enlarged to six; however, if we enlarged the scope to seven, the model would become so large that Alloy would freeze on our system. We also discovered that we spent a signficant portion of our time re-testing checks after making changes to part of our Alloy model to ensure that the change did not accidentally introduce new errors; we believe that this process could be greatly streamlined if Alloy had the ability to let

users execute runs in batches.

5. RELATED WORK

Our work is a novel contribution to using Alloy in the context of security. Earlier attempts at automated reasoning of security focused on analyzing a specific property in a specific protocol, but here we have focused on creating general definitions of security properties that can be applied to different protocols. Each of the steps of a protocol can be checked for violations against any of our definitions. Analyzing and comparing multiple protocols this way will be easier and more likely to yield consistent results than if the security properties had to be redefined for each individual protocol. We believe that a reusable set of security definitions will benefit the Alloy community in the same way that reusable software components benefit software engineering in general.

There have been several approaches to using model checkers and other automated tools for proving properties of protocols. BAN logic proofs are not straightforward to implement, as they require additional work to define precise semantics and prepare the protocol for the proof technique. However, model checking has been applied to protocol analysis [11, 12]. Other approaches to automated reasoning for BAN proofs require making assumptions or changes to the core semantics of the epistemic schema used by BAN. One project, implemented at MIT used knowledge flow logic and Alloy to reason about an authentication protocol [20].

More recent research into security logics have extended the basic framework in BAN with more precise semantics and support for non-repudiation [8, 19]. These logics may be more conducive to automated checking than BAN. Some work has been done using theorem provers to work with cryptographic protocols for authentication [18]. There is promise that a model checking approach might yield good results.

6. DISCUSSION AND FUTURE WORK

We originally intended to use our Alloy model to analyze the security of various network protocols. However, our initial model could only reason about the pre- and post-states of an action. This abstracted away the individual steps of protocols which are their most interesting parts (and the parts most likely to contain vulnerabilities). The current version of the model presented here has the ability to look at individual time steps with the current version of our Alloy model for security. We plan to create Alloy models of network protocols and then test these with our security predicates.

One approach was to convert our model into BAN logic. We originally designed our S5 model to resemble concepts in BAN- such as the generalization of a cryptosystem into a general protection policy-but we encountered semantic disconnect when we attempted the translation of our security model into operational constraints in BAN. They are both epistemic logics, but the difference in their semantics (and possibly the vagueness of BAN) prevented a straight forward translation.

The most important difference is with respect to the difference between knowledge and logic. In S5's possible worlds model, $K_P(\phi)$ exactly means that based on the axioms and inference rules of the system, P has enough knowledge about the current world to determine that in all possible worlds P considers true, ϕ must be true in all of them. Therefore, P *knows* ϕ is true. However, in BAN, $K \models \phi$ (K believes ϕ) means that K may behave as if ϕ is true. This means that the belief operator, $\models$, is weaker than K, though we wonder how secure a system would be if processes believe things that might not necessarily be true.

With respect to our action predicates, *Sent* and *Rcvd*, we find that they are generally similar to BAN's $\mid\sim$ (said message) and $\triangleleft$ (sees message) operators with one significant difference. In BAN, if a message is composed of multiple parts, then a process may both say a message and say each of its parts. Similarly, if a message is encrypted using some key, then a process may say an encrypted message and its contents. The same is true for the sees operator. In our S5 model, *Sent* and *Rcvd* correspond exactly to the messages as sent or as received, not their constituent elements. We think this makes *Sent* and *Rcvd* weaker than $\mid\sim$ and $\triangleleft$, though we cannot find any reason to think that the distinction matters during the conversion from S5 to BAN.

Our notion of policy is intentionally similar to the representation for an encrypted message in BAN. The expression $[M]_i$ corresponds exactly to $\{M\}_K$ when the policy is defined as "encrypt messages using key K". In this way, each distinct key corresponds to a distinct policy in our security model. The set of processes with write permission in our security model is exactly the set of processes in BAN who own a private key. The set of processes with read permission is exactly the set of processes who have the corresponding public key. For a shared key system, the set of processes with read permission and the set of processes with write permission for a given policy is necessarily identical.

The idea of using S5 to express the generalized security model is based on a tool called Sage which was designed to reason about distributed consensus protocols [17]. Sage applied the knowledge logic S5, plus a general-purpose temporal logic [6] to create a specification of a specific distributed coordination problem. The distributed system model uses a local history buffer as well as vector clocks to store what each process does and knows locally at any given moment during the execution of the protocol. Various actions of the system are defined, such that if a given action occurs on a process, then knowledge of that action having occurred is added to that process's local history buffer. Knowledge is distributed through the system using asynchronous message passing.

Sage's reasoning engine is a backward-chaining resolution style theorem prover. Starting with the end of the protocol, i.e., successful termination, Sage applied the inference rules to generate the logical precedents for each true fact in the system. If a predicate corresponding to a local action is needed, then the system assumes the action occurred, and places the action in the appropriate process's local history buffer. Execution continues until the system either generates an incorrect state (that is, the protocol could not have terminated successfully given the execution conditions) or reaches the initial state. If the initial state is reached, then

the protocol terminated successfully, and the set of local history buffers corresponds to a successful run of the protocol. The run may be modified through the user interface to determine how alternate runs or exceptional conditions could affect the outcome of the protocol.

Using a backward-chaining theorem-prover approach, in contrast to the forward generative approach we used with Alloy, should be interesting. In our Alloy approach, we constantly had to evaluate whether new facts were interesting or trivial, but in the backward-chaining approach, only interesting facts are generated. Trivial or inconsequential facts, while no less true, are not generated by this approach.

We attempted to migrate our security model into the Sage engine, but even though our security model was implemented in a similar logic to the one used by Sage, we had difficulty getting the system to recognize the inference rules we needed to support our model. We still think that there is some merit in using a theorem-prover approach, perhaps with a more recent and extensible theorem proving engine.

7. CONCLUSIONS

This work presents a reusable definition of general security using a logic with precise semantics and an extensible Alloy model that implements it. The definition and Alloy model are intended to be reused in more complex analyses. We evolved our model from an original design that used simple pre- and post- condition analysis to one that used time steps to record partial orderings. This change, while increasing the complexity of the analysis execution, provides more opportunities for advanced analysis of runtime behavior. Our hope is that our logical formulations may enable more work with automated verification and validation tools, not just limited to Alloy, but also to automated theorem provers and model checkers. Our results in Alloy show that this is at least conceptually possible.

8. ACKNOWLEDGMENTS

We would like to thank Laurent Hermoye of Département d'Ingénierie Informatique at the Université Catholique de Louvain and Deepika Mahajan of the Department of Electrical and Computer Engineering at the University of Texas at Austin, whose indispensable work on the technical report *Application of Security Constraints to Architecture Design* provided the basis for this work with Alloy.

This research is supported in part by NSF CISE Grant CCR-0306613.

9. REFERENCES

[1] M. Abadi and M. R. Tuttle. A semantics for a logic of authentication (extended abstract). In *PODC '91: Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 201–216, New York, NY, USA, 1991. ACM Press.

[2] G. Bella and L. C. Paulson. Mechanising BAN kerberos by the inductive method. In *Computer Aided Verification*, pages 416–427, 1998.

[3] G. Bella and L. C. Paulson. Mechanical proofs about a non-repudiation protocol. *Lecture Notes in Computer Science*, 2152:91+, 2001.

[4] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, 1990.

[5] K. M. Chandy and J. Misra. How processes learn. *Distrib. Comput.*, 1(1):40–52, 1986.

[6] B. Chellas. *Modal logic: an introduction*. Cambridge University Press, Cambridge, United Kingdom, 1980.

[7] R. Fagin, J. Y. Halpern, M. Y. Vardi, and Y. Moses. *Reasoning about knowledge*. MIT Press, Cambridge, MA, USA, 1995.

[8] L. Gong, R. Needham, and R. Yahalom. Reasoning About Belief in Cryptographic Protocols. In D. Cooper and T. Lunt, editors, *Proceedings 1990 IEEE Symposium on Research in Security and Privacy*, pages 234–248. IEEE Computer Society, 1990.

[9] J. Y. Halpern and A. Ricciardi. A knowledge-theoretic analysis of uniform distributed coordination and failure detectors. In *Symposium on Principles of Distributed Computing*, pages 73–82, 1999.

[10] D. Jackson. Alloy 3.0 reference manual, May 2004.

[11] D. Kindred and J. Wing. Fast, automatic checking of security protocols. In *Second USENIX Workshop on Electronic Commerce*, 1996.

[12] W. Marrero, E. Clarke, and S. Jha. Model checking for security protocols, 1997.

[13] OASIS. *Reference Model for Service Oriented Architecture*.

[14] OASIS. *Web Services Security Core Specification 1.1*.

[15] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[16] A. Ricciardi. Knowledge-theoretic analysis of partitionable coordination, December 1997.

[17] A. Ricciardi and P. Grisham. Toward sofware synthesis for distributed applications. In *TARK '98: Proceedings of the 7th conference on Theoretical aspects of rationality and knowledge*, pages 15–27, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[18] J. Schumann. Automatic verification of cryptographic protocols with setheo. In *CADE-14: Proceedings of the 14th International Conference on Automated Deduction*, pages 87–100, London, UK, 1997. Springer-Verlag.

[19] P. Syverson and P. van Oorschot. A unified cryptographic protocol logic. Technical Report 5540-227, NRL Center for High Assurance Computer Systems, 1996.

[20] E. Torlak, M. van Dijk, B. Gassend, D. Jackson, and S. Devadas. Knowledge flow analysis for security protocols. Technical Report MIT-CSAIL-TR-2005-066, Massachusetts Institute of Technology, 2005.