A Lightweight Formal Analysis of a Multicast Key Management Scheme

Mana Taghdiri and Daniel Jackson

MIT Laboratory for Computer Science,
{taghdiri,dnj}@mit.edu

Abstract. This paper describes the analysis of Pull-Based Asynchronous Rekeying Framework (ARF), a recently proposed solution to the scalable group key management problem in secure multicast. A model of this protocol is constructed in Alloy, a lightweight relational modeling language, and analyzed using the Alloy Analyzer, a fully automatic simulation and checking tool for Alloy models. In this analysis, some critical correctness properties that should be satisfied by any secure multicast protocol are checked. Some flaws, previously unknown to the protocol's designers are exposed, including one serious security breach. To eliminate the most serious flaw, some fixes are proposed and checked using the Alloy Analyzer. The case study also illustrates a novel modeling idiom that supports better modularity and is generally simpler and more intuitive than the conventional idiom used for modeling distributed systems.

Keywords: Lightweight modeling, formal specification, Alloy, secure multicast, key management, asynchronous rekeying

1 Introduction

In this paper, we describe an application of a lightweight formal method[7] to the analysis of a recently proposed protocol[16] for key management in a multicast system. To our knowledge, this is the first formal analysis of such a system that exploits automated verification techniques. In modeling the protocol, we developed a new idiom of specification which allows a rather natural and succinct expression of the protocol and its expected properties. Our paper describes the idiom, presents the analysis of the protocol in full, and discusses the ramifications of the flaws that were identified.

Our modeling language is Alloy[6]. Since Alloy is essentially just a syntax for a rather conventional first-order logic, we believe that our modeling ideas could be applied straightforwardly in other languages. The model is analyzed with the Alloy Analyzer[5], a fully automatic simulation and checking tool built on an underlying SAT engine.

Although our specification closely models the behaviour of a particular protocol – pull based ARF[16] – it introduces a reusable structure to check the correctness properties for a class of multicast key management schemes. These properties are generic, and could be applied to any multicast key management

H. König, M. Heiner, and A. Wolisz (Eds.): FORTE 2003, LNCS 2767, pp. 240–256, 2003.

protocol. Although the model of the protocol is abstract, we have been careful to remain faithful enough to the implementation to ensure that all counterexamples are in fact real, so that flaws detected by the Alloy Analyzer actually exist in this protocol.

Our experience supports the contention that lightweight formal methods are feasible and economical. Our model is less than 100 lines of code in its entirety, and yet it exposed a kind of message loss in this asynchronous protocol that is non-existent in previous synchronous multicast key management protocols. More significantly, it has shown that the protocol violates one of the properties claimed by its own developers, resulting in a security breach that was not previously known to its designers. We suggest a way to fix this breach and reduce it to a less important kind of message loss.

The organization of this paper is as follows: Section 2 compares our proposed modeling idiom with the conventional one. Section 3 gives an overview of the multicast key management problem in general and ARF as a particular solution to that. Section 4 describes our model of this protocol. Section 5 explains the distinguished features of this model. Section 6 describes the analysis of the model, counterexamples found and the evaluation of our suggested fixes. Section 7 explains the related work. Section 8 summarizes and concludes.

2 Tick-Based Modeling vs. Global State Modeling

In this section, we compare *tick-based modeling*, the idiom we shall use in the rest of the paper, to a more conventional idiom, which we will call *global state modeling*.

Figure 1 shows a simple system modeled in the two idioms. The system is a simple distributed system based on message passing. A server sends some encrypted messages to all the agents in the system. To provide security, whenever the server sends a message it generates a fresh key and encrypts the message with it. This key is then sent to some of the agents authorized to decrypt the message. To simplify the example, we ignore the delay in receiving the key.

In Alloy[6], a signature (`sig`) `S` introduces a set of atoms of type `S`. Fields declare relations, and the first column of the relation is always implicit. So, within a signature `S`, a field declared as `r : T` is a binary relation that maps each element of `S` to exactly one element of `T`, while a field `r : set T` is a binary relation that maps each element of `S` to a set of elements of type `T`. For example in Fig. 1-a, the `genTime` field in signature `Key` maps each element of `Key` to one element of `Tick` and in Fig. 1-b, the `genKeys` field in signature `Tick` maps each element of `Tick` to a set of elements of type `Key`. Furthermore, a field declared as `r: T -> U` in a signature `S` is actually a ternary relation `r: S -> T -> U`.

A `fact` paragraph in Alloy is a formula that is always true; a `fun` paragraph is a formula that can be invoked elsewhere. An assertion (`assert`) is a conjecture to be checked: a formula claimed to follow from the facts. The `check` command instructs the Alloy Analyzer to check the validity of an assertion over all the assignments of values to signatures and relations within a given scope, i.e. a

```
1  sig Tick {}
2  sig Key {
3   genTime : Tick }

4  sig Message {
5   encKey : Key,
6   sentTime : Tick }
7  {encKey.genTime = sentTime}

8  sig Agent {
9   knows : Tick -> Key }
10 {all t : Tick |
     knows[t].genTime in
               OrdPrevs(t)+t}

11 assert NoReusedKey {
    all m : Message, t : Tick |
     m.sentTime in OrdNexts(t) =>
      no a : Agent |
       m.encKey in a.knows[t]}

12 check NoReusedKey for 5
```

```
1  sig Tick {
2   sentMsgs : set Message,
3   genKeys : set Key,
4   agentKnows : Agent -> Key }
5  {all m : sentMsgs |
     m.encKey in genKeys
6   all a : Agent | agentKnows[a] in
   (OrdPrevs(this)+this).Tick$genKeys}
7  sig Key {}
8  sig Message { encKey : Key }
9  sig Agent {}

10 fact GeneratedOnce {
    all disj t, t' : Tick |
     no t.genKeys & t'.genKeys}

11 assert NoReusedKey {
    all m : Message, t : Tick |
     m in OrdNexts(t).sentMsgs =>
      no a : Agent |
       m.encKey in t.agentKnows[a]}
12 check NoReusedKey for 5
```

Fig. 1. Tick-based model (a) on left, and Global state model (b) on right, of a simple system

bound on the number of atoms in each signature. The Alloy Analyzer then finds a counterexample for that assertion if any exists in the specified scope.

Alloy is a side-effect-free declarative language. Moreover, no notion of state, transition or even time is built-in. So we define an order over all time ticks in order to be able to talk about the order in which events happen. This total order can be defined by applying generic ordering functions from the Alloy library to signature `Tick`, as in both models above. Using these functions with any signature `T` introduces a fresh relation `next : T -> T` that implicitly defines a total order over all the atoms of type `T`. These ordering functions are: `OrdPrev(t)` that returns the atom immediately preceding `t`, `OrdPrevs(t)` that returns the set of atoms preceding `t` excluding `t` itself, and functions `OrdNext(t)` and `OrdNexts(t)` that return the next atom(s).

In the tick-based model (Fig. 1-a), signature `Tick` is a notion of time; all dynamic parts of the system are modeled as relations involving `Tick`. Signature `Key` represents the keys generated by the server and the field `genTime` gives the time at which the key is generated. Signature `Message` represents sent messages. Each `Message` has an encrypting key (`encKey`) and a sent time (`sentTime`). Line 7 encodes the property that the time at which the encrypting key of a message is generated should be the same as the time at which the message is sent. (In Alloy, the formulas immediately following a signature are facts over all the atoms of that signature.) Signature `Agent` models the agents of the system and the field `knows` gives the set of keys an agent knows at each time. Line 10 encodes the fact that all keys an agent knows at each time t should be generated before or at time t. (Operator `+` is the union and `in` is the subset operator.)

Then we claim the trivial property that if a message is sent at some time after t, then its encrypting key is not known to any of the agents at time t (Line 11). No counterexample is found by the Alloy Analyzer in the scope of 5.

In the global state model (Fig. 1-b), the signature `Tick` is used to model a global state, i.e. a snapshot, of the whole system. All mutable components of the system are modeled as fields of this signature. So here the fields of `Tick` are `sentMsgs`, i.e. the messages sent in this state, `genKeys`, i.e. the keys generated in this state and `agentKnows` that gives the keys known to each agent in this state of the system. Lines 5 and 6 express the same properties as lines 7 and 10 in the previous model. The other signatures have the same semantics as in the previous case.

The fact `GeneratedOnce` (Line 10) is implicit in the tick-based model. It says there is no key generated in two different ticks t and t' (`&` is the set intersection operator and `disj` represents disjoint sets). While in the previous model the field `genTime` in `Key` implicitly encodes, in its declaration, that a key has a single generated time, here this explicit fact is necessary to outlaw false situations in which a key is generated more than once. Without this rule, the assertion would not hold.

The key difference of the two idioms is that, in tick-based modeling, history is maintained local to objects, while in global state modeling, global state is associated with time ticks. In other words, in tick-based models we are able to maintain *all* the attributes of an entity in a single place, i.e. the declaration of that entity. Thus, the tick-based idiom exhibits better modularity. Furthermore, models based on global state may need more non-obvious rules (like `GeneratedOnce` here). This makes them more error-prone. So in general, tick-based models are simpler and more succinct.

3 Overview of the Asynchronous Rekeying Framework

3.1 Secure Multicast and Key Management Problem

Multicast is appropriate when an identical message is to be sent to a group of authorized agents. It has become more important due to the growth of network applications such as teleconferencing, internet newscast, distributed games, and stock quotes.

In sending a message to a large group, multicast is much more efficient than many point-to-point unicasts. However, multicast introduces new problems: the message should not be readable by unauthorized users, while it must be guaranteed that all authorized users are able to read it. One approach is to encrypt the message using a key only known to the members of the group, called the *group key*.

However, in many applications group membership changes often, as members leave and join. In such groups, to provide communication security, we have to make sure that former members do not have access to the current messages and new members do not have access to the previous communications of the group. So after each join or leave, a new group key is generated and distributed to all current group members in order to send or receive the messages securely. Members who do not receive key updates may cause security breaches:

- Receivers failing to receive group key updates will not be able to decrypt new messages, and may also accept messages from members that have been removed from the group.
- Senders failing to receive group key updates will continue to encrypt messages with an outdated group key so that some of the current members of the group may be unable to decrypt the message and also some of the previous members of the group may be able to decrypt it.

Handling group key changes efficiently is a problem in large dynamic groups. Many solutions have been proposed to solve this problem, known as *group key management*, addressing its different aspects of scalability and performance.

3.2 Asynchronous Rekeying – The ARF Protocol

The Asynchronous Rekeying Framework [13, 16], which we call ARF for short, is one of the recent proposed solutions to the key management problem. It tries to reduce the overhead caused by the synchronization of all members during rekeying in a large dynamic group, by distributing the group key to the members only on demand, i.e. asynchronously. To the best of our knowledge, ARF is the only proposed asynchronous rekeying protocol. This motivated us to verify some of its claims formally.

The ARF designers argue that synchronizing all group members for agreement on rekeying after each change in the key, as in other protocols, is costly and unnecessary. Rather, it is sufficient to guarantee that each member is able to receive the key updates before sending or receiving group messages. Thus, they propose to distribute the group key updates on demand, just prior to use.

ARF was designed in two phases. The first phase was *pull-based ARF*. However, since it was not scalable to large groups, it was combined with *push-based ARF*. In this paper we focus on analyzing the pull-based ARF[16].

In the ARF architecture, the group is partitioned into subgroups called *domains*. Each domain has a trusted *Key Distribution Server (KDS)* which has information about its domain membership and is responsible for processing the requests of the domain's members. There is a distinct *individual key* shared between each member of a domain and the corresponding KDS. This key is used when it is needed to send a message not decryptable by others.

It is assumed that the KDS's communicate with each other via a *Reliable and Totally Ordered Multicast Protocol (RTOMP)*. Reliable multicast protocols provide retransmissions and ordering of messages from a source. Totally ordered multicast protocols guarantee that all members receive messages in the same order, ensuring consistency of shared information.

When a group member wants to leave the group, it sends a *leave request* to the KDS of its domain and waits for confirmation. The KDS generates a new group key, distributes it to the other KDS's and then sends back the confirmation. In some cases, a KDS may decide that a current member of its domain should leave and so initiates the leaving process itself.

When a host wants to join the group, it sends a *join request* to one of the KDS's and waits for confirmation. The KDS authenticates it and sends a confirmation if approved. The host then joins the corresponding domain. A new individual key is generated for the new member in this process.

Whenever membership changes in a domain, the KDS of that domain creates a new valid group key and distributes it to other KDS's using the RTOMP. The KDS also sends the new group key to the new members of the domain, if any, using their individual keys.

Each group key carries a unique ID as well as the ID of the KDS which has generated it. All group keys are ordered by the RTOMP. Thus, although different KDS's may create different new group keys simultaneously, there will not be any conflict in the order of key updates.

When a member decides to send a message, it sends a *sequence number request* to the KDS of its domain to check the newness of the group key it owns. The member attaches the ID of its newest group key to this message. If the ID of the member's key is older than the newest group key, the KDS sends back all the newer keys. The individual key of the member is used to encrypt this reply. Then, the member uses the newest key to encrypt its message and multicasts the encrypted message.

When a member receives a message which is not encrypted by any of its valid keys, it sends a request to the KDS of its domain and asks for the newer keys. It attaches the ID of its newest key to this request. The corresponding KDS replies to this request with new keys, as for a sender.

A KDS marks a group key as invalid if it is distributed to all the members of its domain. A member marks a key as invalid t units of time after it receives a message encrypted with a newer key, where t is the accepted delay of the network. To prevent members from keeping very outdated keys, a KDS must multicast a dummy message encrypted by the newest group key if there have been no messages for a while.

4 Alloy Model of the Asynchronous Rekeying Framework

We have made an abstract model of the pull-based ARF in Alloy. Since there is no notion of delay for the RTOMP in the original protocol, we assume that there is no delay in the communications via the RTOMP. Hence, if a KDS generates a new group key, all other KDS's receive it instantly. Furthermore, we assume that a member marks a key as invalid when it receives a message encrypted with a newer key or, more precisely, when it receives a newer key from the KDS[1].

[1] This will eliminate some trivial scenarios in which ARF does not work correctly only because of the notion of t units of delay time in invalidating keys.

```
sig Tick {}
sig Member {
  kds : KDS,
  ownedKeys : Tick -> Key,
  receivedMessages : Tick -> Message }
sig KDS {
  keys : Tick -> Key,
  members : Tick -> Member }
sig Message {
  sender : Member,
  sentTime : Tick,
  encryptingKey : Key }
sig Key { creator : KDS }
```

Fig. 2. Basic components of the model

Basic components of the model are shown in Fig. 2.

- `Member`: This signature models all the members in the system. Field `kds` gives the KDS of the domain this member belongs to (or may join in the future). The model has no explicit notion of domain; all members with the same `kds` field belong to the same domain. The fields `ownedKeys` and `receivedMessages` give respectively all the keys and messages known to this member at a given time. Since, as mentioned before, `Tick` is an ordered type, these fields give the sequence of keys and messages and thus keep a history of them.
- `KDS`: This signature models the key distribution servers in the system. The `keys` field in this signature indicates the keys known to this KDS at a time tick and `members` shows the members present in the corresponding domain at a time tick.
- `Message`: This signature models only multicasted messages. Each message has a sender, a sent time and an encrypting key.
- `Key`: This signature models only group keys. `creator` gives the KDS generating this key.

The `kds` field of the `Member` signature is not based on time: once its value is determined, it does not change. So a member `m` can only join the domain indicated by `m.kds` whenever it wants to join the group. This relies on the fact that different domains are interchangeable. So there is no need to consider all combinations of domains for a member to join over time. With this observation, without loss of generality, many possible cases are pruned away, and analysis time decreases.

To model the behavior of the RTOMP, the previously mentioned ordering functions of Alloy's library are used on signature `Key`. Thus, all the generated keys are ordered. Furthermore, it is implicitly assumed that if key k_2 follows key k_1 in this ordering, then k_2 is generated at some time not before k_1. With this assumption, there is no need to explicitly record the time at which a key is generated. This makes the model simpler and easier to analyze.

```
fun SendMessage(m : Member, t : Tick, msg : Message) {
  let t’ = OrdPrev(t) {
    m.ownedKeys[t] = m.ownedKeys[t’] +
      NewerKeys(m, m.kds, t, NewestKey(m.ownedKeys[t’]))
    m.receivedMessages[t] = m.receivedMessages[t’] + msg }
  msg.sender = m && msg.sentTime = t
  msg.encryptingKey = NewestKey(m.ownedKeys[t])
  ConstantMembership(m, t) }

fun ReceiveMessage(m : Member, t : Tick, msg : Message) {
  CanReceive(m, msg, t)
  let t’ = OrdPrev(t) {
    let newestKey = NewestKey(m.ownedKeys[t’]) |
      msg.encryptingKey = newestKey => m.ownedKeys[t] = m.ownedKeys[t’],
      m.ownedKeys[t] = m.ownedKeys[t’] + NewerKeys(m,m.kds,t,newestKey)
    m.receivedMessages[t] = m.receivedMessages[t’] + msg }}

fun CanReceive(m : Member, msg : Message, t : Tick) {
  ConstantMembership(m, t)
  msg !in m.receivedMessages[OrdPrev(t)]
  msg.sentTime in OrdPrevs(t)
  let newestKey = NewestKey(m.ownedKeys[OrdPrev(t)]) |
    msg.encryptingKey in NewerKeys(m, m.kds, t, newestKey) + newestKey }
```

Fig. 3. Main constraints - part one

In order to support delay in propagating messages, we let a message be sent at some time and received by other members at any later time. Thus, different members may receive the same message at different times.

Figures 3 and 4 show the main operations in this model. The model also makes use of auxiliary functions defined in Appendix 8. Some of these are simple utilities (such as defining the newest key in a set of keys); others represent frame conditions which prevent undesired changes of values in the model.

We use the following fact to constrain the initial values and the behavior of each member at each time. It constrains the histories associated with the tick-based relations to match the defined operations.

```
fact MemberBehavior{
  Init(Ord[Tick].first)
  all m: Member, t: Tick - Ord[Tick].first |
    (some msg: Message | SendMessage(m,t,msg) || ReceiveMessage(m,t,msg))
     || (some k: KDS | Join(m,t,k)) || Leave(m,t) || MemberInactive(m,t)}
```

The function `Init`, presented in the appendix, describes the constraints on the initial state of the model: it makes an empty group in which no host owns any keys or messages. According to `MemberBehavior`, at each time after that, each member can choose what to do from the list of specified actions. (`||`, `&&` and `!` are respectively logical *or*, *and* and *not* operators in Alloy.) Following is a brief description of functions shown in Figures 3 and 4.

```
fun Join(m : Member, t : Tick, k : KDS) {
  k = m.kds
  JoinRequest(m, k, t)
  NoChange(m.receivedMessages, t) }

fun Leave(m : Member, t : Tick) {
  LeaveRequest(m, m.kds, t)
  NoChange(m.receivedMessages, t) }

fun JoinRequest(m : Member, kds : KDS, t : Tick) {
  m !in kds.members[OrdPrev(t)] && m in kds.members[t]
  some k : GeneratedKey(kds, t) {
    m.ownedKeys[t] = m.ownedKeys[OrdPrev(t)] + k
    k in kds.keys[t] } }

fun LeaveRequest(m : Member, kds : KDS, t : Tick) {
  m in kds.members[OrdPrev(t)] && m !in kds.members[t]
  GeneratedKey(kds, t) in kds.keys[t]
  NoChange(m.ownedKeys, t) }
```

Fig. 4. Main constraints - part two

1. `SendMessage(m, t, msg)`
   Without loss of generality, we assume a member sends only one message at each time. To send a message, the member asks its KDS for newer keys. It sends its current newest key with this request. The KDS then checks (in function `NewerKeys` in the appendix) the newness of this key and sends back all the newer keys. Since according to ARF, each KDS knows about the memberships of its domain, the KDS returns an empty set of keys if it finds out that the member is not currently in the group. The member then adds this set of keys to its set of owned keys and sends the message encrypted with the newest key. This message is added to its set of known messages.
2. `ReceiveMessage(m, t, msg)`
   Without loss of generality, we assume a member can receive only one message at each time. A member `m` can receive a message `msg` at time `t` if `CanReceive(m, msg, t)` holds. The member checks if the message is encrypted with its newest owned key. If not, it asks the corresponding KDS for newer keys and adds them to its set of keys. Then, it accepts the message. (`x => y, z` used in this function is a syntax for `if x then y else z`.)
3. `CanReceive(m, msg, t)`
   A member `m` can receive message `msg` at time `t` only if its membership is not being changed at the same time, if the message was not received before, it was sent at some time in the past and the encrypting key is either the member's newest key or a key newer than that.
4. `Join(m, t, kds)`
   To join a domain, the member sends a join request to its assigned KDS. In the `JoinRequest` function, the KDS makes sure that the member was not in

the group in the previous time tick. The KDS adds it to the set of current group members. A new key is generated and sent to this new member as well as all other KDS's.

5. `Leave(m, t)`
   To leave a domain, the member sends a leave request to its KDS. In `LeaveRequest`, it is checked that the member was a group member in the last time tick. Then it is removed from the group. A new key is generated and sent to all other KDS's. (In this case the new key is not sent to the leaving member.)

5 Distinguished Features of the Model

Although this model is specific to the pull-based ARF properties, it suggests a structure for modeling a class of multicast key management protocols. The signatures in Fig. 2 define aspects of any key management protocol. These signatures can be extended to include more features if needed, and the operations can be written in the same way as introduced here to describe the operations of the underlying protocol. In the first author's masters thesis[15], the same structure is used to formally check Iolus[12], a totally different approach to the scalable multicast key management problem. Thus, the model is reusable in this sense.

The structure of this model is based on local behaviour; a member decides what to do at each time, nondeterministically and independently from others. Thus, it allows different members to do different things simultaneously, despite the lack of an explicit modeling of any concurrency mechanism. Because of this flexibility in the model, the analyzer can generate unusual scenarios which are likely to be neglected in a manual review.

On the other hand, the model is constrained enough (by preconditions inherent in the operations) to prevent a single member from doing several different things at the same time. To ensure that this is so, we formulate assertions of this form for each pair of operations:

```
assert NotSimultaneous {
  no m : Member, msg : Message, t : Tick |
    Join(m, t, m.kds) && SendMessage(m, t, msg) }
```

No counterexample is found for any of them.

6 Analysis of the Model

The Alloy Analyzer has been used to automatically check some properties of the model, shown in Fig. 5. The first assertion is checked in a scope of 5 and no counterexample is found. The other ones are checked in a scope of 6. All of them are checked in less than three minutes on a 1.80 GHz Pentium 4 processor.

```
assert OutsiderCantRead {
  no msg : Message, m : Member, t : Tick {
    IsMember(msg.sender, msg.sentTime)
    !IsMember(m, msg.sentTime)
    CanReceive(m, msg, t) } }

assert OutsiderCantSend {
  no msg : Message, m : Member, t : Tick {
    !IsMember(msg.sender, msg.sentTime)
    IsMember(m, t)
    CanReceive(m, msg, t) } }

assert InsiderCanRead {
  no msg : Message, m : Member, t : Tick {
    IsMember(msg.sender, msg.sentTime)
    IsMember(m, msg.sentTime)
    t in OrdNexts(msg.sentTime)
    msg !in m.receivedMessages[OrdPrevs(t)] && !CanReceive(m, msg, t)}}
```

Fig. 5. Basic properties of any secure multicast scheme

6.1 Verified Properties of the Pull-Based ARF

The assertion `OutsiderCantRead` claims that all messages sent inside the group are secure from anyone outside. This is modeled by saying there is no message sent inside the group (`IsMember(msg.sender, msg.sentTime)`) that someone who was not present in the group at that time (`!IsMember(m, msg.sentTime)`) can receive it at some time (`CanReceive(m, msg, t)`).

The Alloy Analyzer could not find any counterexample for this assertion. It should be noted that this assertion subsumes two different desirable properties of any secure multicast protocol: "A new member of the group does not have access to the previous communications" and "A former group member has no access to the current communications".

6.2 Flaws Found

`OutsiderCantSend`: This assertion claims that no message is sent outside the group (`!IsMember(msg.sender, msg.sentTime)`) which can be received by some member at a time he is inside the group (`IsMember(m, t)` and `CanReceive(m, msg, t)`) and thus, interpreted as a valid group message.

For this assertion, the Alloy Analyzer found two different types of counterexamples shown in Tables 1 and 2. In both scenarios the group has only one domain. The first case is when member m_1 joins the group when member m_0 is already there. Thus, m_1 receives a key (k_4) newer than that of m_0 (k_3). Then, m_1 leaves the group and decides to send a message. Since it is not in the group any more, the KDS denies to send it the newer keys. Thus, it encrypts its message with its own newest key, k_4. When m_0 receives the message, since it is encrypted

Table 1. First counterexample for `OutsiderCantSend`

Time, KDS Newest Key	Member m_0, Newest Key	Member m_1, Newest Key
T_1, k_3	join, k_3	-
T_2, k_4	- , k_3	join, k_4
T_3, k_5	- , k_3	leave, k_4
T_4, k_5	- , k_3	send a message , k_4
T_5, k_5	receive the message, k_5	- , k_4

Table 2. Second counterexample for `OutsiderCantSend`

Time, KDS Newest Key	Member m_0, Newest Key	Member m_1, Newest Key
T_1, k_1	join, k_1	join, k_1
T_2, k_2	- , k_1	leave, k_1
T_3, k_2	- , k_1	send a message, k_1
T_4, k_2	receive the message , k_1	- , k_1

with a key newer than its own, it asks the KDS for newer keys. The KDS sends it both k_4 and k_5 according to the ARF scheme and so m_0 decrypts the message just like an ordinary group message.

In the second scenario, two members m_0 and m_1 join the group at exactly the same time and thus, get the same group key (k_1). Then m_1 leaves the group and sends a message encrypted with its newest key (k_1). When m_0 receives this message, it does not contact the KDS for newer keys because the message is encrypted with its own newest key. So it accepts and decrypts the message without realizing that the message is sent from outside the group.

These scenarios indicate a serious security breach. A message from a former group member can influence current communications of the group. We contacted the authors of the original paper [16] to ask about this problem. They were not previously aware of this bug and agreed that the protocol is flawed.

To fix the bug, the model was changed so that in reply to the sequence number request, the KDS attaches only the newest key rather than a key vector. Function `NewerKeys-modified` in the appendix reflects this change. Also, the behavior of each member was modified so that before receiving any messages, it asks the KDS for the newest group key. If the message is not encrypted with the newest group key, it should not be accepted. The new functions are shown in Fig. 6.

After making these changes no counterexample was found. It seems that these changes in the pull-based ARF eliminate this security breach. It should be noted that both of these modifications are necessary to have a working protocol. In other words, neither of these changes alone is enough.

`InsiderCanRead:` This assertion claims that any current member of the group is able to decrypt messages sent inside the group. In order to take care of any possible delay in the network, the model asserts that any messages sent in the group can be decrypted by all present members at *any* time after that.

```
fun ReceiveMessage(m : Member, t : Tick, msg : Message) {
  CanReceive(m, msg, t)
  m.ownedKeys[t] = m.ownedKeys[OrdPrev(t)] +
    NewerKeys(m, m.kds, t, NewestKey(m.ownedKeys[OrdPrev(t)]))
  m.receivedMessages[t] = m.receivedMessages[OrdPrev(t)] + msg }

fun CanReceive(m : Member, msg : Message, t : Tick) {
  ConstantMembership(m, t)
  msg !in m.receivedMessages[OrdPrev(t)]
  msg.sentTime in OrdPrevs(t)
  let futureKeys =
      NewerKeys(m,m.kds,t,NewestKey(m.ownedKeys[OrdPrev(t)])) |
    (some futureKeys) => msg.encryptingKey in futureKeys,
    msg.encryptingKey = NewestKey(m.ownedKeys[OrdPrev(t)]) }
```

Fig. 6. Changed Receiving Functions

Table 3. First counterexample for `InsiderCanRead`

Time, Newest KDS Key	Member m_0, Newest Key	Member m_1, Newest Key
T_1, k_2	join, k_2	-
T_2, k_3	-	join, k_3
T_3, k_3	-	send message by k_3
T_4, k_4	leave, k_2	- , k_3

Here, a message is sent by m_0 while m_1 is in the group. Then m_0 leaves the group. From now on, m_1 is not able to accept that message because the group key is changed to one newer than the encrypting key of the message.

This property does not hold in the pull-based ARF. The counterexample shown in Table 3 describes a scenario in which the group has only one domain. A message is sent by member m_1 when both m_0 and m_1 are in the group, but only m_1 has the latest key, which is used to encrypt the message. Then m_0 leaves the group while it only has the outdated key. Thus, although it was in the group when the message was sent, it can never receive the message.

This problem is caused because of the asynchronous nature of ARF. In a synchronous approach, all receivers have the same key as the sender of the message. So even if, because of some delay, a member leaves the group without being able to receive the message, sooner or later it receives the message and is able to decrypt it. But in the asynchronous approach, the member leaves without having the proper key. So it can not decrypt the message even if it receives it.

To fix this kind of bug, the `Leave` function is changed so that each member updates his keys right before leaving the group. Here is the new function:

```
fun LeaveRequest(m : Member, kds : KDS, t : Tick) {
  m in kds.members[OrdPrev(t)] && m !in kds.members[t]
  some k : GeneratedKey(kds, t) {
    all kds' : KDS | k in kds'.keys[t]
    m.ownedKeys[t] = m.ownedKeys[OrdPrev(t)] +
      NewerKeys(m, kds, t, NewestKey(m.ownedKeys[OrdPrev(t)])) - k } }
```

Table 4. Second counterexample for `InsiderCanRead`

Time, Newest KDS Key	Member m_0, Newest Key	Member m_1, Newest Key
T_1, k_0	join, k_0	-
T_2, k_2	send a message , k_2	join, k_2
T_3, k_4	leave , k_2	- , k_2
T_4, k_4	- , k_2	- , k_2

Even after this change, the counterexample shown in Table 4 was found.

It can be seen that in the original model of pull-based ARF, this particular scenario did not exist because the member could receive any message encrypted with its newest key without having to contact the KDS first.

In other words, the analysis shows that in this protocol a member can not distinguish between a message from outside the group and a late message. Thus, it seems that if we want to keep the asynchronous nature of the protocol, we should either allow this kind of message loss or the security breach mentioned before.

This problem is not as significant as the previous ones. This kind of message loss is inevitable in most proposed secure key management schemes. If a message is sent at some time but received after the group key is changed because of any changes in the group memberships, the key encrypting the message will be considered invalid. Thus, the message can't be accepted.

This kind of message loss can occur with any amount of accepted delay in the network because membership changes might occur right after the message is sent. So although we have allowed *any* amount of delay in our model, which may seem unrealistic, the counterexamples show scenarios that are feasible in reality.

7 Related Work

Global state modeling is the idiom used in most specification languages such as Z[14], VDM [8], and Larch [3]. Compared to that, the tick-based idiom introduced here can be viewed as defining local states for each entity of the underlying system. It has more in common with the communicating process idiom used, e.g., in Promela, the input language of Spin[4]. However, unlike Promela, the Alloy language is a first-order logic which allows for declarative specification of a system.

Note that, like Alloy, Z and Larch are flexible enough to support different idioms, and it should be straightforward to translate our specification into those languages. However, They do not currently have automatic analysis tools.

Group key management schemes are important and useful but their design is error-prone. Although formal verification of security-based protocols is an old topic (e.g. [2, 9]), to our knowledge, none of the secure multicast key management protocols has ever been formally verified with a fully automatic tool.

Meadows et. al.[11] have constructed a model of the GDOI Group Key Management Protocol[1] using the NPATRL language, i.e. a temporal requirement specification language to be used with the NRL Protocol Analyzer[10]. To our knowledge, this is the only attempt to formally verify a group key management protocol. However, the NRL Protocol Analyzer, which combines model checking and theorem-proving techniques, is not fully automatic and relies on the user's interaction to terminate.

Fully automatic checking of formal specifications is not a new concept. Model checkers (such as Spin) have been applied extensively for protocol analysis. Alloy offers the same automation, but in the context of a logic more suited to declarative specification, and the description of structured state in an arbitrary level of details. On the other hand, unlike Spin, Alloy is not tailored to the description of protocols, and does not scale well to long executions.

8 Conclusions and Future Work

The model presented in this paper exposed flaws in pull-based Asynchronous Rekeying Framework (ARF). It showed that ARF violates one of the properties which should be satisfied by any secure multicast protocol. Although some of these flaws might be obvious to those deeply familiar with security protocols, only a complete formal analysis could reveal the behavior of the system in details.

In this model some assumptions were adopted to break the inherent symmetry of the instances without loss of generality, to help the Alloy Analyzer check the properties faster. The model abstracted away some details of ARF, but was constrained enough to prevent generating unreal instances while not eliminating real but subtle and easy to neglect scenarios.

In this model, we exploited the fact that Alloy is a first-order logic without a built-in notion of time or state in a novel idiom in which operations are associated with the time at which their effects are seen. We proposed tick-based modeling as a method that supports better modularity for modeling distributed systems.

Furthermore, the model presented to check the correctness properties of ARF introduced a structure reusable in checking a class of secure multicast key management protocols. A report on the use of the same structure to validate Iolus, a rather different multicast key management scheme, can be found elsewhere[15].

Group key management schemes are important because of their complexity yet increasing number of applications. This work might be further extended by constructing a domain-specific modeling library to make checking various properties of these schemes easier.

Acknowledgement

This research was funded by grant 0086154 from the ITR program of the National Science Foundation.

References

1. M. Baugher, T. Hardjono, H. Harney, and B. Weis. Group domain of interpretation for ISAKMP. *http://search.ietf.org/internet-drafts/draft-irtf-smug-gdoi-01.txt*, 2001.
2. E. M. Clarke and W. Marrero. Using formal methods for analyzing security. *Information Survivability Workshop (ISW)*, Oct. 1998.
3. J. Guttag, J. Horning, and A. Modet. Report on the Larch Shared Language: Version 2.3. *Digital Equipment Corporation, Systems Research Center*, report 58, 1990.
4. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.
5. D. Jackson. Automating first-order relational logic. *Proc. of the 8th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2000.
6. D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. *Proc. of the joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2001.
7. D. Jackson and J. Wing. Lightweight formal methods. *IEEE Computer*, 1996.
8. C. B. Jones. *Systematic Software Development using VDM.* Prentice Hall International, 1990.
9. C. Meadows. A system for the specification and analysis of key management protocols. *Proc. of 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 182–195, 1991.
10. C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.
11. C. Meadows, P. Syverson, and I. Cervesato. Formalizing GDOI group key management requirements in NPATRL. *Proc. of the 8th ACM Conference on Computer and Communications Security*, pages 235–244, 2001.
12. S. Mittra. Iolus: A framework for scalable secure multicasting. *Proc. of ACM SIGCOMM'97*, pages 277 – 288, 1997.
13. F. Sato and S. Tanaka. A push-based key distribution and rekeying protocol for secure multicasting. *Proc. of International Conference on Parallel and Distributed Systems*, pages 214–219, 2001.
14. J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, second edition, 1992.
15. M. Taghdiri. Lightweight modelling and automatic analysis of multicast key management schemes. Master's thesis, MIT, EECS Department, Dec. 2002.
16. S. Tanaka and F. Sato. A key distribution and rekeying framework with totally ordered multicast protocols. *Proc. of the 15th International Conf. on Information Networking*, pages 831–838, 2001.

Appendix: Complementary Parts of the ARF Model

```
sig KDS {
  keys : Tick -> Key,
  members : Tick -> Member }
{ all t : Tick | let t' = OrdPrev(t) {
    keys[t'] in keys[t]
    all m : members[t]-members[t’] | Join(m, t, this)
    all m : members[t’]-members[t] | Leave(m, t) }
  all t : Tick-Ord[Tick].first | let t’ = OrdPrev(t) {
    all k : keys[t]-keys[t’] | some m : Member {
      Join(m, t, m.kds) || Leave(m, t)
      k = GeneratedKey(m.kds, t) }
    all disj k1, k2 : keys[t]-keys[t’] |
      k1.creator != k2.creator } }
fact {
  all k1, k2 : KDS | k1.keys = k2.keys
  all m : Message | SendMessage(m.sender, m.sentTime, m) }
fun Init(t : Tick) {
  no Member.receivedMessages[t]
  no Member.ownedKeys[t]
  no KDS.keys[t]
  no KDS.members[t] }
fun MemberInactive(m : Member, t : Tick) {
  NoChange(m.receivedMessages, t)
  NoChange(m.ownedKeys, t)
  ConstantMembership(m, t) }
det fun NewerKeys(m: Member, kds: KDS, t: Tick, lastKey: Key): set Key {
  m !in kds.members[t] => no result,
  result = kds.keys[t] & OrdNexts(lastKey) }
det fun NewerKeys-modified(m : Member,kds : KDS,t : Tick,lastKey : Key):
                                                              option Key {
  m !in kds.members[t] => no result,
  result = NewestKey(kds.keys[t] & OrdNexts(lastKey)) }
det fun NewestKey(keys : set Key) : option Key {
  some keys <=> some result
  result in keys
  no OrdNexts(result) & keys }
det fun GeneratedKey(kds : KDS, t : Tick) : Key {
  some kds.keys[OrdPrev(t)] =>
    result in OrdNexts(NewestKey(kds.keys[OrdPrev(t)])),
  result in Key
  result.creator = kds }
fun ConstantMembership(m : Member, t : Tick) {
  IsMember(m, t) <=> IsMember(m, OrdPrev(t)) }
fun IsMember(m : Member, t : Tick) {
  some kds : KDS | m in kds.members[t] }
fun NoChange[T](r : Tick -> T, t : Tick) {
  r[OrdPrev(t)] = r[t] }
```