Lightweight Modelling and Automatic Analysis of Multicast Key Management Schemes

by

Mana Taghdiri

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

December 2002

Author .....
Department of Electrical Engineering and Computer Science
December 20, 2002

Certified by .....
Daniel Jackson
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur Smith
Chairman, Department Committee on Graduate Students

Lightweight Modelling and Automatic Analysis of Multicast Key Management Schemes

by

Mana Taghdiri

Submitted to the Department of Electrical Engineering and Computer Science
on December 20, 2002, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Alloy, a lightweight modelling language based on relations, is used to construct a framework for modelling a class of key management schemes used in secure multicast, aimed at checking them against some critical correctness properties that should be satisfied by all secure multicast protocols.

This framework is used to model pull-based Asynchronous Rekeying Framework (ARF) and Iolus, two very different proposed schemes addressing the scalability issue inherently involved in group key management problem. The models are analyzed using the Alloy Analyzer, a fully automatic simulation and checking tool for Alloy models.

These analyses exposed some flaws, including one serious security breach, in ARF, previously unknown to its designers. To eliminate the most serious flaw, some fixes are suggested and checked using the Alloy Analyzer.

The proposed framework introduces a novel idiom for modelling distributed systems. Compared to the conventional way of modelling these systems, our idiom is simpler and more intuitive while supporting better modularity.

Thesis Supervisor: Daniel Jackson
Title: Associate Professor

Acknowledgments

I would like to thank my advisor professor Daniel Jackson for his valuable suggestions and comments during this work. Also, I wish to thank Tina Nolte and Sarfraz Khurshid for their helpful comments on the earlier drafts of this thesis.

But above all, I would like to thank my family: my parents Afsaneh and Hossein, and my sisters Shiva and Sara, who always encouraged me to learn and supported me with love throughout my life, and my husband Mehdi, for all his love, patience, encouragement, and support without which this thesis would have never been possible. This thesis is dedicated to all my family.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Multicast is a preferred way of communication when an identical message is to be sent to a group of agents. It has become considerably more important due to the growth of network applications in which data should be received only by the legitimate members. Some of these applications are: teleconferencing, distributed games, internet newscast, stock quotes, and any network application which requires people to pay for the data.

In order to prevent unauthorized agents from receiving multicasted data, data messages are encrypted using a key known only to the authorized agents. Managing updates of this encrypting key efficiently is a challenging problem in large groups with dynamic memberships.

We describe a framework aimed at checking the correctness properties for a class of multicast key management schemes using a lightweight formal method. To our knowledge, this is the first formal analysis of such schemes that exploits fully automated verification techniques.

Furthermore, in this framework, we introduce a new idiom of specification which allows a rather natural and succinct expression of the dynamic systems. Compared to the conventional way of specifying such systems, our idiom is simpler and supports better modularity.

Our modelling language is Alloy [10, 12], a lightweight modelling language based on relations. Since Alloy is merely a syntax for a rather conventional first-order logic,

we believe that our ideas could be applied straightforwardly in other languages as well. These models can be analyzed with the Alloy Analyzer [9,11], a fully automatic tool built on an underlying SAT engine to simulate and check the Alloy models.

The proposed framework is used to formally check the correctness properties of two different key management schemes: Iolus [19] and pull-based ARF [26]. Iolus was the first proposed scheme to address the scalability problem in multicast key management, and its ideas form the basis of many subsequent schemes.

ARF, in contrast, is a recently proposed scheme for multicast key management that introduces a thoroughly different approach to solving the scalability problem. Our analysis of ARF exposed a kind of message loss in this asynchronous protocol that does not exist in the synchronous schemes. More significantly, it has shown that ARF violates one of the properties claimed by its own designers, resulting in a security breach. We suggest a way to fix this breach and reduce it to a less important kind of message loss.

Although the models of these protocols are abstract, we have been careful to remain faithful enough to ensure that all the scenarios made by the Alloy Analyzer are in fact real, so that flaws detected actually exist in the protocol.

Our experience supports the contention that lightweight formal methods are feasible and economical. Our model of ARF, for example, is less than 100 lines of code in its entirety, and yet it exposed significant flaws previously unknown to its designers.

The outline of the thesis is as follows: Chapter 2 is an overview of the Alloy language and its analyzer. Chapter 3 compares our proposed idiom with the conventional one by a simple example. Chapter 4 explains the key management problem in general and introduces our modelling framework. Chapter 5 describes the ARF scheme, its model and its analysis. Chapter 6 describes Iolus and the analysis of its model. Finally, Chapter 7 summarizes with conclusions and related work.

Chapter 2

Overview of Alloy

Alloy [10, 12] is a declarative modelling language for describing structural properties of a system. The models written in Alloy are usually much smaller than the systems they model but yet they can be used to check important properties of their underlying systems.

Alloy models can be analyzed using the Alloy Analyzer (AA) [9, 11], a fully automatic tool built on a SAT solver to simulate models and check their properties.

2.1 Language

Alloy is a syntax for a first-order logic extended with relational operators. In this section we first explain the legal expressions and formulas in the Alloy language and then describe the language high-level constructs, i.e. signatures, facts, functions, and assertions.

2.1.1 Expressions

Alloy is a strongly typed language that assumes a universe of atoms partitioned into disjoint sets, each of which is associated with a basic type.

The value of any expression in Alloy is always a relation – that is a set of tuples of atoms. Relations are typed and may have any arity. For example, if `S`, `T` and `R` are

three basic types, then a relation of type `<S, T, R>` is a set of tuples with the first atom of type `S`, the second one of type `T` and the third one of type `R`.

Sets are expressed as unary relations, i.e. relations whose tuples have only one element. Scalars are expressed as singleton unary relations, i.e. relations containing only one tuple with only one element.

Relations are combined with a variety of operators to form expressions. These operators are either set operators or relational operators. The standard set operators, '`+`' for union, '`&`' for intersection, and '`-`' for difference, combine two relations of the same type, viewed as sets of tuples, according to their standard definitions.

The main relational operator is the dot operator for join (relational composition). If `r` and `s` are two relations, their composition `r.s` contains a tuple $< r_1, \ldots, r_{m-1}, s_2, \ldots, s_n >$ if and only if $< r_1, r_2, \ldots, r_m >$ is a tuple in `r` and $< s_1, s_2, \ldots, s_n >$ is a tuple in `s` and $r_m = s_1$. When `r` is a unary relation (i.e. a set) and `s` is a binary relation, the composition `r.s` is the standard relational image of `r` under `s`. The expression `r.s` can be also written as `s[r]` with the same meaning but it binds more loosely than the dot operator.

The product operator `r -> s` gives the cross product of the two relations `r` and `s`. When `r` and `s` are unary relations, `r -> s` is just the Cartesian product of the two sets.

The transpose `~r` gives a new relation by reversing the order of atoms in each tuple of `r`. The transitive closure operator `^r` takes a binary relation `r` and gives the smallest transitive relation containing `r`. The reflexive transitive closure `*r` is like `^r` but includes a mapping from every atom to itself, too.

There are also three relational constants: `none[e]`, that represents the empty relation with the same type as `e`, `univ[e]`, that gives the universal relation with the same type as `e`, and `iden[e]` that represents the identity relation with left and right types the same as the type of `e`.

2.1.2 Formulas

Elementary formulas are formed from the subset operator '`in`' and the equivalence operator '`=`'. For two expressions `p` and `q`, the formula `p in q` is true when every tuple in `p` is in `q` and the formula `p = q` is true when `p in q` and `q in p` are both true.

Larger formulas can be obtained using the standard logical connectives: '`&&`' for conjunction, '`||`' for disjunction, '`!`' for negation, '`=>`' for implication, and '`<=>`' for bi-implication. Furthermore, the formula '`p => q, t`' is short for: `if p then q else t` (i.e. `p => q && !p => t`).

The formula '`p: q`' has the same meaning as '`p in q`', but when `q` is a set, adds an implicit constraint that `p` be scalar (i.e. a singleton). This constraint is overridden by writing '`p :option q`' which lets `p` to be empty or scalar, or '`p :set q`' which eliminates the multiplicity constraint entirely.

Quantifications take their usual form. For example, '`all x: e| F`' is true if and only if the formula `F` holds under every binding of the variable `x` to a scalar from the set denoted by `e`. Often used quantifiers are: `all` (universal), `some` (existential) and `no` (i.e. there is no value for `x` for which `F` is true).

For a quantifier `Q` and an expression `e`, the formula '`Q e`' is short for the formula '`Q x: T| x in e`' where `T` is the type of `e`. So '`no e`', for example, says that `e` is empty.

Declaration '`disj` $v_1, v_2, \ldots, v_n$`: e`' is equivalent to a declaration for each of the variables $v_1, v_2, \ldots, v_n$, with an additional constraint that the relations denoted by the variables are respectively disjoint (i.e., share no tuple).

Furthermore, within curly braces consecutive formulas are implicitly conjoined. Thus, for example, '`all x: e| {F G}`' means '`all x: e| F && G`'.

In order to make models simpler, a repeating expression within a formula can be bound to an identifier using the `let` keyword. Then, in the formula, wherever the identifier is used, its corresponding expression will be inlined. For example,

```
all q : Q | let q' = q.r {
  all x : q.s - q'.s | F
  all x : q'.s - q.s | G
}
```

is equivalent to:

```
all q : Q {
  all x : q.s - q.r.s | F
  all x : q.r.s - q.s | G
}
```

2.1.3 High-Level Constructs

An Alloy model is a sequence of paragraphs that can be of two kinds: signatures (`sig`), used for construction of new types, and a variety of formula paragraphs (`fact`, `fun`, and `assert`), used to record constraints.

Signatures

In Alloy, the signature declaration introduces a type and a collection of fields in it along with the types of the fields and constraints on their values. For example,

```
sig Tick {}
```

introduces `Tick` as a basic signature (or an uninterpreted type) with no fields while the signature declaration

```
sig Key {
  creator : KDS
}
```

declares `Key` as a basic signature with a field `creator`. The expression on the right side of the colon after a field denotes the type of that field. Thus, the `creator` field

is of type KDS. Furthermore, each field introduces a new relation in which the type of the first atom of each tuple is the signature type. For example creator is a relation of type <Key, KDS>. The field is interpreted as functional (i.e. each atom of the signature is mapped to exactly one atom by the field relation) unless constrained otherwise using the option or set keywords.

A field can introduce a relation with the arity higher than two. For example,

```
sig KDS {
  keys : Tick -> Key,
  members : Tick -> Member
}
```

declares KDS as a basic signature with fields keys and members where for each KDS k, k.keys, for example, is itself a relation of type <Tick, Key>. Thus, keys is a ternary relation of type <KDS, Tick, Key>.

Some signatures inherit fields and constraints from another signature. The signature declaration

```
sig GSA extends KDS {
  parent : option GSA
}
```

declares GSA as a subtype of KDS with an additional field parent of type GSA. As mentioned before, the option keyword means the value of this field can be empty or a singleton.

Functions

A function (fun) is a formula that can be invoked (that is, imported) elsewhere. Given values of the arguments, the function returns true/false or a relational value. For example, the function

```
fun InitKDS(t : Tick) {
  no KDS.keys[t]
  no KDS.members[t]
}
```

returns true if the argument `t` of type `Tick` satisfies the constraints in curly braces, and false otherwise.

A function may be defined to return a value of a certain type. In that case, keyword `result` stands for the returned value. If the value is defined non-deterministically, the keyword `det` should be used in the function header. For example, the function

```
det fun NewestKey(keys : set Key) : option Key {
  some keys <=> some result
  result in keys
  no OrdNexts(result) & keys
}
```

defines a function which can be also written as:

```
fun NewestKey(keys : set Key, k : option Key) {
  some keys <=> some k
  k in keys
  no OrdNexts(k) & keys
}
```

It is also allowed to define polymorphic functions in Alloy. A polymorphic function must be explicitly parameterized by any signature variable mentioned in its body. For example,

```
fun Monotonic[T] (r : Tick -> T) {
  all t : Tick | r[OrdPrev(t)] in r[t]
}
```

defines the monotonicity property for any `Tick`-based relation.

In order to use a polymorphic function, there is no need to instantiate the signature variables explicitly; this is done automatically based on the types of the arguments. For example,

```
fact {
  for all kds : KDS | Monotonic(kds.keys)
}
```

instructs the analyzer to automatically instantiate the signature variable `T` of the polymorphic function as the `Key` signature.

Facts

A fact (`fact`) is a formula that takes no arguments and need not be invoked explicitly; it is always true. Furthermore, a formula `F` attached to a signature `S` is a fact over all the atoms of the signature type. For example,

```
sig Message {
  sender : Member,
  sentTime : Tick,
  key : Key
}{
  SendMessage(sender, sentTime, this)
}
```

in addition to declaring the `Message` signature, introduces the following fact.

```
fact {
  all msg : Message | SendMessage(msg.sender, msg.sentTime, msg)
}
```

The field names appeared in a fact following a signature refer to the values of those fields in each atom of the signature. In order to refer to the whole relation

defined by a field, the prefix 'signature name$' should be used. Thus, for example, while `sender`, in the above example, refers to the sender of a message (which has the `Member` type), `Message$sender` should be used in order to refer to the relation defined by this field (that has the `Message -> Member` type).

Assertions

An assertion (`assert`) is a formula that is intended to be true. An analyzer tool tries to determine whether assertions follow from facts. A `check` command is used to instruct the analyzer to check an assertion. It takes the name of the assertion and the scope in which the analysis is to be performed. The scope defines the maximum number of atoms for each basic signature. For example

```
check A for 5
```

causes the assertion `A` to be checked for all configurations in which no signature has more than 5 atoms.

2.2 Ordering Relations

Ordering relations are defined in the Alloy library. The command line

```
open std/ord
```

allows using these relations in a model.

Using any of the ordering relations with a basic signature, defines a total order over the atoms of the signature type. For example if an ordering relation is used with signature `S`, a new relation `next` is instantiated as a relation that maps all the atoms of `S` (except one that will be the last element of the order) to exactly one other atom in `S`.

For an arbitrary signature `S` these relations are:

- `Ord[S].first` that gives the first element of the total order.

- `Ord[S].last` that gives the last element of the total order.
- `Ord[S].next` that gives the `next` relation – the total order instantiated over the atoms of `S`.
- `Ord[S].prev` that gives the `~next` relation.

There are also some functions to make orders easier to use. Some of these functions that are used often are as follows. (`x` is a variable of type `S`.)

- `OrdNext(x)` that gives `x.Ord[S].next`
- `OrdPrev(x)` that gives `x.Ord[S].prev`
- `OrdNexts(x)` that gives `x.^(Ord[S].next)`
- `OrdPrevs(x)` that gives `x.^(Ord[S].prev)`

2.3 Analyzer

The Alloy Analyzer (AA) [9, 11] is an automatic tool for analyzing models created in Alloy. Given a formula and a scope, AA determines whether there exists a model of the formula (i.e. an assignment of values to the sets and relations that makes the formula true) that uses no more atoms than the scope permits, and if so, returns it.

First-order logic is undecidable, so AA's analysis cannot be a decision procedure. That's why AA limits its analysis to a finite scope that bounds the size of the carrier sets of the basic types.

Clearly, if AA succeeds in finding a model to a formula, it has demonstrated the formula's consistency. Failure to find a model within a given scope, however, does not prove that the formula is inconsistent. In practice, however, many errors can be found by considering only a small scope (e.g. at most 5 elements of each type). This *small scope hypothesis* is purely empirical of course, since the language is undecidable, for any scope there is a formula whose smallest model occurs in a larger scope. But Andoni et. al. [2] have shown that this hypothesis works well in practice.

AA provides two kinds of analysis: *simulation* in which the consistency of a fact or function is demonstrated by generating a snapshot showing its invocation, and *checking*, in which a consequence of the specification is tested by attempting to generate a counterexample for an assertion. In both cases the tool produces graphical and also textual output.

AA's analysis is based on a translation to a boolean satisfaction problem, and gains its power by exploiting state-of-the-art SAT solvers. The analysis algorithm is described in [9].

One can use AA in the style of traditional model checking, in an attempt to find subtle errors in existing designs. We use this strategy to check the properties of the multicast key management protocols.

Chapter 3

Tick-based Modelling vs. Global State Modelling

Tick-based modelling is a novel idiom for modelling distributed systems. In this chapter, we explain this idiom and compare it to the more conventional idiom [23], which we call *global state modelling*.

Figure 3-1 shows a simple distributed system based on message passing modelled in the two idioms. 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 the tick-based model (Figure 3-1-a), signature `Tick` is defined to model the dynamics of the system. This signature, which is used as a notion of time, introduces a set of ticks which represent the time units. Any part of the system that may change over time is modelled as a relational field based on `Tick`.

Alloy is a side-effect free declarative language. So instead of explicitly writing that time advances, 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 using the ordering functions of the Alloy library with signature `Tick` as explained in the previous chapter.

```
1    module tick_based
2    open std/ord
3    sig Tick {}
4    sig Key { generatedTime : Tick }
5    sig Message {
6      key : Key,
7      sentTime : Tick
8    }{
9      key.generatedTime = sentTime }
10   sig Agent {
11     knows : Tick -> Key
12   }{
13     all t : Tick | knows[t].generatedTime in OrdPrevs(t)+t }
14   assert NoReusedKey {
15     all m : Message, t : Tick |
16       m.sentTime in OrdNexts(t) => no m.key & Agent.knows[t] }
17   check NoReusedKey for 5
```

(a)

```
1    module global_state
2    open std/ord
3    sig State {
4      sentMsgs : set Message,
5      generatedKeys : set Key,
6      agentKnows : Agent -> Key
7    }{
8      all m : sentMsgs | m.key in generatedKeys
9      all a : Agent |
10       agentKnows[a] in (OrdPrevs(this)+this).State$generatedKeys
11   }
12   sig Key {}
13   sig Message { key : Key }
14   sig Agent {}
15   fact GeneratedOnce {
16     all disj s, s' : State | no s.generatedKeys & s'.generatedKeys
17   }
18   assert NoReusedKey {
19     all m : Message, s : State |
20       m in OrdNexts(s).sentMsgs => no m.key & s.agentKnows[Agent] }
21   check NoReusedKey for 5
```

(b)

Figure 3-1: (a) Tick-based Model (b) Global State model of a Simple System

Signature `Key` in this model, represents the keys generated by the server. The `generatedTime` field gives the time at which the key is generated.

Signature `Message` represents sent messages. Each `Message` has an encrypting key (given by the `key` field) and a sent time (given by the `sentTime` field). Each message should be encrypted with a fresh key. This property is stated in Line 9 by constraining the time at which the encrypting key of a message is generated to be the same as the time at which the message is sent.

Signature `Agent` models the agents of the system. Field `knows` gives the set of keys known to an agent at a time tick. At each time `t`, an agent can only know the keys generated at or before time `t`. Line 13 encodes this property.

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` (Lines 14-16). No counterexample is found by the Alloy Analyzer in the scope of 5.

In the model based on the global state (Figure 3-1-b), signature `State` models the dynamics of the system. This signature defines a set of states which are some global states, i.e. snapshots, of the whole system. In this idiom, any time-dependent part of the system is modelled as a field of the `State` signature and the states are ordered using the Alloy ordering functions.

As shown in Figure 3-1-b, the fields of the `State` signature are `sentMsgs`, i.e. the messages sent in this state, `generatedKeys`, i.e. the keys generated in this state, and `agentKnows`, which gives the keys known to each agent in this state of the system. Lines 8-10 express the same properties as lines 9 and 13 in the previous model.

Signatures `Key`, `Message`, and `Agent` respectively give the set of keys (not necessarily *generated* keys), messages (not necessarily *sent* messages), and the agents of the system.

Fact `GeneratedOnce` (Lines 15-17) does not correspond to any explicit rule in the previous model. It says none of the keys can be generated in two different states `s` and `s'`. While in the previous model the field `generatedTime` in `Key` implicitly encodes that a key has a single generated time, here this explicit fact is necessary to outlaw 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 modelling, history is maintained local to objects, while in global state modelling, global state is associated with time ticks. For example in Figure 3-1, relation `knows` in the tick-based model has type `Agent -> Tick -> Key`, while the `agentKnows` relation in the global state model has type `State -> Agent -> Key`. In other words, by shifting the time notion, in the 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.

However, in the global state modelling, the entity declarations define only the static parts of the system and all dynamic attributes are in one other place (i.e. the `State` declaration). Thus, the tick-based idiom generally results in better modularity.

Furthermore, since the attributes related to an entity are all integrated in one place while yet the dynamic attributes are easily distinguishable from the static ones by their types, the tick-based models are more intuitive and easier to understand.

Although, according to the structures of the models, expressing time-independent properties over time-dependent fields is generally easier in global state models, expressing time-dependent properties is easier in the tick-based idiom.

Furthermore, models based on global state may need more non-obvious rules (like `GeneratedOnce` in this example) which makes them more error-prone. In general, a time-varying property `p` of an entity `S` is represented in the tick-based idiom by

```
sig S {
  p : Tick -> P
  ...
}
```

Since `Tick` appears explicitly in such a declaration, multiplicity symbols can be easily used with it whenever necessary. On the other hand, in a global state model, the same property is expressed by

```
Sig State {
  p : S -> P
  ...
}
```

in which no multiplicity constraint can be used with `State` since `p` is a field in the `State` signature. Thus, any such constraint should be expressed as an additional fact in the model.

For example, in Figure 3-1-a, `generatedTime` defines a function `Key ->! Tick` which maps each key to exactly one time tick. However, in Figure 3-1-b, this property should be represented by a separate fact (`GeneratedOnce`) since the corresponding relation (`generatedKeys`) is a field of the `State` signature and thus, multiplicity symbols can't be used with `State`.

So in general, the tick-based models are simpler, more succinct, and less error-prone while providing better modularity.

Chapter 4

A Modelling Framework for Key Management Schemes

Multicast key management protocols, in general, are very important and useful but error-prone. A framework is constructed to model a class of these protocols in Alloy, aimed at checking their correctness.

In this chapter, we first explain the multicast key management problem and then describe the modelling framework and its general properties.

4.1 Multicast Key Management Problem

In many network applications such as teleconferencing, distributed games and internet newscast, a group of authorized agents communicate via message passing. All members of the group should be able to receive all the messages sent within the group.

The conventional way of message passing, i.e. *unicast*, requires the sender to transmit a separate copy of the data to each recipient. However, this mechanism is inefficient in a group where the number of recipients is large. Thus, *multicast* is introduced as an efficient group communication mechanism in which the sender sends only one copy of data to a group address and the network is responsible for delivering it to all receivers.

4.1.1 Secure Multicast

In comparison with point-to-point unicasts, multicast reduces the overhead of message transmission considerably. At the same time, multicast introduces new problems: messages should be secured from unauthorized users while all authorized ones should be able to receive them. This is known as *secure multicasting*.

To provide security, group messages are encrypted using a key only known to the members of the group, called the *group key*. In this way, everyone in the group can decrypt the messages while no one else is able to decrypt them.

4.1.2 Key Management

In many applications, the group is very *dynamic*, i.e. membership changes often; current members of the group leave and new members join. In order to provide communication security in such groups, we have to make sure that the former members do not have access to the current messages and the 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 messages securely[1].

Members who do not receive group key updates may cause security breaches [19]:

- 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 while some of the previous members are able to decrypt it.

[1]It is possible to generate new keys once a while rather than after each membership change, but it should be guaranteed that a new key is used in later message transmissions.

Handling group key changes efficiently is a problem in large dynamic groups. Many solutions are suggested to solve this problem, known as *group key management*, addressing its different aspects of scalability and performance (e.g. [19, 21, 22, 25–27]).

4.2 Description of the Framework

Figure 4-1 shows the main structure of the framework. This framework is most suitable for checking group key management schemes in which security is achieved by the policy taken for maintenance and distribution of keys rather than by specific key properties. For example, the framework is not suitable for checking protocols with group keys based on complicated numerical computations (e.g. [25]) since Alloy is a first order logic without strong support for numerical properties.

This framework models only the basic entities that are used in key management protocols. To use it as a model of a real protocol, it may be necessary to extend some of the signatures to make them more specialized or introduce entirely new entities. The basic signatures are as follows:

- `Tick`: As mentioned before, this framework is a tick-based model. Signature `Tick` is the notion of time. It introduces a set of ticks which are ordered by the ordering functions.

- `Key`: This signature represents the set of keys that can be used to encrypt multicast messages.

- `Message`: This signature represents the multicasted messages. Each message has a sender (which, in general, can be any group member), a sent time, and an encrypting key.

- `KDS`: This signature represents the set of servers in the system. These servers, called *Key Distribution Servers (KDS)*, are the entities responsible for creating new keys when necessary.

```
1   sig Tick {}
2   sig Key {}
3   sig Message {
4     sender : Member,
5     sentTime : Tick,
6     key : Key
7   }
8   sig KDS {
9     keys : Tick -> Key,
10    members : Tick -> Member
11  }{
12    Monotonic(keys)
13    all t : Tick | let t' = OrdPrev(t) {
14      all m : members[t]-members[t'] | Join(m, t, this)
15      all m : members[t']-members[t] | Leave(m, t)
16    }
17  }
18  sig Member {
19    ownedKeys : Tick -> Key,
20    receivedMessages : Tick -> Message
21  }{
22    Monotonic(ownedKeys)
23    Monotonic(receivedMessages)
24  }
25
26  fact MemberBehavior {
27    Init(OrdFirst(Tick))
28    all m : Member, t : Tick - OrdFirst(Tick) |
29      (some msg : Message |
30         SendMessage(m, t, msg) || ReceiveMessage(m, t, msg)) ||
31      (some kds : KDS | Join(m, t, kds)) ||
32      Leave(m, t) || MemberInactive(m, t)
33  }
34  fun Monotonic[T](r : Tick -> T) {
35    all t : Tick | OrdPrev(t).r in t.r
36  }
37  fun OrdFirst[T](s : set T) : T {
38    result = Ord[s].first
39  }
40  fun OrdLast[T](s : set T) : T {
41    result = Ord[s].last
42  }
```

Figure 4-1: The Modelling Framework - Basic Structure

In most key management schemes, to provide better scalability, the group is divided into smaller subgroups called *domains*, each with a different KDS that is responsible for managing key changes only in its corresponding domain. Thus, the framework allows for *a set* of servers by default. However, some systems may have only one KDS. In that case, it is better to add the following fact to explicitly constrain the number of KDS's to be one.

```
fact OneKDS { one KDS }
```

Field `keys` in signature `KDS` gives the keys known to this server by the end of each time tick. As encoded in line 12, this field maintains a history of keys.

Since the group is assumed to be dynamic, each KDS keeps track of the set of members present in its corresponding domain at each time tick. This is given by field `members`. Lines 13-16 constrain all changes in this field over time to either be because of members joining this domain or leaving it.

- `Member`: This signature gives all the agents of the system that can be group members. Field `ownedKeys` gives all the keys known by a member by the end of a time tick. Field `receivedMessages` gives the set of messages received by the member by the end of a time tick. Similar to the `keys` relation, these two fields keep histories. This is encoded in lines 22 and 23.

The main rule of the model is the `MemberBehavior` fact. It constrains the histories associated with the tick-based relations to match the defined operations. Line 27 invokes the `Init` function to construct the initial configuration of the system. Lines 28-32 express the behavior of each member at any time after that: at each time tick, each member can (if the necessary conditions hold) send a message, receive a message, join a domain, leave the group or simply be inactive (i.e. do nothing). Each member selects what to do nondeterministically.

The structure of this framework is based on local behavior. Different members behave simultaneously but independently from each other. This flexibility in the

```
fun Init(t : Tick) {}
fun Join(m : Member, t : Tick, kds : KDS) {}
fun Leave(m : Member, t : Tick) {}
fun SendMessage(m : Member, t : Tick, msg : Message) {}
fun ReceiveMessage(m : Member, t : Tick, msg : Message) {}
fun MemberInactive(m : Member, t : Tick) {}
fun CanReceive(m : Member, t : Tick, msg : Message) {}
fun IsMember(m : Member, t : Tick) {
  some kds : KDS | m in kds.members[t]
}
```

Figure 4-2: The Modelling Framework - Main Functions

model allows the analyzer to generate unusual scenarios that may be neglected in manual verification of such a protocol. As a result, subtle bugs in the design of the system can be exposed.

The main functions of this framework are given in Figure 4-2. The body of these functions should be defined according to the rules of the underlying system. The following is a brief overview of each function:

- `Init`: This function is used to outlaw undesired configurations at a specific time tick. When invoked with the first time tick, it defines the constraints on the initial values of the time dependent fields. For example, this function may simply define a trivial initial configuration in which all the domains are empty and nobody owns any keys.

- `Join`: This function encodes the conditions under which a particular member can join the domain corresponding to a KDS at a given time tick, as well as the effects of this action.

- `Leave`: Similar to the `Join` function, this function encodes conditions and effects of a member leaving the group.

- `SendMessage`: This function encodes what conditions should hold and how different relations are affected if a member sends a message at a specific time.

```
1    assert OutsiderCantRead {
2      no msg : Message, m : Member, t : Tick {
3        IsMember(msg.sender, msg.sentTime)
4        !IsMember(m, msg.sentTime)
5        CanReceive(m, t, msg)
6      }
7    }
8    assert OutsiderCantSend {
9      no msg : Message, m : Member, t : Tick {
10       !IsMember(msg.sender, msg.sentTime)
11       IsMember(m, t)
12       msg !in m.receivedMessages[OrdPrev(t)]
13       CanReceive(m, t, msg)
14     }
15   }
16   assert InsiderCanRead {
17     all msg : Message, m : Member |
18       some t : Tick - OrdLast(Tick) | all t' : OrdNexts(t) |
19         (IsMember(msg.sender, msg.sentTime) &&
20          IsMember(m, msg.sentTime)) =>
21            CanReceive(m, t', msg)
22   }
```

Figure 4-3: Correctness Properties

- `ReceiveMessage`: Similar to the `SendMessage` function, this function gives the conditions and effects if a member receives a message at a given time.

- `MemberInactive`: This function says that the values of the relations relevant to this member at the given time are the same as in the previous time tick.

- `CanReceive`: This function is later used in the assertions. Since each member behaves nondeterministically at each time tick, there is no guarantee that it receives a sent message. Thus, this function is defined to check if the conditions necessary for a member to receive a message at a time tick hold, regardless of whether or not the member actually receives the message at that time.

- `IsMember`: This function checks if a member is in the group at a given time tick or not. It is used in the assertions.

Figure 4-3 shows the critical properties that should be satisfied by any secure multicast scheme. Any feasible counterexample for any of these assertions shows an error in the underlying system. The assertions are as follows:

- `OutsiderCantRead`: This assertion claims that no one outside the group has access to the group communications.

  This assertion is in fact expressing two different correctness properties: "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". `OutsiderCantRead` claims a more general statement than any of these two, yet includes both of them.

- `OutsiderCantSend`: This assertion claims that a member from outside the group can not influence group communications, i.e. by sending a message to the group that can be interpreted as a group message by some members.

- `InsiderCanRead`: This assertion claims that all present members of the group are able to decrypt the current group messages after a certain delay time is passed.

  This assertion claims something stronger than needed. It allows *any* amount of delay more than a base amount in message passing which is not usually the case. However, in practice, the scenarios violating this assertion can happen even within a fixed amount of delay time.

Chapter 5

ARF

The Asynchronous Rekeying Framework [21, 26], which we call ARF for short, is one of the recent proposed solutions to the multicast key management problem. It tries to reduce the overhead caused by the synchronization of all members during the changing of group keys in a large dynamic group. This is done by distributing group keys to the members only on demand, i.e. asynchronously. To the best of our knowledge, ARF is the only proposed rekeying scheme that is asynchronous. This motivated us to formally check its correctness.

5.1 Description

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

ARF was designed in two phases. The first phase was *pull-based ARF*. However, since it is not scalable to large groups, it was combined with *push-based ARF*. In this paper we focus on analyzing pull-based ARF [26].

In the ARF architecture, the group is partitioned into domains. Each domain has a trusted Key Distribution Server (KDS) that has information about its domain mem-

bership and is responsible for processing the requests of the domain members. There is a distinct *individual key* between each member of a domain and the corresponding KDS. This key is used when it is necessary 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 receivers receive the same messages, in the same order. They are used to provide consistency of shared information.

When a group member decides to leave the group, it sends a *leave request* to the KDS of its domain and waits for a confirmation. Subsequently, 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 initiates the leaving process itself.

When an agent decides to join the group, it sends a *join request* to one of the KDS's and waits for a confirmation. The KDS authenticates it and sends the confirmation if approved. The member then joins the corresponding domain. An individual key is made between the KDS and the new member in this process. Furthermore, the KDS creates a new group key and distributes it to other KDS's. The KDS also sends the new group key to the new member using its individual key.

Each group key carries a unique ID as well as the ID of the KDS that generated it. All group keys are ordered by the RTOMP. As a result, although different KDS's may create different group keys simultaneously, there will still be a consistent 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 member's key is older than the newest group key, the KDS sends back all the keys newer than it. 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 that is not encrypted by any of its keys, it sends a request to the KDS of its domain asking for the newer keys. It attaches the ID of its newest key to this request. The corresponding KDS replies to this request in the same way as mentioned above.

In order to use less memory, group members and KDS's do not maintain invalid group keys. 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 multicasts a dummy message encrypted by the newest group key if there have been no messages for a while.

5.2 Model

The framework described in the previous chapter is used to model this scheme. Figure 5-1 shows the signatures and facts added to the framework.

Signature `Client` is an extension of the `Member` signature of the framework. Each client is associated with a server that corresponds to the domain the client is allowed to join. This field is not time-dependent; whenever the member wants to join the group it should join the same domain. This relies on the observation that since in ARF there is no domain-specific behavior, different domains are symmetric and 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.

Signature `GroupKey` is an extension of the `Key` signature. The `creator` field is added to indicate the KDS that generated the key. To model the behavior of the RTOMP, the ordering functions of Alloy are used with the 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 was generated at some time not before k_1. With this assumption, there is no need to explicitly record the time at which a key was generated. This makes the model simpler and easier to analyze.

```
1    sig Client extends Member {
2      server : KDS
3    }
4    sig GroupKey extends Key {
5      creator : KDS
6    }
7
8    fact ARFProperties {
9      all k1, k2 : KDS | k1.keys = k2.keys
10     all t : Tick - OrdFirst(Tick) | let t' = OrdPrev(t) {
11       all disj k1, k2 : KDS.keys[t] - KDS.keys[t'] |
12         k1.creator != k2.creator
13       all kds: KDS, k: kds.keys[t]-kds.keys[t'] | some c: Client {
14         Join(c, t, c.server) || Leave(c, t)
15         k = GeneratedKey(c.server, t)
16       }
17     }
18     all m : Message | SendMessage(m.sender, m.sentTime, m)
19   }
```

Figure 5-1: ARF Model - Added Signatures and Facts

Fact `ARFProperties` represents the rules added to the framework to instruct the Alloy Analyzer to only consider the scenarios feasible in the ARF scheme.

Since there is no notion of delay for the RTOMP in ARF, we assume that there is no delay in communications via the RTOMP. Hence, if a KDS generates a new group key, all other KDS's receive it instantly. This is encoded in line 9 by saying: at each time, all the KDS's know the same set of keys.

Lines 10-12 say: for any two different keys generated at the same time, the generating KDS's should differ. This encodes the fact that in the ARF scheme, a KDS generates only one key at a time tick even if there are different members joining or leaving the corresponding domain at that time.

Lines 13-16 outlaw any undesired changes in the set of keys known to a KDS over time. This rule says for any key added to the set of known keys of a KDS, there should have been a membership change that prompted generation of that key.

Line 18 constrains each message in the system to be a legal sent message.

Figures 5-2 and 5-3 show the functions necessary in the framework. Here is a brief description of each function:

- `init(t)`:

  This function is used to construct the initial configuration. It constrains the initial state to represent an empty group with no domain members where no one knows any keys or messages.

- `Join(m, t, kds)`:

  As mentioned before, each member can only join the domain corresponding to its assigned KDS. In order to join that domain, the member sends a join request to the KDS. In the `JoinRequest` function, the KDS makes sure that the member was not in the group in the previous time tick. It then generates a new group key, sends it to this new member and adds the member to the set of current domain members.

  It should be noted that, in this function, in order to model that a member `c` is added to the domain corresponding to a server `kds`, we have written:

  ```
  c in kds.members[t]
  ```

  rather than conventionally writing:

  ```
  kds.members[t] = kds.members[OrdPrev(t)] + c
  ```

  In this way we allow multiple members to join and leave the group simultaneously and thus, subtle scenarios can be generated by the Alloy Analyzer. Undesired changes in the value of the `members` relation are outlawed by the constraints discussed in the previous chapter.

- `Leave(m, t)`:

  To leave the group, a member sends a leave request to its KDS. In the `LeaveRequest` function, it is checked that the member was a group member in the previous

```
fun Init(t : Tick) {
  no Member.receivedMessages[t]
  no Member.ownedKeys[t]
  no KDS.keys[t]
  no KDS.members[t]
}

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

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

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

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

Figure 5-2: ARF Model - Actions - Part 1

time tick. A new key is generated but is not sent to the leaving member. Then the member is omitted from the domain members.

- `SendMessage(m, t, msg)`:

  Without loss of generality, we assume a member sends only one message at each time tick. To send a message, the member asks its KDS for the keys newer than its own newest key. The member adds these keys to its set of owned keys and sends the message encrypted with the newest key. This message is then added to the member's set of known messages.

- `ReceiveMessage(m, t, msg)`:

  Without loss of generality, we assume a member can receive only one message at each time tick. Furthermore, we assume a member marks a key as invalid when it receives a newer key[1]. Thus, a member can receive a message only if it was encrypted with either the member's newest owned key or a newer key that can be obtained from the KDS. Also, the message should have been sent before the given time and not received yet. These conditions are encoded in the `ReceiveConditions` function.

  When receiving a message, the member checks if the message was encrypted with its newest owned key. If not, it asks the corresponding KDS for newer keys and adds them to its set of keys and then accepts the message.

- `MemberInactive(m, t)`:

  If a member does nothing at a time tick, then there is no change in its set of received messages or owned keys. Furthermore, its membership does not change.

- `CanReceive(m, msg, t)`:

[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.

```
fun SendMessage(m : Member, t : Tick, msg : Message) {
  m.ownedKeys[t] = m.ownedKeys[OrdPrev(t)] +
    NewerKeys(m, m.server, t, NewestKey(m.ownedKeys[OrdPrev(t)]))
  msg.sender = m
  msg.sentTime = t
  msg.key = NewestKey(m.ownedKeys[t])
  m.receivedMessages[t] = m.receivedMessages[OrdPrev(t)] + msg
  ConstantMembership(m, t)
}

fun ReceiveMessage(m : Member, t : Tick, msg : Message) {
  ReceiveConditions(m, t, msg)
  let newestKey = NewestKey(m.ownedKeys[OrdPrev(t)]) |
    m.ownedKeys[t] = m.ownedKeys[OrdPrev(t)] +
      if msg.key = newestKey then none[GroupKey]
      else NewerKeys(m, m.server, t, newestKey)
  m.receivedMessages[t] = m.receivedMessages[OrdPrev(t)] + msg
  ConstantMembership(m, t)
}

fun ReceiveConditions(c : Client, t : Tick, msg : Message) {
  let newestKey = NewestKey(c.ownedKeys[OrdPrev(t)]) |
    msg.key in newestKey + NewerKeys(c, c.server, t, newestKey)
  msg.sentTime in OrdPrevs(t)
  msg !in c.receivedMessages[OrdPrev(t)]
}

fun MemberInactive(m : Member, t : Tick) {
  NoChange(m.receivedMessages, t)
  NoChange(m.ownedKeys, t)
  ConstantMembership(m, t)
}

fun CanReceive(m : Member, t : Tick, msg : Message) {
  msg in m.receivedMessages[OrdPrev(t)] ||
  { ReceiveConditions(m, t, msg)
    ConstantMembership(m, t)
  }
}
```

Figure 5-3: ARF Model - Actions - Part 2

This function, which is used in the assertions, returns true if the member received the message before the given time or the conditions explained in the `ReceiveConditions` function hold.

The auxiliary functions used in the model are shown in Figure 5-4. These functions are as follows:

- `NewerKeys`:

  This function determines the set of group keys returned by a KDS in reply to a client's request for newer keys. The client's newest key is sent with this request. If at the given time the client is not present in the domain corresponding to the KDS, the KDS does not send any keys. Otherwise it sends all the keys newer than the client's key.

- `NewestKey`:

  This function determines the newest key of a given set of keys by considering the total order defined over the keys.

- `GeneratedKey`:

  This function gives the key generated by a KDS at a given time. The generated key should succeed the last generated key in the total order of the keys.

- `ConstantMembership`:

  This function returns true if the status of the membership of a given member at a given time has not changed compared to the previous time tick.

- `NoChange`:

  This polymorphic function returns true if at the given time, the given tick-based relation has the same value as the previous time tick.

```
det fun NewerKeys(c : Client, kds : KDS, t : Tick,
                                lastKey : GroupKey) : set GroupKey {
  c !in kds.members[t] => no result,
  result = kds.keys[t] & OrdNexts(lastKey)
}
det fun NewestKey(keys : set GroupKey) : option GroupKey {
  some keys <=> some result
  result in keys
  no OrdNexts(result) & keys
}
det fun GeneratedKey(kds : KDS, t : Tick) : GroupKey {
  some kds.keys[OrdPrev(t)] =>
    result in OrdNexts(NewestKey(kds.keys[OrdPrev(t)])),
  result in Key
  result.creator = kds
}
fun ConstantMembership(m : Client, t : Tick) {
  IsMember(m, t) <=> IsMember(m, OrdPrev(t))
}

fun NoChange[T](r : Tick -> T, t : Tick) {
  r[OrdPrev(t)] = r[t]
}
```

Figure 5-4: ARF Model - Auxiliary Functions

5.3 Analysis

The Alloy Analyzer has been used to automatically check the model against the generic correctness properties explained in the previous chapter.

5.3.1 Verified Properties

Assertion `OutsiderCantRead` was checked in a scope of 5 but with 2 KDS's and 2 Members. The analysis took less than one minute on a 1.80 GHz Pentium 4 processor. The Alloy Analyzer could not find any counterexamples for this assertion.

5.3.2 Flaws Found

OutsiderCantSend:

Assertion `OutsiderCantSend` was checked in the scope of 6 in less than two minutes on the same processor. For this assertion, the Alloy Analyzer found two different types of counterexamples shown in Tables 5.1 and 5.2.

In both scenarios, the group has only one domain and two members. The first scenario illustrates the case when member m_1 joins the group while member m_0 is already there. Thus, m_1 receives a key (k_4) which is newer than m_0's key (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 does not send it newer keys. As a result, m_1 encrypts the message with its own newest key, k_4. When m_0 receives the message, since it is encrypted 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 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 (i.e. k_1). When m_0 receives this message, since it is encrypted with its own newest key, it does not contact the KDS for newer keys. It accepts and decrypts the message without realizing that the message is sent

Table 5.1: 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 5.2: 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

from outside the group.

These scenarios describe 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 to ask about this problem. They were not previously aware of this bug in the protocol and agreed that it suffers from this flaw.

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 `NewerKeys2` reflects this change:

```
det fun NewerKeys2(c : Client, kds : KDS, t : Tick,
                                   lastKey : GroupKey) : option GroupKey {
  c !in kds.members[t] => no result,
  result = NewestKey(kds.keys[t] & OrdNexts(lastKey))
}
```

Additionally, the behavior of each member was modified so that before receiving any messages, the member would ask its KDS for the newest group key. If the message is not encrypted with the newest group key, it is not accepted. The new functions are shown in Figure 5-5.

After making these changes, no counterexample was found. It seems that these

```
fun ReceiveMessage2(m : Member, t : Tick, msg : Message){
  ReceiveConditons2(m, t, msg)
  m.ownedKeys[t] = m.ownedKeys[OrdPrev(t)] +
    NewerKeys(m, m.server, t, NewestKey(m.ownedKeys[OrdPrev(t)]))
  m.receivedMessages[t] = m.receivedMessages[OrdPrev(t)] + msg
  ConstantMembership(m, t)
}

fun ReceiveConditions2(c : Client, t : Tick, msg : Message) {
  let futureKeys =
    NewerKeys(c, c.server, t, NewestKey(c.ownedKeys[OrdPrev(t)])) |
      (some futureKeys) => msg.key in futureKeys,
      msg.key = NewestKey(c.ownedKeys[OrdPrev(t)])
  msg.sentTime in OrdPrevs(t)
  msg !in c.receivedMessages[OrdPrev(t)]
}

fun CanReceive2(m : Member, t : Tick, msg : Message) {
  msg in m.receivedMessages[OrdPrev(t)] ||
  { ReceiveConditions2(m, t, msg)
    ConstantMembership(m, t)
  }
}
```

Figure 5-5: Changed Receiving Functions

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; neither alone is enough.

InsiderCanRead:

Assertion `InsiderCanRead` is checked in the scope of 5 on the same processor in less than 3 minutes. This assertion does not hold in the pull-based ARF.

The counterexample shown in Table 5.3 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 (i.e. k_3), which was used to encrypt the message. Then m_0 leaves the group while it only has the outdated key k_2. Thus,

Table 5.3: 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

although it was in the group when the message was sent, it can never receive the message.

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

To fix this kind of bug, the `LeaveRequest` function is changed so that the KDS updates the member's keys right before approving its leaving the group. (The KDS does not send the new key generated because of the member's leave to the leaving member.) The following is the new function.

```
fun LeaveRequest2(c : Client, kds : KDS, t : Tick) {
    c in kds.members[OrdPrev(t)]
    c !in kds.members[t]
    some k : GeneratedKey(kds, t) {
      k in kds.keys[t]
      c.ownedKeys[t] = c.ownedKeys[OrdPrev(t)] +
        NewerKeys(c, kds, t, NewestKey(c.ownedKeys[OrdPrev(t)])) - k
    }
}
```

Even after this change, the counterexample shown in Table 5.4 was found. Here, a message is sent by m_0 while m_1 is in the group. Then m_0 leaves the group. From now

Table 5.4: 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

on m_1 is not able to accept that message because the group key has been changed to one newer than the encrypting key of the message.

It can be seen that this counterexample is due to the changes in the receiving functions shown in Figure 5-5. In the original model of pull-based ARF this particular scenario did not exist because a member could receive any message encrypted with its newest key without having to contact the KDS first.

In other words, the analysis of the pull-based ARF shows that in this protocol a member can't distinguish between a message from outside the group and a late message. It seems that if we want to keep the asynchronous nature of the protocol, we must either allow this kind of message loss or the security breach mentioned before.

This kind of message loss is not as significant as the previous problems. It 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 membership, 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.

Chapter 6

Iolus

Iolus [19] was the first multicast key management scheme proposed to address the inherent scalability problem associated with frequent key changes in a large dynamic group. It reduces the overhead of key changes by avoiding the use of a common group key. Instead, the group is divided into small subgroups each with its own session key. This idea formed the basis of many subsequent schemes which motivated us to formally check its correctness.

6.1 Description

As mentioned before, in order to provide security in group communications, the group key is changed due to any change in the group membership. Since all the members using this key should be informed of the key updates, there is an inherent scalability problem associated with the group key management [19].

In order to improve the scalability, Iolus's designers proposed to divide the group into small subgroups that are relatively independent (i.e. each of them has its own *local key*) and avoid using a global group key. Thus, after each membership change, only the local key of the corresponding subgroup should be changed and the other subgroups are not affected.

In this scheme, the small subgroups are arranged in a hierarchy to form a *secure distribution tree* (see Figure 6-1). The top-level subgroup (the root of the tree) has a

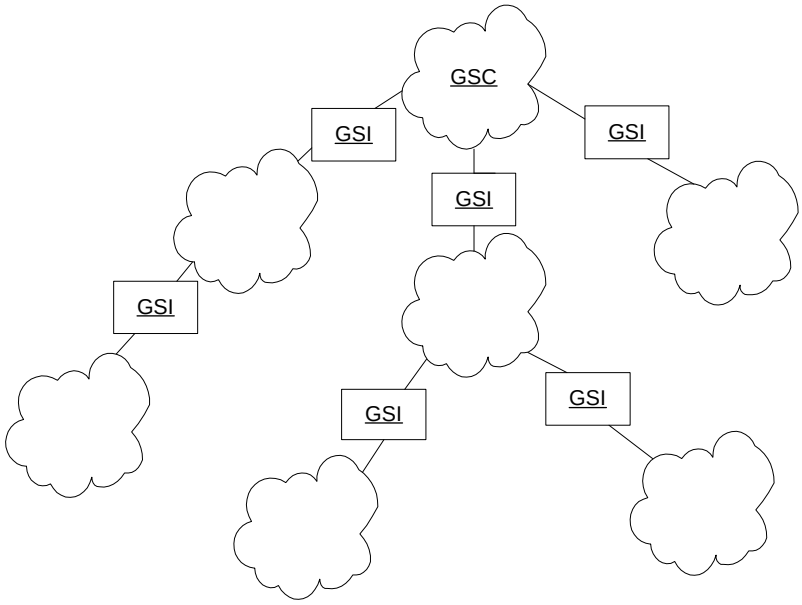

Figure 6-1: An Example of a Secure Distribution Tree – Each cloud represents a subgroup of clients.

trusted *Group Security Controller (GSC)* responsible for controlling that subgroup. Furthermore, there is a set of trusted *Group Security Intermediaries (GSI's)* connecting the subgroups in the secure distribution tree. Each GSI is responsible for controlling its child subgroup as well as delivering group messages from its child subgroup to its parent subgroup or vice-versa. The term *GSA* for *Group Security Agent* is used to refer to both GSC and the GSI's.

When a member decides to join the group, it sends a *join request* to one of the GSA's. The GSA authenticates the member using its previously provided *Access Control List (ACL)*. If approved, it generates an *individual key* to be shared only with this member, and sends it to the member. The GSA then generates a new local key and sends it to the previous subgroup members using the previous local key. It sends this new key to the new member using its individual key.

Leaves may occur either when a member wants to leave the subgroup, in which case the member sends a *leave request* to the corresponding GSA, or when a GSA decides that a member should leave, in which case it sends the member a notification. In either case, the GSA generates a new local key and sends it to all remaining members of the subgroup using their individual keys.

If a member decides to send a message, it unicasts the data to the GSA of its subgroup, encrypted with its individual key. The GSA then re-encrypts the data using the local key of the subgroup, signs it, and multicasts it to its subgroup. Furthermore, it multicasts another copy of the data to its parent subgroup (if any) encrypted with the local key of its parent.

A GSA listens for multicasted data in both its parent and child subgroups. If it receives a message in any of them, it decrypts the message, re-encrypts it with the local key of the other subgroup, and re-multicasts it to that subgroup. Because of the connectivity of the underlying tree, the data will eventually reach all the subgroups. The tree structure of the group guarantees that none of the messages are multicasted more than once in a subgroup.

Since all group messages are multicasted by a GSA, there is no possibility for a message to be encrypted using an outdated key. Furthermore, the receivers see the GSA's signature as proof that the message is from a valid source.

6.2 Model

The proposed framework is used to model the Iolus behavior in order to check its correctness. Figure 6-2 shows how the framework signatures are extended to model the necessary entities of Iolus. The following is a brief explanation of these extensions.

- `GroupKey`:

  This signature is an extension of the `Key` signature. Each group key has a `generator` field, that gives the GSA that generated the key, and a `generatedTime` field, that gives the time tick at which the key is generated.

  Lines 5-7 say that each group key should be generated because of a membership change in the corresponding subgroup.

- `DataMessage`:

  This signature is an extension of the `Message` signature. For each data message, the previously defined fields of the `Message` signature, i.e. `sender`, `sentTime`

```
1   sig GroupKey extends Key {
2     generator : GSA,
3     generatedTime : Tick
4   }{
5     some c : Client |
6       (Join(c, generatedTime, c.server) || Leave(c, generatedTime)) &&
7       c.server = generator
8   }
9   sig DataMessage extends Message {
10    gsaID : GSA,
11    retransmitTime : Tick
12  }{
13    SendMessage(sender, sentTime, this) ||
14    (some msg' : DataMessage |
15      this = Remulticast(gsaID, msg', retransmitTime))
16  }
17  sig GSA extends KDS {
18    parent : option GSA
19  }{
20    keys[Tick].generator = this
21    all t : Tick, k: keys[t]-keys[OrdPrev(t)] | k.generatedTime = t
22  }
23  sig Client extends Member {
24    server : GSA
25  }{
26    all t : Tick, k : ownedKeys[t] - ownedKeys[OrdPrev(t)] |
27      k.generator = server && k.generatedTime = t
28  }
29
30  fact IolusProperties {
31    no disj k, k' : GroupKey |
32      k.generator = k'.generator &&
33      k.generatedTime = k'.generatedTime
34    all g : GSA, msg : DataMessage, t : Tick |
35      RemulticastConditions(g, msg, t) => some Remulticast(g, msg, t)
36  }
37  fact GSATree {
38    let root = {g : GSA | no g.parent} {
39      one root
40      GSA in root.*~parent
41    }
42  }
```

Figure 6-2: Iolus Model - Added Signatures and Facts

and `key`, respectively give the group member who initiated sending the message, the original sending time of the message and the key by which the message was encrypted. Furthermore, the `gsaID` field gives the GSA that signed the message (i.e. the GSA that re-multicasted this message – or multicasted if it is the original message), and `retransmitTime` gives the time tick at which the message was re-multicasted.

Lines 13-15 say that each data message should be either the original message multicasted in the sender's subgroup, or a re-multicasting of another message.

It should be noted that the `Message` type is only used for multicasted messages; the possible delay associated with sending and processing unicasted messages is not modelled. Thus, the model is abstract in this sense.

- `GSA`:

  This signature is an extension of the `KDS` signature. For each GSA, the `parent` field gives the GSA corresponding to the parent subgroup of that GSA in the secure distribution tree. Lines 20 and 21 outlaw the infeasible values of the `keys` field of the `KDS` signature.

  It should be noted that in order to make the model simpler, a GSA is not modelled as a *bridge* between two subgroups. Instead, each GSA is associated with a subgroup and has an additional parent field that gives the subgroup associated with the parent GSA. The example shown in Figure 6-1 is modelled as the tree shown in Figure 6-3.

- `Client`:

  This signature is an extension of the `Member` signature. The `server` field gives the GSA of the subgroup that the member is allowed to join. Similar to the ARF model, because of the symmetry of the different subgroups, this field is not time-dependent to reduce the analysis time.

  Furthermore, to make the model simpler, we ignore the delay in receiving the keys generated by a GSA. Thus, whenever a key is generated, all the members of

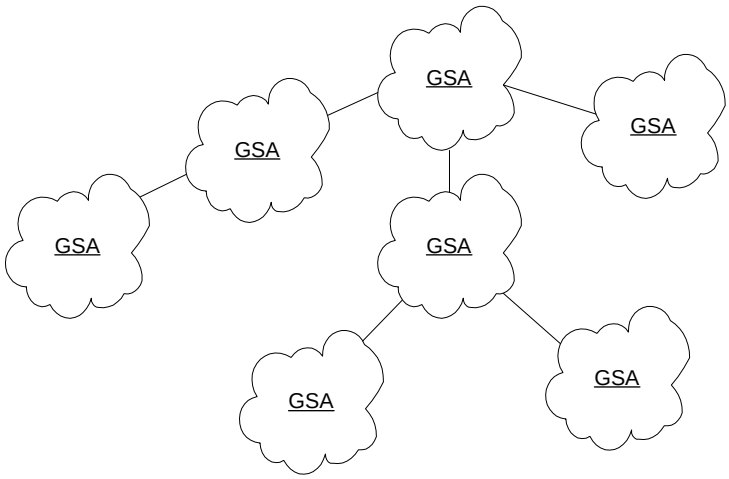

Figure 6-3: The Model of the Example Secure Distribution Tree

the corresponding subgroup receive it immediately. Lines 26 and 27 encode that at each time tick, a client receives only the keys generated by its corresponding GSA at that time.

The `IolusProperties` fact defines two more rules: lines 31-33 express the fact that a GSA does not generate more than one key per time tick (although there might be different members joining and leaving the corresponding subgroup at the same time), and lines 34 and 35 encode the rule that whenever a GSA is able to remulticast a message, it should remulticast it which decreases the analysis time of the model.

The `GSATree` fact constrains the topology defined by the `parent` relation to be a tree by encoding that there is exactly one node without a parent, i.e. the root, and any node is reachable from the root by following the *child* mapping, i.e. the transpose of the `parent` relation.

Figures 6-4 and 6-5 show the necessary functions of the framework. Here is a brief description of each function:

- `Init`:

  This function is used to define the initial values of the time-dependent fields. It constructs a group with no members where no one knows any keys or messages.

```
fun Init(t : Tick) {
  no Member.receivedMessages[t]
  no Member.ownedKeys[t]
  no KDS.keys[t]
  no KDS.members[t]
}

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

fun JoinRequest(c : Client, gsa : GSA, t : Tick) {
  c !in gsa.members[OrdPrev(t)]
  KeyUpdate(gsa, t)
  c in gsa.members[t]
}

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

fun LeaveRequest(c : Client, gsa : GSA, t : Tick) {
    c in gsa.members[OrdPrev(t)]
    KeyUpdate(gsa, t)
    c !in gsa.members[t]
}
```

Figure 6-4: Iolus Model - Actions - Part 1

- `Join`:

  This function constrains a member to only join the subgroup associated with its server. To join the group, the member sends a join request. In the `JoinRequest` function, the corresponding GSA makes sure that the member was not in the group at the previous time tick, updates the subgroup key, and adds the member to the subgroup.

- `Leave`:

  In order to leave the group, a member sends a leave request to its server. In the `LeaveRequest` function, if the member was in the group in the previous time tick, the subgroup key gets updated and then the member leaves.

- `SendMessage`:

  If a member wants to send a message at a time tick, it sends the data to its server in the form of a request. In reply, the member gets the actual multicasted message, and adds it to its set of known messages.

  In the `SendRequest` function, the corresponding GSA creates a data message with the requesting member and the time tick as its sender and sent time. Furthermore, the GSA encrypts the message with the newest subgroup key and signs the message. This message is considered the original message which is multicasted in the GSA's subgroup. Also, the GSA remulticasts the message in its parent subgroup if it has any members.

- `ReceiveMessage`:

  If the receive conditions hold, that is, if the member is not leaving the group or joining it at the same time, it has not received the message before, the message has been remulticasted (and sent) some time before, and the member has the necessary key, the member receives the message and adds it to its set of received messages.

- `MemberInactive`:

```
fun SendMessage(m : Member, t : Tick, msg : Message) {
  msg = SendRequest(m, m.server, t)
  m.receivedMessages[t] = m.receivedMessages[OrdPrev(t)] + msg
  ConstantMembership(m, t)
}

det fun SendRequest(c : Client, gsa : GSA, t : Tick) : DataMessage {
  c in gsa.members[t]
  result.sender = c
  result.sentTime = t
  result.key = NewestKey(gsa.keys[t])
  result.gsaID = gsa
  result.retransmitTime = t
  (some gsa.parent.members[t]) => some Remulticast(gsa, result, t)
}

fun ReceiveMessage(m : Member, t : Tick, msg : Message) {
  ReceiveConditions(m, t, msg)
  m.receivedMessages[t] = m.receivedMessages[OrdPrev(t)] + msg
}

fun ReceiveConditions(m : Member, t : Tick, msg : Message) {
  ConstantMembership(m, t)
  msg !in m.receivedMessages[OrdPrev(t)]
  msg.retransmitTime in OrdPrevs(t)
  msg.key in m.ownedKeys[t]
}

fun MemberInactive(m : Member, t : Tick) {
  NoChange(m.receivedMessages, t)
  ConstantMembership(m, t)
}

fun CanReceive(m : Member, t : Tick, msg : Message) {
  some msg' : DataMessage {
    msg'.sentTime = msg.sentTime
    msg'.sender = msg.sender
    msg' in m.receivedMessages[OrdPrev(t)] ||
      ReceiveConditions(m, t, msg')
  }
}
```

Figure 6-5: Iolus Model - Actions - Part 2

If a member does nothing at a time tick, then its set of received messages as well as its membership does not change. It should be noted that the member's known keys are not required to stay the same as in the previous time tick. That is because if a new key is generated in the subgroup because of a membership change at the same time, the server will send the new key to all the members regardless of whether they were active or inactive at that time.

- `CanReceive`:

  The semantics of this function, which is used in the assertions, is a bit different from what is suggested in the framework. This function returns true if at the given time, the given member can receive the piece of data encrypted in the given message, not necessarily the given message itself. That is because in Iolus, a member can receive only the messages multicasted in its own subgroup rather than all the messages. But since data is re-multicasted in all subgroups, any piece of data sent should be received by all the group members.

  Thus, `CanReceive` returns true if and only if there exists a message encrypting the same piece of data (i.e. a message with the same sender and sent time) that was either previously received by the given member or is receivable at the given time (i.e. the `ReceiveConditions` function holds).

The auxiliary functions used in the model are shown in Figure 6-6. These functions are as follows:

- `RemulticastConditions`:

  A GSA listens to the messages sent in both its own subgroup and its parent's subgroup (if any). If the GSA receives a message in any of these subgroups, it remulticasts the message to the other subgroup (if any).

  A GSA can remulticast a message if it has been sent (multicasted) at some previous time tick, the encrypting key of the message is either the GSA's key or its parent's key, and there is some other adjacent subgroup to remulticast the message to.

```
fun RemulticastConditions(g : GSA, msg : DataMessage, t : Tick) {
  msg.retransmitTime in OrdPrevs(t)
  msg.key in g.keys[t] + g.parent.keys[t]
  some g.parent + g - msg.gsaID
}

det fun Remulticast(g : GSA, msg : DataMessage, t : Tick) :
                                                option DataMessage {
  RemulticastConditions(g, msg, t)
  let g' = g.parent + g - msg.gsaID |
    result.key = NewestKey(g'.keys[msg.sentTime])
  result.sender = msg.sender
  result.sentTime = msg.sentTime
  result.retransmitTime = t
  result.gsaID = g
}

fun KeyUpdate(g : GSA, t : Tick) {
  some k : GeneratedKey(g, t) {
    all c : Client | c in g.members[t] <=> k in c.ownedKeys[t]
    k in g.keys[t]
  }
}

det fun NewestKey(keys : set GroupKey) : option GroupKey {
  some keys <=> some result
  result in keys
  no OrdNexts(result.generatedTime) & keys.generatedTime
}

det fun GeneratedKey(g : GSA, t : Tick) : GroupKey {
  result.generator = g
  result.generatedTime = t
}

fun ConstantMembership(c : Client, t : Tick) {
  IsMember(c, t) <=> IsMember(c, OrdPrev(t))
}

fun NoChange[T](r : Tick -> T, t : Tick) {
  r[OrdPrev(t)] = r[t]
}
```

Figure 6-6: Iolus Model - Auxiliary Functions

- `Remulticast`:

  If the conditions for remulticasting a message hold, the GSA creates another message with the same sender and sent time, encrypts it with the local key of the destination subgroup at the time the original message was sent, and signs it.

  In order to simplify the model, we assume that a GSA has access to the entire set of keys known to its parent. In reality, the GSA may have to send a request to its parent and ask for the necessary key.

- `KeyUpdate`:

  A GSA updates the local key of its subgroup by generating a new key and sending it to all the current members of its subgroup.

- `NewestKey`:

  This function takes a set of keys and determines the most recently generated key.

- `GeneratedKey`:

  This function returns a key generated by the given GSA at the given time tick.

- `ConstantMembership`:

  This function return true if a given member does not join or leave the group at a given time.

- `NoChange`:

  This polymorphic function returns true if the value of the given relation at a given time is the same as the previous time tick.

6.3 Analysis

In order to reduce the analysis time, the `EasierAnalysis` fact (shown in Figure 6-7 was added. Without loss of generality, it constrains any element of the super-type

```
fact EasierAnalysis {
  Member = Client
  KDS = GSA
  Message = DataMessage
  Key = GroupKey
  no disj m, m' : DataMessage {
    m.sender = m'.sender
    m.sentTime = m'.sentTime
    m.key = m'.key
    m.gsaID = m'.gsaID
    m.retransmitTime = m'.retransmitTime
  }
}
assert Acyclic {
  all g : GSA | g !in g.^parent
}
assert Connected {
  all g, g' : GSA | g in g'.*(parent + ~parent)
}
assert TimeProceeds {
  no msg : DataMessage | msg.retransmitTime in OrdPrevs(msg.sentTime)
}
```

Figure 6-7: Iolus Model - Basic Properties

sets to be an element of the sub-type sets. Furthermore, it outlaws any cases with two different messages with exactly the same fields.

The basic assertions shown in Figure 6-7 were checked to make sure the model has the required properties. All of these claims were checked in the scope of 6 in less than one minutes in a 1.80 GHz Pentium 4 processor and no counterexample was found.

Furthermore, assertions `OutsiderCantRead` and `OutsiderCantSend` were checked, both in the scope of 5 but with 2 members. The analyses took less than 3 minutes and no counterexamples were found.

The `InsiderCanRead` assertion did not hold in this model but none of the counterexamples showed an error in the underlying system. This is not surprising because this assertion expresses a liveness property: a message sent by a member can be eventually received by all other members given enough time. For any scope specified for

```
fun LoopFree() {
  no disj t, t' : Tick {
    all k : KDS | k.members[t] = k.members[t']
    all m : Member | m.receivedMessages[t] = m.receivedMessages[t']
    all m : DataMessage | m.retransmitTime = t =>
      some m' : DataMessage {
        m'.retransmitTime = t'
        m.sender = m'.sender
        m.sentTime = m'.sentTime
        m.gsaID = m'.gsaID
        m.key = m'.key
      }
  }
}

assert Loop { !LoopFree() }
```

Figure 6-8: Constraints Required to Check the Liveness Property

the `Tick` signature, there exists a scenario in which more time is needed to achieve this property.

A trick to check such properties in Alloy was recently presented by Shlyakhter et. al. [23]. As explained in that paper, for a given specification, checking a property for traces of length up to the *recurrence diameter* is equivalent to proving the property for traces of *any* length, while the recurrence diameter of a state machine is the smallest number r such that all traces of length $r + 1$ must contain a loop, i.e. have two equivalent states.

Using the proposed method, we define the `LoopFree` function shown in Figure 6-8. This function returns true if there are not two different time ticks at which the group configurations are the same. Two configurations of our model are considered the same if the sets of present subgroup members in both configurations are the same, all the members have the same set of received messages in both configurations, and the sets of multicasted messages in both configurations are the same.

To find the recurrence diameter, the `Loop` assertion was checked for a fixed number of members, KDS's, and messages and the number of ticks was incremented by one

whenever a counterexample was found. A counterexample to this assertion represents a scenario in which there is no loop in the trace. Thus, if k is the smallest scope of Tick for which the analyzer can't find any counterexamples, then $k - 1$ is the recurrence diameter.

The recurrence diameter of this model with 2 members, 1 KDS and 1 Message is 12 and the `InsiderCanRead` assertion holds in that scope. Checking this assertion with more KDS's, members or messages requires a larger number of time ticks which can't be processed by the current version of the Alloy Analyzer in a reasonable time.

Chapter 7

Conclusions and Related Work

7.1 Summary

We constructed a framework in Alloy for modelling a class of multicast key management schemes. The framework aims at checking the underlying schemes against the critical correctness properties that should be satisfied by all secure multicast protocols.

This framework was then used to verify two very different protocols addressing the scalability problem inherently involved in multicast key management. These protocols are Iolus, which was the first protocol addressing scalability and formed the basis of many subsequent proposed protocols, and pull-based ARF, which is a recently proposed protocol with an entirely different approach.

Checking pull-based ARF using the Alloy Analyzer uncovered a serious security breach in this scheme previously unknown to its designers. We suggested some modifications to the protocol to resolve the problem. Furthermore, the analysis exposed a kind of message loss in this asynchronous protocol which does not exist in the synchronous ones.

The framework proposed here introduces a novel idiom, called *tick-based modelling*, for modelling dynamic systems. Compared to the more conventional way of modelling such systems, our idiom is simpler, more intuitive and provides better modularity.

7.2 Discussions and Related Work

Global state modelling is the idiom used in most specification languages such as Z [24], VDM [1], and Larch [7]. Compared to that, tick-based idiom introduced here, can be viewed as defining local states for any individual entity of the underlying system. This idea is similar to the Spin model checker [8] in which each process has its own local state. However, unlike Promela (the input language of Spin), the Alloy language is a first-order logic which allows for declarative specification of a system in an arbitrary level of details.

Group key management protocols are very important and useful but their design is error-prone. Although verification of security-based protocols is an old topic in formal verification [5, 6, 14–16], to our knowledge, none of the secure multicast key management protocols has been formally verified with a fully automatic tool before.

Meadows et. al. [18] have constructed a formal model of GDOI Group Key Management Protocol [3] using the NPATRL language, i.e. a temporal requirement specification language to be used with the NRL Protocol Analyzer [17]. NRL Protocol Analyzer is a prototype special-purpose verification tool, written in Prolog, for the analysis of cryptographic protocols that are used to authenticate principals and services and distribute keys in a network. To our knowledge, this work 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, works automatically only when the state space of the problem is small enough for an exhaustive search. Thus, in checking most real protocols, it heavily depends on the user's interaction to terminate. More precisely, in order to find a feasible path from an initial state to a goal state, the analyzer performs a breadth-first search and in each level it asks the user to prune the states. Although, it can use some user provided as well as some built-in rules to detect unreachable states, the search can be done automatically only in very small state spaces.

On the other hand, fully automatic checking of formal specifications is not a new concept. Model checking applied to a finite state machine is well known. The Alloy

Analyzer is an analog, based on a systematic search of bindings for specifications in the setting of first-order logic and set theory. Such a framework is especially well-suited to modelling complex structures.

Khurshid and Jackson [13] managed to find some serious flaws in the design and implementation of an intentional naming scheme using Alloy. Also, using the Alloy Analyzer, they were able to establish conditions under which that version of the INS algorithm works.

Nolte [20] used the Alloy Analyzer to explore the behavior of some filesystem synchronizers aiming at exploring the policies these tools employ and what they guarantee in the subtle situations with conflicting updates made to multiple filesystem replicas.

Recently Alloy has been used for model-checking of some distributed algorithms [23] which involved checking liveness and safety properties of the algorithms running in any arbitrary topology. In this paper, it is explained how to embed Biere et.al's idea [4] in Alloy in order to use the Alloy Analyzer for checking liveness properties despite its finite-scope analysis.

We believe that checking the design of systems using lightweight modelling is very useful since it can result in considerable savings by detecting errors prior to implementation.

Group key management schemes are especially important because of their complexity yet increasing number of applications. Thus, verifying their expected properties before implementing them can be considerably helpful. This work might be further extended by constructing a domain-specific modelling library to make checking various properties of these schemes easier.

Bibliography

[1] Sten Agerholm and Peter Gorm Larsen. The IFAD VDM tools: Lightweight formal methods. In *proc. International Workshop on Current Trends in Applied Formal Methods (FM-Trends98)*, pages 326–329, Boppard, October 1998.

[2] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the small scope hypothesis. *submitted for publication.*

[3] Mark Baugher, Thomas Hardjono, Hugh Harney, and Brian Weis. Group domain of interpretation for ISAKMP. In *http://search.ietf.org/internet-drafts/draft-irtf-smug-gdoi-01.txt*, January 2001.

[4] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking wihout BDDs. Technical Report CS-99-101, Carnegie Mellon University, January 1999.

[5] Edmund Clarke and Will Marrero. Using formal methods for analyzing security. In *Information Survivability Workshop*, 1998.

[6] Ernie Cohen. TAPS: A first-order verifier for cryptographic protocols. In *Proc. 13th IEEE Computer Security Foundations Workshop*, pages 144–158, Cambridge, July 2000.

[7] John V. Guttag, James J. Horning, and Andres Modet. Report on the larch shared language: Version 2.3. Technical Report TR58, Compaq Systems Research Center, April 1990.

[8] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[9] Daniel Jackson. Automating first-order relational logic. In *Proc. 8th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, San Diego, November 2000.

[10] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, April 2002.

[11] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. Alcoa: the alloy constraint analyzer. In *Proc. 22nd International Conference on Software Engineering*, Limerick, June 2000.

[12] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *Proc. 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering/European Software Engineering Conference (FSE/ESEC'01)*, Vienna, September 2001.

[13] Sarfraz Khurshid and Daniel Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *15th IEEE Internation Conference on Automated Software Engineering*, Grenoble, September 2000.

[14] Will Marrero, Edmund Clarke, and Somesh Jha. A model checker for authentication protocols. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, Piscataway, September 1997.

[15] Catherine Meadows. A system for the specification and analysis of key management protocols. In *Proc. 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 182–195, Los Alamitos, 1991.

[16] Catherine Meadows. Formal verification of cryptographic protocols: A survey. In *Advances in Cryptology – ASIACRYPT'94: 4th International Conference on the*

Theory and Application of Cryptology, pages 135–150, Wollongong, November 1994.

[17] Catherine Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[18] Catherine Meadows, Paul Syverson, and Iliano Cervesato. Formalizing GDOI group key management requirements in npatrl. In *Proc. 8th ACM Conference on Computer and Communications Security*, pages 235–244, Philadelphia, November 2001.

[19] Suvo Mittra. Iolus: A framework for scalable secure multicasting. In *Proc. ACM SIGCOMM'97*, pages 277–288, Cannes, September 1997.

[20] Tina Nolte. Exploring filesystem synchronization with lightweight modeling and analysis. Master's thesis, Massachusettes Institute of Technology, September 2002.

[21] Fumiaki Sato and Shinya Tanaka. A push-based key distribution and rekeying protocol for secure multicasting. In *Proc. 8th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 214–219, 2000.

[22] Sanjeev Setia, Samir Koussih, Sushil Jajodia, and Eric Harder. Kronos: A scalable group rekeying approach for secure multicast. In *Proc. IEEE Symposium on Security and Privacy*, pages 215–228, 2000.

[23] Ilya Shlyakhter, Manu Sridharan, and Daniel Jackson. Analyzing distributed algorithms with first-order logic. *submitted for publication.*

[24] J. Michael Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, second edition, 1992.

[25] Michael Steiner, Gene Tsudik, and Michael Waidner. Cliques: A new approach to group key agreement. In *Proc. 18th International Conference on Distributed Computing Systems*, pages 380–387, Amsterdam, May 1998.

[26] Shinya Tanaka and Fumiaki Sato. A key distribution and rekeying framework with totally ordered multicast protocols. In *Proc. 15th International Conference on Information Networking*, pages 831–838, 2001.

[27] Chung Kei Wong, Mohammad Gouda, and Simon S. Lam. Secure group communications using key graphs. *IEEE/ACM Transactions on Networking*, 8(1):16–30, February 2000.