Formal Specification and Analysis of the Group Domain of Intrepretation Protocol Using NPATRL and the NRL Protocol Analyzer [*]

Catherine Meadows, Paul Syverson

Center for High Assurance Computer Systems
Naval Research Laboratory
Washington, DC 20375-5320 — USA
{*meadows, syverson*}*@itd.nrl.navy.mil*

Iliano Cervesato

Advanced Engineering and Sciences Division
ITT Industries, Inc.
2560 Huntington Avenue, Alexandria, VA 22303-1410 — USA
iliano@itd.nrl.navy.mil

Abstract

Although research has been going on in the formal analysis of cryptographic protocols for a number of years, they are only slowly being integrated into the protocol design process. In this paper we describe how we furthered the integration of analysis and design by working closely with the Multicast Security Working Group in the Internet Engineering Task Force on the analysis of a proposed Internet Standard, the Group Domain Of Interpretation (GDOI) Protocol. We describe the challenges that had to be met before the analysis could be successfully completed, and some of the challenges that still remain. Perhaps not surprisingly, some of the most challenging work was in understanding the security requirements for group protocols in general. We give a detailed specification of the requirements for GDOI, describe our formal analysis of the protocol with respect to these requirements, and show how our analysis impacted the development of GDOI.

[*]Meadows and Syverson were supported by ONR and DISA. Cervesato was partially supported by NSF grant INT98-15731 "Logical Methods for Formal Verification of Software" and NRL under contract N00173-00-C-2086.

1 Introduction

Although research has been going on in the formal analysis of cryptographic protocols for a number of years, it is only slowly being integrated into the protocol design process. In this paper we describe how we furthered the integration of analysis and design by working closely with the Multicast Security Working Group in the Internet Engineering Task Force on the analysis of a proposed Internet Standard, the Group Domain Of Interpretation (GDOI) Protocol, using the NRL Protocol Analyzer (NPA) and the associated NPATRL requirements language. We found the formal analysis to be extremely beneficial when done in close cooperation with the protocol designers, and we outline our findings and the impact on the protocol in this paper. We also outline the various challenges that had to be met before we could perform a successful analysis.

Perhaps not surprisingly, the most important challenge turned out to be getting the requirements right. Requirements are in general well understood for key distribution protocols involving two or three parties, and a number of formalizations of such requirements exist. But they are not as well understood for group key distribution protocols, where keys may possibly be distributed among an arbitrarily large group of principals that may join or leave the group at any time. Concepts such as secrecy and freshness that can be tied to a single session in a two or three party protocol become more elusive when the number of principals which are permitted to know a secret is unspecified, and when a principal's permission to know a secret can change over time as it joins or leaves the group.

We attempted to fill this gap by developing a set of formal requirements for the GDOI group key management protocol [1], a protocol which we have been formally specifying and verifying as part of a joint effort with the IETF MSec working group. To do this, we used the NPATRL requirements language [21], a temporal language for cryptographic protocol requirements intended for use with the NRL Protocol Analyzer [15, 16]. What we found as a result of this effort was that requirements for group key distribution protocols were little understood, and that as much or more work needed to be put into developing a set of formal security requirements as into the formal specification of the protocol itself. For GDOI, the requirements for authentication and freshness turned out to be rather similar to those for pairwise protocols. This we believe is the result of the fact that GDOI provides authentication only for the Group Controller/Key Server (GCKS), which is responsible for distributing keys to the group, rather than for authentication among the group members themselves. However, we found the secrecy requirements radically different from those for pairwise protocols. This has to do with the fact that GDOI was developed for the distribution of keys to dynamic groups, so the set of principals who have permission to know the keys is constantly changing. This forced us to address the issue of how much permission a new or departed member should have to keys that were not current when he was a member of the group. The added complexity of the secrecy requirements resulting from the needs of secure group communication has also induced us to develop a deductive system for NPATRL so that it can be used to derive simple requirements from complex ones. We describe the logic and show how it can be applied in simplifying requirements.

The actual course we followed in the analysis of GDOI was first to write a formal specification of the protocol in the NPA specification language, then to write a formal specification of its requirements in NPATRL, and finally use NPA to verify that the protocol met its requirements. This deviates somewhat from the recommended practice, which is to deal with the requirements first, but was motivated by the fact that we already had a draft of the protocol to work with when we started. We found various problems at each stage of the analysis. When we did, we would bring the matter up with the protocol designers; this would lead to a discussion of various options to take, and would usually result in a change to a protocol. The final decision was usually motivated by a consideration (and sometimes a re-evaluation) of the protocol's requirements, showing again the importance of having a clear unambiguous way of stating them.

The rest of this paper is organized as follows. In Section 2 we give an overview of the GDOI protocol. In Section 3 we introduce the NRL Protocol Analyzer and the NPATRL logic, showing how it can be considered as a subset of Linear Temporal Logic (LTL), and we describe the normal form for NPATRL requirements for the NRL Protocol Analyzer. In Section 4 we present the NPATRL requirements for GDOI, and we also describe our system for building secrecy requirements. In Section 5 we describe our specification and analysis of the protocol, and the impact that this had on the protocol's design. In Section 6 we conclude the paper and sum up the impact of our work and some of the open problems that remain to be solved. In Appendix A we show how we were able to use the logic to remove recursiveness from the secrecy requirements and reduce them to normal form.

2 An Overview of GDOI

The GDOI (for Group Domain Of Interpretation) protocol [1] is intended to be used with the Internet Key Exchange (IKE) protocol [9, 5] to allow a Group Controller and Key Server (GCKS) to distribute keys to members of a group. Although it does not specify any mechanisms such as key hierarchies [2] for efficiently distributing keys to group members or for expelling or adding members, it is designed to be compatible with the use of such techniques.

GDOI uses three categories of keys. *Category 1* keys are the pairwise keys shared between the GCKS and potential members. *Category 2* keys are key-encryption keys that are used to protect the *Category 3*, or traffic encryption keys.

2.1 Group-Key Push Subprotocol

For GDOI, the Category 1 (pairwise) keys are distributed via IKE Phase 1, which is described in [9, 5]. Key-encryption keys and traffic-encryption keys are created by the GCKS. The GCKS distributes these keys to the group as a whole by a *groupkey-push* message encrypted with the current key-encryption key. The GCKS maintains a sequence number `SEQ` that is incremented every time a new groupkey-push message is sent. The current value of the sequence number is included in the groupkey-push message. This allows group members to verify that a message is not a replay of one that they have already received. The groupkey-push message is also digitally signed by the GCKS using its private key so that receivers can verify that it was sent by the GCKS and not by another group member.

Since GDOI is based on IKE, it makes use of the ISAKMP [13] header format. Some details about this header turned out to be relevant to our analysis, so we include them here. First, each header begins with a random or pseudo-random number. In IKE Phase 2 this is a cookie pair contributed to by the initiator and the responder. Since this is not practical for the groupkey push message, it instead begins with a random or pseudo-random number generated by the GCKS. Secondly, each header includes a message ID, which is is a random number generated by the initiator. If a protocol contains more than one message, that message ID remains the same throughout the entire exchange.

The groupkey-push message appears as follows in [1]:

Member		GCKS or Delegate
	$\longleftarrow$	`HDR*, SEQ, SA, KD [,CERT], SIG`

The term `HDR*` indicates that everything is encrypted after the header, in this case using the current key encryption key. `SEQ` is the sequence number, `SA` the security association for this key payload, which gives such information as algorithms used, key lifetimes, etc., and `KD` the new keying material. `SIG` is a digital signature taken over the message and the header, and `CERT` is an optional certificate for the signature key.

2.2 Group-Key Pull Subprotocol

When a principal wants to join a group, it takes part in a four-message *groupkey-pull* exchange with the GCKS. All messages are encrypted and authenticated with the pairwise key shared between the two principals. In the first message, the principal sends a request to join the group, including the group identifier and a nonce `Ni` to help in verifying freshness.

The GCKS responds with its own nonce `Nr` and with the group security association `SA`, which describes the mechanisms (e.g. encryption algorithms) and policies used by the group. It holds off on sending the keying material itself until it can verify that the request is recent. This helps prevent denial of service attacks, and if POPs (see below) are used, allows the GCKS to wait until it knows the member has the appropriate credentials. The group member responds with a hash taken over the two nonces, as well as some optional information, described below. The GCKS sends the keying material and the current value of the sequence number in the last message.

There are also some optional fields in the last two messages. If it is required by the group policy, the member can send its own part of a Diffie-Hellman key exchange in the third message (`KE_I`), and the GCKS can respond with its part of the exchange in the fourth message (`KE_R`). The resulting Diffie-Hellman key is used to encrypt the group keying material by use of exclusive-or. The purpose of this is to provide perfect forward secrecy: even if a pairwise key is compromised, the intruder can learn only keys distributed after the compromise, not those distributed before.

Another option allows the two principals to verify that each is authorized to act in their respective roles. This is the proof-of-possession (POP) option, where each party includes a public key certificate signed by a relevant authority, and proves his or her possession of the key by using it to sign the two nonces that were exchanged earlier in the protocol.

The four messages sent in the groupkey-pull exchange appear as follows in [1]:

Initiator (Member)		Responder (GCKS)
`HDR*, HASH(1), Ni, ID`	$\longrightarrow$	
	$\longleftarrow$	`HDR*, HASH(2), Nr, SA`
`HDR*, HASH(3) [, KE_I] [,CERT] [,POP_I]`	$\longrightarrow$	
	$\longleftarrow$	`HDR*, HASH(4) [,KE_R], SEQ, KD [,CERT] [,POP_R]`

where `Ni` and `Nr` are the two nonces, `SA` is the security association, `KE_I` and `KE_R` are the optional Diffie-Hellman halves, `CERT`, `POP_I`, `POP_R` are the certificates and signatures used in the optional proof-of-possession exchange, and `SEQ` and `KD` are the sequence number and keying material (exclusive-ored with the Diffie-Hellman key if that is used), respectively. The way in which `POP_I`, and `POP_R` were computed varied as the protocol evolved, but in the version we verified, they were created by computing a digital signature over a hash taken over the initiator's nonce followed by the responder's nonce. The notation `HDR*` means, as before, that all information after the header is encrypted, this time with the a key derived from Category 1 key already shared between the GCKS and the member. The hashes in the exchange are computed over message ID from the header plus the payloads sent in the body of the message (minus any formatting information), with the exception of certificates, in the order that they appear. The key used in computing the hashes is also derived from the Category 1 key, but is different from the encryption key. More detail may be found in [1].

2.3 About Keys

Elimination of a member from a group is accomplished via the groupkey push message, which can be used to transmit the a new key to the group in a way such that the expelled member does not receive it. How exactly GDOI accomplishes this is beyond the scope of the GDOI specification itself. However, this can be done using something called a *key hierarchy* in the key download field. Basically, a key hierarchy is a tree, the root of which is the actual key used for encryption. Nodes of the tree encrypt and authenticate the nodes above it. When a principal is admitted to the group, it is assigned a leaf of the tree. When it leaves the group, only the (limited) portion of the tree it needs to compute the group key ought to be updated. This allows access control for both entering and leaving members to be enforced in an efficient way, as well as providing extra security beyond that provided by the key-encryption key used to encrypt the push message, since a new key will be protected by the keys below it in the hierarchy. See [2] for a discussion and overview of key hierarchies.

2.4 A Preview of the GDOI Requirements and Analysis

GDOI must satisfy requirements for authentication, freshness, and secrecy.

Authentication The authentication requirements are similar to those for pairwise protocols. Indeed, the groupkey pull protocol *is* a pairwise protocol. The key distributed is shared with other group members, but this does not affect the authentication requirements. Furthermore, for the groupkey push message, the only authentication requirement is that its single message must have been generated by the GCKS for that group. In spite of this, we found the most interesting violations when we attempted to verify authentication properties. These had to do with possible type confusion attacks between POP and groupkey push messages, and were realistic enough to motivate the designers to change the specification of both the POP signatures and the groupkey push message to avoid it. These attacks are described in Section 5.2.3.

Freshness Freshness requirements turned out to be a little more subtle, mainly because of the slightly different requirements for the groupkey pull protocol and groupkey push message. Since the latter could not make use of challenge-response, its freshness requirements were of necessity slightly weaker. We identified two different types of freshness: recency freshness, the requirement that a principal accept the most recently current key, and sequential freshness, the requirement that a principal should not accept a key that is older than any key it has already accepted. These correspond closely to notions of freshness we developed earlier in [21]. The groupkey pull protocol should satisfy both kinds of freshness, while the groupkey push message can only satisfy sequential freshness. We found several problems here. In one case, the incorrect placement of the sequence number meant that freshness could not be guaranteed for traffic-encryption keys delivered under certain circumstances. This is detailed in Section 5.2.1. In another case, we found that specification was written in such a way that it was possible to construct compliant implementations of the groupkey pull protocol that were in violation of sequential freshness. This is detailed in Section 5.2.2. In both cases, the specification was modified to eliminate these flaws.

Secrecy The secrecy requirements were probably the most interesting, and the most different from pairwise protocol requirements. GDOI does not support a single set of secrecy requirements. Rather, it can be configured to implement different combinations of requirements. These include perfect forward secrecy (if a pairwise key is compromised, then the intruder learns only the keys it is used to distribute after the compromise) and forward and backwards access control (a member should not have access to keys current after it leaves the group (forward access control) or before it joins the group (backward access control)). Some of these options, such as perfect forward secrecy, can be enforced by GDOI itself. Others, such as forward and backwards access control, can only be enforced by the use of the appropriate key hierarchy and are beyond the scope of GDOI, as well as the current capabilities of NPA. Thus we only attempted to analyze the simplest secrecy properties in our NPA analysis, but since a precise specification of requirements is helpful whether or not it is accompanied by an analysis, we provide a means for specifying secrecy requirements in Section 4.5. Since there were so many ways requirements could be mixed and matched, we found it helpful to specify them separately in terms of different sequences of events that could cause the compromise of a key, and then describe the ways in which they could be combined. We then needed ways of simplifying the resulting requirements. This necessitated the development of a deductive system for NPATRL, which is described in Section 3.

3 The NPATRL Logic

In this section, we define the concepts, languages and tools used in the rest of the paper. More precisely, we give an informal account of the NRL Protocol Analyzer, which was used to perform the actual analysis of GDOI, in Section 3.1. We then introduce NPATRL in Section 3.2, relate it to mainstream temporal logics in Section 3.3, and present an axiomatization for it in Section 3.4. We conclude with a breif discussion of the fragment of NPATRL used in the NPA in Section 3.5.

3.1 The NRL Protocol Analyzer

The NRL Protocol Analyzer, or NPA for short, is a computer-assisted verification tool for security protocols which combines model checking and theorem-proving techniques to establish authentication and secrecy properties. We present merely a brief overview here. The interested reader is invited to consult [15, 16] for further details.

A protocol is modeled as a number of communicating state machine templates, each associated with a different role. Their transitions correspond to the actions that comprise the corresponding role. At run time, roles are executed by *honest principals* who faithfully follow the protocol. Several instances of a role involving the same principal can be executing at the same time, and they are distinguished by means of a *round number*. Round numbers are implemented in the NRL Protocol Analyzer in the following way: each principal possesses a local counter that is incremented every time that principal engages in a transition. When a principal starts to execute a protocol instance, that instance is assigned a round number equal to value of the counter at the time it begins to execute that instance. All further transitions involving that instance of a role will have the same round number. This ensures that principal name and round number will be enough to identify any instance of a role uniquely.

The intruder is modeled after the Dolev-Yao adversary [4]. *Dishonest principals* share their keys and other confidential information with the adversary. Thus a dishonest principal is assumed to be part of the adversary, since anything a dishonest principal could do the adversary could do as well.

The messages in transit, the information held by each principal and the intruder, the runs currently being executed, and the point that each of them has reached constitute the global *state* for the NPA. A protocol action implements a local transformation with global effects on the state. The initial state is implicit in the protocol specification.

In order to verify a protocol, a specification is fed into the run-time system of the NRL Protocol Analyzer together with the description of a family of states that correspond to attack situations. The system applies protocol actions backwards from these target states until it either reaches the initial state, or it exhausts all possibilities for doing so. As it regresses back towards the initial state, the NPA maintains a *trace* of the sequence of actions that, when executed, lead to the target state. If the initial state is ever reached, the resulting trace is a potential attack. If all possibilities are exhausted, there is no attack of the kind sought. Although the search space is in general infinite, the NPA incorporates techniques based on theorem proving that have, in most cases, the effect of soundly restricting the search to a finite abstraction.

Traces in NPA are sequences of *events*. Each event is associated with an NPA transition. An event consists of five components: the name T of the transition the event is associated with, the principal P executing the transition, a list Q of other parties involved in the transition (e.g. the putative sender of a message P is receiving or the intended receivers of a message P is sending), the set L of relevant words chosen by the specification writer, and the local round number N associated with the transition.

NPA goal states can be expressed by referring to terms known by the intruder, values of local state variables, and conditions on traces of events. Examples of goal states would be one in which the same key has been accepted twice by a principal (two events occurring) or a state in which a responder B accepts a key as good for communicating by an initiator A, but in which A never initiated the protocol (one event having occurred and another event not having occurred previously), a state in which the intruder knows a term K and a principal A has accepted K as a key (the intruder knowing a term together with an event having occurred).

The NPA determines whether a protocol rule could be used to produce a state by means of a process known as *narrowing*. Terms (that is expressions made out of variables, constants, and function symbols) used in protocol specifications are assumed to obey a set of explicitly defined rewrite rules, that is, equations such that the right-hand side of the equation is "simpler" than the left-hand side according to some well-defined measure. An example of a rewrite rule would be one saying that the result of encrypting a term and then decrypting it with the same key reduces to the original term. Terms used in state descriptions are assumed to be irreducible (no further rewrite rules apply), while terms used as output of rules may possibly not be reducible. The narrowing algorithm is used to find all substitutions to the variables involved such that the terms in the rule output become reducible to the terms in the state description. The narrowing algorithm is very dependent upon the fact that the entities used are rewrite rules, as well as the fact that they obey certain other well-defined properties.

The NPA makes no assumptions about limits on the number of protocol executions, the number of principals performing the different executions, the number of interleaved executions, or the number of times cryptographic functions are applied. This results in a search space that is originally infinite. However, the NPA provides means for specifying and proving inductive lemmas about the unreachability of infinite classes of states. This allows the user to narrow down the search space so that in many cases an exhaustive search is possible. These inductive lemmas are formulated in terms of formal languages. The user gives the NPA a seed term, and the NPA uses the seed term to construct a language and prove that, if the intruder learns a term in the language, then it must have already known a term in that language, thus inductively proving that the intruder can never learn a term in the language. For protocols involving public and shared-key encryption, we have developed a standard set of seed terms: master keys, encrypted data where the data is not known by the intruder, decrypted data where the term decrypted is not known by the intruder, concatenation of two terms where one of the terms is not yet known by the intruder, and signed data.

3.2 The NPATRL Syntax

The NRL Protocol Analyzer has successfully analyzed a number of protocols, sometimes uncovering previously unknown flaws [15, 16]. But, secrecy and authentication goals are awkwardly expressed, as states that should not be reachable from the initial state. This unintuitive and occasionally error prone way of writing requirements would have made it very difficult to use the NPA for large protocols.

The *NRL Protocol Analyzer Temporal Requirements Language*, better known as NPATRL (and pronounced "N Patrol"), was designed to address these shortcomings [21]. This formalism makes available the abstract expressiveness of a logical language to specify requirements at a high enough level to capture intuitive goals precisely, and yet it can be interpreted in the NPA search engine.

NPATRL requirements are logical expressions whose atomic formulas are *event statements*, which mostly correspond to events in the NRL Protocol Analyzer; they include events denoting actions by honest principals that can be found in the trace of an NPA search, and the special learn event that indicates the acquisition of information by the adversary. NPATRL's syntax for events is similar but not identical to the NPA's. In NPATRL, the NPA accept event given above is written:

$$\mathsf{initiator\_accept\_key}(\mathsf{user}(A, \mathsf{honest}), \mathsf{user}(B, H), K, N)$$

The logical infrastructure of NPATRL consists of the usual connectives $\neg$, $\wedge$, $\rightarrow$, etc, and the temporal modality $\diamond\!\!\!\!-$ which is interpreted as "happened at some time before" or "previously".

For example, we may have the following requirement:

> *If an honest principal A[1] accepts a key K for communicating with another principal B, then a server must have previously generated and sent this key with the intention that it should be used for communications between A and B*

We can use NRL Protocol Analyzer events to construct an NPATRL formula that expresses it:

$$\begin{array}{l} \mathsf{initiator\_accept\_key}(\mathsf{user}(A, \mathsf{honest}), \mathsf{user}(B, H), K, N) \\ \rightarrow \;\diamond\!\!\!\!-\; \mathsf{svr\_send\_key}(\mathsf{server}, (\mathsf{user}(A, \mathsf{honest}), \mathsf{user}(B, H)), K, N) \end{array}$$

This formula is a simple expression of the above requirement.

Intuitively, the protocol verification process changes from what we discussed in the previous section by using NPATRL requirements where the final state appeared. More precisely, we first need to map every NPATRL event statement to an actual event or set of events in the NPA specification of the protocol. However, this mapping does not need to be one-to-one; it is possible to have one NPATRL event map to more than on NPA event, and more than one NPATRL event map to the same NPA event. In particular, if the intruder learns more than one term as a result of an NPA transition, this can be expressed by mapping two NPATRL learn events to the NPA event describing that transition. Conversely, if there are two ways in which a principal might accept a key as valid, and there are described by two different NPA transitions, we might have a single NPATRL event statement map to the two NPA events describing the two transitions.

Once we have performed the mapping, we use the negation of each NPATRL requirement to provide a way to characterize the states that should be unreachable if and only if that requirement is satisfied. At this point, we perform the analysis as in the previous section: if the NPA proves that this goal is unreachable, the protocol satisfies the original requirement. Otherwise, it returns a trace corresponding to an attack on the protocol that potentially invalidates the requirement.

A couple of particular points about NPATRL expressions: events occur exactly once. This means that atomic formulas are true at exactly one point in a trace (if at all). There is nothing in NPATRL syntax to automatically guarantee

[1]See Section 3.1 for the definition of an honest principal.

this uniqueness; it is assumed that event statements contain enough individuating information in their arguments or predicate to enforce this. Note that NPA guarantees this uniqueness, in part by having all events indexed both by local runs and timestamps. Second, "$\diamondminus$" is a strict operator; it includes times prior to the present time but does not include the present time. It is also convenient, especially when stating axioms, to have the dual operator in our language, "$\boxminus$", read as "at all previous times" or "always previously". It can be defined logically by, $\boxminus\varphi \leftrightarrow \neg\diamondminus\neg\varphi$, where φ is an formula.

NPATRL has been extensively used in the last few years to analyze protocols with various characteristics. Among these, generic requirements have been given for two-party key distribution protocols [19, 20] and two-party key agreement protocols [21]. The most ambitious specification undertaken using NPATRL has involved the requirements of the credit card payment transaction protocol SET (Secure Electronic Transactions) [14]. SET proved particularly difficult to specify for several reasons. One of these was that the objects to be authenticated are dynamic: unlike keys, what is agreed upon changes as it passes from one principal to another. This exercise revealed several ambiguities [14].

Our current task, formalizing group key management requirements, has its own dynamics. Even when the data objects (keys) are constant, the principals sharing them are not. And the very notion of a session is much less well defined than in previously studied cases. Perhaps most significantly, until this point we had been able to use NPATRL as just a language. All statements were interpreted into the NPA and evaluated there. However, we have found it necessary to reason at the level of NPATRL itself. This requires a deductive system for our logical language.

3.3 Interpreting NPATRL into LTL

A number of temporal logics [6, 8] have been investigated and successfully used in recent years. In this section, we will relate NPATRL with the closest proposals in this field. Specifically, we will focus our attention to *Linear Temporal Logic* (*LTL*). We show that NPATRL can be seen as a sublanguage of LTL. This observation opens the doors to importing theoretical results and practical techniques developed for LTL, and in particular the efficient implementation methods available for this language [7].[2]

LTL belongs to the class of logics that postulate that any moment has a single successor (as opposed to branching temporal logics that admit several possible futures for a given state). Therefore time is represented as a line, and each instant is characterized by the formulas that are true in it. An LTL specification omits the time-line in favor of the relations between formulas at different instants. We have the following syntax:

$$\varphi \;::=\; p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi \mid \Box\varphi \mid \Diamond\varphi \mid \varphi_1\,\mathcal{U}\,\varphi_2.$$

LTL adds a number of temporal modalities to atomic formulas and the familiar

[2]We are grateful to Lenore Zuck for pointing out this connection to us.

operators from classical logic. $\bigcirc\varphi$ ("next φ") holds if φ is true in the next instant of time; $\Box\varphi$ ("always φ") holds if φ is currently true and will remain true at every instant in the future; $\Diamond\varphi$ ("sometimes φ") holds if φ is true at some time from now on; finally $\varphi_1 \,\mathcal{U}\, \varphi_2$ ("φ_1 until φ_2") holds if φ_1 will continue to be true until φ_2 becomes true. We included $\mathcal{U}$ for completeness but we will not make use of it in this section. Notice however $\Box$ and $\Diamond$ can be expressed in terms of $\mathcal{U}$. The interested reader is referred to [6, 8] for a further discussion of LTL and other temporal logics.

This presentation, which we tried to keep intuitive, hints at a number of differences between LTL and NPATRL. We will now examine them. The most evident, but also least invasive, mismatch is that LTL is a logic of the future while NPATRL's temporal operators are used to talk about the past. Duality principles effortlessly reconcile these views. Some proposals, e.g. [12], have considered explicit past operators, but interpreting LTL "the other way around" will be sufficient for us.

NPATRL's events model transitions between the implicit states of this language. The effect of an event is to alter the values of unmentioned state components such as the messages known by a principal (or the intruder) or what protocol actions can be fired next. The atomic formulas of LTL (and of the modal logic discussed in Section 3.4) represent instead state properties, i.e. the observable local effect of implicit transitions. We bridge the gap between transition-oriented NPATRL and state-oriented LTL by interpreting the events of the former as meta-level state properties in the latter: event e will be mapped to the property $p(e)$ that we will read as "e is the next transition that will be applied in this state" (alternatively, "e is the transition that lead to this state"). Observe that both events and states are explicit in NPA.

The next discrepancy has to do with the interpretation of the temporal modalities. In NPATRL, $\boxminus$ and $\diamondminus$ are strict: their scope does not encompass the current instant, which does contribute to the interpretation of $\Box$ and $\Diamond$ in LTL. Therefore, $\boxminus\varphi$ is true in a model where φ holds at every moment but the current one, while $\Box\varphi$ is false. Yet, the "next" operator allows for a simple embedding $\ulcorner\_\urcorner$ of NPATRL into LTL:

$$\begin{array}{rcl|rcl}
\ulcorner e \urcorner & = & p(e) & \ulcorner \boxminus\varphi \urcorner & = & \bigcirc\Box\ulcorner\varphi\urcorner \\
\ulcorner \neg\varphi \urcorner & = & \neg\ulcorner\varphi\urcorner & \ulcorner \diamondminus\varphi \urcorner & = & \bigcirc\Diamond\ulcorner\varphi\urcorner \\
\ulcorner \varphi_1 \vee \varphi_2 \urcorner & = & \ulcorner\varphi_1\urcorner \vee \ulcorner\varphi_2\urcorner & & &
\end{array}$$

The left column translated the traditional fragment directly. The more interesting right column uses the $\bigcirc$ modality of LTL to shift the time-line by one point to simulate the strict operators of NPATRL with the corresponding non-strict modalities of LTL. This encoding shows that NPATRL can be thought of as a sublanguage of LTL.

Notice that $\Box$ and $\Diamond$ can easily be expressed in NPATRL. If $\llcorner\_\lrcorner$ denotes our inverse translation,

$$\llcorner\Box\varphi\lrcorner = \llcorner\varphi\lrcorner \wedge \boxminus\llcorner\varphi\lrcorner \qquad \text{and} \qquad \llcorner\Diamond\varphi\lrcorner = \llcorner\varphi\lrcorner \vee \diamondminus\llcorner\varphi\lrcorner.$$

However, there is no way to render "next" in NPATRL ("until" does not seem expressible either).

We conclude with a note about the form of atomic formulas. Clearly, NPATRL relies on a first-order language. While the propositional instance of LTL is most commonly used, various first-order extensions have been investigated [6]. This brings us to the decidability issue: since the non-temporal fragment of NPATRL embeds the language of first-order Horn clauses, which is well-known to be Turing complete, validity is undecidable in NPATRL.

3.4 NPATRL Axioms

We give axioms of a normal modal logic adequate to capture the needed temporal reasoning. As we noted in our discussion of NPATRL and LTL, for our purposes that means reasoning about discrete linear strict orderings of events. We present a standard axiomatization as set out in [8] that is sound and complete with respect to a semantics of time indexed and ordered by the natural numbers. Following [8], the axioms are given their traditional names, some tracing back to the early twentieth century. Readers are referred to standard texts for details on systems of modal and temporal logic, e.g., [3, 8, 11].

Our logic has two inference rules:

Modus Ponens: From φ and $\varphi \to \psi$ infer ψ.

Necessitation: From $\vdash \varphi$ infer $\vdash \boxminus\varphi$.

'$\vdash$' is a meta-linguistic symbol. '$\Gamma \vdash \varphi$' means that φ is derivable from the set of formulae Γ (and the axioms as stated below). '$\vdash \varphi$' means that φ is a theorem, i.e., derivable from axioms alone. Axioms are all instances of tautologies of classical propositional logic, and all instances of the following axiom schemata

K $\boxminus(\varphi \to \psi) \to (\boxminus\varphi \to \boxminus\psi)$

4 $\boxminus\varphi \to \boxminus\boxminus\varphi$

D $\boxminus\varphi \to \diamondminus\varphi$

L $\boxminus((\varphi \wedge \boxminus\varphi) \to \psi) \vee \boxminus((\psi \wedge \boxminus\psi) \to \varphi)$

Z $\boxminus(\boxminus\varphi \to \varphi) \to (\diamondminus\boxminus\varphi \to \boxminus\varphi)$

The first axiom, **K**, guarantees that our temporal operators respect the non-temporal part of the logic. The **4** axiom guarantees that temporal reasoning is transitive. **D** guarantees that there are always more points in time; thus something does not end up being true for all previous points in time simply because there are no previous points.[3] The fourth axiom, **L**, guarantees that events are weakly-connected (comparable), specifically that any two points connected to any third are also connected to each other. The last, **Z**, guarantees that time is discrete: between any two points there are at most finitely many other points.

[3]This is entirely compatible with the usual assumption that protocols start with a finite number of principals and in an initial state, which can be reflected by the non-logical assumption of a point prior to which the state is always the same, i.e., with the schema $\diamondminus(\varphi \to \boxminus\varphi)$.

There is some discussion of logics containing these axioms in [8] and [11]. In [11], axiom **L** is called "$\mathbf{Lem}_0$" after Lemmon.

3.5 NPA's Use of NPATRL and NPA Acceptable Expressions

NPATRL requirements are used with NPA by having the user translate the NPATRL requirements into states that the NPA will attempt to prove unreachable. This is currently done by hand, although we believe that it should be possible to write software to achieve this. Since NPATRL requirements express conditions on events and sequences of events, it is necessary to take the negation of NPATRL requirements, and then to have NPA prove that no state for this negated requirement can be reached from an initial state. However, although NPATRL was originally designed to be used with the NRL Protocol Analyzer, one of its main motivations was intuitiveness in specifying requirements while NPA was originally designed simply for effective automated protocol analysis. Thus it is not surprising that NPATRL is actually much more expressive than the set of specifications whose negations are accepted by the tool. So, in order to make NPATRL usable with NPA, it is necessary to identify a subset of NPATRL requirements that are acceptable by NPA, and to put them into a normal form that is parsable by NPA.

An NRL Protocol Analyzer query can be specified in terms of three things: terms known by the intruder, values of local state variables, and sequences of events that did or did not occur. The part of the query concerning sequences of events corresponds most closely to the NPATRL events. However, these events only correspond to user actions, and do not include learn events, which correspond to intruder actions. In order to capture intruder learn events, we will need to make use of the part of the query that specifies terms known by the intruder. This can cause some difficulties, since an NPA query does not specify when the terms were learned by the intruder. However, we can simplify matters by limiting ourselves to queries which specify the learning of only one term, so we only have to worry about the placement of learn events relative to non-learn events, whose order it is possible to specify explicitly in NPA. We also have several tricks available with which we can get around some of the difficulties posed by this discrepancy between NPATRL and NPA. In the case that a specification is of the form

$$\mathsf{learn}(P, (), X, N) \rightarrow \diamondminus\, E$$

where E is an event or sequence of events, the negation would be

$$\mathsf{learn}(P, (), X, N) \ \wedge\ \neg \diamondminus\, E$$

The question that will actually be put to NPA is whether or not there is a state in which the intruder knows X and E did not occur. NPA will search for X, at each step checking whether or not the event E occurred. Unless it is possible for E to be the same event as the one in which the intruder learns X, this will guarantee that NPA will only look for sequences in which E occurs before the learning of X.

Conversely, we may want to ask whether or not it is possible for the intruder to learn X without some other sequence of events F occurring previously. As it turns out, we cannot ask this, but we can ask if it is possible for the intruder to learn X and for some sequence of events F to happen. In many cases, this is what we want any way. For example, if the intruder learns a key that is shared between two honest principals, we do not care whether or not the key was learned before or after it was accepted by the principals. This condition is then

$$\diamondminus\ \mathsf{learn}(P, (), X, N) \rightarrow \neg(\diamondminus\ F)$$

and its negation, whose unreachability we want to check, is

$$\diamondminus\ \mathsf{learn}(P, (), X, N)\ \wedge\ \diamondminus\ F$$

The question that we ask NPA is whether or not the intruder knows X and F *did* occur. Then NPA will start up two lines of inquiry, one in which the last event is the intruder learning X (and so F occurs previously or is the same event as the intruder learning X), and one in which the last event is the last event in F, so that the intruder learning X predates the last event in F, or is the same event as F.

In most cases, we have found this kind of workaround to be adequate. However, in our analysis of GDOI we found a case where a precise specification of the timing of learn events with respect to other events was critical. This was the requirement that one condition under which the intruder could learn a group key was that a dishonest member must have joined the group and not have left yet. As it turned out, we did not attempt to verify this particular condition for other reasons, but our inability to even pose this question to NPA has motivated us to implement the learn event as part of the NPA query language; we are currently in the process of doing this.

Given that caveat, we specify a normal form $\mathbf{R}$ for expressions acceptable by the current version of the NPA in a BNF grammar. Note that this normal form simply functions as an interface between NPATRL requirements specifications and NPA expressions in the context of the current expansions in the class of protocols to which they are applied. It is not claimed to be, e.g., maximal or especially intuitive per se, but merely one such possible interface. Its BNF grammar is as follows. We let w stand for a learn event, a stand for any atomic event that is not a learn event, and let b stand for any atomic event.

$$\begin{array}{rcl}
\mathbf{E} & ::= & \diamondminus a \ \big|\ \diamondminus(a\ \wedge\ \mathbf{E}) \\
\mathbf{F} & ::= & \mathbf{E}\ \big|\ \mathbf{E}\ \vee\ \mathbf{F}\ \big|\ \mathbf{E}\ \wedge\ \mathbf{F} \\
\mathbf{G} & ::= & \neg\mathbf{E}\ \big|\ \neg\mathbf{E}\ \wedge\ \mathbf{G} \\
\mathbf{R} & ::= & \neg b \ \big|\ a \rightarrow \mathbf{G}\ \big|\ b \rightarrow \mathbf{F}\ \big|\ a \rightarrow \mathbf{G}\ \vee\ \mathbf{F}\ \big|\ a \rightarrow \mathbf{G}\ \wedge\ \mathbf{F} \\
 & & \big|\ \diamondminus w \rightarrow \mathbf{G}\ \big|\ \diamondminus w \rightarrow \mathbf{G}\ \vee\ \mathbf{F}\ \big|\ \diamondminus w \rightarrow \mathbf{G}\ \wedge\ \mathbf{F}
\end{array}$$

The special handling of "b" events (which may be learn events) is an artifact of the fact that learn events are not currently explicitly specified in the NPA query language. Instead, they are specified by means of terms that the intruder knows, and thus must always be referred to as something that happened in the past. Once learn events are implemented explicitly in the NPA query language, it will be possible to handle them in the same way as other events, which will allow us to simplify the normal form language.

4 Requirements for GDOI

In this section we give NPATRL specifications of the various relevant events in the GDOI protocol as well as specifications of the various GDOI security requirements. For the purpose of exposition, we begin with the requirements that are closest to those for two-party protocols, the authentication requirements. We then proceed to freshness requirements, which are somewhat more complex, and finally to secrecy requirements, which are the most complex and differ the most from requirements for two-party protocols.

4.1 Assumptions

We assume that each group is managed by one GCKS (it is possible to have more, but the means for doing this are not specified in the GDOI document). We assume that a GCKS may manage more than one group, and that a member may belong to more than one group. We assume that members may both join and leave a group, and a member may have concurrent and/or overlapping memberships in the same group.

We assume the usual Dolev-Yao style intruder, who can read, alter, destroy, and create traffic, and is in league with any dishonest principals, who share all data with it. We assume that all GCKSs are honest, but that some members may be dishonest. Note that as a result of this assumption we make no distinction between the intruder's learning a key and a principal learning a key to which it is not entitled. Only dishonest principals will attempt to gain access to keys to which they are not entitled, and dishonest principals are assumed to share all information with the intruder.

We assume that there are two ways in which a key can be compromised that cannot be prevented by the protocol. One is by stealing: the intruder may learn the key by cryptanalysis, theft, etc. even if all possessors of the key are honest. The other is by having a dishonest member join the group.

Finally, in order to simplify matters, we only define events and requirements for key encryption keys, not traffic encryption keys.

4.2 GDOI Events

In general, events map to actual messages and vice versa. However, since the second and third messages of the groupkey-pull exchange simply defer computation in order to resist forms of denial-of-service attacks [17] and the NPA does

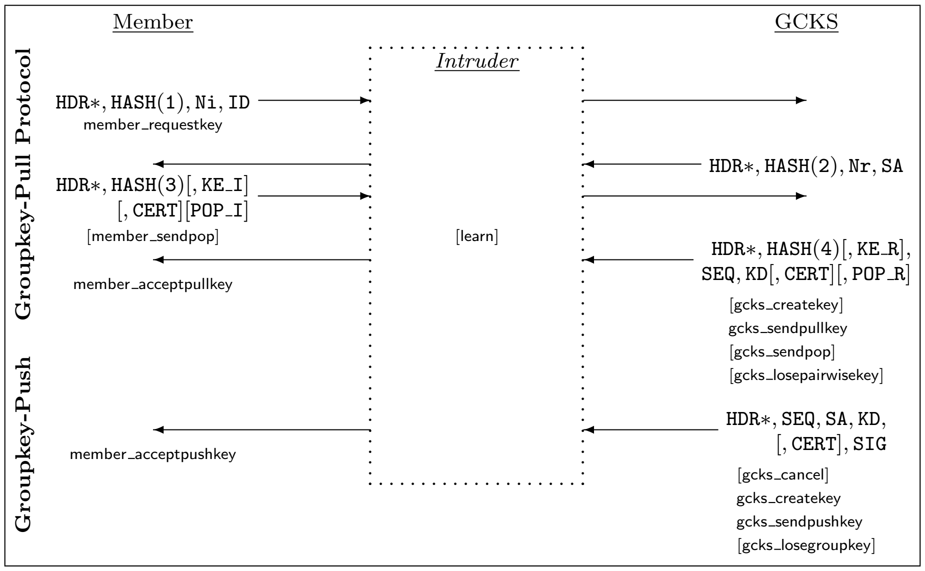


Figure 1: Event Annotations for GDOI

not currently support reasoning about denial-of-service, we behave as if their information load were compounded with the outer messages of this exchange.

We divide the possible GDOI events according to the principals that engage in them. There are four types of principals: the intruder, the GCKS, the group member, and an authorization server responsible for issuing credentials. Since a group member may be honest or dishonest, we represent a general group member as $\mathsf{member}(M, H)$ with H a variable, an honest member as $\mathsf{member}(M, \mathsf{honest})$, and a dishonest group member as $\mathsf{member}(M, \mathsf{dishonest})$ where honest and dishonest are constants.

We summarize the events considered in the present formalization and give their approximate location within the execution of GDOI in Figure 1. Events enclosed in brackets may or may not happen, depending on which options are enable (*e.g.*, [gcks_sendpop]) or the scenario we are considering (*e.g.*, [learn]). Other events will occur when the protocol reaches the point where they appear.

4.2.1 Intruder Events

There is only one intruder event of interest to us here: the event in which the intruder P learns a word W. We represent that as follows:

$$\mathsf{learn}(P, (), W, N)$$

4.2.2 The GCKS

The GCKS performs a number of actions of interest. It can create a key encryption key. It can admit and expel members. It can also cause a key to become current, and cause a key to expire. It can send a key, either in response to a member's request, or as part of a group-key push datagram. We represent these as follows:

Creating a key:

$$\mathsf{gcks\_createkey}(GCKS, (), (G, K_G), N)$$

where G is the group for which $GCKS$ is creating the key encryption key K_G.

Sending a key as a result of a pull exchange:

$$\mathsf{gcks\_sendpullkey}(GCKS, M, (K_G, N_M, N_{GM}, G, K_{GM}, PoP), N)$$

where M is the member, K_G is the key, G is the group, K_{GM} is the pairwise key, N_M is the nonce M uses in initiating the exchange, N_{GM} is the nonce G uses in responding, and PoP is a Boolean variable indicating whether or not a Proof-of-Possession is required. We also use the gcks_sendpullkey event to cover the GCKS's admitting M to the group, since M requests membership by initiating a pull protocol. We use N_{GM} to identify M's particular membership in the group. Note that this may not be the identifier used in a real application (as a matter of fact, GDOI does not specify any kind of membership identifier); however it is useful from a requirements point of view in that it allows us to distinguish between different and possibly overlapping memberships on the part of the same individual. Finally, PoP denotes a variable which is used to determine whether or not M was expecting a Proof-of-Possession (POP) from the GCKS.

Sending a key in a push message:

$$\mathsf{gcks\_sendpushkey}(GCKS, (), (G, K_G, K'_G), N)$$

The event gcks_sendpushkey causes one key, K'_G, to expire for group G and causes the next one K_G to become current. The initial key created for a group is first sent in a pull-key message. Except for such initial keys, we assume for convenience that a push-key message making K_G current is sent immediately after the create event that produced K_G (without the possibility of an intervening distribution in a pull-key message). Indeed, we could easily have made them map to the same transition, but this would have complicated things by forcing us to express our requirements differently for the initial group key and the subsequent group keys. We also assume that the initial key is sent in at least one pull-key response that takes place immediately after its creation. We say the initial key becomes current when that first pull-key response containing it is sent.

We note that neither gcks_sendpullkey nor gcks_sendpushkey tell the whole story about the keying material passed in these two messages. In actual fact, the pull-key message will contain, not only the current key, but also the relevant part of the key hierarchy that the member M needs to access the key. Likewise the gcks_sendpushkey message will also contain the portion of the key hierarchy that needs to be changed to give members access to the new key and prevent former members from accessing the new key, if this is desired.

Canceling a membership:

$$\mathsf{gcks\_cancel}(GCKS, M, (G, N_{GM}), N)$$

where M is the member, G is the group, and N_{GM} identifies the membership. Note that expulsion cancels only the membership with identifier N_{GM}, not all memberships of that member. In order to truly expel the member, all its memberships would have to be canceled.

We note that gcks_cancel would be achieved in GDOI by having the GCKS send out a push message containing a new key hierarchy from which M is excluded. We choose to specify gcks_cancel separately from gcks_sendpushkey since this allows us to avoid issues such as canceling multiple memberships in one message, etc.

Sending a POP:

$$\mathsf{gcks\_sendpop}(GCKS, M, (G, N_{GM}, N_M, C_G), N)$$

This event describes a GCKS sending a POP. C_G stands for G's credentials.

Stealing a Key:

Finally, we need to specify the stealing of a key. We think of this not as something done by the intruder, but as something done by the GCKS. In other words, the action of stealing a key needs to be precipitated by the GCKS "losing" a key. This appears paradoxical, but it is a result of our model's assumption that actions involving a piece of data can only be initiated by those in possession of it.

Note that it would also seem to make sense to specify the member's losing a key. However, in the case in which the key is actually generated by the GCKS, this is redundant, since compromise of that key is already described by a GCKS lose event. The only sort of member lose event that is *not* covered by a GCKS lose event is one in which the member accepts something that is not generated by the GCKS and then loses it. This would be of interest, but if we discovered that a member could accept a bogus key as genuine, it would make more sense to fix the protocol than to attempt to discover whether an intruder could make use of compromised bogus keys. So, for the sake of avoiding unnecessary complication of the analysis we did not include it.

We have two event statements, one describing the loss of a key-encryption key, and one describing the loss of a pairwise key:

$$\mathsf{gcks\_losegroupkey}(GCKS, (), (G, K_G), N)$$

where G is the group and K_G is the key.

$$\mathsf{gcks\_losepairwisekey}(GCKS, (), (GCKS, M, K_{GM}), N)$$

where K_{GM} is the pairwise key and M is the member who shares the key with the GCKS.

4.2.3 Member Actions

The relevant member actions involve accepting a key and requesting a key. A member can only request a key by initiating a group-key pull exchange, but it may accept a key as a result of receiving the final message of a group-key pull exchange or as a result of receiving a group-key push message.

The events are specified as follows.

Requesting a Key:

$$\mathsf{member\_requestkey}(M, GCKS, (G, N_M, K_{GM}), N)$$

where $GCKS$ is the GCKS, G is the group, N_M is the nonce M uses in initiating the request (to distinguish it from other requests), and K_{GM} is the pairwise key shared between $GCKS$ and M. This corresponds to M sending the first message in a group-key pull exchange.

Accepting a Key From a Group-key Pull Exchange:

$$\mathsf{member\_acceptpullkey}(M, GCKS, (G, K_G, N_{GM}, N_M, K_{GM}, PoP), N)$$

where $GCKS$ is the GCKS, G is the group, K_G is the key, and K_{GM} is the pairwise key shared between M and the GCKS. Again, this does not give the whole picture, as it leaves out the portion of the key hierarchy that the GCKS sends to M. The variable PoP is a Boolean indicating whether a Proof-of-Possession is requested.

Accepting a Key From a Group-key Push Message:

$$\mathsf{member\_acceptpushkey}(M, GCKS, (G, K_G, K'_G), N)$$

where $GCKS$ is the GCKS, G is the group, K_G is the new key, and K'_G is the current key encryption key. Again, we leave out the portion of the key hierarchy that M uses to authenticate the message and decrypt K_G.

For conciseness, we set

$$\begin{aligned}
&\mathsf{member\_acceptkey}(M, GCKS, (G, K_G), N)\\
\leftrightarrow\quad &\mathsf{member\_acceptpullkey}(M, GCKS, (G, K_G, \_, \_, \_, \_), N)\\
&\vee\, \mathsf{member\_acceptpushkey}(M, GCKS, (G, K_G, \_), N)
\end{aligned}$$

Here and in the rest of this paper, we write "_" for an argument whose actual value is irrelevant. Each occurrence can be thought of as a distinct variable.

Sending a POP:

$$\textsf{member\_sendpop}(M, \mathit{GCKS}, (G, N_M, N_{GM}, C_M), N)$$

This event describes a member M sending a POP. C_M stands for M's credentials.

4.3 Authentication Requirements

Since GDOI is only intended to address the problem of secure distribution of group keys, not the authentication of group members to each other, its authentication requirements are simple and rather similar to those for two-party protocols, as described in [19, 21]. Thus we give them first. There are authentication requirements for both the group member and the GCKS. The group member will want to know, if it accepts a key, that that key was generated by the GCKS for that group. The GCKS will want to know that, if it sends a key to a group member, then that group member requested a key. Finally, there are authentication requirements on the Proof-of-Possession (POP) algorithm. Initially, we took this requirement to mean that if a principal A received a POP from a principal B for use in a group G, then B should have generated that POP using its signature key, and intended it for A and group G. As we will see, this turned out to be too restrictive a requirement if we assume that pairwise keys can be compromised; we will discuss what we actually wound up verifying in Section 5.

4.3.1 Authentication of a Key to a Group Member

Since there are two different ways a group member can receive a key, we have two different sets of requirements. In the case of the group member M accepting a key K_G for group G as a result of the pull protocol, we require that one of two things must have happened; either the pairwise key shared between the member and the GCKS was lost, or the GCKS did send K_G to M for use in G:

$$\begin{aligned}&\textsf{member\_acceptpullkey}(M, \mathit{GCKS}, (G, K_G, N_M, N_{GM}, K_{GM}, \mathit{PoP}), \_)\\ &\rightarrow \quad \diamondminus\textsf{gcks\_losepairwisekey}(\mathit{GCKS}, (), (M, K_{GM}), \_)\\ &\quad\ \vee \diamondminus\textsf{gcks\_sendpullkey}(\mathit{GCKS}, M, (K_G, \_, \_, G, K_{GM}, \mathit{PoP}), \_)\end{aligned}$$

In the case of the member M accepting the key K_G for group G as the result of receiving a push datagram, we again require that the GCKS has sent K_G for use in G in a push datagram protected by key K'_G:

$$\begin{aligned}&\textsf{member\_acceptpushkey}(M, \mathit{GCKS}, (G, K_G, K'_G), \_)\\ &\rightarrow \diamondminus\textsf{gcks\_sendpushkey}(\mathit{GCKS}, (), (G, K_G, K'_G), \_)\end{aligned}$$

Observe that this requirement does not make any provision for losing an old group key K'_G since gcks_sendpushkey messages are authenticated with the signature of the GCKS.

4.3.2 Authentication of a Group Member's Request

Although the GCKS has two ways of sending keys, it has only one way of sending a key to a specific group member: via a pull protocol. Thus we need only one requirement here, saying that if the GCKS sent a key to a group member in response to a pull protocol request, then either the pairwise key between the GCKS and group member was lost, or the group member actually sent that request. We need a unique way of identifying the group member's request, and so we will use the nonce the group member sends in the first message of the pull protocol:

$$\begin{aligned}
&\mathsf{gcks\_sendpullkey}(GCKS, M, (K_G, N_M, N_{GM}, G, K_{GM}, PoP), \_)\\
\rightarrow\quad &\diamondminus\mathsf{gcks\_losepairwisekey}(GCKS, (), (M, K_{GM}), \_)\\
&\vee \diamondminus\mathsf{member\_requestkey}(M, GCKS, (G, N_M, K_{GM}), \_)
\end{aligned}$$

4.3.3 Authentication of a Proof of Possession

For Proofs-of-Possession, we want to show that, for either the GCKS or a member, if A accepts a key requiring a Proof-of-Possession from B, then B sent the POP, and B obtained the credentials from the appropriate authority. The act of obtaining credentials is outside the scope of GDOI; however, we leave it in the requirement specification because it is clearly the intent of the POP.

$$\begin{aligned}
&\mathsf{gcks\_sendpullkey}(GCKS, M, (K_G, N_M, N_{GM}, G, K_{GM}, \mathsf{true}), \_)\\
&\rightarrow\diamondminus\ (\quad \mathsf{member\_sendpop}(M, GCKS, (G, N_M, N_{GM}, M), \_)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{member\_acceptpullkey}(M, GCKS, (K_G, N_M, N_{GM}, G, K_{GM}, \mathsf{true}), \_)\\
&\rightarrow\diamondminus\ (\quad \mathsf{gcks\_sendpop}(GCKS, M(G, N_M, N_{GM}), \_)
\end{aligned}$$

4.4 Freshness Requirements

For GDOI, we can identify two types of freshness. One, we call *recency freshness*. This is the requirement that, if a principal receives a piece of information, such as a key, then it must have been current at some specified point in time according to the principal's local clock, for example when the principal requested it. The other, we call *sequential freshness*. This is the requirement that, if a principal accepts a key K_G, then it could not have previously accepted a key that became current after K_G.

Again, Freshness requirements for GDOI do not differ greatly from those that have been defined for two-party protocols. Indeed, recency and sequential freshness correspond closely to the notions of "freshness" and "virginity", respectively, defined in [21]. The main difference is that, without some notion of globally reliable timestamps, only sequential freshness is possible for the push message, since it does not allow for a handshake.

4.4.1 Recency Freshness for Pull Protocol

$$\begin{array}{l}
\mathsf{member\_acceptpullkey}(M, GCKS, (G, N_M, N_{GM}, K'_G, K_{GM}, PoP), N) \\
\rightarrow \quad \diamondminus \mathsf{gcks\_losepairwisekey}(GCKS, (), (M, K_{GM}), \_) \\
\quad \vee \neg(\diamondminus \; (\quad \mathsf{member\_requestkey}(M, GCKS, (G, N_M, K_{GM}), N) \\
\qquad\qquad \wedge \diamondminus \mathsf{gcks\_sendpushkey}(GCKS, (), (G, K_G, K'_G), \_)))
\end{array}$$

Note that the definition of recency freshness is one of the few places we make use of round numbers, since the member requests and accepts the key in the same round. Note also that the GCKS's act of sending a key K_G protected by K'_G using the push protocol results in the expiration of K'_G.

4.4.2 Sequential Freshness for Pull Protocol

$$\begin{array}{l}
\mathsf{member\_acceptpullkey}(M, GCKS, (G, N_M, N_{GM}, K_G, K_{GM}, PoP), \_) \\
\rightarrow \quad \diamondminus \mathsf{gcks\_losepairwisekey}(GCKS, (), (M, K_{GM}), \_) \\
\quad \vee \neg(\diamondminus \; (\quad \mathsf{member\_acceptkey}(M, GCKS, (G, K'_G), \_) \\
\qquad\qquad \wedge \diamondminus \; (\quad \mathsf{gcks\_createkey}(GCKS, (), (G, K'_G), \_) \\
\qquad\qquad\qquad \wedge \diamondminus \mathsf{gcks\_createkey}(GCKS, (), (G, K_G), \_)))
\end{array}$$

Recall from Section 4.2.2 that a group key is sent (and therefore made current) immediately after it is created by the GCKS.

4.4.3 Sequential Freshness for Push Protocol

$$\begin{array}{l}
\mathsf{member\_acceptpushkey}(M, GCKS, (G, K_G, K'_G), \_) \\
\rightarrow \neg(\diamondminus \; (\quad \mathsf{member\_acceptkey}(M, GCKS, (G, K''_G), \_) \\
\qquad\qquad \wedge \diamondminus \; (\quad \mathsf{gcks\_createkey}(GCKS, (), (G, K''_G), \_) \\
\qquad\qquad\qquad \wedge \diamondminus \mathsf{gcks\_createkey}(GCKS, (), (G, K_G), \_)))
\end{array}$$

4.4.4 Freshness of a Member's Key Request

We now consider a freshness requirement from the GCKS's point of view. When the GCKS responds to a member's request with a key, it must be sure that this is a new request, not a replay of some old request. Since a member's request contains a nonce which is intended to be unique, we make this into a requirement that a GCKS should not have previously distributed a key to that member using that nonce.

Note that this freshness requirement can only be guaranteed for an honest member, since there is nothing preventing a dishonest member from replaying an old request and then participating in the protocol to obtain a key. Since honest members are the only ones we are interested in protecting anyway, this is not a problem for us. However, we need a way of distinguishing between honest and dishonest members. We do this by borrowing a trick from the NRL Protocol Analyzer specification language, and referring to principals as $\mathsf{member}(M, H)$ where H is a variable that can be instantiated to honest or $\mathsf{dishonest}$. At this point we are only interest in $\mathsf{member}(M, \mathsf{honest})$:

$$\begin{array}{l} \mathsf{gcks\_sendpullkey}(GCKS, M_{\mathsf{h}}, (K_G, N_M, N_{GM}, G, K_{GM}, PoP), \_) \\ \rightarrow\ \diamondminus\mathsf{gcks\_losepairwisekey}(GCKS, (), (M_{\mathsf{h}}, K_{GM}), \_) \\ \quad \vee\neg\diamondminus\mathsf{gcks\_sendpullkey}(GCKS, M_{\mathsf{h}}, (K'_G, N_M, N'_{GM}, G, K_{GM}, PoP'), \_) \end{array}$$

where $M_{\mathsf{h}} = \mathsf{member}(M, \mathsf{honest})$.

4.4.5 Freshness of Proof of Possession

Freshness requirements for Proof-of-Possession are more similar to two-party freshness requirements than some of the others we have visited. Since POPs are computed on nonces supplied by sender and receiver we require that, if a principal accepts a POP for two nonces, then it should not have accepted it previously. Since the POP is computed on the sender's and receiver's nonces, this can be enforced by requiring that the GCKS does not engage in a sendpullkey event based on the same nonces twice, and that a member does not engage in an $\mathsf{member\_acceptpullkey}$ event based on the same nonces twice. It is not necessary for the GCKS (resp. member) to check prior use of nonces to satisfy this requirement provided that it generates and sends a fresh nonce for each sendpullkey message it sends. Note that the GCKS's freshness requirement is similar, but somewhat stronger than, the requirement for freshness of a member's key request; it is not dependent on the pairwise key being uncompromised.

$$\begin{array}{l} \mathsf{gcks\_sendpullkey}(GCKS, M_{\mathsf{h}}, (K_G, N_M, N_{GM}, G, K_{GM}, PoP), \_) \\ \rightarrow\neg\diamondminus\mathsf{gcks\_sendpullkey}(GCKS, M_{\mathsf{h}}, (K'_G, N_M, N_{GM}, G', K'_{GM}, PoP'), \_) \end{array}$$

$$\begin{array}{l} \mathsf{member\_acceptpullkey}(M, GCKS, (K_G, N_M, N_{GM}, G, K_{GM}, PoP), \_) \\ \rightarrow\neg\diamondminus\mathsf{member\_acceptpullkey}(M, GCKS, (K'_G, N_M, N_{GM}, G', K'_{GM}, PoP), \_) \end{array}$$

where again $M_{\mathsf{h}} = \mathsf{member}(M, \mathsf{honest})$.

4.5 Secrecy Requirements

GDOI has one basic secrecy requirement, that keys should only be learned by members of the group. However, we may want to put other conditions on this requirement. For example, we may require that new members should not have access to old keys (*backward access control*), and that expelled members will not have access to any keys generated after they were expelled (*forward access control*). GDOI also allows for an option that provides a degree of protection against compromise of pairwise keys; it allows for the optional use of Diffie-Hellman to assure *perfect forward secrecy*: if a pairwise key is stolen, then the intruder should only be able to learn key encryption keys distributed after the event.

As we can see, the different secrecy requirements are not quite orthogonal, and they can interact with each other in different ways. For example, one would not want to waste time with perfect forward secrecy if one did not also have backwards access control. In general, it is assumed that it is more likely that a dishonest member will join the group than that a pairwise key shared between only two principals will be stolen. So it makes little sense to use perfect forward secrecy to protect old keys, if they could be compromised by having a group key distributed to a dishonest principal. Likewise, requirements such as forward and backward access control should not only govern the effects of the distribution of keys, but other events such as the stealing of keys. For example, if members should no longer have access to new keys after leaving the group, then an intruder's stealing a key should not give it access to subsequent keys either.

Our solution to this problem is to define a number of conditions describing sequences of events that define the situation under which an intruder might learn a key. These conditions can then be mixed and matched to put together the appropriate requirement. We can then use the NPATRL logic to reduce the requirements to normal form, when necessary.

In the remainder of this section, we describe the various sequences. These include five "base cases" that describe some simple sequences of events that could lead to key compromise, as well as two recursively defined cases that describe forward access control without backward access control, and vice versa. We also give several examples showing how the various cases can be combined to produce different types of requirements.

4.5.1 The Base Cases

The five base cases are as follows:

$\mathsf{BC}_{\mathsf{lose}}(K_G, G)$**:**

$$\mathsf{gcks\_losegroupkey}(\mathit{GCKS}, (), (G, K_G), \_)$$

This describes the a group key-encryption key being stolen.

$\mathsf{BC}_{\mathsf{dishonest\_mempush}}(K_G, G)$**:**

$$\begin{aligned} &\diamondminus(\ \ \mathsf{gcks\_sendpushkey}(\mathit{GCKS}, (), (G, K_G, K'_G), N) \\ &\quad \wedge \diamondminus \mathsf{gcks\_sendpullkey}(\mathit{GCKS}, M_{\mathsf{d}}, (G, \_, N_{GM}, K''_G, K_{GM}, \mathit{PoP}), \_)) \\ \wedge \neg &\diamondminus(\ \ \mathsf{gcks\_sendpushkey}(\mathit{GCKS}, (), (G, K_G, K'_G), N) \\ &\quad\quad \wedge \diamondminus \mathsf{gcks\_cancel}(\mathit{GCKS}, M_{\mathsf{d}}, (G, N_{GM}), \_)) \end{aligned}$$

where $M_{\mathsf{d}} = \mathsf{member}(M, \mathsf{dishonest})$. This describes a group key being distributed while a dishonest member is in the group. Note that it is in two parts. The first says that the dishonest member has joined the group; the second says that the member has not left it yet. In order to take care of the possibility of multiple joinings and leavings, we give both join and leave the same index N_{GM}, which uniquely identifies M's joining the group.

$\mathsf{BC}_{\mathsf{dishonest\_mempull}}(K_G, G)$**:**

$$\diamondminus \mathsf{gcks\_sendpullkey}(\mathit{GCKS}, M_{\mathsf{d}}, (G, \_, N_{GM}, K_G, K_{GM}, \mathit{PoP}), \_)$$

for $M_{\mathsf{d}} = \mathsf{member}(M, \mathsf{dishonest})$. This describes a group key K_G being distributed to a dishonest member via a pull protocol, that is, the dishonest member is being admitted to the group.

$\mathsf{BC}_{\mathsf{pairwiselose}}(K_G, G)$:

$$\begin{aligned} &\diamondminus \mathsf{gcks\_losepairwisekey}(GCKS, (), (M, K_{GM}), \_) \\ \wedge\, &\diamondminus \mathsf{gcks\_sendpullkey}(GCKS, M, (G, \_, \_, K_G, K_{GM}, PoP), \_) \end{aligned}$$

This describes the result of a pairwise key being lost and a key being sent using that pairwise key.

$\mathsf{BC}_{\mathsf{prev\_pairwiselose}}(K_G, G)$:

$$\begin{aligned} \diamondminus\, (\quad &\mathsf{gcks\_sendpullkey}(GCKS, M, (G, \_, \_, K_G, K_{GM}, PoP), \_) \\ \wedge\, &\diamondminus \mathsf{gcks\_losepairwisekey}(GCKS, (), (M, K_{GM}), \_)) \end{aligned}$$

This describes a pairwise key being lost and a key being sent using that pairwise key *after* the pairwise key is lost.

4.5.2 The Recursive Cases

There are two additional cases that we call recursive as they are defined on the basis of the learn event. The first describes an intruder learning an old key after a later key has become current. The second describes the intruder learning a key before another key expires. We call these two cases "backward inference" and "forward inference."

$\mathsf{BI}(K'_G, G)$:

$$\begin{aligned} &\diamondminus \mathsf{learn}(P, (), (K'_G, G), \_) \\ \wedge\, &\diamondminus (\quad \mathsf{gcks\_createkey}(GCKS, (), (G, K_G), \_) \\ &\quad \wedge\, \diamondminus \mathsf{gcks\_createkey}(GCKS, (), (G, K'_G), \_)) \end{aligned}$$

Note that when a new key is sent, the old key expires. And, we assume any (non-initial) key is sent in a push-key message as soon as it is created. Thus BI for K'_G describes an intruder learning a key K_G that became current after a key K'_G was current.

$\mathsf{FI}(K_G, G)$:

$$\begin{aligned} &\diamondminus \mathsf{learn}(P, (), (K''_G, G), \_) \\ \wedge\, &\diamondminus (\quad \mathsf{gcks\_sendpushkey}(GCKS, (), (G, K_G, K'_G), \_) \\ &\quad \wedge\, \diamondminus \mathsf{gcks\_sendpushkey}(GCKS, (), (G, K''_G, K'''_G), \_)) \end{aligned}$$

FI describes an intruder learning a key K''_G that expired before a later key K_G was generated. We explicitly write K_G and K'''_G for the additional keys (instead of "_") for readability.

Backward Inference will be used to specify forward access control without backward access control: If an intruder learns a key K_G, then $\mathsf{BI}(K_G, G)$ will be listed among the set of possible paths to that event, but not $\mathsf{FI}(K_G, G)$, that is, the intruder may have learned K_G as a result of learning a key K'_G that expired previously to K_G, but not a key K^*_G that was generated after K_G expired. Similarly, Forward Inference will be used to specify backward access control without forward access control: if an intruder learns a key K_G, then $\mathsf{FI}(K_G, G)$ will be listed among the set of of possible paths to that event, but not $\mathsf{BI}(K_G, G)$.

We note that there appear to be some major changes from the original, informal, definition of forward and backward access control. The original definition put the requirement on the knowledge of any group member, not on the intruder. Also, the original requirement discussed a member learning a key as a result of joining the group, while we simply consider the results of the intruder learning a key without specifying how it was learned.

Our rationale for changing the focus from member to intruder can be expressed in two steps. In the NRL Protocol Analyzer model, we assume that dishonest group members can do everything honest group members do and more, since honest members can only obey the rules of the protocol. Thus any conditions on a dishonest member's learning a key should also hold for an honest member. Secondly, we assume that all dishonest members share information with the intruder, so that any conditions on the intruder's learning a key would imply the same condition for a dishonest member learning that information.

4.5.3 Sample Requirements

In this section, we show how the various "cases" can be combined into requirements.

Weak Secrecy

The weakest form of secrecy requirement simply requires that the protocol should protect against key compromise given the most benign assumptions possible: that is, that neither pairwise or key encryption keys have been lost, and no dishonest members have ever joined the group. This can be described in terms of three separate conditions:

$$\begin{aligned} & \mathsf{learn}(P, (), (K_G, G), \_) \\ \rightarrow\ & \mathsf{BC}_{\mathsf{lose}}(K'_G, G) \ \vee\ \mathsf{BC}_{\mathsf{dishonest\_mempull}}(K'_G, G) \ \vee\ \mathsf{BC}_{\mathsf{pairwiselose}}(K'_G, G) \end{aligned}$$

In other words, the intruder should not learn a key K_G for G unless some group key has previously been lost, a dishonest member joined the group at some time, or a pairwise key that was used to distribute a group key was stolen, either before or after being used.

Strong Secrecy

We can also use the base cases to formulate the strongest type of secrecy possible. In strong secrecy, the intruder learns a key K_G only if K_G is lost, a dishonest member received K_G, either when it joined the group or while it was a member of the group, or if a pairwise key was stolen and used to distribute K_G. We may or may not wish to require perfect forward secrecy.

Here, for example, is strong secrecy with perfect forward secrecy:

$$\begin{aligned} & \mathsf{learn}(P, (), (K_G, G), \_) \\ \rightarrow \quad & \mathsf{BC}_{\mathsf{lose}}(K_G, G) \ \vee \ \mathsf{BC}_{\mathsf{dishonest\_mempush}}(K_G, G) \ \vee \ \mathsf{BC}_{\mathsf{dishonest\_mempull}}(K_G, G) \\ & \vee \mathsf{BC}_{\mathsf{prev\_pairwiselose}}(K_G, G) \end{aligned}$$

Forward Access Control

Forward access control (without backward access control) can be thought of as strong secrecy together with added condition of backward inference: An intruder can learn a key, not only if the key was lost, distributed to a dishonest member, or distributed using a lost pairwise key, but if the key became current before the intruder learned a later key, *e.g.*, because a dishonest member joined the group. We do not include perfect forward secrecy, since protecting against old keys being compromised as a result of a stolen pairwise key makes no sense if the keys could be learned as a result of a dishonest member joining the group at any point:

$$\begin{aligned} & \mathsf{learn}(P, (), (K_G, G), \_) \\ \rightarrow \quad & \mathsf{BC}_{\mathsf{lose}}(K_G, G) \ \vee \ \mathsf{BC}_{\mathsf{dishonest\_mempush}}(K_G, G) \ \vee \ \mathsf{BC}_{\mathsf{dishonest\_mempull}}(K_G, G) \\ & \vee \mathsf{BC}_{\mathsf{pairwiselose}}(K_G, G) \ \vee \ \mathsf{BI}(K_G, G) \end{aligned}$$

Backward Access Control

Backward access control (without forward access control) can be specified similarly to forward access control, except that we replace backward with forward inference. We can require perfect forward secrecy or not. Here, for example, is backward access control without forward access control but with perfect forward secrecy:

$$\begin{aligned} & \mathsf{learn}(P, (), (K_G, G), \_) \\ \rightarrow \quad & \mathsf{BC}_{\mathsf{lose}}(K_G, G) \ \vee \ \mathsf{BC}_{\mathsf{dishonest\_mempush}}(K_G, G) \ \vee \ \mathsf{BC}_{\mathsf{dishonest\_mempull}}(K_G, G) \\ & \vee \mathsf{BC}_{\mathsf{prev\_pairwiselose}}(K_G, G) \ \vee \ \mathsf{FI}(K_G, G) \end{aligned}$$

If we wanted to omit the perfect forward secrecy requirement we would substitute $\mathsf{BC}_{\mathsf{pairwiselose}}(K_G, G)$ for $\mathsf{BC}_{\mathsf{prev\_pairwiselose}}(K_G, G)$.

In appendix A we show how to put the requirements for Forward and Backward Access Control into normal form.

5 GDOI Specification and Analysis

5.1 Specification

In our specification of GDOI, we used a feature of NPA that we had not used before, the notion of master protocols and subprotocols. A subprotocol executes using information generated by a master protocol. An example would be Phase 1 and Phase 2 in the Internet Key Exchange Protocol (IKE) [9]. Phase 1 generates keying material that is used to protect the message exchanges in the Phase 2 protocol. Indeed, we had originally intended to use subprotocols in our specifications separately.[4]

We used subprotocols as follows (see also the diagram below). From the point of the view of the GCKS, the master protocol is the one in which it sets up a group. There are two subprotocols: the handshake protocol in which it distributes the current group key to a new member requesting to join the group, and the formation and sending of the key update message. For the group member, the master protocol is the one by which it joins the group, and the subprotocol is the one in which it receives and verifies the key update message.

$$\begin{tikzcd}
 & \underline{\text{GCKS}} & & \underline{\text{Member}} \\
 & \boxed{\text{create\_group}} \arrow[dl] \arrow[dr] & & \boxed{\text{pull\_key}} \arrow[d] \\
\boxed{\text{pull\_key}} & & \boxed{\text{push\_key}} & \boxed{\text{push\_key}}
\end{tikzcd}$$

We modeled GCKS's who could set up an arbitrary number of groups, although each group would be managed by at most one GCKS. For any of those groups, it can receive and process a request from a member at any time. It can also distribute keys to the group as a whole at any time. The only restriction is that it will only process one handshake from a member at a time per group. This is a restriction on subprotocols that was inherited from their original intent that they be used for to describe distribution of pairwise keys between two principals using a master key that had been distributed between the two principals using the parent protocol. In such a case it is reasonable to expect that two session keys would not be generated at the same time for the same two principals. As a result of our experience with GDOI, we plan to make it possible to do without this restriction in subsequent versions of NPA.

We also modeled group members who could request membership in a group at any time, and who could belong to more than one group. A group member could also receive and process a new group key at any time after joining the group.

One thing we did not specify was a member leaving the group. This was

[4]We are currently using the subprotocol-style specification to analyze IKE version 2, however.

because NPA does not currently have the ability to reason about the key hierarchies that are used to revoke access to keys. As a result, in this analysis, we only verified one of the secrecy requirements to NPA: weak secrecy. We are, however, currently investigating ways of expressing key hierarchies as abstract data types whose properties are described by rewrite rules, with the idea of developing equational unification algorithms for them in a way similar to what we do now in the NPA with encryption and digital signatures. This will allow us to integrate key hierarchies into the NPA.

5.2 Analysis

Our formal analysis actually consisted of three parts: specification of the protocol, specification of its requirements, and verification that the protocol satisfied its requirements. We were able to identify problems at all three stages of our analysis, which we communicated to the authors of GDOI. All of these problems were fixed in subsequent versions. Since our analysis proceeded from protocol specification to requirements specification to formal verification, we will present our results in that order.

5.2.1 Problems Found During Protocol Specification

Most of the problems we found during the protocol specification were minor inconsistencies and ambiguities. For example, the Diffie-Hellman keying material generated by the responder and sender was denoted by `KE` in each case, instead of `KE_I` and `KE_R`, allowing for possible confusion. Also, it was unclear in the `SIG` payload definition of the groupkey-push message what fields the signature was taken over, and whether signing or encryption was done first. All of these were quite easy to correct once they were pointed out to the authors. More seriously, a flaw was discovered in the way sequence numbers were handled. It required some more intensive thought and redesign before it could be fixed.

Specifically, the groupkey-push message contains a sequence number `SEQ` that is intended to guarantee freshness. A group member would find out the current value of the sequence number when it joined a group. From then on, the sequence number would be incremented every time a new key-encryption-key `SA` was created (in the pull-key exchange), and it would be included in that `SA`. The problem was that not every group-key push message contained a key-encryption key. Some might contain only traffic-encryption keys. When this was pointed out, it was realized that the key-encryption `SA` was not the appropriate place to put the sequence number, and the specification was changed so that sequence number payloads have their own separate place in the groupkey-push message, instead of appearing inside the key-encryption-key `SA`'s, and sequence numbers are updated every time a new groupkey-push message is created.

While this was a minor change conceptually, from the point of view of implementation it was a major overhaul. Catching this problem earlier allowed the designers of the protocol to avoid the expense that would have come with a major redesign and reimplementation at a later stage, or from cumbersome workarounds.

5.2.2 Problem Found During Requirements Specification

We found two problems during the requirements specification. Both of these emerged when we started asking ourselves under what conditions a requirement should hold.

The first problem concerned the Proof-of-Possession signature. In the original protocol, a principal proved possession of its private key by signing the nonce that it had created. This worked fine as long as the pairwise key that protected the conversation between GCKS and member was not compromised. However, if the key was compromised, it would be possible for an intruder to insert a fake Proof-of-Possession signature into the conversation, for example by replaying an old one.

After the problem was pointed out, there was some discussion as to whether it was necessary to protect against such an eventuality, since if pairwise keys were compromised, the group keys would be compromised as well, causing other problems besides facilitating attacks on Proof-of-Possession. However, there are advantages to making Proof-of-Possession secure against pairwise key compromise, since it makes it possible to prevent intruders from impersonating honest principals even when such compromise has taken place. At the end, it was decided to modify the protocol so that each principal signed a hash of its own and another's nonce. As we will see in the next section, this led to some other, more subtle problems.

The other problem emerged as we were considering the different types of freshness. We found that the groupkey-pull protocol did not satisfy sequential freshness if a member was allowed to send out a second request to join a group before it had received an answer to the first request. In that case, if the responses from the GCKS were received out of order, and the group-key was changed between those two responses, the member might accept the newest key first, and then replace it with the old key when it received the earlier response. Again, there were several possible solutions to this, one of which was not to allow a principal to repeat a request to join a group until the first one had timed out. At the end, though, it was decided that it would allow more flexibility to require the group member to compare the sequence number it received in a final message from a GCKS with its current sequence number value, and only to accept the new key if the incoming sequence number was greater than the current one.

5.2.3 Problems Found During Formal Analysis

Once we had the protocol and requirements specification written, we were ready to use NPA to ask questions. We found fewer problems at this stage, but they tended to be more subtle, and, in at least one case, we uncovered a problem that we believe would have been extremely unlikely to have been discovered without mechanical aid.

We begin with the more obvious problem first. This was a problem found in one of our own requirements specifications, instead of in the protocol itself. In our specification of Proof-of-Possession authentication as presented in Section 4 we found that we had given too restrictive a requirement. For example, for the GCKS's Proof-of-Possession, we had said:

$$\begin{aligned}&\mathsf{member\_acceptpullkey}(M, GCKS, (K_G, N_M, N_{GM}, G, \_, \mathsf{true}), \_)\\ &\quad\rightarrow \quad \Diamond\mathsf{gcks\_sendpop}(GCKS, M, (G, N_M, N_{GM}), \_)\end{aligned}$$

This meant that the Proof-of-Possession was assumed not only to authenticate the GCKS's key, but also to verify that member and GCKS were talking about the same group. This was true only if each group was associated with a unique signature key, since the proof-of-possession proves only possession of that key. As a matter of fact, unless the GCKS expected and received a proof-of-possession from the group member, it would not even necessarily know that it was talking to that member. Our assumption that the Proof-of-Possession should authenticate the group as well as possession of the key was our own mistake, however, since the GDOI specification only says that the Proof-of-Possession verifies possession of the key. Thus, we changed the requirement to something more realistic: the only data agreed upon was the key and the nonces it was used to sign.

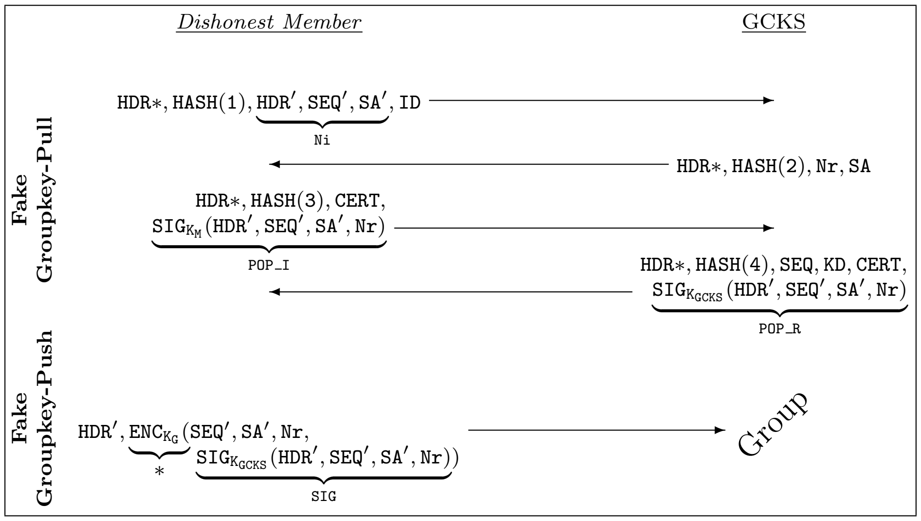
Thus, our requirement would now look like this:

$$\begin{aligned}&\mathsf{member\_acceptpullkey}(M, GCKS, (K_G, N_M, N_{GM}, G, \_, \mathsf{true}), \_)\\ &\quad\rightarrow \quad \Diamond\mathsf{gcks\_sendpop}(GCKS, M', (G', N_M, N_{GM}), \_)\end{aligned}$$

The other set of problems we found were more subtle, and did require a rewriting of the protocol. The decision to require principals to sign nonces generated by others in the Proof-of-Possession opens up a vulnerability to oracle attacks. This vulnerability is usually countered by including information generated by the signer, as the GDOI authors did by including the signer's nonce. However in this particular case this precaution did not always prevent such attacks.

We found two oracle attacks. In both, we assume that the same private key is used by the GCKS to sign Proofs-of-Possession and groupkey-push messages. In the first of these, we assume a dishonest group member who wants to pass off a signed Proof-of-Possession from the GCKS as a groupkey-push message. To do this, she creates a fake plaintext groupkey-push message GPM, which is missing only the last (random) part of the Key Download Payload. She then initiates an instance of the groupkey-pull protocol with the GCKS, but in place of her nonce, she sends GPM. The GCKS responds by appending its nonce N_B and signing it, to create a signed (GPM, N_B). If N_B is of the right size, this will look like a signed groupkey-push message. The group member can then encrypt it with the key encryption key (which she will know, being a group member) and send it out to the entire group. This attack is summarized in Figure 2.

The second attack, detailed in Figure 3, is similar to the first, although it requires a few more assumptions to make it work. It relies upon the fact that GDOI is built on top of the ISAKMP protocol [13], which requires that any



Dishonest Member
GCKS
Fake Groupkey-Pull
HDR∗, HASH(1), HDR′, SEQ′, SA′, ID
Ni
HDR∗, HASH(2), Nr, SA
HDR∗, HASH(3), CERT, SIG_KM(HDR′, SEQ′, SA′, Nr)
POP_I
HDR∗, HASH(4), SEQ, KD, CERT, SIG_KGCKS(HDR′, SEQ′, SA′, Nr)
POP_R
Fake Groupkey-Push
HDR′, ENC_KG(SEQ′, SA′, Nr, SIG_KGCKS(HDR′, SEQ′, SA′, Nr))
∗
SIG
Group

Figure 2: First Oracle Attack on GDOI

message begin with a random number, usually contributed by both sender and receiver of the message. Since the groupkey-push message has one sender and many receivers, the random number beginning it is generated by the sender only, that is, by the GCKS. We assume that there is a group member A who can also act as a GCKS, and that the pairwise key between A and another GCKS, B, is stolen, but that B's private key is still secure. Suppose that A, acting as a group member, initiates a groupkey-pull protocol with B. Since their pairwise key is stolen, it is possible for an intruder P to insert a fake nonce for B's nonce N_B. The nonce he inserts is a fake groupkey-push message GPM' that it is complete except for a prefix of the header consisting of all or part of the random number beginning the header. A then signs (N_A, GPM'), which, if N_A is of the right length, will look like the signed part of a groupkey-push message. The intruder can then find out the key encryption key from the completed groupkey-pull protocol and use it to encrypt the resulting (N_A, GPM') to create a convincing fake groupkey-push message.

The NPA found the second attack. Indeed, the attack is subtle enough that we do not know when it would have been found if some sort of automated assistance had not been used. However, the NPA could not have found the first attack, since to do so would have required an ability on the part of the NPA to model associativity of concatenation, which would have caused problems with the termination of the narrowing algorithm. However, once the NPA had found the second attack, we were able to mimic the reasoning used by it to find the first one.

Both attacks also required further analysis before it could be determined whether or not they were realistic. The NPA does not model any constraints on

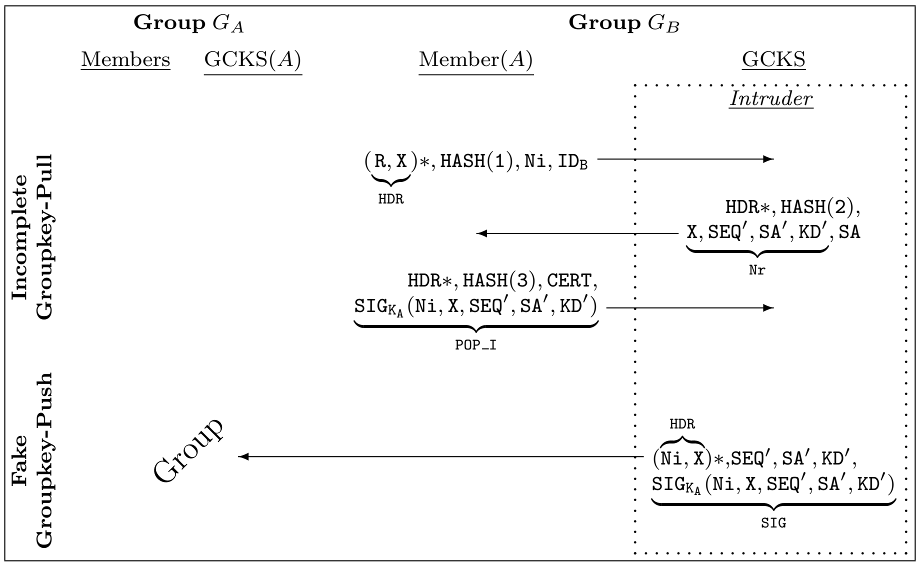

Group G_A
Group G_B
Members
GCKS(A)
Member(A)
GCKS
Intruder
Incomplete Groupkey-Pull
$\underbrace{(\mathtt{R},\mathtt{X})}_{\mathtt{HDR}}*,\mathtt{HASH(1)},\mathtt{Ni},\mathtt{ID_B}$
$\mathtt{HDR}*,\mathtt{HASH(2)},\underbrace{\mathtt{X},\mathtt{SEQ'},\mathtt{SA'},\mathtt{KD'}}_{\mathtt{Nr}},\mathtt{SA}$
$\mathtt{HDR}*,\mathtt{HASH(3)},\mathtt{CERT},\underbrace{\mathtt{SIG_{K_A}}(\mathtt{Ni},\mathtt{X},\mathtt{SEQ'},\mathtt{SA'},\mathtt{KD'})}_{\mathtt{POP\_I}}$
Fake Groupkey-Push
Group
$\overbrace{(\mathtt{Ni},\mathtt{X})}^{\mathtt{HDR}}*,\mathtt{SEQ'},\mathtt{SA'},\mathtt{KD'},\underbrace{\mathtt{SIG_{K_A}}(\mathtt{Ni},\mathtt{X},\mathtt{SEQ'},\mathtt{SA'},\mathtt{KD'})}_{\mathtt{SIG}}$

Figure 3: Second Oracle Attack on GDOI

the lengths of fields, and thus it is not possible to tell from its output whether or not the attacks it finds put impossible conditions on field lengths. For example, the first attack requires that the GCKS accept a nonce from a group member that is nearly as long as a groupkey-push message. Is this possible or not? For GDOI, this turned out to be possible, since GDOI puts no restrictions on lengths of nonces used in the groupkey-push protocol.

Again, there were several possible responses to this problem. One would be to require that different signature keys be used for Proof-of-Possession and groupkey-push messages. This would be difficult to enforce, and would put an extra burden on users of the protocol. Another would be to put constraints on lengths of the nonces and other fields used that would make such attacks impossible. This would also be difficult to enforce, however, and might conflict with future security requirements on lengths of nonces and keys. The third, and the one that was finally chosen, was to require that, instead of taking the signatures over a hash a message (such as the two nonces, or the groupkey-push message), to require that the signature be taken over a hash of the concatenation of a field indicating the type of the message (member Proof-of-Possession, GCKS Proof-of-Possession, or groupkey-push) with the message itself. This was a more substantial change than the other two possibilities, but had the advantage of being more robust and less restrictive.

Our discovery of these flaws also led us to consider developing techniques for detecting problems of these types. Attacks that rely upon confusing data of one type with data of another, known as “type confusion” attacks, can usually be defeated by introducing a tagging scheme in which each data item has a field attached to it indicating its type, as was done in this case for GDOI (see [10] for a further discussion of this and how it can simplify an automated protocol analysis), but problems can arise when a protocol is interacting with another protocol that does not use a tagging scheme, or tags data in a different way. Such an interaction contributed to the second type confusion attack on GDOI, since it was built on top of ISAKMP, which did not tag the random number at the beginning of the header. Thus, we have recently been investigating techniques for verifying that protocols are free of type confusion attacks. Our preliminary results may be found in [18].

6 Conclusions

We have described an formal analysis of the Group Domain Of Interpretation protocol. In the course of this analysis, we not only added value to GDOI, and demonstrated how a formal analysis could be beneficial when performed in close cooperation with the protocol designers, but we were able to develop a set of formal requirements that could be possibly applied to other group protocols involving a central key distributor, and led us to a better understanding of the nature of requirements for open-ended cryptographic protocols in general. This has motivated us to develop the NPATRL requirements language into a full-scale logic that can be used to reason about and simplify requirements as well as specify them.

We also discovered a number of issues that still remain to be resolved. Some of them were relatively minor, such as changes that need to be made to NPA to better accommodate reasoning about subprotocols. Others are more challenging, such as the research that is needed to help us reason about open-ended data structures such as key hierarchies. But probably the most important problem is how to better integrate the formal analysis with the protocol design. Probably the promising avenue for this is through the requirements specification process. We found specifying requirements very helpful in aiding our understanding of the protocols. Moreover, when we presented problems we found to the GDOI designers, the discussion of what approach to take often hinged on a decision on the protocol requirements. For example, with respect to the problems we found with the first version of Proof-of-Possession, the issue was whether or not Proof-of-Possession should be secure against pairwise key compromise. It would be helpful to have a common language to reason about requirements that was both precise and convenient to use. However, although the NPATRL language allowed us to state requirements precisely and reason about them in a rigorous way, it is, like most temporal languages, difficult to write and read. Recently, we have been investigating graphical representations of NPATRL requirements. The NRL normal form of the requirements can for the most part be naturally represented in terms of fault trees. We found fault tree representations much easier to read than the original NPATRL requirements, and we often used them to help us understand the requirements better. We are currently further investigating fault tree representations of NPATRL requirements and their applicability.

References

[1] M. Baugher, T. Hardjono, H. Harney, and B. Weis. Group domain of interpretation for ISAKMP. Archived at `http://www.watersprings.org/pub/id/draft-irtf-smug-gdoi-01.txt`, January 2001.

[2] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast security: A taxonomy and some efficient constructions. In *Proc. of INFOCOM'99, vol. 2*, pages 708–716, March 1999.

[3] Brian F. Chellas. *Modal Logic: An Introduction.* Cambridge University Press, 1980.

[4] Danny Dolev and Andrew C. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29):198–208, March 1983. Preliminary version in *Proc. 22nd Annual IEEE Symp. Foundations of Computer Science,* 1981, 350–357.

[5] Naganand Doraswamy and Dan Harkins. *IPSEC: The New Security Standard for the Internet, Intranets, and Virtual Private Networks.* Prentice Hall, 1999.

[6] E. A. Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 997–1072. MIT Press, 1990.

[7] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of CAV'01*, pages 53–65, Paris, France, 2001. Springer-Verlag LNCS 2102.

[8] Robert Goldblatt. *Logics of Time and Computation, 2^{nd} edition*, volume 7 of *CSLI Lecture Notes.* CSLI Publications, Stanford, 1992.

[9] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC 2409, IETF, November 1998. Available at `ftp://ftp.isi.edu/in-notes/rfc2409.txt`.

[10] James Heather, Gavin Lowe, and Steve Schneider. How to prevent type flaw attacks on security protocols. In *Proceedings of 13th IEEE Computer Security Foundations Workshop*, pages 255–268. IEEE Computer Society Press, June 2000. A revised version is to appear in the *Journal of Computer Security.*

[11] G.E. Hughes and M.J. Creswell. *A New Introduction to Modal Logic.* Routledge, 1996.

[12] O. Lichtenstein, A Pnueli, and L. Zuck. The glory of the past. In *Proceedings of the Conference on Logics of Programs*, pages 196–218. Springer-Verlag LNCS 193, 1985.

[13] D. Maughan, M. Schertler, M. Schneider, and J. Turner. Internet Security Association and Key Management Protocol (ISAKMP). Request for Comments 2408, Network Working Group, November 1998. Available at http://ietf.org/rfc/rfc2408.txt.

[14] C. Meadows and P. Syverson. A formal specification of requirements for payment transactions in the SET protocol. In R. Hirschfeld, editor, *Financial Cryptography, FC'98*, pages 122–140. Springer-Verlag, LNCS 1465, 1998.

[15] Catherine Meadows. A model of computation for the NRL Protocol Analyzer. In *Proceedings of the 7th Computer Security Foundations Workshop*, pages 84–89. IEEE CS Press, June 1994.

[16] Catherine Meadows. The NRL Protocol Analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, February 1996.

[17] Catherine Meadows. A cost-based framework for analysis of denial of service in networks. *Journal of Computer Security*, 9(1–2):143–164, 2001.

[18] Catherine Meadows. Identifying potential type confusion in authenticated messages. In *Proceedings of Foundations of Computer Security 02*, July 2002.

[19] P. Syverson and C. Meadows. A logical language for specifying cryptographic protocol requirements. In *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy*, pages 165–177. IEEE CS Press, May 1993.

[20] P. Syverson and C. Meadows. Formal requirements for key distribution protocols. In A. De Santis, editor, *Advances in Cryptology — EUROCRYPT '94*, pages 32–331. Springer-Verlag, LNCS 950, 1994.

[21] P. Syverson and C. Meadows. A formal language for cryptographic protocol requirements. *Designs, Codes, and Cryptography*, 7(1 and 2):27–59, January 1996.

A Removing Recursion

In this appendix we show how we use the NPATRL logic to remove recursion from the requirements for forward and backwards access control. Removing recursion is not only desirable from the point of view of the NRL Protocol Analyzer, but also because a recursively defined condition may cause an infinite regression in other model checkers and theorem provers.

We present only the proof for backward access control; that for forward access control is similar. In the rest of this section, we will rely on the abbreviation

$$\begin{aligned}\mathsf{BC}(K,G) = \;& \mathsf{BC}_{\mathsf{lose}}(K,G) && \vee\, \mathsf{BC}_{\mathsf{dishonest\_mempush}}(K,G)\\ & \vee\, \mathsf{BC}_{\mathsf{dishonest\_mempull}}(K,G) && \vee\, \mathsf{BC}_{\mathsf{pairwiselose}}(K,G).\end{aligned}$$

where K is a key and G is a group.

Lemma 1
For any group G and group key K_G, the backward access control condition $\mathsf{BAC}(K_G,G) =$

$$\mathsf{learn}(P,(),(K_G,G),\_) \;\;\rightarrow\;\; \mathsf{BC}(K_G,G) \;\vee\; \mathsf{FI}(K_G,G)$$

is equivalent to the conditions $\mathsf{NRFAC}(K_G,G) =$

$$\mathsf{learn}(P,(),(K_G,G),\_) \;\;\rightarrow\;\; \mathsf{BC}(K_G,G) \;\vee\; \mathsf{NRFI}(K_G,G)$$

where $\mathsf{NRFI}(K_G,G) =$

$$\begin{aligned}&\diamondminus \mathsf{BC}(K''_G,G)\\ \wedge\, &\diamondminus(\;\; \mathsf{gcks\_sendpushkey}(GCKS,(),(G,K_G,K'_G),\_)\\ &\quad\wedge\, \diamondminus \mathsf{gcks\_sendpushkey}(GCKS,(),(G,K''_G,K'''_G),\_)).\end{aligned}$$

Proof:
In order to prove the implication in the forward direction, let us assume that the premise, $\mathsf{learn}(P,(),(K,G),\_) \rightarrow \mathsf{BC}(K,G) \;\vee\; \mathsf{FI}(K,G)$, holds for every key K and group G. We proceed by induction on the age of K.

In the base case, suppose that K is the first key used by the GCKS for G (i.e. the first key that was made current). Then, since there is no K'' that was distributed before K, neither "$\mathsf{FI}(K,G)$" nor "$\mathsf{NRFI}(K,G)$" holds, and so "$\mathsf{BC}(K,G) \;\vee\; \mathsf{FI}(K,G)$" is trivially equivalent to "$\mathsf{BC}(K,G) \;\vee\; \mathsf{NRFI}(K,G)$".

Suppose now that the result holds up to the n'th key K_n used by the GCKS. Now, by unfolding the hypothesis relative to K_{n+1}, we have

$$\begin{aligned}&\mathsf{learn}(P,(),K_{n+1},G),\_)\\ \rightarrow\quad &\quad \diamondminus \mathsf{BC}(K_{n+1},G)\\ &\vee(\;\; \mathsf{learn}(P,(),(K_k,G),\_)\\ &\quad \wedge\, \mathsf{gcks\_sendpushkey}(GCKS,(),(G,K_{n+1},K'),\_)\\ &\quad \wedge\, \diamondminus \mathsf{gcks\_sendpushkey}(GCKS,(),(G,K_k,K'''),\_)),\end{aligned}$$

for some $k < n{+}1$. By induction hypothesis, we know that $\mathsf{learn}(P,(),(K_k,G),\_)$ $\rightarrow \mathsf{BC}(K_k,G) \;\vee\; \mathsf{NRFI}(K_k,G)$, which we can use in the above formula obtaining

$$\begin{aligned}&\mathsf{learn}(P,(),K_{n+1},G),\_)\\ \rightarrow\quad &\quad \diamondminus \mathsf{BC}(K_{n+1},G)\\ &\vee(\;\; (\mathsf{BC}(K_k,G) \;\vee\; \mathsf{NRFI}(K_k,G))\\ &\quad \wedge\, \mathsf{gcks\_sendpushkey}(GCKS,(),(G,K_{n+1},K'),\_)\\ &\quad \wedge\, \diamondminus \mathsf{gcks\_sendpushkey}(GCKS,(),(G,K_k,K'''),\_)),\end{aligned}$$

Now, by expanding $\mathsf{NRFI}(K_k, G)$, we get:

$$\begin{array}{l}
\mathsf{learn}(P, (), (K_{n+1}, G), \_) \\
\rightarrow \quad \mathsf{BC}(K_n, G) \\
\qquad \vee \diamondminus(\quad \diamondminus(\quad \mathsf{BC}(K_i, G) \\
\qquad\qquad\qquad \wedge\ \mathsf{gcks\_sendpushkey}(GCKS, (), (G, K_k, K^*), \_) \\
\qquad\qquad\qquad \wedge\ \diamondminus\mathsf{gcks\_sendpushkey}(GCKS, (), (G, K_i, K^{***}), \_))) \\
\qquad\qquad \wedge\ (\quad \mathsf{gcks\_sendpushkey}(GCKS, (), (G, K_{n+1}, K'), \_) \\
\qquad\qquad\quad \wedge\ \mathsf{gcks\_sendpushkey}(GCKS, (), (G, K_k, K'''), \_)))).
\end{array}$$

for keys K_i, K^* and K^{***}. Using the tautologies "$\diamondminus(A \wedge B) \rightarrow \diamondminus A \wedge \diamondminus B$", that "$\diamondminus\diamondminus A \rightarrow \diamondminus A$", and that "$(A \wedge \diamondminus B) \wedge (C \wedge D) \rightarrow C \wedge \diamondminus B$", that follow immediately from the NPATRL axioms, this reduces to

$$\begin{array}{l}
\mathsf{learn}(P, (), K_{n+1}, G), \_) \\
\rightarrow \quad \mathsf{BC}(K_{n+1}, G) \\
\qquad \vee \diamondminus(\quad \mathsf{BC}(K_i, G) \\
\qquad\qquad \wedge\ \mathsf{gcks\_sendpushkey}(GCKS, (), (G, K_{n+1}, K'''), \_) \\
\qquad\qquad \wedge\ \diamondminus\mathsf{gcks\_sendpushkey}(GCKS, (), (G, K_i, K^{***}), \_)),
\end{array}$$

i.e., $\mathsf{learn}(P, (), (K_{n+1}, G), \_) \rightarrow \mathsf{BC}(K_{n+1}, G) \vee \mathsf{FI}(K_{n+1}, G)$, which is the result we need.

In the reverse direction, it is sufficient to show that "$\mathsf{NRFI}(K, G)$" implies "$\mathsf{FI}(K, G)$", or, after expanding these formulas and simplifying common terms, that "$(\mathsf{BC}_{\mathsf{lose}}(K'', G) \vee \mathsf{BC}_{\mathsf{dishonest\_mempush}}(K'', G) \vee \mathsf{BC}_{\mathsf{dishonest\_mempull}}(K'', G) \vee \mathsf{BC}_{\mathsf{pairwiselose}}(K'', G)$" implies "$\mathsf{learn}(P, (), (K'', G), \_)$". This last statement holds by virtue of the semantics of the events: all the above cases describe transitions in which the key is released to the intruder, so of course the occurrence of any of them would imply the learn event. □

Lemma 2
For any group G and key K_G, the forward access control condition $\mathsf{FAC}(K_G, G) =$

$$\mathsf{learn}(P, (), (K_G, G), \_) \ \rightarrow\ \mathsf{BC}(K_G, G) \ \vee\ \mathsf{BI}(K_G, G)$$

is equivalent to the conditions $\mathsf{NRBAC}(K_G, G) =$

$$\mathsf{learn}(P, (), (K_G, G), \_) \ \rightarrow\ \mathsf{BC}(K_G, G) \ \vee\ \mathsf{NRBI}(K_G, G)$$

where $\mathsf{NRBI}(K'_G, G) =$

$$\begin{array}{l}
\quad \diamondminus\mathsf{BC}(K''_G, G) \\
\wedge \diamondminus(\quad \mathsf{gcks\_createkey}(GCKS, (), (G, K_G), \_) \\
\qquad \wedge \diamondminus\mathsf{gcks\_createkey}(GCKS, (), (G, K'_G), \_).
\end{array}$$

Proof:
The proof goes as for backward access control, except the base induction case is the most recent key instead of the first key, and the induction is on distance from the most recent key instead of on distance from the earliest key. □