Model-Checking A Secure Group Communication Protocol: A Case Study

Alan J. HU, Rui LI, Xizheng SHI, and Son VUONG
Department of Computer Science, University of British Columbia, Canada

Abstract: With the explosive growth of the Internet and the distributed applications it supports, there is a pressing need for secure group communications – the ability of a group of agents to communicate securely with each other while allowing members to join or leave the group. Prompted by the success of other researchers in applying finite-state model-checking tools to the verification of small security protocols, we decided to attempt a larger security protocol: a recently published protocol for secure group communication. Not surprisingly, creating an ad hoc abstract model suitable for model-checking required cleverness, and state explosion was always a threat. Nevertheless, with minimal effort, the model checking tool discovered two flaws in the protocol, one of which has not been reported previously. We conclude our paper with a discussion of possible fixes to the protocol, as well as suggested verification tool improvements that would have simplified our task.

1. INTRODUCTION

Secure group communication is emerging as an important problem in networking. The objective is to allow messages to be sent securely to a set of authorized users. Such a mechanism underlies many new applications such as teleconferencing or pay-per-view content distribution. Trivially, we can build an n-party secure group communication protocol using $\binom{n}{2}$ normal point-to-point secure links and requiring each message to be sent individually to all other group members. For efficiency reasons, however, we would like to exploit the broadcast or multicast capabilities a network may offer, and we would like to introduce hierarchy to manage very large groups

--- hence, the need for secure group communication protocols. In this paper, we consider a recently published secure group communication protocol [13] that offers substantial performance benefits over previously published protocols (e.g. [10]).

Security protocols are extremely tricky and have long been an important application domain for formal specification and verification techniques. Recently, a number of researchers have reported excellent results applying finite-state model-checking tools developed for hardware and general protocol verification to the verification of simple security protocols (e.g. key exchange, authentication) [8, 9]. The obvious promise of such an approach is a great reduction in labor and expertise required to carry out the verification. The obvious question is whether these automatic tools are sufficiently powerful in practice to handle interesting security protocols. This paper helps to answer that question by describing our experience using a model-checking tool on the above mentioned secure group communication protocol [13], which is more complex than has previously been attempted via model-checking. We will discuss the protocol and how we modeled it, the benefits and limitations of the model-checking approach and the tool we used, and the errors we discovered in the protocol.

2. THE PROTOCOL

In this section, we will briefly sketch the protocol we are studying. We refer interested readers to the original paper for details [13].

The protocol presumes a trusted server that administers the group. Requests to join or leave the group are sent to this server. The server has some mechanism for authenticating new users who request to join the group and for distributing or negotiating a private session key with the new user. These are standard operations and are assumed to work correctly. All cryptographic keys are assumed to be symmetric, private keys.

The basic idea of the protocol is to construct a hierarchy of keys organized in a tree.[1] (See Figure 1.) Each user is associated with a leaf of the tree, and the key at that leaf is the private session key between the server and that user alone. Each internal node of the tree corresponds to a subgroup key that is known by all users descending from that node. At the root is the group key that is shared by all users in the group. Accordingly, we can send a secure message to all the users in any subtree by using the key at the root of

[1] The paper also proposes a star graph, which is just a special case of a tree, and a complete graph, which is discarded as impractical.

that subtree. Each user needs to know all the keys on the path from itself to the root.

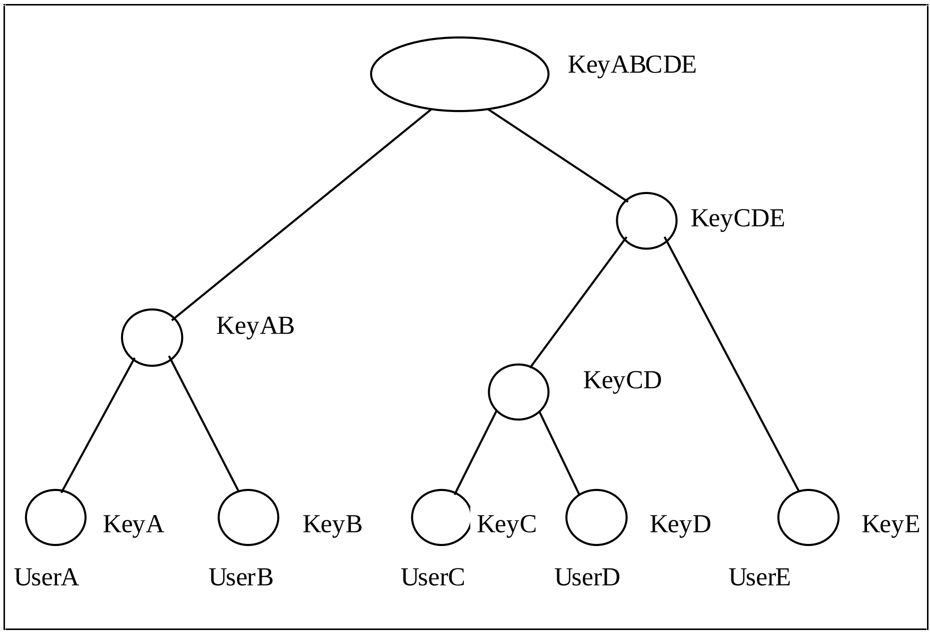

Figure 1. Key Tree: Each user is associated with a leaf in the tree and knows all the keys on the path from itself to the root. A secure message can be sent to all (and only the) users in a subtree by using the key at the root of that subtree.

When a new user wishes to join the group, it sends a join request to the server. The server authenticates the user, finds a position in the tree to attach the new user, and then must rekey all the nodes on the path from the root to the new user to ensure that the new user cannot read old messages from before it joined the group. (For example, in a pay-per-view application, a user should not be able to decode and view saved copies of previously broadcast programs using the key purchased for a more recent program.) Note that the rekeying means that every user must update at least one of its keys whenever a new user joins the group. Similarly, when a user wishes to leave the group, it sends a leave request to the server. Again, the server must rekey all nodes on the path from the root to the leaving user to ensure that it can no longer read any messages from the group.

The paper proposed three different rekeying schemes. For simplicity, we considered only “group-oriented” rekeying, the scheme with the best server-side performance. Intuitively, the scheme handles a join by sending a single message with all the needed keys to the new user and by broadcasting a single message to the whole group that contains all the changed keys, appropriately encrypted so that no user learns any keys that it shouldn't know. Similarly, the scheme handles a leave by broadcasting a single message to the whole group that contains all the changed keys, again with each key encrypted separately so that no user (especially the one that just left) learns new keys that it shouldn't.

More formally, let $s \rightarrow u : m$ denote sending message m from s to u. Generalized, $s \rightarrow \{u_1, \ldots, u_n\}$: m denotes broadcasting message m to all the users $u_1, \ldots, u_n$. A message (or portion of a message) enclosed in subscripted braces $\{m\}_k$ denotes message m encrypted using key k. Only users who know k can encrypt or decrypt such a message. The join protocol is as follows:

Join Protocol

1. $u \rightarrow s$: join request
2. $s \leftrightarrow u$: authenticate and negotiate key k_u
3. s attaches u to tree. Let $k_0, \ldots, k_j$ be the keys from root (k_0) to the node where we attached u. (The node with k_j is the parent of the node with k_u .)
4. s generates new keys $k_0', \ldots, k_j'$
5. $s \rightarrow$ {all users} : $\{k_0'\}_{k0}, \ldots, \{k_j'\}_{kj}$
6. $s \rightarrow u : \{k_0', \ldots, k_j'\}_{ku}$

The leave protocol is more complex. The server broadcasts a message containing all the new subgroup keys. The difficulty is that the normal way to send secret information to a subgroup is to use that subgroup's key. The leaving user, however, knows its current subgroup keys and could thereby learn the new subgroup key as well. To avoid this problem, each new subgroup key is sent encrypted by the (possibly new) keys of its subgroups. For example, in Figure 1, if User D requested to leave the group, the new KeyABCE' (replacing KeyABCDE) would be included in the rekey message twice: once encrypted by KeyAB and once encrypted by the new KeyCE', which replaces KeyCDE. Similarly, the new KeyCE' (which Users C and E need in order to decrypt the new KeyABCE') would be included twice: once encrypted by the new KeyC' and once encrypted by KeyE. This process continues to the leaves of the key tree. More formally, the leave protocol is as follows:

Leave Protocol

1. $u \rightarrow s$: leave request
2. $s \rightarrow u$: leave granted
3. s detaches u from tree. Let $k_0, \ldots, k_j$ be the keys from root (k_0) to the node where we attached u. (The node with k_j is the parent of the node with k_u .) Let $x_0, \ldots, x_j$ be the corresponding nodes.
4. s generates new keys $k_0', \ldots, k_j'$

5. We will now construct the rekey message in several pieces. For i=0..j, we create m_i to be the portion of the message containing k_i'. In particular, for each child of x_i (the node corresponding to k_i), m_i will contain a copy for k_i' encrypted by key for that child. In the case of x_{i+1} (one of the children of x_i), the key used will be the new key k_{i+1}'.
6. $s \rightarrow$ {all users} : $m_0, \ldots, m_j$

Note that as described, there is an obvious flaw: an adversary can forge rekey messages and "black out" authorized users from reading subsequent messages in the group. The original paper suggested that rekey messages be signed using a secure digital signature scheme to prevent forging.

3. MODELING AND MODEL-CHECKING

In this section, we describe how we modeled the protocol and applied model checking.

Our first decision was which model checker to use. We chose the Murphi verifier [4] rather arbitrarily, but in part because others have used it successfully on security protocols [9]. We expect that other state-of-the-art model-checkers such as Spin [5] or FDR [8] would have produced similar results. We chose not to use a BDD-based model checker such as SMV [2]. BDD-based model checking has done very well for hardware, but not nearly as well on high level protocols [6]. We will discuss some of the advantages and disadvantages of choosing Murphi later in this paper.

Having chosen Murphi, we were able to follow the general approach of Mitchell et al. [9], who had first used Murphi on small security protocols. Murphi is a guarded-command language [3], so we model the various components of the system (e.g., server, users, adversary) as sets of commands for the possible actions of each part of the system. This modeling is somewhat tedious, but generally straightforward. For example, the commands to model the users allow an idle user to request to join the group (which entails sending a join message to the server and changing state to wait for the reply), a user waiting for a reply to process that reply, etc. Murphi has no notion of communication channels, so the network must be modeled explicitly using shared variables as a multiset of messages.

The adversary is the most interesting part of the modeling. Standard practice in the security community is to allow the adversary to intercept, modify, and generate arbitrary messages. However, the paper describing the protocol proposed using digital signatures to prevent forged messages, so we limited our adversary to eavesdropping and remembering messages, and then sending out previously saved messages later. Thus, our adversary is limited to “replay attacks”, which cannot be prevented by digital signatures. The adversary can remember a finite number of previous messages. The adversary cannot encrypt or decrypt a message unless it knows the key.

The main modeling challenge we faced was incomplete information. The original paper emphasized performance, so many implementation details were missing. For example, the properties of the underlying communication network were never specified in the paper. We chose to model a network that guarantees delivery of messages, but does not guarantee preservation of message order --- properties that are typical of broadcast and multicast on wide-area networks. We expect such networks to be the typical underlying infrastructure for a secure group communication system. We also modeled a network that preserves message order; we will discuss this briefly in the next section.

Another challenge created by incomplete information was specifying properties to verify. Formal verification is only as good as the properties being verified. The original paper, however, never stated a security policy, other than a parenthetical comment about privacy, integrity, and authenticity. For our correctness properties, we specified that all members of each subgroup shared the subgroup key, and that all non-members did not have that key. Furthermore, the adversary should never obtain a usable key, and the adversary’s messages should never be processed. Clearly, these are necessary conditions for correctness of the protocol; we do not claim sufficiency.

Throughout the verification process, state explosion was a constant threat. We were forced to make countless simplifying assumptions, such as limiting the size of the model to only a few users total, allowing the adversary to remember only a small finite history of previous messages, having a small number of possible keys and purging the adversary’s memory when we are forced to recycle any keys, and many, many others. Altogether, these simplifications mean that countless bugs in the real protocol may have been simplified out of the Murphi model. This fact combined with the incomplete specifications means that model checking must be viewed as a debugging aid, rather than a certification of correctness. Success is achieved by finding bugs that had otherwise eluded detection.

4. RESULTS

Surprisingly, the adversary did not play a major role in the bugs we found. Instead, the bugs were the result of subtle ordering problems with the messages.[2]

A number of trivial bugs appeared very early on. For example, the protocol as stated allows a user to join and then immediately leave the group. Thus, a leave request might reach the server before the join request for that user. Adding a simple handshake would eliminate this problem. A similar problem is that the adversary can replay previous join/leave requests and overload the server and network, creating a denial-of-service attack. A proper authentication procedure (which the protocol presumes) would use nonces (one-time random numbers) in the authentication to prevent replaying previous joins/leaves. In order to explore other possible bugs, and to reduce state explosion, we simply modified the model to prohibit these behaviors.

Murphi found two more serious bugs. The first bug was mentioned in the original paper—a forged rekey message can be used to “black out” a legitimate user from subsequent communication in the group. The paper proposed using digital signatures to prevent forged rekey messages, but neglected to consider replay attacks. The Murphi model found that the adversary can save an old rekey message (properly signed by the server) and send it out again later, switching the victim to old, no-longer-valid keys. A solution is to timestamp the messages or to use sequence numbers, encrypted into the message to prevent tampering. Note that the timestamp must be large enough to prevent the adversary from reusing old messages whenever the clock wraps around. It is interesting that the authors of the protocol noticed the problem, proposed a solution, but didn’t realize that the solution was inadequate, whereas the verification tool found the error quite easily.

A more serious bug was also discovered by Murphi. To our knowledge, this bug was previously unknown. The problem is that when multiple transactions are occurring, the respective rekey messages can arrive at a user out-of-order. Depending on the details of the message format (i.e. are there known fields in the encrypted parts of messages in addition to the randomly-generated key?) the result is at best that the user receives a bunch of unintelligible messages, at worst that the user switches to an incorrect key based on decrypting the new key with the wrong old key. Unfortunately, there are no easy solutions to this problem. Unencrypted sequence numbers would be vulnerable to attack by the adversary, causing the messages to be decrypted with the wrong keys. Encrypted sequence numbers are useless,

[2] Given that the original paper did not specify an underlying network model, it is perhaps unfair to label these as bugs, rather than merely as surprises.

because we need to know the ordering of the messages in order to decrypt them correctly. For a small network with infrequent rekey messages, a user could save all unintelligible messages and retry each one whenever a new rekey message arrives, but this approach is obviously impractical if rekey messages are frequent. Another approach would be to implement a scheme to provide order-preserving broadcast on top of the non-order-preserving network, such as ABCAST [1]. Unfortunately, such schemes impose substantial overhead and hinder scalability to large networks.

Note that the original paper did not specify the underlying network model. We chose a model that does not preserve message order, as in practice, that would be typical for an underlying network supporting multicast. When we changed our network model to preserve message order and reran the model checker, the above bugs disappeared. Thus, the protocol may have been designed assuming preservation of message order, but that assumption was never stated.

Altogether, the verification process required less than two months of part-time effort. Despite the fact that the protocol was published and hence presumed free of obvious bugs, we were able to discover two difficult bugs with very little effort. Furthermore, the two student authors, who carried out the bulk of the verification, had had no previous formal verification experience. In short, we had a very good bug/effort ratio.

5. DISCUSSION AND CONCLUSION

In retrospect, the choice of Murphi as our model-checking tool proved quite reasonable. Although we feel that other contemporary model checkers would also have served well, some positive features of Murphi stood out. Ease-of-use was perhaps the most important point. Murphi provides a very simple, small language, which makes the learning curve easy. For a novice user, or if we wish verification to be adopted by non-experts, initial ease-of-learning and ease-of-use can be critical. Similarly, the guarded command semantics of Murphi proved to be very convenient for modeling the protocol. Finally, Murphi provides special data types that allow powerful symmetry reductions to be applied automatically [7]. We made extensive use of these data types, resulting in an enormous savings in the number of states explored.

The hash compaction scheme, a probabilistic verification technique that compresses states stored in the hash table, proved to be absolutely vital to avoid running out of memory. The scheme is supported in both Spin and Murphi [12].

Murphi also has downsides. For example, Murphi has a very weak language for specifying properties to check, allowing only simple invariants that must hold in all reachable states. We did not find this restriction to be burdensome, but if more powerful properties could be specified, perhaps we would have found them indispensable. Similarly, Murphi has no communication primitives. The protocol we considered is broadcast-oriented, so the lack of communication primitives was not a problem. For a more connection-oriented protocol, however, communication primitives, such as the channels provided by Spin, would be very useful.

More generally, as the importance of security protocols increases, there is a need for verification tools tailored for this domain. The tool could provide primitives for communication, encryption/decryption, and the adversary, as well as opening the door to domain-specific reduction techniques. Shmatikov and Stern have recently published some provably-sound reduction techniques specifically designed for model checking security protocols [11]. Much more progress along these lines should be possible.

The greatest challenge we faced was incomplete informal specifications. The paper describing the protocol did not provide all the information needed. In particular, no security policy was given, nor were all assumptions about the environment stated. Greater attention to these issues is certainly necessary for security protocols. Formal description techniques, even in the absence of automated tool support, could be helpful. The saying comes to mind, “A program that has not been specified cannot be incorrect; it can only be surprising.” [14] For security protocols, surprises are dangerous.

We were able to find bugs in a published security protocol with minimal effort. The two student authors, who performed the bulk of the verification work, had no prior formal verification experience. Clearly, model checking tools and methodology are now robust enough that they can be a routine part of security protocol design. There are no guarantees that the model checking will succeed, but the likelihood of finding bugs is high, and the effort expended is low.

Finally, we note that the numerous ad hoc simplifications one needs in order to use model checking imply that an error-free model-checking run does not ensure “correctness”. For critical applications, conventional proof-based methodologies --- using a more expressive formal description technique and perhaps automated proof assistants --- are still needed. Nevertheless, the ease-of-use and excellent bug/effort ratio of model checking makes this approach an excellent tool for designing and debugging security protocols.

References

[1] J.-M. Chang and N. F. Maxemchuk. Reliable broadcast protocols. ACM Transactions on Computer Systems, 2(3):251--273, August 1984.

[2] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In R. Alur and T. A. Henzinger, editors, Computer-Aided Verification: Eighth International Conference, pages 419--422. Springer-Verlag, July 1996. Lecture Notes in Computer Science Number 1102.

[3] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM, 18(8):453--457, August 1975.

[4] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In International Conference on Computer Design. IEEE, October 1992.

[5] G. J. Holzmann. Design and Validation of Computer Protocols. Prentice-Hall, 1991.

[6] A. J. Hu and D. L. Dill. Reducing BDD size by exploiting functional dependencies. In 30th Design Automation Conference, pages 266--271. ACM/IEEE, 1993.

[7] C. N. Ip and D. L. Dill. Efficient verification of symmetric concurrent systems. In International Conference on Computer Design, pages 230--234. IEEE, October 1993.

[8] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In Tools and Algorithms for the Construction and Analysis of Systems, pages 147-- 166. Springer-Verlag, 1996. Also published in Software Concepts and Tools Volume 17, pp. 93--102.

[9] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murphi. In Symposium on Security and Privacy, pages 141--151. IEEE, 1997.

[10] S. Mittra. Iolus: A framework for scalable secure multicasting. In SIGCOMM. ACM, 1997.

[11] V. Shmatikov and U. Stern. Efficient finite-state analysis for large security protocols. In 11th Computer Security Foundations Workshop, pages 106--115. IEEE, 1998.

[12] P. Wolper, U. Stern, D. Leroy, and D. Dill. Reliable probabilistic verification using hash compaction. Submitted for publication. A draft is available from http://sprout.Stanford.EDU/uli/publications.html.

[13] C. K. Wong, M. Gouda, and S. S. Lam. Secure group communications using key graphs. In SIGCOMM. ACM, 1998.

[14] W. D. Young, W. E. Boebert, and R. Y. Kain, Proving a Computer System Secure, *The Scientific Honeyweller*, vol 6, no 2 (July 1985), pp. 18—27. Reprinted *in Computer and Network Security*, M. D. Abrams and H. J. Podell, eds., IEEE Computer Society Press, 1986.