2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing

Formal Analysis of Fault-tolerant Group Key Management using ZooKeeper

Stephen Skeirik, Rakesh B. Bobba, Jose Meseguer
University of Illinois at Urbana-Champaign, USA
{*skeirik2,rbobba,meseguer*}*@illinois.edu*

***Abstract*—Security-as-a-Service (SecaaS) is gaining popularity, with cloud-based anti-spam and anti-virus leading the way. In this work we look at key management as a security service and focus on group key management with a central group key manager. Specifically, we analyze a rewriting logic model of a ZooKeeper-based group key management service specified in Maude and study its tolerance to faults and performance as it scales to service larger groups using the PVeStA statistical model checking tool.**

***Keywords*-Security-as-a-Service, Group Key Management, Formal Verification, Maude**

I. Introduction

With the increasing popularity and adoption of cloud computing, security of cloud computing has become an important issue. As a result many companies are starting to provide security services in the cloud, leading to the emergence of "security-as-a-service (SecaaS)" as a paradigm. Cloud-based anti-spam and anti-virus services are reported to be the most popular SecaaS services in use today. Cloud Security Alliance (CSA), a non-profit coalition of practitioners, corporations, and other stake holders, recently identified ten categories of SecaaS services [5] that included identity and access management, and encryption among others. Key-management is considered in scope by CSA in the Encryption category although they recommend that key-management be undertaken by the customer if possible for security reasons.

In this work we look at the design of key-management as a service without sacrificing security. Specifically, we focus on key distribution for secure group communication with a central group key controller. Secure group communication, as the name indicates, is used whenever an entity (or multiple entities) needs to securely communicate with multiple authorized entities, for example, in distributed interactive computations such as collaborative work. In settings with a central group key controller, a *group key* is generated and distributed to authorized group members by the group key controller [16]. However, in such settings the group key controller is a single point of failure. Replicating the group controller (*e.g.* [3], [17]), and eliminating the group controller through decentralized key distribution (*e.g.* [2], [14]) or distributed key agreement (*e.g.* [6], [10], [15]) are alternative approaches proposed in literature to alleviate this problem[1]. However, decentralized key distribution and distributed key agreement approaches typically increase the complexity of group members and are not commonly used. There are many fault-tolerant replication algorithms in the literature (*e.g.* [3]) that can be used to replicate a group key controller.

Our goal in this work is to determine whether a fault-tolerant group key-management service can be built by leveraging existing coordination services commonly available in cloud infrastructures. We model a group key distribution service built using Zookeeper [9], a reliable distributed coordination service. ZooKeeper is used by many distributed applications such as Apache Hadoop MapReduce, Apache HBase and by many companies including Yahoo and Zynga in their infrastructures. One advantage of using freely available off-the-shelf components is a possible reduction in development and maintenance time for application designers. Another is ease of deployment in the cloud.

Specifically, we analyze a Maude [4] based rewriting logic model of a ZooKeeper-based group key management service using the PVeStA [1] statistical model checking tool. In our experiments, Maude provides the language necessary to describe and run formal models of distributed systems while PVeStA executes and observes the model over the course of tens to hundreds of thousands of runs to understand expected model behavior through statistical model checking within a specified confidence interval.

In this model, key generation is handled by a centralized key management server while key distribution is offloaded to a ZooKeeper cluster. Regarding this choice, our analysis seeks to answer two key questions: Firstly, can a ZooKeeper-based group key management service handle faults more reliably than a traditional centralized group key manager? Secondly, can such a service scale to a large number of concurrent clients with a low enough latency to be useful?

Our preliminary analysis shows that a scalable and fault-tolerant key-management service can indeed be

[1]Please refer to [13] for a survey on group key management schemes

978-0-7695-4996-5/13 $26.00 © 2013 IEEE
DOI 10.1109/CCGrid.2013.98

built using ZooKeeper. However, our analysis also showed that a naive design may run into performance bottlenecks, highlighting the need for formal modeling and analysis.

Organization: We provide some necessary background information in Section II and present our design approach in Section III. The formal model and analysis of the design are in Section IV. A discussion of limitations, conclusions and future work are in Section V.

II. Background

Group Key Management: Group key management refers to the management of cryptographic keys for secure group communication. Typically a *group key* is established for encrypting group communications either through the help of a centralized controller or using decentralized key distribution or distributed key agreement approaches. In either case the group key needs to be updated i) whenever a new member joins the group to preserve *backward secrecy*, ii) whenever a group member leaves the group to preserve *forward secrecy*, and iii) periodically even when there are no group membership changes to ensure the secrecy of the group key.

Apart from generating and distributing group keys to group members, another function of a group controller is admission control to the group, that is, to authenticate and admit only authorized users to the group. The group controller maintains pairwise security associations (SA) (*e.g.*, pairwise keys) with group members, and the group key is securely transmitted leveraging these pairwise security associations. Specifically, a *key encrypting key (KEK)* for the group is distributed to each group member using pairwise security associations. The group key is periodically updated and securely distributed using the KEK. While in this work we focus on this simple key management scheme, more sophisticated schemes like the key graphs or logical key hierarchy [16] could be supported as well.

In settings with a centralized group controller, failure of the group controller would not only hinder group dynamics (*i.e.*, joins and leaves) but would also impact periodic key updates, leaving the group key vulnerable. Further, if the group controller were to fail in the middle of a key update, then the group might be left in an inconsistent state with some group members migrating to the updated key when others are using the old key. This is especially significant when considering the design of group key management as a cloud-based service, since such a service will manage a lot of groups. While maintaining a hot-backup might alleviate this problem, a central controller still suffers from being a performance bottleneck for large groups and when managing multiple groups as it has to individually update the keys of each member.

ZooKeeper: Zookeeper [9] is a high-performance coordination service to support internet-scale distributed applications. It is a centralized service but replicated to achieve high availability and performance. From a bird's eye view, the ZooKeeper system consists of two kinds of entities: servers and clients. All of the servers together form an ensemble - a distributed and fault-tolerant key/value store which the clients may read data from or write data to. In ZooKeeper terminology, the key/value pairs located in the distributed store are called *znodes*. These znodes are then organized in a tree structure similar to that of a standard filesystem. ZooKeeper provides an event-driven mechanism by supporting *watches*. A client can set a watch on a znode which will be triggered when the the znode changes.

In order to achieve fault-tolerance, ZooKeeper requires that a majority of servers—that is, a quorum—acknowledge and log every write to disk before it is committed. Since ZooKeeper enforces that a quorum of servers must be alive and aware of each other for the system to operate, this means that updates will not be lost. A detailed description of ZooKeeper design and features can be found in [9] and in ZooKeeper documentation[2].

ZooKeeper provides a set of guarantees to its clients which are useful for key management applications. Updates to ZooKeeper state are atomic – they either succeed or fail. Further, once an update succeeds it will persist until it is overwritten. A client will have the same view of the system regardless of which ZooKeeper server it connects to and is guaranteed to be up-to-date within a certain time-bound. For our purposes, this means keys, once written, will not be lost. The watch event mechanism will enable ZooKeeper to easily propagate key change events to interested clients, a fact we will explore further later.

Rewriting Logic and Maude: Rewriting logic is a framework for both logical deduction and system specification—particularly concurrent systems [11]. A conditional rewrite rule has the general form:

$$\texttt{crl}\ [L] : t(\overrightarrow{x}) \Rightarrow t'(\overrightarrow{x})\ \texttt{if}\ C(\overrightarrow{x})$$

where t, t' are terms, C is a condition, and they are all parameterized by variables $\overrightarrow{x}$. Informally, the rule states that a state matching the pattern t may transition to a corresponding state t' if the match for $\overrightarrow{x}$ satisfies condition C.

We can view the state of a distributed system in rewriting logic as an associative-commutative soup of messages and actors where rewrite rules define transitions from one state to the next. Note that rewrites

[2]http://zookeeper.apache.org/doc/trunk/

may be both local and concurrent. Because of this, rewrite rules are intrinsically non-deterministic; rewrite rules may overlap on different pieces of distributed state so that one rewrite may forbid a different rule from rewriting and vice versa. The nondeterministic, local nature of rewrites closely models real distributed systems.

Maude [4] is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming for a wide range of applications. Besides supporting equational specification and programming, Maude also supports rewriting logic computation.

PVeStA: While non-deterministic rewrite rules can be used to describe distributed systems in a very general way, it is sometimes useful to restrict the classes of systems we consider. Though concurrent systems are in general nondeterministic, there is a very large class of interesting probabilistic systems. The great advantage of these systems is that they are amenable to statistical analysis.

This fact is exploited by PVeStA [1], a parallel statistical model checking and quantitative analysis tool for probabilistic systems, which we use to formally analyze our rewriting logic specification. We ensure our model is probabilistic by forcing rules to apply in a deterministic order and using probabilistic rewrite rules to express and quantify all non-determinism.

A conditional probabilistic rewrite rule has the general form:

$$\texttt{crl}\ [L] : t(\overrightarrow{x}) \Rightarrow t'(\overrightarrow{x}, \overrightarrow{y})$$
$$\texttt{if}\ C(\overrightarrow{x})\ \texttt{with probability}\ \overrightarrow{y} := \pi_r(\overrightarrow{x})$$

where $\overrightarrow{x}, \overrightarrow{y}$ are variables, t, t' are terms, C is a condition, and π_r is a probability distribution. A probabilistic rewrite rule operates in the exact same way as a standard rewrite rule with the exception of the new variables $\overrightarrow{y}$ in the resultant term. If condition $C(\overrightarrow{x})$ is satisfied, then t rewrites to t' parameterized by $\overrightarrow{x}$ and additional variables $\overrightarrow{y}$ are randomly chosen according to the probability distribution π_r. Because of the extra variables $\overrightarrow{y}$, the outcome of the above rewrite rule is non-deterministic. However, this non-determinism is quantified by the probability distribution $\pi_r(\overrightarrow{x})$, which is parametric on the matching parameters $\overrightarrow{x}$.

III. Approach

In this section we discuss the high-level design of a group controller service for clouds that leverages the ZooKeeper service. In this design, based on [8], the group key generation and group key distribution functions of the group controller are separated. Group key generation is still performed by the group controller,

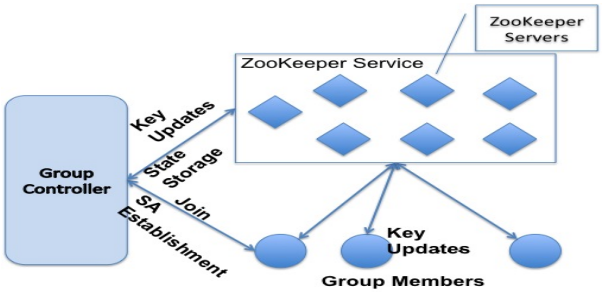

Figure 1. Group Key Management Service Architecture

while key distribution is delegated to the ZooKeeper service. The group admission control function is also left with the group controller. Further, the group controller also uses the ZooKeeper service to store its state in a replicated fashion to enable quick recovery from failure.

More specifically, as shown in Figure 1, users wishing to join a group will contact the group controller who performs the necessary admission control (*i.e.*, authentication and authorization checks). If the user is authorized, the group controller adds the user to the group and establishes a long term security association with the user. The group controller then updates its state with the ZooKeeper service, generates and updates the group key through the ZooKeeper service (as explained next), and provides the new user with the updated group key and necessary information to establish connection with ZooKeeper and obtain future key updates.

The group controller uses znodes to maintain group information with the ZooKeeper service. Whenever a new user joins the group, new znodes corresponding to that user are added as children to the znode representing the group. The znode corresponding to the group contains the group key in an encrypted form (encrypted with a key encrypting key or KEK). When the group key needs to be updated (it is updated periodically) the new group key encrypted with the KEK is written to the group znode by the group controller. All group members have a watch set on this group znode and are notified whenever this znode changes, so that they may obtain the updated key. When a new member joins or an existing member leaves the group, the group controller updates the KEK and distributes it using pairwise keys. Specifically, the group controller updates znodes corresponding to each user in the group with the KEK encrypted under the pairwise key corresponding to the user. Each group member has a watch on the znode corresponding to him and is notified whenever this znode changes. In this manner, the group controller uses the ZooKeeper service to maintain group information and to distribute and update cryptographic keys. The group controller also saves its operational state with the ZooKeeper service (*e.g.*, group member znodes it has updated, last key update time etc.) so that if the

controller were to fail a back-up controller may take over without interruption in the service. Furthermore, group admission control and key generation are performed by the group controller, which can reside on customer premises, alleviating security and compliance concerns. However, the performance implications of the design still need to be understood.

IV. Formal Analysis with Maude and PVesta

Model

We now present our Maude executable specification of a group key management system built on top of ZooKeeper using the Russian-doll actor model [12]. After a few common preliminaries, we can naturally subdivide the specification into two parts: one defining ZooKeeper and the other defining the group key management system.

The distributed state of the system is an associative-commutative multiset or "soup" structure, called a configuration, populated by three basic sorts of objects: actors, messages, and the scheduler. Actors represent addressable fragments of a distributed computation. Actors may also contain other actors; in this case, computation may occur simultaneously at any actor level. Actor addresses are assigned hierarchically; each actor's address is prefixed by the address of its parent actor. Messages are state fragments passed from one actor to another marked with a timestamp, an address, and a payload. Finally, the scheduler's purpose is to deliver messages to an actor at the appropriate time, that is, a message's timestamp; it also provides a total ordering of messages in case of identical timestamps.

The ZooKeeper system state is modeled by three classes of actors: the ZooKeeper *service*, *servers*, and *clients*. The service actor represents the entire cluster of ZooKeeper servers and contains each server actor which represents a single server in the ZooKeeper cluster. The ZooKeeper client actors represent the ZooKeeper client library, and each client contains a single actor representing an application of this library. We refer to this actor as the client's *application* object.

The group key management system also consists of three different classes of actors: the group key *wrapper*, *managers*, and *clients*. These three actors share a relationship very similar to that of the ZooKeeper service, server, and client. Specifically, the wrapper actor contains and manages several manager actors which group key clients connect to in order to request service. Note that the term client is overloaded here; we will specify which client we mean when it is ambiguous.

All of these actors are designed to operate according to the approach discussed in Section III, but we must be careful to clarify a few details. Firstly, we do not model ZooKeeper leader election. Instead, the ZooKeeper leader is assumed not to fail. This abstraction dramatically reduces the complexity of the protocol we must model, and is safe because leadership is not an externally visible property. In addition, each server actor in the service can maintain connections with an arbitrary number of ZooKeeper clients. A ZooKeeper server object may fail; when that happens, it will no longer respond to any messages. After a variable repair timeout, the server will come back online.

Clients connect to servers and then send read and write requests for key/value pairs. With any read or write request, a client can also optionally set a watch on the read/written key. If a server fails, all of its connected clients will attempt to migrate to another randomly chosen server. If a key manager fails while a client is joining or leaving the secure group, that client will simply time out and try again.

A key manager buffers clients' requests and answers them in order. Before answering each request, it saves information about the requested operation (either a join or a leave) to ZooKeeper. If a manager dies, the succeeding manager merely needs to load the stored ZooKeeper state and replay the last operation.

For our model to accurately reflect the performance of a real ZooKeeper cluster, we choose parameters that agree with the data gathered in [9]. In particular, we set the total latency of a read or write request to ZooKeeper on average takes under 2msec (which corresponds to a ZooKeeper system under *full* load as in [9]). Note that this average time only occurs in practice without server failure; our model permits servers to fail. We also assume that all of the ZooKeeper servers are located in the same data center, which is the recommended way to deploy a ZooKeeper cluster.

Here we show an example illustrating how we can translate an informal, English specification into a Maude rewrite rule.

```
crl [ZKSERVER-PROCESS] :
  < A : ZKServer | leader: LDR,
    live: true, zxid: Z,
    store: ST, clients: CS,
    requests: RL, updates: UL, AS >
  {t, A <- process}
=>
  < A : ZKServer | leader: LDR,
    live: true, zxid: Z',
    store: ST'', clients: CS,
    requests: null, updates: null, AS >
    M M'
if (Z',ST',M) := commit(CS,UL,ST)   /\
    M'        := process(LDR,RL,ST') .
```

This rule in Maude illustrates how a ZooKeeper server object processes a batch of updates and requests during a processing cycle. Essentially, this rule will apply whenever an actor with address `A` and type `ZKServer` is sent a message with payload `process`.

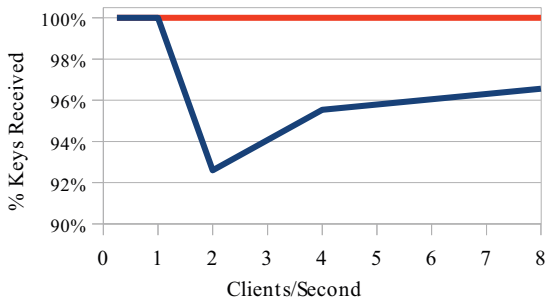

Figure 2. Keys Received/(Keys Sent) Vs. Client Join Rate

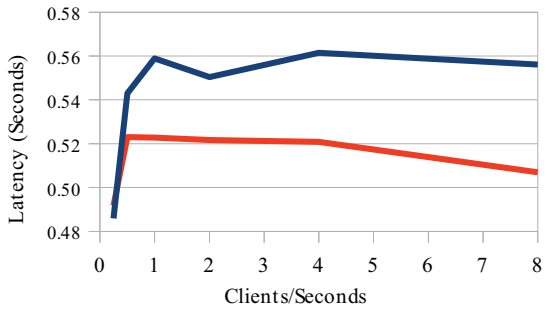

Figure 3. Latency Vs. Client Rate

Then the condition of this rule invokes two auxiliary functions, `commit` and `process`, which repeatedly commit updates from the leader and process requests from connected clients. When this rule completes, the server will have a new zxid and store, an empty update and request list, and a set of messages to be sent `M` and `M'` generated by the auxiliary functions.

Analysis and Results

We now present our testing methodology and analysis results. For flexibility in testing, the model was designed with a variety of parameters which vary the number of servers, the rate of client arrival, failure probabilities, the test duration, and network and processing latencies. Whenever possible, we attempted to estimate conservatively.

The first experiment demonstrates that saving snapshots of the group key manager state in the ZooKeeper store can increase the overall reliability of the system. In the experiment, we set the failure rate for the key manager such that it would fail in half of the experiments, the time to failover from one server to another at 5 seconds, and the experiment duration to 50 seconds (we keep the experiment duration short to speedup analysis). We then compare the key manager availability (i.e. the time it is available to distribute keys to clients) between a single key manager and two key managers where they share a common state saved in the ZooKeeper store. In the single case, the average availability was 32.3 seconds whereas in the shared case, the average availability was 42.31 seconds. This represents an improvement from a 65% to 85% available even with our conservative parameters. Of course, we expect our system with replicated state to be more reliable because there is no longer a single point of failure.

Our second experiment demonstrates that using ZooKeeper to distribute shared keys is efficient and scalable enough for real-world use. There are two important system parameters the experiment measures. The first parameter measured is the percentage of keys successfully received by group key session members; here we define success by the percentage of keys that a client is expected to receive over its lifetime versus the amount it actually receives. Because our key distribution protocol is so simple, instead of retrying on error, a client simply fails. This failure model is conservative; in a real implementation, clients would be able to recover. The second parameter measured is key distribution latency—from the time a key is generated by the key manager until that key is received by the session members. This latency should be low so clients receive keys in a timely fashion. We then measure these parameters against increasing numbers of clients joining the secure group per second. As more clients try to join at once, the key manager and ZooKeeper must scale to deliver keys more quickly.

In our experiment, we sample at join rates of once every 4 seconds, once every 2 seconds, once every second, twice a second, 4 times a second, and 8 times a second. We keep the same experiment duration, but we pick failure probabilities such that system will have 99.99% availability. We let link latency between the ZooKeeper service and ZooKeeper clients vary according to a uniform distribution on the interval from .05 to .25 seconds.

We present our results in Figure 2 and Figure 3. Figure 2 shows key reception rate versus client join rate while Figure 3 shows key distribution latency versus client join rate. The blue line corresponds to our initial experiments. The red line corresponds to a slightly modified model where we added a 2 second wait time in-between key updates from the key manager. While our initial experiments show that naively using ZooKeeper as a key distribution agent works well, at high client join rates, the key reception rate seems to level out around 96%. This occurs because ZooKeeper can apply key updates internally more quickly then clients can download them because all of the ZooKeeper servers enjoy a high-speed intra-cloud connection. By adding extra latency between key updates, the ZooKeeper servers are forced to wait enough time for the correct keys to propagate to clients. As shown in Figure 2, this change achieves a 99% key reception in all cases.

On the other hand, key distribution latency remains relatively constant, at around half a second, regardless of the join rate because ZooKeeper can distribute keys at a much higher rate than a key manager can update them [9]. Of course, the artificial latency added in the second round of experiments has a cost; it increases the time required for a client to join or leave the group by the additional wait time.

V. Conclusion and Future Work

In this work we analyze whether a fault-tolerant and scalable group key management service can be built using a commonly used coordination service. Specifically, we modeled and analyzed a group key management service built using ZooKeeper for fault-tolerance and performance and showed that a scalable and fault-tolerant key-management service can indeed be built using ZooKeeper. However, our analysis also showed that if this is not done carefully a naive design may run into performance bottlenecks highlighting the need for formal modeling and analysis. Specifically, the group key manager must add a small amount of latency before distributing keys to ZooKeeper to ensure that they propagate correctly to clients. Indeed, the results presented here settle the various doubts raised about the effectiveness of ZooKeeper for key management by an earlier, but considerably less-detailed, model and analysis [7]. While our model accurately represents ZooKeeper and the key manager, in order to ease modeling and analysis, our model only considered crash-failures. A more complete analysis would model network failure effects as well and is planned for future work. Enhancing the key distribution protocol to be efficient in the face of membership churn is also left as future work; for example, batching group membership changes and applying them all at once would be a good first effort. In addition, we chose to model a very simplistic single layer group key protocol. Extending ZooKeeper key distribution to more complex hierarchical protocols to increase scalability is another direction for fruitful future work.

Acknowledgment

This work has been partially supported by the Air Force Research Laboratory and the Air Force Office of Scientific Research under agreement number FA8750-11-2-0084. We thank Jonas Eckhardt for his earlier work in [7], which raised questions stimulating our own work and Z. Kalbarczyk for insightful discussion that helped us improve our analysis.

References

[1] M. AlTurki and J. Meseguer. PVeStA: A parallel statistical model-checking and quantitative analysis tool. in Proc. CALCO 2011, Springer LNCS 6859, 386–392, 2011.

[2] R. Bobba and H. Khurana. DLPKH - Distributed Logical Public-Key Hierarchy. In P. D. McDaniel and S. K. Gupta, editors, *ICISS*, volume 4812 of *Lecture Notes in Computer Science*, pages 110–127. Springer, 2007.

[3] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, Nov. 2002.

[4] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude – A High-Performance Logical Framework*. Springer LNCS Vol. 4350, 2007.

[5] Cloud SecurityAlliance: Security as a Service Working Group. Defined Categories of Service 2011.

[6] R. Dutta and R. Barua. Dynamic group key agreement in tree-based setting. In *ACISP*, pages 101–112, 2005.

[7] J. Eckhardt. Security analysis in cloud computing using rewriting logic.

[8] J. Gupta. Available group key management for naspinet. Master's thesis, Univeristy of Illinois at Champaign-Urbana, 2011.

[9] P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, volume 10, 2010.

[10] Y. Kim, A. Perrig, and G. Tsudik. Tree-based group key agreement. *ACM Trans. Inf. Syst. Secur.*, 7(1):60–96, 2004.

[11] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[12] J. Meseguer and C. Talcott. Semantic models for distributed object reflection. In *Proceedings of ECOOP'02, Málaga, Spain, June 2002*, pages 1–36. Springer LNCS 2374, 2002.

[13] S. Rafaeli and D. Hutchison. A survey of key management for secure group communication. *ACM Comput. Surv.*, 35(3):309–329, 2003.

[14] O. Rodeh, K. P. Birman, and D. Dolev. Using avl trees for fault-tolerant group key management. *Int. J. Inf. Sec.*, 1(2):84–99, 2002.

[15] M. Steiner, G. Tsudik, and M. Waidner. Key agreement in dynamic peer groups. *IEEE Trans. Parallel Distrib. Syst.*, 11(8):769–780, 2000.

[16] C. K. Wong, M. G. Gouda, and S. S. Lam. Secure group communications using key graphs. *IEEE/ACM Trans. Netw.*, 8(1):16–30, 2000.

[17] L. Zhou, F. B. Schneider, and R. Van Renesse. Coca: A secure distributed online certification authority. *ACM Trans. Comput. Syst.*, 20(4):329–368, Nov. 2002.