Intrusion-Tolerant Group Management for Mobile Ad-Hoc Networks

Jonathan Kirsch
Department of Computer Science
The Johns Hopkins University
Baltimore, Maryland 21218
Email: jak@cs.jhu.edu

Brian Coan
Distributed Computing Group
Telcordia Technologies
Piscataway, New Jersey 08854
Email: coan@research.telcordia.com

***Abstract*—This paper presents PICO, a distributed protocol that manages group membership and keying in mobile ad-hoc networks (MANETs). PICO tolerates a limited number of Byzantine nodes and an additional limited number of crashed nodes. It allows clients to join or leave a dynamically changing group and provides group members with a dynamically updated group encryption key. Since MANETs are characterized by relatively high message loss and frequent network partitions, PICO is built around a new Byzantine fault-tolerant agreement protocol designed to cope with these conditions. The agreement protocol leverages weak (commutative) semantics to allow multiple partitions to continue operating in parallel without sacrificing correctness; it also copes well with unreilable communication links because it uses cumulative messages instead of needing the retransmission of prior lost messages.**

I. INTRODUCTION

This paper addresses the problem of building a robust and highly available group management system, providing services for group membership management, cryptographic key generation, and secure key distribution. The group managment system is designed to work in MANETS that might have high packet loss, temporary network partitions, a limited number of compromised processors and a limited number of crashed processors.

Applications that use this service can join a group and encrypt messages for one another using the group's shared encryption key, thus facilitating secure communication among group members. This problem arose as part of our work on the DARPA IAMANET (Intrinsically Assurable Mobile Ad-Hoc Networks) program. In our system, ZODIAC [1], dynamically-formed groups of nodes must be able to communicate securely with one another. The system is intended to operate in a MANET with short-lived links, high packet loss, and transient network partitions. It must operate despite a limited number of compromised participants.

The key properties of PICO are as follows. It uses threshold cryptography to achieve intrusion tolerance. PICO uses the threshold coin-tossing scheme of Cachin, Kursawe, and Shoup [2], as adapted for the group membership problem by Dutertre et al. [3], to generate the group encryption key, and it uses a threshold digital signature scheme to construct proofs that can be used to verify the messages of PICO participants. Second, it uses limited tamper-proof hardware to assemble shares of the generated group key, hold the generated group key, and use the current group key to encrypt and decrypt traffic. This limited use of trusted hardware prevents a compromised client from divulging the group key to outsiders. Third, it uses a new Byzantine fault-tolerant agreement protocol to agree on the current group membership. This agreement protocol avoids the need for acknowledgements or queues of undelivered messages in the face of partitions and message loss. PICO uses cumulative threshold cryptographic proofs that allow efficient reconciliation by requiring only the "last" message to be delivered. These proofs also allow a member to know who is in the group at the time it encrypts a message. Only those processors in the group when a message is encrypted can potentially decrypt that message because a change in group membership is tied to a change in the shared group key.

This material is based upon work supported by the Defense Advanced Research Projects Agency and Space and Naval Warfare Systems Center, San Diego, under Contract No. N66001-08-C-2012. Approved for Public Release;Distribution Unlimited.

II. RELATED WORK

Several secure group communication systems, such as Ensemble [4], [5] and Secure Spread [6], have been built in the so-called "fortress model," where the group members are assumed to be correct and use cryptography to protect their communication from external attackers. Ensemble uses group key distribution protocols to distribute a shared group key, while Secure Spread uses a contributory key agreement protocol in which every group member contributes an equal share of the group secret.

Group communication systems have also been developed in the Byzantine fault model [7]. In the Byzantine model, faulty processes can fail arbitrarily. The Rampart system [8] and the SecureRing system [9] provide services for membership and ordered message delivery, and they depend on failure detectors to remove faulty processes from the membership. They rely on synchrony for both safety and liveness, since inconsistency can arise if a membership is installed that has one-third or more faulty processes. Unlike Rampart and SecureRing, PICO guarantees safety and liveness without relying on synchrony assumptions.

We emphasize that PICO is not a "group communication system" as the term applies to the systems above; it does not provide the strong membership semantics or the reliable, ordered message delivery of these systems. Rather, it allows applications to join a logical group and encrypt messages for one another using a dynamically generated symmetric group encryption key. PICO provides security against both external and insider attacks, as Rampart and SecureRing do. PICO does not provide any functionality to support the sending, retransmission, or ordering of application data messages.

At the core of PICO is a Byzantine fault-tolerant agreement protocol. Over the last several years, much of the work in Byzantine fault-tolerant agreement has focused on Byzantine fault-tolerant state machine replication (SMR) protocols (e.g., [10]–[14]). In the state machine approach [15], [16], a group of servers totally orders all updates that cause state transitions, and then the servers apply the updates in the agreed upon order. If the servers begin in the same initial state and the updates are deterministic, the servers will remain consistent with one another. SMR protocols provide strong consistency semantics, but they allow at most one partition to continue executing new updates at a time. In contrast, PICO's agreement protocol guarantees weaker, commutative semantics but allows multiple partitions to operate in parallel, which is desirable in MANETs.

PICO uses threshold cryptography [17] to implement its security services. Using threshold cryptography to provide security in peer-to-peer and MANET settings is not new (see, for example, [18]–[22]). Narasimha et al. [21] discuss the use of threshold cryptography for admission control in malicious environments. In the work of Narasimha et al. the current group members run a voting protocol (based on a threshold digital signature scheme) to decide whether or not to admit a potential group member. PICO also uses a threshold digital signature scheme, but the voting is conducted among group controller processes only. In addition to admission control, PICO requires a coordination protocol for group key generation.

The work most closely related to PICO is the Intrusion-Tolerant Enclaves protocol of Dutertre et al. [3], [23]. We use a similar protocol architecture as Intrusion-Tolerant Enclaves, and we adopt the same threshold key generation scheme [2]. We highlight the differences between the two protocols in Section V.

III. SYSTEM MODEL AND ASSUMPTIONS

We assume a Byzantine fault model. Processes are *correct, crashed,* or *faulty*; correct processes follow the protocol specification, crashed processes simply stop, while faulty processes can deviate from the protocol specification arbitrarily. Processes communicate by passing messages in an asynchronous communication network. Messages can be delayed, lost, or duplicated.

We assume that each process has tamper-proof hardware that can hold a public/private key pair and can assemble and verify key shares in the threshold key generation scheme. The process, even if it is Byzantine, cannot read the private key. When a controller sends a key share to a client, it encrypts the key share with the public key of the client's hardware, establishing a secure channel between a correct controller and the trusted hardware of the receiving client. The client's hardware decrypts the key share and verifies the correctness proof. When the hardware combines $f + 1$ valid key shares, it generates the group encryption key. Clients can use the hardware to encrypt application-level messages using the group key, but they cannot read the group key, even if they are Byzantine. The same physical machine can host both a client process and a controller process.

The network may be divided into multiple *partitions*. In an infinite execution, we say that there is a partition, P, if (1) P contains a subset (not necessarily proper) of the processes, (2) for any two correct processes a and b in P, if a sends an infinite number of messages to b then b delivers an infinite number of messages from a, and (3) there is some time after which no process in P receives any message from a process outside of P. Although we define partitions in terms of properties that hold forever (beginning at some point in the execution), real executions may go through many different partition configurations. In practice we are interested in proving that the properties of PICO hold in those partitions that last "long enough."

PICO supports secure group communication by generating and distributing a group encryption key. The group services for a group, G, are implemented by a collection of *group controller* processes. Each group has a fixed number of group controllers, C_G, uniquely identified from the set $\mathcal{R}_G = \{1, 2, \ldots, C_G\}$. At most f of the group controllers may be Byzantine. Each group can support an arbitrary but finite number of *clients*, which communicate with the group controllers to join or leave the group. Clients are uniquely identified from the set $\mathcal{S}_G = \{1, 2, \ldots\}$. Any number of client processes may be Byzantine.

As discussed in greater detail in Section IV, we make use of two threshold cryptosystems. First, each group uses an $(f + 1,\ C_G)$ threshold digital signature scheme. Each group controller knows one share of the private key, which it can use to generate partial signatures and proofs of correctness. We assume threshold signatures are unforgeable without knowing at least $f + 1$ secret shares. Second, each group uses an $(f + 1,\ C_G)$ threshold key generation scheme. Each group controller knows one secret share, which it can use to generate key shares and proofs of correctness. We assume one cannot construct the group encryption key without knowing at least $f+1$ key shares. Also as discussed in greater detail in Section IV, we make use of a public key infrastructure.

Coping with Faulty Clients: Like membership and key management systems, PICO must make an assumption about the behavior of client processes. With no assumptions, faulty group members can engage in two behaviors to compromise confidentiality: (1) broadcasting the group encryption key to non-group members, and (2) decrypting application messages using the group key and then re-broadcasting them to non-

group members. There are two possible approaches to dealing with this problem. The approach taken by the Intrusion-Tolerant Enclaves protocol [3] is to assume that all clients are correct, in which case no enforcement is necessary. We make a different (weaker) assumption, constraining the behavior of faulty clients, by requiring that they incorporate a limited trusted computing base. To cope with the first problem, we assume trusted hardware for key manipulation, storage, and application. We believe this assumption is reasonable in certain military environments and is likely to become more generally applicable in the future (see [24] for a description of mechanisms in this direction). To cope with the second problem, one can use the approach of the ZODIAC system [1] (which we do not describe in this paper) that leverages host security, virtual machines, and non-bypassable encryption implemented in trusted hardware. PICO can be deployed using either set of assumptions, although some aspects of the protocol (including trusted hardware) are not needed if all clients are assumed to be correct.

IV. CRYPTOGRAPHIC RESOURCEES

PICO makes use of two threshold cryptosystems: a threshold digital signature scheme (used to enforce correct client behavior and facilitate efficient reconciliation) and a threshold key generation scheme (used to generate the shared group key that group members use to encrypt application-level messages for one other). We now describe both cryptosystems and their associated security properties. We also describe the way in which PICO makes use of a public key infrastructure for simple message signing.

Threshold digital signatures: A $(k,\ n)$ threshold digital signature scheme allows a set of k out of n processes to generate a digital signature; any set of fewer than k processes is unable to generate a valid signature. When $k \geq f+1$, where f is the maximum number of processes that may be malicious, generating a threshold signature on a message implies that at least one correct process participated in the protocol and assented to the content of the message.

In a typical threshold signature scheme, a private key is divided into n key shares, where each process knows one key share. To sign a message, m, each process uses its key share to generate a *partial signature* on m. Any process that collects k partial signatures can then combine them to form a threshold signature on m. An important property provided by some threshold signature schemes, especially in malicious environments, is verifiable secret sharing [25]: each process can use its key share to generate a proof of correctness, proving that the partial signature was properly generated using a share from the initial key split.

Our current implementation of PICO uses the Shoup RSA threshold digital signature scheme [26]. The signatures generated using this scheme are standard RSA signatures [27], which can be verified using the public key corresponding to the divided private key. The scheme assumes a trusted dealer to divide the private key and securely distribute the initial key shares (after which the dealer is no longer needed), and it provides verifiable secret sharing.

Threshold key generation: A $(k,\ n)$ threshold key generation scheme allows a set of k out of n processes to generate a group encryption key, while any set of fewer than k processes is unable to do so. Similar to the case of threshold digital signatures, setting $k \geq f+1$ ensures that the group key was generated using a share from at least one correct process.

PICO uses the Diffie-Hellman based threshold coin-tossing scheme of Cachin, Kursawe, and Shoup [2] for key generation; the coin-tossing scheme was adapted for the group membership problem by Dutertre et al. [3]. A trusted dealer generates n shares of an initial secret (as in [28]) and securely distributes one share to each process (after which the dealer is no longer needed). To generate a group key, each process computes a *key share* as a function of its secret share and some common state. In PICO, this common state is based on the current group membership. Any process that combines k key shares can combine them to form the group key. As in [26], the scheme provides verifiable secret sharing, allowing each process to generate a proof that its key share was created using a valid secret share.

Public Key Infrastructure: Each process has a public/private key pair signed by a trusted certification authority. We employ digital signatures, and we make use of a cryptographic hash function for computing message digests. We denote a message m signed by process i as $\langle m \rangle_{\sigma_i}$. We assume that all adversaries, including faulty controllers and clients, are computationally bounded such that they cannot subvert these cryptographic mechanisms.

V. SYSTEM ARCHITECTURE AND DESIGN

In this section we describe the PICO architecture and its security properties. We then discuss the design of one of the core algorithmic components of PICO, the *group controller coordination protocol*.

A PICO group consists of a collection of clients that share an encryption key, which the clients use to protect their application-level data. This key is dynamically constructed by PICO and is dynamically changed when the group membership changes. A pre-defined set of group controllers is responsible for providing security services to the clients, including handling join and leave requests according to group policy and distributing shares of the group key to the group members. Each group member is presented with a *view* of the membership, which is a list of the processes currently in the group. Any change in group membership will be acompanied by a key change.

The PICO architecture is inspired by the architecture of the Intrusion-Tolerant Enclaves protocol [3]. It has the following security goals:

PROPERTY 5.1: VALID AUTHENTICATION – *Only an authorized client can join the group.*

Protocol Step	Entity Taking Action
1. Client submits request to group controllers	Joining or leaving client
2. Request validation	Each group controller that receives the client request
3. Group Controller Coordination Protocol	All group controllers
4. Key share generation and dissemination	Each group controller that accepts the operation
5. Combining of key shares, group key generation	Trusted hardware of each group member

Fig. 1: Outline of the PICO protocol.

PROPERTY 5.2: SECURE-KEYING – *If group member i is given $f+1$ shares for group encryption key k for view v, only the members of v will ever generate k.*

Figure 1 presents an outline of the PICO protocol. When a client wants to join or leave the group, it sends a request to the group controllers. If a group controller determines that the request is authorized (i.e., if it *approves* the request), it proposes that the request be agreed upon by sending a message in the group controller coordination protocol. A controller *accepts* the requested operation when it becomes agreed upon as a result of the coordination protocol. Once a controller accepts an operation, it updates its view of the group membership and sends a message, containing a share of the group key, to each group member. The message is encrypted with the public key of the trusted hardware of the receiving group member. Each group member combines a threshold number of key shares (in its trusted hardware) to construct the group key.

A critical property of the threshold key generation protocol is that, in order for key shares to be combinable, they must be computed based on some common state. In PICO, the common state on which the controllers compute their key shares is the set of operations (join and leave requests) that have been accepted. Thus, the group controller coordination protocol must facilitate agreement, among the group controllers, on the set of accepted operations.

Several factors make Intrusion-Tolerant Enclaves unsuitable for use in the PICO environment. First, the coordination protocol is not partitionable. Although it leverages weak semantics to avoid synchrony assumptions, it still requires collecting messages from all correct servers ($N-f$) in order to guarantee that a new join or leave request can be accepted. Second, we identified a flaw in the coordination protocol where, simply due to network asynchrony, there are scenarios in which an authorized client will never be admitted into the group. Due to space limitations, we describe this flaw in the extended version of this paper [29]. Finally, the coordination protocol assumes reliable communication links between correct servers; *all* protocol-level messages must eventually be delivered in order to ensure that all valid operations are eventually agreed upon.

In both Intrusion-Tolerant Enclaves and PICO, key shares are only guaranteed to be combinable when the membership stabilizes. If join and leave requests are continuously submitted too quickly, then there is the potential for livelock if the controllers are unable to converge on the set of accepted operations. This is the price of forgoing the total ordering of SMR. Note, however, that a steady stream of joins and leaves would cause the encryption key to change very rapidly even if SMR were used for coordination. Therefore, in practice these systems must be augmented with mechanisms to rate limit the joins and leaves from both correct and faulty processes.

To capture this requirement in PICO, we define a partition P as *stable* with respect to time t if no client in P submits a new join or leave request after t. In practice, we want to provide liveness during sufficiently long stable periods. PICO guarantees the following liveness property:

PROPERTY 5.3: PICO-LIVENESS – *Let P be a partition with at least $f+1$ correct group controllers, where P is stable at time t. Let M be the set of correct clients in P whose last submitted operation is a join. Then there exists a time $t' > t$ after which the members of M share an encryption key.*

VI. THE PICO PROTOCOL

In this section we describe the PICO protocol in detail. In Section VI-A, we introduce the terminology used in our protocol description, and we present several key data structures. In Section VI-B, we present the three basic components of PICO: the *client protocol*, used to join or leave the group; the *group controller coordination protocol*, used to agree upon join and leave requests; and the *rekey protocol*, used to generate a new group key when the membership changes. Section VI-C addresses the problem of how a client can determine which encryption key is the most recent, which is made difficult by the fact that operations are not totally ordered and communication is asynchronous. Section VI-D presents techniques for efficient state reconciliation and garbage collection. Finally, Section VI-E discusses how the PICO architecture can support process ejections.

A. Terminology and Data Structures

As mentioned above, the group controllers must agree on the set of *operations* (join and leave requests) that have been accepted. Operations are uniquely identified by *(clientID, operationID)* pairs. PICO enforces that clients submit operations with increasing, contiguous operation identifiers, beginning with 1, which must correspond to a join request. As explained below, this prevents faulty clients from prematurely exhausting the space of operation identifiers, and it allows for the use of cumulative threshold-signed proofs for efficient state reconciliation. All valid join operations have odd identifiers, and all valid leaves have even identifiers.

Each controller maintains the state of accepted operations in an array, *lastOpsAccepted[]*, where lastOpsAccepted[i] contains the operation identifier of the last operation that the controller has accepted for client i. By agreeing on lastOpsAccepted[], the controllers implicitly agree on the current

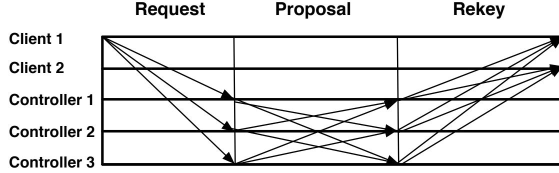

Fig. 2: Basic operation of the PICO protocol, with $f = 1$. Client 1 is requesting a new operation; Client 2 is already in the group. When a controller collects $f + 1$ valid PROPOSAL messages, it accepts the requested operation and sends a REKEY message to the requesting client and all current group members. The REKEY sent to Client 1 only contains a key share if this is a join request. The REKEY sent to existing group members (i.e., Client 2) contains an updated key share to reflect the new group membership.

membership of the group: Client i is currently in the group if lastOpsAccepted[i] corresponds to a join operation. In addition, the controllers implicitly agree on the total number of operations that have been accepted for all clients, which (following [3]) we call the *view number*. As described in Section VI-C, clients use the view number to determine which group encryption key is the most up to date.

B. Basic Protocol Operation

Figure 2 depicts the basic protocol operation of PICO. When a client wants to join or leave the group, it broadcasts a REQUEST message to the group controllers. As we describe below, although the client broadcasts the REQUEST, PICO provides liveness as long as the message is received by at least $f + 1$ correct controllers in the partition to which the client belongs. The group controllers then exchange PROPOSAL messages to agree to accept the requested operation. Upon accepting the operation, the group controllers send a REKEY message to the client and all current group members. We now examine each phase of the protocol in more detail.

Client Protocol: When client i wants to join or leave the group, it broadcasts a $\langle$REQUEST, opID, proof$\rangle_{\sigma_i}$ message to the controllers. The *opID* field is the operation identifier chosen by the client for this operation. If this request has an operation identifier of 1, then the *proof* field is empty. Otherwise, *proof* is a threshold-signed proof that operation (i, *opID* $- 1$) was legitimately accepted by at least one controller. Thus, to request an operation with identifier j, the client must present proof that operation $j - 1$ was accepted.

After submitting the request, the client waits for $f+1$ valid REKEY messages from the group controllers, indicating that they have accepted the operation. The responses contain partial signatures that can be combined to generate proof that the operation was accepted. In addition, if the operation was a join request, the responses contain key shares that can be combined to form the group encryption key. The client retransmits its request if it does not receive the necessary replies within a timeout period.

Group Controller Coordination Protocol: Upon receiving REQUEST message r from client i, controller c performs the following validation steps. In each step, if the validation fails, the request is discarded.

1) Verify the signature on r using client i's public key, and consult the group policy to determine if the operation is authorized.
2) If r should contain a proof, confirm that one is present.
3) If r contains a proof, verify it using the group's public key, and confirm that it proves that operation (i, *opID* $- 1$) was accepted.
4) If c has already accepted an operation (i, j), $j >$ opID, discard the request, because (i, j) must have already been accepted.

If all of the above checks succeed, then controller c broadcasts a $\langle$PROPOSAL, clientID, opID, partialSig$\rangle_{\sigma_c}$ message to the rest of the controllers. The *clientID* and *opID* fields uniquely identify the requested operation. The *partialSig* field is a partial signature computed over the hash of the (clientID, opID) pair, along with a proof that the partial signature was computed correctly.

A controller considers a PROPOSAL message as valid if it is properly signed and contains a partial signature with a valid correctness proof. Upon collecting $f + 1$ valid PROPOSAL messages for operation (i, j) from distinct controllers, a controller accepts the operation and takes several steps. First, it combines the partial signatures to construct a threshold-signed proof that (i, j) was legitimately accepted. Since this proof is on a single operation, we refer to it as a *singleOp proof*. As described in Section VI-D, the singleOp proof can be passed to other controllers to convince them that the operation was legitimately accepted. Second, the controller sets lastOpsAccepted[i] to j and updates the view number. Finally, the controller performs the requested operation by either adding client i to, or removing client i from, the membership list.

The group controller coordination protocol (GCCP) meets the following two correctness properties:

PROPERTY 6.1: GCCP-VALIDITY – *If some correct controller accepts operation* (i, j)*, then some (potentially different) correct controller approved the operation.*

PROPERTY 6.2: GCCP-AGREEMENT – *If some correct controller in partition* P *accepts operation* (i, j)*, then all correct controllers in* P *eventually accept the operation.*

Observe that the group controller coordination protocol requires a controller to collect only $f+1$ matching PROPOSAL messages in order to accept an operation, instead of the typical ($N - f$) messages required by Byzantine fault-tolerant state machine replication protocols and Intrusion-Tolerant Enclaves. The implication of this difference is that PICO guarantees that any partition with at least $f+1$ correct controllers can accept new join and leave operations, provided there is sufficient connectivity among the controllers and clients. More formally:

PROPERTY 6.3: GCCP-LIVENESS – *Let* P *be a partition with at least* $f + 1$ *correct group controllers. Then if a correct client in* P *submits an operation* (i, j)*, some correct controller in* P *accepts the operation.*

If $N > 3f+1$, then multiple partitions, operating in parallel, can guarantee the liveness of join and leave requests. The controllers eventually agree on the set of accepted operations. This is a weaker agreement problem than consensus, because controllers never need to make an irrevocable decision; they give their best guess of what the current set is and only need to converge eventually. This allows PICO to circumvent the FLP impossibility result [30] and guarantee safety and liveness without relying on synchrony.

Rekey Protocol: After accepting an operation, controller c generates a $\langle$REKEY, partialSig, lastOpsAccepted, keyShare$\rangle_{\sigma_c}$ message. The *partialSig* field is a partial signature computed over the hash of c's lastOpsAccepted[] data structure. There are two cases to consider. If the operation being accepted is a join, then *keyShare* is a key share computed over the hash of lastOpsAccepted[], and the REKEY message is sent to all current group members, including the client that just joined. If the operation being accepted is a leave, then controller c generates two distinct REKEY messages. The first is sent only to the leaving group member and does *not* contain a key share; this message serves only to allow the leaving member to obtain proof that the leave operation was accepted. The second REKEY message contains a new key share and is sent to all remaining group members. To overcome message loss, a controller periodically retransmits the REKEY messages for its last accepted operation.

A client validates a REKEY message by verifying the signature, along with the proof of correctness of the partial signature and the key share (if one is present). When a client collects $f+1$ valid REKEYs for the same *lastOpsAccepted* data, from distinct controllers, it first combines the partial signatures to form a threshold-signed proof reflecting the acceptance of the operation. Since this proof is generated on the array of last accepted operations, we refer to it as an *arrayOp proof*. We denote the ith entry of proof p as $p[i]$. If the REKEY messages contain key shares, the client combines them to compute the group encryption key. We refer to the sum of the entries in the arrayOp proof on which the key shares were computed as the view number of the key.

C. Choosing an Encryption Key

In this section we address the following practical problem: Given that client requests are not totally ordered, and that clients collect key shares asynchronously, how does a client know which group encryption key is the most up-to-date? Our solution is to leverage the threshold cryptographic proofs already used by the protocol so that a client can choose the correct key by using the one with the highest view number.

Recall that a REQUEST message sent by client i for operation j contains an arrayOp proof, p, where $p[i] = j - 1$. More generally, $p[k]$ contains the last accepted operation for client k at the time the REKEY messages containing the partial signatures combined to form p were generated. Thus, proof p can be viewed as a *snapshot* of the state of $f + 1$ group controllers, at least one of which is correct. Therefore, a controller receiving a REQUEST message containing p knows that, if $p[m] = n$, then the operation $(m,\ n)$ was legitimately accepted in the controller coordination protocol. Further, since we force clients to use contiguous sequence numbers, all operations $(m,\ n'), n' < n$, have been legitimately accepted (i.e., the proof is *cumulative*).

The preceding discussion implies that group controllers can use the proofs contained in REQUEST messages to perform reconciliation on the set of accepted operations. Upon receiving a $\langle$REQUEST, opID, p$\rangle_{\sigma_i}$ message from client i, a controller performs the following two steps (in addition to those described in Section VI-B). First, for each client k, the controller sets lastOpsAccepted[k] to $\max($lastOpsAccepted[k], $p[k])$. We say that the controller *applies* the arrayOp proof to its data structures. Second, if any entry in lastOpsAccepted[] changed, the controller updates the view number and membership list, and it computes a new REKEY message. We also impose the rule that a client only processes a REKEY message if the view number implied by the lastOpsAccepted field is higher than the view number of the last group key it adopted.

Each group member periodically broadcasts the arrayOp proof corrresponding to its current group key in a reconciliation message, $\langle$RECONC, proof$\rangle_{\sigma_i}$. When a controller receives a RECONC message, it applies the proof to its data structures and generates a new REKEY message if it learned of new accepted operations. Thus, when client c moves from one partition to another, it carries with it the snapshot (i.e., the proof) corresponding to key it is currently using. Eventually, the clients in the new partition will either adopt a key with the same view number as the one c was using (in which case they will install the exact same membership as c) or a greater view number (in which case they all converge on a new membership). We formalize this property as:

PROPERTY 6.4: REKEY-FORWARD-PROGRESS: *Let P be a partition with at least $f + 1$ correct group controllers. If a correct client in P ever successfully generated a group key with view number v, then there exists a time after which each correct group controller in P only sends* REKEY *messages corresponding to a view number $v' \geq v$.*

To help elucidate the intuition behind the mechanism described above, we conclude this section with an example. Figure 3 depicts a system with four clients, where the network is split into two partitions, A and B. Suppose all controllers in A agree on the set of accepted operations (with a lastOpsAccepted array of $[5, 4, 1, 0]$), all controllers in B agree on a different set of accepted operations ($[0, 1, 1, 1]$), and no new join or leave requests are submitted. Clients 1 and 2 are currently in partition A. Client 1 is using a group key corresponding to the array $[5, 4, 1, 0]$ (with a view number of 10). Client 2 is not currently a member of the group, and last had a group key corresponding to $[5, 3, 1, 0]$ (with a view number of 9). It has an arrayOp proof corresponding to $[5, 4, 1, 0]$, which it collected after completing the operation $(2,\ 4)$ (i.e., after it left the group). Clients 3 and 4 are in

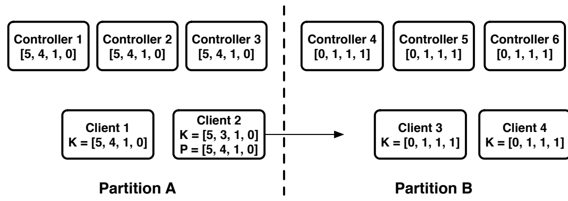

Fig. 3: A PICO system with 6 group controllers and four clients. Controllers 1, 2, and 3 have lastOpsAccepted $= [5, 4, 1, 0]$, and Controllers 4, 5, and 6 have lastOpsAccepted $= [0, 1, 1, 1]$. Client 1 is a member of the group and is using the key corresponding to the array $[5, 4, 1, 0]$. Client 2 is not a member of the group and last had a key corresponding to $[5, 3, 1, 0]$; it has an arrayOp proof for $[5, 4, 1, 0]$. Clients 3 and 4 are currently members of the group and share the key corresponding to $[0, 1, 1, 1]$.

partition B and are using a group key corresponding to the array $[0, 1, 1, 1]$ (with a view number of 3).

Now suppose client 2 moves to partition B. We would like the client to be able to share a group key with clients 3 and 4. Since client 2 was last using a group key with view number 9 in partition A, it must have an arrayOp proof, p, corresponding to a key with a view number of at least 9. In this case, p consists of the array $[5, 4, 1, 0]$ and a corresponding threshold signature. When client 2 requests to join in partition B, its REQUEST message contains p. After applying p, the controllers in partition B will update their view number to 11, since they compute the maximum of each slot in the array. Thus, when client 2's new join request is accepted, it will compute a group key based on the array $[5, 5, 1, 1]$, which has a view number of 12. In addition, clients 3 and 4 receive the corresponding REKEY messages (since they are members of the group) and will adopt the same group key.

D. *Reconciliation and Garbage Collection*

The constraints imposed by the MANET environment dictate that PICO should meet two important properties. First, it should not rely on reliable communication links. Given that message loss can be high and partitions long-lived, reliable links would consume bandwidth with acknowledgements and would require unbounded message queues. Second, PICO must provide efficient reconciliation when two partitions merge. Again, since partitions can be long-lived, PICO should specifically avoid passing all of the operations that were accepted in one partition to the other partition when the network heals.

We now describe how we use the threshold-signed proofs already in PICO to build a simple and efficient reconciliation and garbage collection mechanism. Each group controller maintains a data structure called a *Reconciliation Vector*, or RV. The RV is simply an array of proofs, where $RV[i]$ contains the proof reflecting the latest accepted operation for client i. For convenience, we denote the operation identifier of this operation as $RV[i]$.opID. Note that a proof might be a singleOp proof (constructed during the group controller coordination protocol) or an arrayOp proof (constructed by a client during the rekey protocol and passed to the controller in either a REQUEST or a RECONC message).

Each controller, c, periodically broadcasts the contents of its RV, wrapping each proof, p, in a $\langle \text{RECONC, p} \rangle_{\sigma_c}$ message. Upon receiving a RECONC message, a controller applies p, updating RV and lastOpsAccepted[] if p reflects more knowledge than what it currently has in its data structures. More formally, if p is a singleOp proof for operation $(i,\ j)$, then if $j >$ lastOpsAccepted[i], the controller replaces $RV[i]$ with p and sets lastOpsAccepted[i] to j. If p is an arrayOp proof, then for each slot k in p, if $p[k] >$ lastOpsAccepted[k], then the controller sets $RV[k]$ to p and lastOpsAccepted[k] to $p[k]$.

Since proofs are cumulative, PICO requires only the *last* reconciliation message to be received for each client in order to reconcile all of that client's accepted operations. This facilitates efficient reconciliation when two partitions merge; rather than requiring state proportional to the number of operations that were accepted in each partition to be transferred, each controller must transfer at most one message per client (multiple slots may have the same proof, which can be sent only once). This also makes the coordination protocol tolerant of message loss: once any correct controller in a partition, P, collects $f + 1$ PROPOSAL messages for an operation, $(i,\ j)$, all subsequent PROPOSAL messages for $(i,\ j)$ need not be delivered in order for all controllers in P to accept it.

Observe that PICO avoids the need for unbounded message queues. Each controller must retransmit at most one proof per client, and old PROPOSAL messages do not need to be reliably delivered. Thus, garbage collection in PICO is implicit and is done simply by updating the RV and discarding PROPOSAL messages for operations (i, j) if $RV[i]$.opID $> j$. In contrast, protocols requiring reliable links operating in a partitionable environment would require an explicit garbage collection mechanism to determine which messages had been delivered to all processes and could be deleted.

E. *Support for Process Ejection*

PICO can be extended to support the ejection (irreversible revocation) of both controller and client processes.

We first consider the ejection of faulty clients. We assume that some trusted entity generates and signs an ejection message, which contains the process identifier of the client being ejected. This entity can be made fault-tolerant via threshold cryptographic techniques. Ejection messages impact whether or not (1) a controller sends REKEY messages to a client, and (2) a controller processes a REQUEST message from a client. A correct controller never sends a REKEY message to a client it knows to be ejected, and it ignores subsequent REQUEST messages from clients it knows to be ejected.

Note, however, that correct controllers continue to accept join and leave operations for ejected clients when knowledge of these operations comes from any other source (i.e., in proofs received from other processes). In this way, the ejection does not impact the properties guaranteed by the rest of the protocol. The join/leave status agreed upon for an ejected client does not matter because clients are treated as group members only if (1) their last operation is a join and (2) they have not been ejected.

Group controllers within a partition must also converge on the set of ejected processes (in addition to the set of accepted operations). To facilitate this convergence, ejection messages can be periodically transmitted by extending the Reconciliation Vector to include the ejection status of each process.

PICO supports the ejection of group controllers in the same way. A correct process will ignore messages sent by an ejected controller. However, if too many group controllers are ejected, then PICO will no longer guarantee liveness. That is, PICO only guarantees liveness in partitions with at least $f+1$ correct (i.e., not faulty and not ejected) controllers.

VII. PERFORMANCE CONSIDERATIONS

The PICO protocol is being implemented as part of DARPA's Intrinsically Assurable Mobile Ad-Hoc Networks program. Although integration with our full system, ZODIAC, is not yet complete, in this section we briefly comment on some of the implementation and performance considerations of PICO. We first evaluate the cryptographic overhead of our implementation. We then describe a simple optimization that can be used to reduce the computational load.

Our implementation is written in C and uses the OpenSSL library [31]. We measured the latency of the different types of cryptographic operations when running on a 3.2 GHz, 64-bit Intel Xeon computer. Each computer can generate a 1024-bit standard RSA signature in 1.3 ms and verify a signature in 0.07 ms.

Threshold RSA Signatures: As described in Section VI, a group controller combines $f + 1$ partial signatures when it accepts an operation, and a client combines $f + 1$ partial signatures when its operation completes. We used the OpenTC implementation of Shoup's threshold RSA signature scheme [26]. The cost of generating a partial signature, along with its proof of correctness, was measured to be 3.9 ms. This cost remains fixed as the number of tolerated faults increases, because the number of exponentiations required to compute the partial signature remains the same.

On the other hand, the cost of combining $f+1$ partial signatures grows as f increases. We optimized for the common-case operation by attempting to combine partial signatures without first verifying their correctness proofs. If the resulting threshold signature verifies, then the shares were correct. However, if the signature does not verify, then we check each proof and can detect which shares were invalid. Since all messages are digitally signed, the invalid share can be broadcast as a proof that the corresponding controller is compromised, and the controller can subsequently be blacklisted. Using this technique, we measured the latency for combining to be 1.3 ms when $f = 1$, 2.1 ms when $f = 3$, and 3.4 ms when $f = 5$.

Threshold Key Generation: We implemented the threshold key generation scheme of Cachin, Kursawe, and Shoup [2]. We generated a 1024-bit safe prime and performed operations in its prime order subgroup. We measured the cost of generating a key share in this setting to be 11.3 ms. This cost is independent of the number of tolerated faults. The cost of combining the key shares into the group key increases as f increases. We measured the latency for combining to be 23.7 ms when $f = 1$, 50 ms when $f = 3$, and 91 ms when $f = 5$.

Aggregating Membership Changes: In many settings, join and leave operations are not likely to require real-time latencies. Therefore, we believe the latencies presented above are likely to be acceptable for many applications. Nevertheless, if membership changes are frequent, the cost of generating and combining partial signatures and key shares can become high. To help reduce this cost, a controller can aggregate several membership change operations before generating a REKEY message, which contains its partial signature and key share. This amortizes the cryptographic cost over several operations, reducing the average load per operation.

VIII. PROOF OF CORRECTNESS

A. *Proof of Liveness Properties*

Proof Strategy: We first prove GCCP-AGREEMENT (Property 6.2) and GCCP-LIVENESS (Property 6.3) of the group controller coordination protocol. Using these properties, we prove Lemma 8.1, which states that all correct controllers in a stable partition eventually converge on the set of accepted operations (i.e., their lastOpsAccepted[] data structures become identical). Once the correct controllers converge, we prove REKEY-FORWARD-PROGRESS (Property 6.4), which shows that correct controllers will eventually generate REKEY messages for a view number that will be adopted by the correct group members. The liveness of the overall PICO protocol, PICO-LIVENESS (Property 5.3), follows directly from these two properties.

Proof of GCCP-Agreement: When a correct controller, c, in partition P accepts operation $(i,\ j)$, it obtains a proof, p, that $(i,\ j)$ was legitimately accepted. We must show that all correct controllers in P eventually accept $(i,\ j)$. If c never accepts a later operation for i, then it continues to periodically retransmit p, which will eventually be received by all correct controllers in P. If c does accept a later operation for i, it will replace $RV[i]$ with a new proof, p', for some operation $(i,\ j')$. In turn, c may replace p' with a later proof, p'', and so on. Eventually, a correct controller will receive one of these proofs (call it p^*, for operation j^*), at which point it will implicitly accept all operations $(i,\ j'')$ with $j'' \leq j^*$, including $(i,\ j)$, because proofs are cumulative.

Proof of GCCP-Liveness: We must show that if client i submits request $(i,\ j)$ in a partition, P, with at least $f + 1$ group controllers, then $(i,\ j)$ will eventually be accepted. Client i periodically retransmits the request until it receives proof that $(i,\ j)$ was accepted. The request is eventually received by at least $f + 1$ correct group controllers, each of which will approve it and send a PROPOSAL for $(i,\ j)$. Each correct controller thus eventually receives at least $f + 1$ valid PROPOSALs from distinct controllers and will therefore accept the operation.

Lemma *8.1: Let P be a partition with at least $f + 1$ correct group controllers, where P is stable at time t. Then all correct group controllers in P eventually agree on the set of accepted operations.*

Proof of Lemma 8.1: Since P is stable, no new join or leave requests are submitted. By GCCP-LIVENESS, any pending operation from a correct client will eventually be accepted by some correct controller in P, and by GCCP-AGREEMENT, all correct controllers will eventually accept these operations. If any pending operation from a faulty client is accepted by a correct controller, all correct controllers in P will accept it.

For each client i, let i_c be the highest operation identifier for which a correct process in P has a proof, and let i_f be the highest operation identifier for which a faulty process in P has a proof. If $i_c \geq i_f$, then let r be a correct process in P that has proof that $(i,\ i_c)$ was accepted. Any other correct controller, s, will eventually accept this operation because r continues to retransmit the proof.

If $i_f > i_c$, then for each operation j, with $i_c < j \leq i_f$, a faulty process can either choose to make the proof of $(i,\ j)$ known to a correct process (in which case it will be accepted by all correct controllers) or it never makes the proof known. Thus, there exists some maximum such j that a faulty process makes known, which implies that the correct controllers eventually agree on the set of operations for which only faulty processes had proof of acceptance. Therefore, the correct controllers eventually agree on the set of accepted operations for each client.

Proof of Rekey-Forward-Progress: By Lemma 8.1, all correct group controllers in partition P eventually agree on the set of accepted operations. When each correct controller in P accepts the last operation, it generates a REKEY message with a key share based on the same membership as each other correct controller in P. Let v_{final} be the view number implied by the lastOpsAccepted field, L, of these REKEY messages. We must show that v_{final} will be at least as high as the view number, v, of the key currently being used by any of the correct group members. We can prove this by showing that no correct group member has proof of an operation $(i,\ j)$ where $j > L[i]$. The proof is by contradiction. If any correct group member had this proof, then it would eventually be received in a RECONC message by a correct controller, which would cause the controller to increase its view number and generate a REKEY message with a higher view number, which violates the assumption that v_{final} is the convergence point established by Lemma 8.1.

Proof of PICO-Liveness: By Lemma 8.1, all group controllers in a partition P eventually converge on the set of accepted operations and generate a REKEY message based on the same membership. Since there are at least $f + 1$ correct controllers in P, and since correct controllers periodically retransmit their last REKEY message, all correct group members will eventually collect $f + 1$ combinable REKEY messages based on the stable membership. By Property 6.4, the view number of this key, v_{final}, will be at least as high as the one currently being used by any correct group member. Any group member in M that previously had a group key with a view $v < v_{final}$ will adopt the group key corresponding to v_{final}. Any group member already using a key with a view number $v = v_{final}$ must already be using this group key, since otherwise there exists some operation that has not been converged upon. Since the convergence view is v_{final}, no correct controller sends a REKEY message corresponding to a higher view number, so all members of M will continue using the established group key.

B. Proof of Security Properties

Proof Strategy: We first prove GCCP-VALIDITY (Property 6.1), the validity property of the group controller coordination protocol. We then use this to prove VALID-AUTHENTICATION (Property 5.1), which states that only authorized clients are able to join the group. Finally, we prove SECURE-KEYING (Property 5.2), the security of the keying process.

Proof of GCCP-Validity: A correct controller accepts an operation $(i,\ j)$ after (1) collecting $f + 1$ PROPOSAL messages, (2) collecting a singleOp proof for operation $(i,\ j)$, or (3) collecting an arrayOp proof p with $p[i] \geq j$. In the first case, since at most f controllers are faulty, at least one correct controller sent a PROPOSAL and therefore approved the operation. In the second case, a singleOp proof is constructed by collecting $f + 1$ PROPOSAL messages, each with a partial signature on the hash of $(i,\ j)$. Again, since at most f controllers are faulty, at least one correct controller must have sent a PROPOSAL message that contributed to the construction of the singleOp proof.

In the third case, the arrayOp proof was constructed by collecting $f + 1$ REKEY messages. In each message, the ith entry of the lastOpsAccepted field contained $j' \geq j$. Thus, at least one correct controller had lastOpsAccepted$[i]\ = j'$. In order for $(i,\ j')$ to have been accepted, client i must have submitted a REQUEST containing proof that $(i,\ j' - 1)$ was accepted, which implies that at least one correct controller had lastOpsAccepted$[i]\ = j' - 1$. Using a simple induction, each operation from $(i,\ 1)$ through $(i,\ j')$ was accepted by at least one correct controller, including $(i,\ j)$. Consider the first correct controller to accept $(i,\ j)$. This controller must have done so through either Case 1 or Case 2, since no arrayOp proof, p, with $p[i] \geq j$, can yet exist. By Case 1 and Case 2 above, some correct controller must have sent a PROPOSAL message for $(i,\ j)$.

Proof of Valid-Authentication: By GCCP-VALIDITY, a client can only join the group if its operation was approved by some correct controller. A correct controller consults group policy in deciding whether to approve a client join request. Thus, only an authorized client can join the group.

Proof of Secure-Keying: We show that only members of a given view, v, can generate the group key k. Group member i uses its trusted hardware to encrypt messages with k. When i adopted k, it obtained an arrayOp proof, p, from which the current group view can be deduced. To obtain k, a process must combine $f + 1$ key shares all based on the

same lastOpsAccepted data, which is the same data as in p. A correct controller only sends a REKEY message containing a key share to the members of v. Each REKEY is encrypted with the public key of the trusted hardware of the receiving group member. Thus, a faulty client not in v will never be sent the necessary $f + 1$ REKEY messages. Faulty clients cannot decrypt the key shares of REKEY messages sent to correct clients. Further, since they cannot learn the decryption key of their own trusted hardware, even faulty group members cannot divulge their own key shares to processes not in v. The security of the keying process thus follows from the fact that only processes in v are able to generate k, and no process is able to learn k.

IX. CONCLUSION

This paper presented PICO, a distributed protocol that manages group membership and keying in mobile ad-hoc networks. PICO uses a weakly consistent Byzantine fault-tolerant agreement protocol to provide a partitionable service, and it leverages threshold cryptographic proofs to tolerate message loss and avoid requiring reliable communication links. We highlighted several pragmatic issues associated with integrating PICO as a component in a secure system, which must be addressed in practical deployments.

REFERENCES

[1] S. Alexander, S. Bellovin, Y.-H. Cheng, B. Coan, A. Ghetie, V. Kaul, N. F. Maxemchuk, H. Schulzrinne, S. Schwab, B. Siegell, A. Stavrou, and J. M. Smith, "The dynamic community of interest and its realization in ZODIAC," *IEEE Communications Magazine*, vol. 47, no. 10, accepted for publication, 2009.

[2] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in constantipole: Practical asynchronous byzantine agreement using cryptography (extended abstract)," in *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing (PODC '00)*, Portland, Oregon, 2000, pp. 123–132.

[3] B. Dutertre, V. Crettaz, and V. Stavridou, "Intrusion-tolerant enclaves," in *Proceedings of the 2002 IEEE Symposium on Security and Privacy (SP '02)*, 2002, p. 216.

[4] O. Rodeh, K. P. Birman, and D. Dolev, "The architecture and performance of security protocols in the ensemble group communication system: Using diamonds to guard the castle," *ACM Trans. Inf. Syst. Secur.*, vol. 4, no. 3, pp. 289–319, 2001.

[5] ——, "Using avl trees for fault tolerant group key management," *International Journal on Information Security*, vol. 1, pp. 84–99, 2002.

[6] Y. Amir, C. Nita-rotaru, J. Stanton, and G. Tsudik, "Secure spread: An integrated architecture for secure group communication," *IEEE Transactions on Dependable and Secure Computing*, vol. 2, pp. 248–261, 2005.

[7] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982.

[8] M. K. Reiter, "The Rampart Toolkit for building high-integrity services," in *Selected Papers from the International Workshop on Theory and Practice in Distributed Systems*, 1995, pp. 99–110.

[9] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "The SecureRing protocols for securing group communication," in *Proceedings of the IEEE 31st Hawaii International Conference on System Sciences*, vol. 3, Kona, Hawaii, January 1998, pp. 317–326.

[10] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999, pp. 173–186.

[11] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for Byzantine fault-tolerant services," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, USA, October 2003, pp. 253–267.

[12] J.-P. Martin and L. Alvisi, "Fast Byzantine consensus," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 202–215, 2006.

[13] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative Byzantine fault tolerance," in *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, Washington, USA, 2007, pp. 45–58.

[14] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Byzantine replication under attack," in *Proceedings of the 38th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '08)*, Anchorage, AK, USA, June 2008, pp. 197–206.

[15] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[16] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[17] Y. G. Desmedt and Y. Frankel, "Threshold cryptosystems," in *CRYPTO '89: Proceedings on Advances in Cryptology*, Santa Barbara, California, United States, 1989, pp. 307–315.

[18] L. Zhou and Z. J. Haas, "Securing ad hoc networks," *IEEE Network*, vol. 13, pp. 24–30, 1999.

[19] J. Kong, P. Zerfos, H. Luo, S. Lu, and L. Zhang, "Providing robust and ubiquitous security support for mobile ad-hoc networks," in *Proceedings of the 9th International Conference on Network Protocols (ICNP '01)*, 2001, pp. 251–260.

[20] H. Luo, P. Zefros, J. Kong, S. Lu, and L. Zhang, "Self-securing ad hoc wireless networks," in *Proceedings of the Seventh IEEE Symposium on Computers and Communications (ISCC '02)*, 2002.

[21] M. Narasimha, G. Tsudik, and J. H. Yi, "On the utility of distributed cryptography in p2p and manets: the case of membership control," in *Proceedings of the 11th International Conference on Network Protocols (ICNP '03)*, 2003, pp. 336–345.

[22] N. Saxena, G. Tsudik, and J. H. Yi, "Efficient node admission for short-lived mobile ad hoc networks," in *Proceedings of the 13th IEEE International Conference on Network Protocols (ICNP '05)*, 2005, pp. 269–278.

[23] B. Dutertre, H. Sadi, and V. Stavridou, "Intrusion-tolerant group management in enclaves," in *In International Conference on Dependable Systems and Networks (DSN '01)*, 2001, pp. 203–212.

[24] "Trusted platform module (TPM) specifications, http://www.trustedcomputinggroup.org/specs/tpm/."

[25] P. Feldman, "A practical scheme for non-interactive verifiable secret sharing," in *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*. Los Angeles, CA, USA: IEEE Computer Society, October 1987, pp. 427–437.

[26] V. Shoup, "Practical threshold signatures," *Lecture Notes in Computer Science*, vol. 1807, pp. 207–223, 2000.

[27] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 26, no. 1, pp. 96–99, 1983.

[28] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[29] J. Kirsch and B. Coan, "Intrusion-tolerant group management for mobile ad-hoc networks," Tech. Rep. CNDS-2009-2, Johns Hopkins University, www.dsn.jhu.edu, 2009.

[30] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[31] "The openssl project, http://www.openssl.org."