GREP: a Group REkeying Protocol Based on Member Join History

Marco Tiloca
SICS Swedish ICT AB, Security Lab
Isafjordsgatan 22, Kista (Sweden)
Email: marco@sics.se

Gianluca Dini
Dipartimento di Ingegneria dell'Informazione
University of Pisa, Largo Lazzarino 1, Pisa (Italy)
Email: gianluca.dini@ing.unipi.it

***Abstract*—This paper presents GREP, a highly scalable and efficient group rekeying protocol with the following merits. First, it rekeys the group with only two messages, introducing an overhead which is small, constant, and independent of the group size. Second, GREP considers collusion as a first-class attack. Third, GREP efficiently recovers the group from a collusion attack without recourse to a total member reinitialization. The recovery cost smoothly grows with the group size, and gradually increases with the attack severity. GREP achieves these results by organizing nodes into logical subgroups and exploiting the history of node joining events. This allows GREP to establish a total ordering among subgroups and among nodes in each subgroup, so making collusion recovery highly scalable and efficient. We evaluate performance from several standpoints, and show that GREP is deployable in large-scale networks of customary, even resource constrained, platforms.**

1. Introduction

Group communication is a powerful and efficient paradigm that can be used in a range of application scenarios, from wireless sensor networks (WSNs) to large scale distribution of contents. According to this model, a node becomes member of the group by explicitly joining it. After that, it may send/receive broadcast messages to/from other group members. Later on, the node may voluntarily leave the group or be forced to, if compromised or suspected so.

It is generally required that only group members can access group communication. To this end, group members secretly share a cryptographic *group key* to securely exchange messages in the group. When a node joins the group, it must be prevented from deciphering previous messages even if it has recorded them (*backward security*). When it leaves the group, or is forced to leave, the node must be prevented from further accessing group communication (*forward security*). Backward and forward security are generally guaranteed by *rekeying*. That is, when a node joins or leaves the group, the group key is revoked and a new one is distributed. In large dynamic groups where joining and leaving events are frequent, rekeying must be efficient and highly scalable.

Besides, a *collusion attack* occurs when multiple compromised group members share their security material, in order to regain access to the group key. No group rekeying scheme is exempt from collusion attacks, and different schemes display different levels of resilience. However, only a few of them consider collusion as a first-class attack, and provide countermeasures to recover from successful instances of this kind of attack. In many schemes, recovering from collusion requires a total member reinitialization, i.e. all non compromised group members have to be separately reinitialized, in a one to one fashion. It follows that the recovery overhead grows linearly with the group size, with negative impact on the overall system performance and scalability.

In this paper, we take these challenges and present GREP, a novel rekeying scheme for large-scale dynamic groups that levers on *logical subgrouping* and *join history*. Group members are partitioned into non overlapping logical subgroups that become the units of rekeying and collusion recovery. Unlike other schemes, subgroups only support efficient group key management, have no application meaning and are transparent to the application layer. Also, GREP exploits the *history* of joining events to establish a total ordering among subgroups and among nodes in each subgroup, in order to efficiently recover from collusion attacks.

GREP displays the following benefits. First, it is *secure*, as it fulfills the backward and forward security requirements. Second, it is *highly scalable*, as it requires a number of rekeying messages which is small, constant, and independent of the group size, i.e. $\mathcal{O}(1)$. Third, GREP has a $\mathcal{O}(\sqrt{n})$ storage and computing overhead, where n is the group size. This makes the scheme deployable on a large spectrum of platforms, including resource scarce sensor nodes. Fourth, GREP considers collusion as a first-class attack. Fifth, GREP efficiently recovers from collusion attacks, displaying a communication overhead that gradually increases with the attack severity, and grows as $\mathcal{O}(\sqrt{n})$ only in the unlikely worst case. This is possible by exploiting the history of joining events in the group. To the best of our knowledge, GREP is the first group rekeying protocol that exploits the join history to achieve highly efficient collusion recovery.

. This work was carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme. The research leading to these results has received funding from the *European Union* Seventh Framework Programme (*FP7/2007-2013*) under *grant agreement* n° 246016.

The paper is organized as follows. Section 2 discusses related works. Section 3 describes the system architecture. We present the GREP protocol in Section 4, provide a security analysis in Section 5, and evaluate performance in Section 6. Section 7 draws our conclusive remarks.

2. Related work

Like other rekeying schemes suitable for large groups [2] [3] [6] [7] [10], GREP takes a centralized approach to group key management and relies on logically organized *administrative keys* to provide scalable and efficient rekeying.

LKH organizes administrative keys in a hierarchical logical tree [3], where the root contains the group key, the leaves contain group members' individual keys, and the internal tree nodes contain additional administrative keys. Given a group size n and a balanced key tree with arity a, the leave communication overhead and the storage overhead at the node side grow as $\mathcal{O}(\log_a n)$. In case of collusion attack, LKH incurs the risk of a total member reinitialization when at least $\lceil \frac{n}{a} \rceil$ nodes are captured before they are detected.

Other schemes deriving from LKH have been proposed, but none of them achieves better performance [2] [6] [10]. Key Graphs is a generalization of LKH, where performance depends on the specific graph topology [2]. LARK is based on Key Graph, and relies on logically organizing administrative keys as a tool for application design [6]. That is, the key graph topology reflects cooperation within the group, and is considered to provide efficient rekeying. KTR adopts an approach similar to the one of LARK, and generalizes LKH to manage multiple subscriptions in content distribution applications and wireless broadcast services [10].

HISS [7] and GREP display similarities, as they both rely on logical subgrouping to support efficient and scalable rekeying. They achieve the same efficiency as to computing and storage overhead, i.e. $\mathcal{O}(\sqrt{n})$, and rekey the group with a number of messages which is reduced, constant, and independent of the group size. However, unlike HISS, GREP introduces the notion of member *join history*, and exploits it to recover from collusion attack in a much more efficient and scalable way. That is, the HISS recovery overhead always grows as $\mathcal{O}(n)$. Instead, the GREP recovery overhead gradually increases with the collusion attack severity, and grows as $\mathcal{O}(\sqrt{n})$ only in the unlikely worst case condition. Also, the chance of a total member reinitialization requires at least $2 \cdot \lceil \sqrt{n} \rceil$ colluding nodes, but is practically an unlikely event, so making GREP extremely efficient against collusion attacks even when several nodes collude.

3. System architecture

We consider a set G of nodes that communicate according to the *group communication* paradigm. A node becomes member of G by explicitly *joining* the group. Then, it may send/receive broadcast messages to/from other group members. If a member *leaves* the group, or is forced to, it cannot send/receive messages to/from the group anymore. Group members secretly share a cryptographic *group key* that they use to encrypt/decrypt messages within the group. We denote by K_G the group key associated to the group G. In general, it is required that the backward and forward security requirements are guaranteed [11]. In order to fulfill them, when a new node joins the group or a current member leaves it, the current group key is revoked and a new one is distributed. Hereafter, we refer to this operation as *rekeying*.

The group G is managed by a *Group Controller* (GC), which is composed of three services: i) a *Group Membership Service* (GMS); ii) a *Key Management Service* (KMS); and iii) an *Intrusion Detection Service* (IDS). In short, the GMS maintains the group membership by keeping track of nodes that join and leave. The IDS component monitors network activities to detect possible compromised nodes. Since there is no sure and efficient way to readily detect a single node capture [12], the IDS may report multiple compromised nodes at the same time. Upon detecting a set of compromised nodes G_c, the IDS notifies the GMS in order to have them evicted from the group. Further details about the IDS and the monitoring process are beyond the scope of this paper, and we refer the reader to, e.g., [5] [13]. Upon being notified of any membership change, the GMS activates the rekeying process. The KMS is responsible for performing such a task.

The GC is typically implemented according to a centralized approach, as a resourceful computing node which is generally more powerful than group nodes. In particular, the GC is considered trustworthy and properly designed, implemented and managed to be reliable and secure, hence practically infeasible to compromise. Although server security and reliability are still an open research issue, the literature provides well established techniques to keep servers secure [4] [9]. As an alternative, the GC can be practically implemented according to a distributed architecture, which is beneficial in terms of robustness and availability, and avoids a single GC instance from being a single point of failure. However, this requires that the different GC replicas are kept synchronized with one another, especially as to the current group membership and established key material. In this paper, we consider a centralized GC, and detail the *Key Manager* (KM) component implementing the KMS. Further details about practical architectural design choices for the GC are out of the scope of this work.

4. The rekeying protocol

The group G is *partitioned* into a set $\mathcal{S}$ of non empty subgroups, such that each group member is exactly in one of these subgroups. Subgroups have no meaning to applications and are never merged nor split. Each member of G is assigned to a given subgroup S upon joining the group, and is never moved to a different subgroup. We denote two nodes in the same subgroup as *cognates*.

GREP totally orders the members of each subgroup S according to their joining time, to reflect the *node join history* of S. In particular, upon joining a subgroup S, a node u is associated with a numeric node ID, nid_u, which

is unique within the subgroup, and reflects its members' total order. That is, given two nodes u and v in S, $nid_u < nid_v$ if and only if node u has joined S before node v. We refer to the nodes that have joined subgroup S before or after node u as *elder cognates* and *junior cognates* of u, respectively.

GREP also totally orders subgroups according to their addition time to group G, to reflect the *subgroup addition history* of G. In particular, upon its addition to the group G, a subgroup S is associated with a numeric subgroup ID, sid_S, which is unique within the group G, and reflects subgroups' total order. That is, given two subgroups S and S', $sid_S < sid_{S'}$ if and only if subgroup S has been added to G before subgroup S'. We refer to the subgroups that have been added to the group G before or after subgroup S as *elder kindreds* and *junior kindreds* of S, respectively.

Each node and subgroup is associated with secret quantities called *tokens*. That is, *node tokens* and *subgroup tokens* are associated with nodes and subgroups, respectively. Each node u in S is associated to two node tokens, i.e. a *forward node token* t_u^F and a *backward node token* t_u^B. Elder cognates of u store token t_u^F, while junior cognates of u store token t_u^B. Similarly, each subgroup S is associated to two subgroup tokens, i.e. a *forward subgroup token* st_S^F and a *backward subgroup token* st_S^B. All nodes in elder kindreds of S store st_S^F, while all nodes in junior kindreds of S store st_S^B. Also, every node u *a priori* shares a *node key* K_u with the KM. Every subgroup S is associated to a *subgroup key* K_S, which is shared between the KM and every node in the subgroup. Finally, the KM stores: i) all node tokens in the *Node Token Set* (NTS); ii) all subgroup tokens in the *Subgroup Token Set* (STS); iii) all node keys in the *Node Key Set* (NKS); and iv) all subgroup keys in the *Subgroup Key Set* (SKS). The KM and nodes keep tokens and keys secret.

Let us consider a node u in a subgroup S. Node u holds its node key K_u, the subgroup key K_S, and the node tokens associated to its cognate nodes. Such tokens are partitioned into two sets, i.e. *Backward Node Token Set* (NTS_u^B) and *Forward Node Token Set* (NTS_u^F). In particular, NTS_u^B includes all backward node tokens associated to the elder cognates of u. Instead, NTS_u^F includes all forward node tokens associated to the junior cognates of u. Also, node u holds the subgroup tokens of all subgroups belonging to the absolute complement of S in $\mathcal{S}$. Such tokens are partitioned into two sets, i.e. *Backward Subgroup Token Set* (STS_S^B) and *Forward Subgroup Token Set* (STS_S^F). In particular, STS_S^B includes all backward subgroup tokens associated to the elder kindreds of S, while STS_S^F includes all forward subgroup tokens associated to the junior kindreds of S.

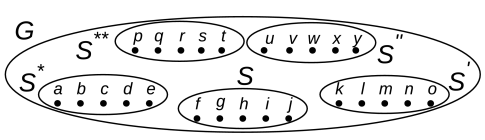

Figure 1. A group G partitioned into five subgroups.

To fix ideas, we consider the example in Figure 1, where a group G is partitioned into five subgroups, each one including five nodes. Figure 2(A) shows the *node join history* of subgroup S, while Figure 2(B) shows the *subgroup addition history* of G. Finally, Table 1 shows the node and subgroup tokens held by nodes in subgroups S^*, S and S'.

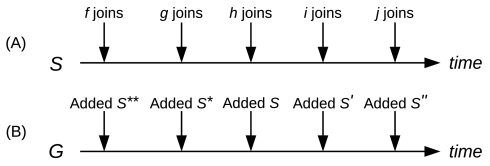

Figure 2. (A) Node join history of S; (B) Subgroup addition history.

TABLE 1. NODE AND SUBGROUP TOKENS.

Subgroup	Node	NTS^B	NTS^F	STS^B	STS^F
S^*	a	–	$t_b^F, t_c^F, t_d^F, t_e^F$	$st_{S^{**}}^B$	$st_S^F, st_{S'}^F, st_{S''}^F$
	b	t_a^B	t_c^F, t_d^F, t_e^F	$st_{S^{**}}^B$	$st_S^F, st_{S'}^F, st_{S''}^F$
	c	t_a^B, t_b^B	t_d^F, t_e^F	$st_{S^{**}}^B$	$st_S^F, st_{S'}^F, st_{S''}^F$
	d	t_a^B, t_b^B, t_c^B	t_e^F	$st_{S^{**}}^B$	$st_S^F, st_{S'}^F, st_{S''}^F$
	e	$t_a^B, t_b^B, t_c^B, t_d^B$	–	$st_{S^{**}}^B$	$st_S^F, st_{S'}^F, st_{S''}^F$
S	f	–	$t_g^F, t_h^F, t_i^F, t_j^F$	$st_{S^{**}}^B, st_{S^*}^B$	$st_{S'}^F, st_{S''}^F$
	g	t_f^B	t_h^F, t_i^F, t_j^F	$st_{S^{**}}^B, st_{S^*}^B$	$st_{S'}^F, st_{S''}^F$
	h	t_f^B, t_g^B	t_i^F, t_j^F	$st_{S^{**}}^B, st_{S^*}^B$	$st_{S'}^F, st_{S''}^F$
	i	t_f^B, t_g^B, t_h^B	t_j^F	$st_{S^{**}}^B, st_{S^*}^B$	$st_{S'}^F, st_{S''}^F$
	j	$t_f^B, t_g^B, t_h^B, t_i^B$	–	$st_{S^{**}}^B, st_{S^*}^B$	$st_{S'}^F, st_{S''}^F$
S'	k	–	$t_l^F, t_m^F, t_n^F, t_o^F$	$st_{S^{**}}^B, st_{S^*}^B, st_S^B$	$st_{S''}^F$
	l	t_k^B	t_m^F, t_n^F, t_o^F	$st_{S^{**}}^B, st_{S^*}^B, st_S^B$	$st_{S''}^F$
	m	t_k^B, t_l^B	t_n^F, t_o^F	$st_{S^{**}}^B, st_{S^*}^B, st_S^B$	$st_{S''}^F$
	n	t_k^B, t_l^B, t_m^B	t_o^F	$st_{S^{**}}^B, st_{S^*}^B, st_S^B$	$st_{S''}^F$
	o	$t_k^B, t_l^B, t_m^B, t_n^B$	–	$st_{S^{**}}^B, st_{S^*}^B, st_S^B$	$st_{S''}^F$

Hereafter, we adopt the following notation. By $P \rightarrow u : m$ we denote a principal P sending a unicast message m to node u. By $P \twoheadrightarrow S : m$ we denote P that broadcasts a message m to (sub)group S. We denote by $H(\cdot)$ a one-way hash function, and by $KDF(\cdot)$ a pseudo-random key derivation function that derives one cryptographic key from a secret value. By $\{x\}_K$, we denote the symmetric encryption of x by means of key K. We assume that cryptographic primitives are secure, and secrets have a size that discourages an exhaustive search, thus no analytical attack against rekeying and data traffic is practically feasible. Due to space constraints, we do not cover how to assure integrity and authenticity of rekeying messages. Possible mechanisms to provide them are digital signatures and hash-chains [6].

4.1. Rekeying upon joining

Let us consider a node u joining the group G. We assume that u is not malicious or compromised, and has been authorized to join the group by the GMS. Due to space constraints, we consider the association of u to an already existing subgroup S. The KM renews the group security material as follows, in order to assure *backward security*.

The KM randomly generates a *refresh key* K_R, a node key K_u, a backward node token t_u^B and a master node token t_M. Then, it derives a forward node token $t_u^F = KDF(t_M || K_R)$. After that, it determines a node ID nid_u associated to u, and computes a new group key

$K_G^+ = KDF(K_G||K_R)$ and a new subgroup key $K_S^+ = KDF(K_S||K_R)$. Finally, the KM broadcasts the following messages:

$$\text{JM1} : KM \twoheadrightarrow S : < nid_u, \{t_M, K_R\}_{K_S} >$$
$$\text{JM2} : KM \twoheadrightarrow G : < \{K_R\}_{K_G} >$$

That is, the KM rekeys S by means of JM1, and the remaining subgroups by means of JM2. Then, it installs K_G^+ as the current group key, and K_S^+ as the current subgroup key of subgroup S. Finally, it adds t_u^B and t_u^F to NTS, and discards t_M and K_R.

Upon receiving message JM1, any node v in S uses K_S to retrieve t_M and K_R. Then, it derives $t_u^F = KDF(t_M||K_R)$ and adds it to NTS_v^F. Also, it generates the two keys $K_G^+ = KDF(K_G||K_R)$ and $K_S^+ = KDF(K_S||K_R)$, and installs them as the current group key and subgroup key, respectively. Finally, it discards t_M and K_R. Upon receiving message JM2, any node v in S', $S' \neq S$: i) uses K_G to retrieve K_R; ii) generates $K_G^+ = KDF(K_G||K_R)$ and installs it as the current group key; and iii) discards K_R.

Before joining the group, u initializes its token sets as empty sets. Then, the KM provides u with K_u, K_G^+ and K_S^+. Upon receiving them, u installs K_u as its own node key, K_G^+ as the current group key, and K_S^+ as the current subgroup key. Then, node u receives from the KM the backward node tokens associated to its cognates in S, and adds them to NTS_u^B. Finally, the KM provides u with the backward and forward subgroup tokens associated to the elder and junior kindreds of S. Upon receiving them, u accordingly adds the subgroup tokens to STS_S^B and STS_S^F. We assume that node u receives the cryptographic material through a pre-existing secure channel, so assuring authentication and confidentiality. Possible implementations include a pre-shared cryptographic key or out-of-band means.

4.2. Rekeying upon leaving

With reference to Figure 1, let us suppose that node h in subgroup S leaves the group G. Then, all the cryptographic material held by h, including the group key K_G, gets compromised and must be revoked. In particular, a new group key must be distributed to all nodes in G but h. Due to space constraints, we do not discuss the case when h is the only member of subgroup S, which never becomes empty.

To rekey the group in a scalable way, the KM uses tokens, and bases the rekeying on the following observation. When node h leaves the group, all the tokens it holds get compromised (see Table 1). However, by construction, four tokens remain secret, i.e. t_h^B, t_h^F, st_S^F and st_S^B. The KM can thus rely on these tokens to efficiently rekey the group. In fact, i) all elder cognates of h (nodes f and g) hold t_h^F; ii) all junior cognates of h (nodes i and j) hold t_h^B; iii) nodes in all elder kindreds of S (subgroups S^* and S^{**}) hold st_S^F; iv) nodes in all junior kindreds of S (subgroups S' and S'') hold st_S^B; v) node h does not know either t_h^F or t_h^B; and vi) no node in S, including node h, holds st_S^F or st_S^B.

Practically, the KM randomly generates a refresh key K_R, and computes the new group key $K_G^+ = KDF(K_G\|K_R)$ and the new subgroup key $K_S^+ = KDF(K_S\|K_R)$. Then, it computes the four key encryption keys $K_F = KDF(t_h^F)$, $K_B = KDF(t_h^B)$, $K_F^S = KDF(st_S^F)$, and $K_B^S = KDF(st_S^B)$. Finally, it broadcasts the following messages:

$$\text{LM1} \quad KM \twoheadrightarrow S : < nid_h, \{K_R\}_{K_F}, \{K_R\}_{K_B} >$$
$$\text{LM2} \quad KM \twoheadrightarrow G : < sid_S, \{K_R\}_{K_F^S}, \{K_R\}_{K_B^S} >$$

That is, the KM rekeys S by means of LM1, and the remaining subgroups by means of LM2. Then, it installs K_G^+ as the current group key and K_S^+ as the current subgroup key of subgroup S. Then, it updates its node token set and subgroup token set by means of K_R, as follows. First, it removes the two node tokens t_h^B and t_h^F associated to h from its node token set NTS. Then, it updates all the node tokens in NTS associated to the remaining nodes in S, i.e. $\forall\ t_u \in NTS$, s.t. u belongs to S and $u \neq h$, $t_u \leftarrow H(t_u||K_R)$. Also, the KM updates all subgroup tokens in its subgroup token set STS, i.e. $\forall\ st \in STS, st \leftarrow H(st||K_R)$. Finally, the KM discards K_R, K_F, K_B, K_F^S and K_B^S.

Upon receiving message LM1, any node u in S, $u \neq h$, computes either $K_F = KDF(t_h^F)$, if $nid_u < nid_h$, or $K_B = KDF(t_h^B)$ otherwise. Then, it retrieves K_R, and computes $K_G^+ = KDF(K_G\|K_R)$ and $K_S^+ = KDF(K_S\|K_R)$. Such keys are installed as the current group key and subgroup key, respectively. Also, u removes either t_h^B from the node token set NTS_u^B or t_h^F from the node token set NTS_u^F. Then, u updates all tokens in its token sets, i.e. $\forall\ TS \in \{NTS_u^B, NTS_u^F, STS_S^B, STS_S^F\}, \forall\ t \in TS, t \leftarrow H(t\|K_R)$. Finally, u discards K_R, as well as either K_F or K_B.

Upon receiving message LM2, any node v in S', $S' \neq S$, computes either $K_F^S = KDF(st_S^F)$, if $sid_{S'} < sid_S$, or $K_B^S = KDF(st_S^B)$ otherwise. Then, it retrieves the key K_R, computes $K_G^+ = KDF(K_G\|K_R)$, and installs it as the current group key. Also, v updates its subgroup token sets, i.e. $\forall\ TS \in \{STS_{S'}^B, STS_{S'}^F\}, \forall\ t \in TS, t \leftarrow H(t\|K_R)$. Finally, node v discards K_R, as well as either K_F^S or K_B^S.

4.3. Recovering from collusion attack

In case of collusion attack, multiple compromised nodes may share their individual pieces of information to regain access to the group key. The compromised nodes can all belong to the same subgroup, or be spread among different subgroups. Hereafter, we call *compromised subgroup* any subgroup containing at least one compromised node. Generally, recovering from collusion might require a *total member reinitialization*, i.e. all non compromised nodes in the group require to be reinitialized one by one. This would greatly limit efficiency and scalability of the rekeying process.

On the contrary, GREP efficiently recovers from collusion attacks, by following two intuitive observations. First, with reference to Figure 2, let us assume that nodes g and i, $nid_g < nid_i$, belonging to subgroup S, are colluding, i.e.

an adversary can collect all the tokens and keys they hold. By construction, all node tokens in S get compromised but two, i.e. t_i^B and t_g^F (see Table 1). That is, all junior cognates of i, e.g. node j, hold the former, while all elder cognates of node g, e.g. node f, hold the latter. Second, with reference to Figure 2, let us assume that S^* and S', $sid_{S^*} < sid_{S'}$, are compromised. By construction, all subgroup tokens get compromised but two, i.e. $st_{S'}^B$ and $st_{S^*}^F$ (see Table 1). That is, all junior kindreds of S', e.g. S'', hold the former, while all elder kindreds of S^*, e.g. S^{**}, hold the latter.

Let us denote with G_c the set of colluding nodes to be evicted, with $\mathcal{C}$ the set of compromised subgroups, and with $\mathcal{U} = \mathcal{S} \setminus \mathcal{C}$ the set of non compromised subgroups. Due to space constraints, we assume that: i) all compromised subgroups include at least one non compromised node, i.e. no compromised subgroups become empty after the recovery has been completed; and ii) no previously evicted nodes collude with nodes in G_c. Then, the KM revokes the current group key K_G and distributes a new one K_G^+ as follows.

1) Initially, the KM randomly generates a refresh key K_R. Then, it computes a new group key $K_G^+ = KDF(K_G||K_R)$ and installs it as the current group key.

2) The KM rekeys the compromised subgroups as follows. For each subgroup $S \in \mathcal{C}$, it determines: i) the subset S_c of compromised nodes in S; and ii) u_y^S and u_e^S, i.e. the youngest and the eldest node in S_c, respectively. Practically, the set S_c can be represented as a list of node IDs. According to the first observation, node tokens $t_{u_e^S}^F$ and $t_{u_y^S}^B$ are not compromised. Also, all elder cognates of u_e^S hold token $t_{u_e^S}^F$, while all junior cognates of u_y^S hold token $t_{u_y^S}^B$. Hence, these tokens can be used to rekey these nodes as follows. The KM generates the key encryption keys $K_F = KDF(t_{u_e^S}^F)$ and $K_B = KDF(t_{u_y^S}^B)$, and broadcasts the message

$$\text{RM1} \quad KM \twoheadrightarrow S : < S_c, \{K_R\}_{K_F}, \{K_R\}_{K_B} >$$

Every non compromised node u which is elder cognate of u_y^S and junior cognate of u_e^S holds only compromised tokens. Thus, it must be rekeyed in a one-to-one fashion, by means of its user key K_u. Then, $\forall\ u$ in $S \setminus S_c$, such that $nid_{u_e^S} < nid_u < nid_{u_y^S}$, the KM sends the message

$$\text{RM2} \quad KM \rightarrow u : < \{S_c, K_R\}_{K_u} >$$

3) The KM rekeys the non compromised subgroups in $\mathcal{U}$, as follows. It determines the subgroups $S_y^{\mathcal{C}}$ and $S_e^{\mathcal{C}}$, i.e. the youngest and the eldest subgroup in $\mathcal{C}$, respectively. By construction, subgroup tokens $st_{S_e^{\mathcal{C}}}^F$ and $st_{S_y^{\mathcal{C}}}^B$ are not compromised. Also, all elder kindreds of $S_e^{\mathcal{C}}$ hold token $st_{S_e^{\mathcal{C}}}^F$, while all junior kindreds of $S_y^{\mathcal{C}}$ hold token $st_{S_y^{\mathcal{C}}}^B$. These tokens are used to rekey these subgroups, i.e. the KM generates the key encryption keys $K_F^S = KDF(st_{S_e^{\mathcal{C}}}^F)$ and $K_B^S = KDF(st_{S_y^{\mathcal{C}}}^B)$, and broadcasts the message

$$\text{RM3} \quad KM \twoheadrightarrow G :< sid_{S_e^{\mathcal{C}}}, sid_{S_y^{\mathcal{C}}}, \{K_R\}_{K_F^S}, \{K_R\}_{K_B^S} >$$

Every other non compromised subgroup S which is elder kindred of $S_y^{\mathcal{C}}$ or junior kindred of $S_e^{\mathcal{C}}$ holds only compromised subgroup tokens. Thus, it must be rekeyed by means of its subgroup key K_S. Then, $\forall\ S \in \mathcal{U}$, such that $sid_{S_e^{\mathcal{C}}} < sid_S < sid_{S_y^{\mathcal{C}}}$, the KM broadcasts the message

$$\text{RM4} \quad KM \twoheadrightarrow S : < sid_{S_e^{\mathcal{C}}}, sid_{S_y^{\mathcal{C}}}, \{K_R\}_{K_S} >$$

4) The KM updates its token sets. First, it removes the node tokens t_v^B and t_v^F associated to each compromised node $v \in G_c$ from NTS. Then, it updates the remaining node tokens in NTS, i.e. $\forall\ t_u \in NTS$, s.t. u belongs to $S \in \mathcal{C}$ and $u \notin G_c$, $t_u \leftarrow H(t_u||K_R)$. Also, the KM updates all subgroup tokens in STS, i.e. $\forall\ st \in STS, st \leftarrow H(st||K_R)$. Then, for each $S \in \mathcal{C}$, the KM computes $K_S^+ = KDF(K_S||K_R)$ and installs it as the current subgroup key of S. Finally, the KM discards K_R.

On their side, nodes perform the following steps.

• Every non compromised node u in a compromised subgroup S, i.e. $\forall\ S \in \mathcal{C}, \forall\ u$ in $S \setminus S_c$, retrieves K_R from RM1 if ($nid_u < nid_{u_e^S}$ or $nid_u > nid_{u_y^S}$), or from RM2 otherwise. Then, u computes $K_S^+ = KDF(K_S||K_R)$ and installs it as the current subgroup key. Also, u removes the node tokens associated to all compromised nodes in S, i.e. $\forall\ v$ in $S_c, NTS_u^B \leftarrow NTS_u^B \setminus \{t_v^B\}$ and $NTS_u^F \leftarrow NTS_u^F \setminus \{t_v^F\}$. Finally, node u updates all remaining node tokens, i.e. $\forall\ TS \in \{NTS_u^B, NTS_u^F\}, \forall\ t \in TS, t \leftarrow H(t||K_R)$.

• Every (non compromised) node u in a non compromised subgroup S, i.e. $\forall\ S \in \mathcal{U}, \forall\ u$ in S, retrieves K_R from RM4 if $sid_{S_e^{\mathcal{C}}} < sid_S < sid_{S_y^{\mathcal{C}}}$, or from RM3 otherwise.

• Every rekeyed node u computes $K_G^+ = KDF(K_G||K_R)$ and installs it as the current group key. Also, u updates its subgroup token sets, i.e. $\forall\ ST \in \{STS_u^B, STS_u^F\}, \forall\ t \in ST, t \leftarrow H(t||K_R)$. Finally, u discards K_R.

5. Security analysis

Backward security. Let us consider a node u that joins the group G as a member of subgroup S. The issue is to prove that u cannot access the current group key K_G and subgroup key K_S, or any previous incarnation of them. In the join rekeying procedure, the KM achieves this goal by first rekeying S and G, and then initializing u. Actually, before u can join the group G, the KM efficiently rekeys all the current members of G (but u), by means of messages JM1 and JM2, so causing the installation of the new keys K_S^+ and K_G^+. Confidentiality of JM1 and JM2 is protected by means of K_S and K_G, respectively. Thus, only members of S and G (but u) can decrypt them, retrieve the secret material therein contained, and generate the new keys K_S^+ and K_G^+. Only once this process has been successfully completed, u receives the new keys K_S^+ and K_G^+. Therefore, u never gets

knowledge of the security material used before its join, and therefore is not able to access old group communication.

Forward security. There are two cases to consider, i.e. node leaving and recovery from collusion attack. Let us first consider the former case of a leaving node h in a subgroup S. The issue is to prove that the leaving node h cannot take part to the rekeying process, and hence cannot get access to the new subgroup key K_S^+ and group key K_G^+, or any future incarnation of them. In the leave rekeying procedure, the KM achieves this goal in two steps. In the first step, the KM rekeys subgroup S by means of message LM1, which is encrypted by means of (keys deriving from) node tokens t_h^F and t_h^B. Then, in the second step, the KM rekeys the remaining subgroups by means of message LM2, which is encrypted by means of (keys deriving from) subgroup tokens st_S^F and st_S^B. By construction, h does not hold such tokens. As a consequence, it cannot access messages LM1 and LM2, and thus is excluded from the rekeying process.

Let us now consider the case of recovery from collusion attack, and the consequent eviction of multiple colluding nodes. The issue here is to prove that the leaving nodes, even when colluding, cannot take part to the rekeying process, and hence cannot get access to the new subgroup key K_S^+ and group key K_G^+, or any future incarnation of them. In the collusion recovery procedure, the KM achieves this goal in two steps. In the first step, the KM rekeys every compromised subgroup $S \in \mathcal{C}$ by sending a single message RM1, and one message RM2 for each node u in $S \setminus S_c$ such that $nid_{u_e^S} < nid_u < nid_{u_y^S}$. Messages RM1 and RM2 are encrypted by means of (keys deriving from) node tokens $t_{u_e^S}^F$ and $t_{u_y^S}^B$, and node keys K_u, respectively. By construction, colluding nodes in S do not hold such tokens and keys, and thus cannot access messages RM1 and RM2. Then, in the second step, the KM rekeys the remaining non compromised subgroups, by sending a single message RM3, and one message RM4 for each subgroup $S \in \mathcal{U}$ such that $sid_{S_e^C} < sid_S < sid_{S_y^C}$. Messages RM3 and RM4 are encrypted by means of (keys deriving from) subgroup tokens $st_{S_e^C}^F$ and $st_{S_y^C}^B$, and subgroup keys K_S, respectively. By construction, colluding nodes in G do not hold such tokens and keys, and thus cannot access messages RM3 and RM4. Hence, leaving nodes can never access rekeying messages. Since we assume that cryptanalytical and key exhaustive attacks are practically infeasible, leaving nodes cannot derive K_G^+, nor any future group key, and thus are not able to access future group communication.

6. Performance evaluation

We analytically evaluate GREP in terms of storage, communication, and computing overhead of rekeying upon node joining, node leaving and recovering from collusion. In particular, we evaluate storage and communication overhead as the number of information items that protocol actors store and transmit/receive, respectively, and the computing overhead as the number of performed cryptographic operations, i.e. encryptions, decryptions and hash function executions.

We consider a group G composed of p subgroups with m nodes each, i.e. $n = p \cdot m$. GREP well supports heterogeneous subgrouping, but a homogeneous one allows us to evaluate performance with no significant lack of generality. We assume that node IDs and subgroup IDs have the same size of tokens and keys, and that the key generator, $H(\cdot)$ and $KDF(\cdot)$ result in a comparable computing overhead.

In order to give a concrete insight of the high scalability and practical sustainability of GREP, we discuss the overheads with reference to a WSN application that features a group composed of $n = 1024$ TmoteSky sensor nodes, interconnected through an IEEE 802.15.4 wireless network and equipped with the Skipjack cipher [8]. Although GREP is general and not especially designed for WSNs, they constitute a challenging case study, as they are composed of a large set of interconnected resource scarce devices.

6.1. Storage overhead

The KM stores the group key, p subgroup keys, n node keys, $(2{\cdot}n{-}2{\cdot}p)$ node tokens, and $(2{\cdot}p{-}2)$ subgroup tokens. The resulting storage overhead for the KM is $O_{s,km} = (3 \cdot n{+}p{-}1)$, i.e. it grows linearly with n. This is not a problem in practice, since the KM has plentiful of resources.

Instead, it is vital that the storage overhead is affordable at the node side. Each node u in a subgroup S stores: i) its node key K_u; ii) the group key K_G; iii) the subgroup key K_S; iv) $(m-1)$ node tokens associated to its cognate nodes; and v) $(p-1)$ subgroup tokens associated to all subgroups but S. The resulting storage overhead for a node is $O_{s,u} = (p{+}m{+}1)$. If we consider $p \gg 1$ or $m \gg 1$, then $O_{s,u} \simeq p + m$. Hence, if nodes are uniformly distributed in p subgroups of m members each ($p = m = \sqrt{n}$), the minimum storage overhead is $O_{s,u}^{(\min)} = (2 \cdot \sqrt{n})$, i.e. it grows as $\mathcal{O}(\sqrt{n})$.

In the WSN application, the minimum storage overhead is $O_{s,u}^{(\min)} = 64$. If we consider 80 bit tokens and Skipjack keys, then the storage overhead is 640 bytes. As TmoteSky nodes feature 48 Kbytes of memory, the storage overhead is equal to 1.30% of the total memory. It follows that GREP is practically affordable even in constrained sensor nodes.

6.2. Overhead of node joining

Node u's joining of group G as a member of subgroup S requires to: i) broadcast message JM1 carrying nid_u, the master node token t_M, and the refresh key K_R; and ii) broadcast message JM2 carrying the refresh key K_R. Thus, the communication overhead amounts to four, i.e. $O_c^{(j)} = 4$.

The worst case for the computing overhead regards the current nodes in S. Each of them performs: i) one decryption to retrieve t_M and K_R from message JM1; and ii) three hash function executions to compute t_u^F, K_G^+ and K_S^+. Instead, the KM: i) computes four keys (i.e. K_G^+, K_S^+, K_R and K_u), the master node token t_M, and two node tokens (i.e. t_u^B and t_u^F); and ii) encrypt messages JM1 and JM2. Thus, the KM performs 2 encryptions and 7 hash function executions.

6.3. Overhead of node leaving

Let us consider a node h in subgroup S that leaves the group G. Message LM1 introduces a communication overhead equal to three, as it conveys nid_h and two copies of K_R. Message LM2 introduces a communication overhead equal to three, as it conveys sid_S and two copies of K_R. The total communication overhead is equal to $O_c^{(l)} = 6$. Thus, GREP efficiently rekeys the group, displaying a small and constant communication overhead which is independent of the group size, i.e. $\mathcal{O}(1)$. This makes GREP highly efficient and scalable with the number of nodes in the group.

The worst case for the computing overhead regards a node $u \neq h$ in S. Such node: i) computes either K_F or K_B; ii) decrypts either $\{K_R\}_{K_F}$ or $\{K_R\}_{K_B}$ to retrieve K_R from message LM1; iii) computes K_G^+ and K_S^+; and iv) updates its node token sets and subgroup token sets by executing $(m-2)$ and $(p-1)$ hash functions, respectively. Thus, a node performs at most one decryption and $(p+m)$ hash function executions. Instead, the KM: i) computes the keys K_G^+, K_S^+, K_R, K_F, K_B, K_F^S and K_B^S; ii) encrypts two copies of K_R in message LM1 and two copies of K_R in message LM2; and iii) updates its node token set NTS and subgroup token set STS by executing $(2 \cdot m - 4)$ and $(2 \cdot p - 2)$ hash functions, respectively. Thus, the KM performs 4 encryptions and $(2 \cdot p + 2 \cdot m + 1)$ hash function executions.

The computing overheads on the KM and nodes grow both as $\mathcal{O}(\sqrt{n})$. In particular, in the WSN application, a node performs at most 1 decryption and 64 hash function executions, which is practically affordable for sensor platforms.

6.4. Overhead of collusion recovery

Let us consider a collusion attack with C compromised subgroups and $(p - C)$ non compromised subgroups. Also, let us assume that each compromised subgroup contains c compromised nodes. In general, the communication overhead of the collusion recovery depends on the specific compromised nodes and subgroups, i.e. the relation between their node ID and their cognates' and between their subgroup ID and their kindreds', respectively. In the following, we discuss the collusion recovery in the worst case condition.

We have the worst case condition when the following two events occur at the same time: 1) non compromised nodes in every compromised subgroup $S \in \mathcal{C}$ are rekeyed by means of one unicast message RM2 each, i.e. each of the C compromised subgroups requires a *total subgroup recovery*; and 2) the $(p - C)$ non compromised subgroups $S \in \mathcal{U}$ are rekeyed by means of one broadcast message RM4 each, i.e. group G requires a *total group recovery*.

In the worst case, $(m-c)$ unicast messages RM2 are sent within each of the C compromised subgroups. The resulting communication overhead is equal to $C \cdot (m-c) \cdot (c+1)$. Also, one broadcast message RM4 is sent to each of the $(p - C)$ non compromised subgroups. The resulting communication overhead is equal to $3 \cdot (p - C)$. Thus, the total communication overhead is $O_c^{(r)} = C \cdot (m-c) \cdot (c+1) + 3 \cdot (p - C)$. If we reasonably assume that i) each subgroup includes a non negligible number of members, i.e. $m \gg 1$; ii) only a few nodes per subgroup are captured, i.e. $m \gg c$; iii) the group G includes a non negligible number of subgroups, i.e. $p \gg 1$; iv) only a few subgroups are compromised, i.e. $p \gg C$; and v) $p = m = \sqrt{n}$, for storage optimisation, then the communication overhead can be approximated as $O_c^{(r)} \simeq \sqrt{n} \cdot (C \cdot (c+1) + 3)$. Thus, in the worst case, the communication overhead smoothly grows as $\mathcal{O}(\sqrt{n})$, and gradually increases with the severity of the attack scenario.

TABLE 2. COMMUNICATION OVERHEAD (KB).

Compromised subgroups	**Compromised nodes per subgroup**				
	$\mathbf{c = 2}$	$\mathbf{c = 4}$	$\mathbf{c = 6}$	$\mathbf{c = 8}$	$\mathbf{c = 10}$
$\mathbf{C = 1}$	1.88	2.50	3.13	3.75	4.38
$\mathbf{C = 10}$	10.31	16.56	22.81	29.06	35.31

Table 2 shows the communication overhead in the WSN application. Even if 10 nodes in 10 different subgroups collude, i.e. $C = c = 10$ and 100 nodes collude, then $O_c^{(r)} = 35.31$ KB. In IEEE 802.15.4, unsecured frames have a payload with maximum size 102 bytes, and an implementation displays an effective data rate (excluding headers, CRCs, and control packets) of about 8.4 Kbps (out of 250 Kbps). Thus, even if $C = c = 10$, i.e. 100 nodes collude, the communication overhead requires 355 frames and results in 33.63 s (per hop). Hence, also in the worst case, the communication overhead is sustainable in a WSN environment. Note that IEEE 802.15.4 can display better performance. That is, Latré *et al.* showed that a throughput of about 140 Kbps can be achieved, even if acknowlegment frames are trasmitted [1]. In that case, when $C = c = 10$, the communication overhead results in 2.02 s (per hop).

In the worst case, the KM performs $C \cdot (m-c) + p - C$ encryptions and $C \cdot (2 \cdot (m-c) - 1) + 2 \cdot p - 1$ hash function executions. While this is generally not a problem on the KM, the computing overhead must be practically sustainable on the node side. A non compromised node processes only one rekeying message, i.e. either RM2 or RM4, and retrieves the refresh key K_R by performing one decryption. The highest computing overhead is experienced by a node in a compromised subgroup S, as it computes: i) the new group key K_G^+; ii) the new subgroup key K_S^+; iii) $(m - c - 1)$ new node tokens associated to its non compromised cognate nodes; and iv) $(p - 1)$ new subgroup tokens associated to the kindred subgroups of S. Hence, a node performs at most one decryption and $(p + m - c)$ hash function executions.

In the WSN application, the Skipjack key K_R is only 80 bits in size. If hash functions are implemented through the same cipher Skipjack used for encryptions, then hash function executions require to process at most 640 bytes. On TmoteSky nodes, a software version of Skipjack takes 77 μs per encrypted/decrypted byte [6]. Thus, decrypting K_R takes 0.77 ms, and performing all the hash function

executions takes at most 49.28 ms, so making collusion recovery affordable from the computing standpoint.

Now, we discuss the probability $P_{wc}(G)$ of a worst case recovery to occur. Due to space constraints, we only present the results for the WSN application, and give intuitions of the attack configurations that lead to a worst case recovery.

Let us refer to Figure 2. Subgroup S requires a total subgroup recovery if any of these pairs of nodes is compromised: i) $\{f, j\}$; ii) $\{f, i\}$; or iii) $\{g, j\}$, i.e. either $f = u_e^S$ or $g = u_e^S$, and either $i = u_y^S$ or $j = u_y^S$. Thus, one can not exploit node tokens in S to rekey multiple nodes through a single broadcast message $\mathrm{RM1}$, and each non compromised node in S is rekeyed through one unicast message $\mathrm{RM1}$ or $\mathrm{RM2}$. Also, the more compromised nodes in a given subgroup, the more it requires a total subgroup recovery.

The group G requires a total group recovery if any of these pairs of subgroups is compromised: i) $\{S^{**}, S''\}$; ii) $\{S^{**}, S'\}$; or iii) $\{S^{*}, S''\}$, i.e. either $S^{**} = S_e^{\mathcal{C}}$ or $S^{*} = S_e^{\mathcal{C}}$, and either $S' = S_y^{\mathcal{C}}$ or $S'' = S_y^{\mathcal{C}}$. Thus, one can not exploit subgroup tokens to rekey multiple subgroups through a single broadcast message $\mathrm{RM3}$, and each non compromised subgroups is rekeyed through one broadcast message $\mathrm{RM3}$ or $\mathrm{RM4}$. Also, the more compromised subgroups, the more likely the group G requires a total group recovery.

TABLE 3. PROBABILITY $P_{wc}(G)$ OF WORST CASE RECOVERY.

$\mathbf{C = 3}$		$\mathbf{C = 5}$		$\mathbf{C = 10}$	
$\mathbf{c = 2}$	$\mathbf{c = 10}$	$\mathbf{c = 2}$	$\mathbf{c = 10}$	$\mathbf{c = 2}$	$\mathbf{c = 10}$
$0.39 \cdot 10^{-8}$	$0.1 \cdot 10^{-3}$	$0.45 \cdot 10^{-12}$	$0.31 \cdot 10^{-4}$	$< 1 \cdot 10^{-20}$	$0.7 \cdot 10^{-7}$

Table 3 shows the probability $P_{wc}(G)$ that a worst case recovery occurs. Given a number C of compromised subgroups, the probability of a worst case recovery increases with c. This is consistent with the presence of more compromised nodes per compromised subgroup. However, given c compromised nodes per compromised subgroup, the probability $P_{wc}(G)$ decreases with C. Intuitively, the more subgroups are compromised, the less it is likely that i) each of them requires a total subgroup recovery; and ii) all other subgroups must be separately rekeyed. If 10 nodes collude in 10 different subgroups, i.e. $C = c = 10$ and 100 nodes out of 1024 collude, we have $P_{wc}(G) = 0.7 \times 10^{-7}$.

Thus, a worst case recovery is an extremely unlikely event, even when a non negligible number of nodes and subgroups is compromised. Besides, a total member reinitialization is necessary only when all subgroups are compromised, i.e. $C = p$, and each of them requires a total subgroup recovery. This, together with the limited overheads displayed even in worst case conditions, makes GREP extremely efficient when recovering from collusion attacks.

7. Conclusion

We have presented GREP, a novel group rekeying protocol that efficiently rekeys a group with a number of messages which is small, constant and independent of the group size. In case of collusion attack, GREP recovers the group by exploiting the history of joining events. This avoids a total member reinitialization and results in an overhead which smoothly grows with the group size, and gradually increases with the attack severity. We have provided an analytical performance evaluation and shown that GREP is deployable on large-scale networks of constrained devices.

Acknowledgments

This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no. 607109. This work was also supported by the EIT DIGITAL HII project ACTIVE; "Progetti di Ricerca di Ateneo - PRA 2016" of the University of Pisa; and the PRIN project TENACE (20103P34XC) funded by the Italian Ministry of Education, University and Research.

References

[1] B. Latré, P. De Mil, I. Moerman, N. Van Dierdonck, B. Dhoedt and P. Demeester, "Maximum Throughput and Minimum Delay in IEEE 802.15.4," in *The First international conference on Mobile Ad-hoc and Sensor Networks, Wuhan, China*, vol. 3794. Springer, 2005, pp. 866–876.

[2] C. K. Wong, M. Gouda and S. S. Lam, "Secure group communications using key graphs," *IEEE/ACM Trans. on Networking*, vol. 8, no. 1, pp. 16–30, 2000.

[3] D. Wallner, E. Harder and R. Agee, *Key Management for Multicast: Issues and Architectures*, IETF, 1999.

[4] E. Cole, *Network Security Bible, 2nd Edition*. Wiley, 2009.

[5] F. Bao, I. Chen, M. Chang and J. Cho, "Hierarchical Trust Management for Wireless Sensor Networks and its Applications to Trust-Based Routing and Intrusion Detection," *IEEE Trans. on Network and Service Management*, vol. 9, no. 2, pp. 1–15, 2012.

[6] G. Dini and I. M. Savino, "LARK: A Lightweight Authenticated ReKeying Scheme for Clustered Wireless Sensor Networks," *ACM Trans. on Embedded Computing Systems*, vol. 10, no. 4, pp. 41:1–41:35, 2011.

[7] G. Dini and M. Tiloca, "HISS: A HIghly Scalable Scheme for Group Rekeying," *The Computer Journal*, vol. 56, no. 4, pp. 508–525, 2013.

[8] J. Doumen, Y. W. Law and P. H. Hartel, "Survey and benchmark of block ciphers for wireless sensor networks," *ACM Trans. on Sensor Networks*, vol. 2, no. 1, pp. 65–93, 2006.

[9] K. Birman, *Guide to Reliable Distributed Systems. Building High-Assurance Applications and Cloud-Hosted Services*. Springer, 2012.

[10] P. Liu, W.-C. Lee, Q. Gu and C.-H. Chu, "KTR: An Efficient Key Management Scheme for Secure Data Access Control in Wireless Broadcast Services," *IEEE Trans. on Dependable and Secure Computing*, vol. 6, no. 3, pp. 188–201, 2009.

[11] S. Rafaeli and D. Hutchison, "A Survey of Key Management for Secure Group Communication," *ACM Computing Surveys*, vol. 35, no. 3, pp. 309–329, 2003.

[12] S. Setia, S. Zhu and S. Jajodia, "LEAP+: Efficient security mechanisms for large-scale distributed sensor networks," *ACM Trans. on Sensor Networks*, vol. 2, no. 4, pp. 500–528, 2006.

[13] Y. Wang, X. Wang, B. Xie, D. Wang, and D. P. Agrawal, "Intrusion Detection in Homogeneous and Heterogeneous Wireless Sensor Networks," *IEEE Transactions on Mobile Computing*, vol. 7, no. 6, pp. 698–711, June 2008.