HISS: A HIghly Scalable Scheme for Group Rekeying

GIANLUCA DINI and MARCO TILoca

Department of Ingegneria dell’Informazione, University of Pisa, Via Diotisalvi 2, 56100 Pisa, Italy
*Corresponding author: g.dini@iet.unipi.it

Group communication is a suitable and effective communication model for large-scale distributed systems. To be fully exploitable, group communication must be protected. This is normally achieved by providing members with a group key which is revoked and redistributed upon every member’s joining (backward security) or leaving (forward security). Such a rekeying process must be efficient and highly scalable. In this paper, we present HISS, a highly scalable group rekeying scheme that efficiently rekeys the group into two broadcast rekeying messages. HISS features two novel contributions. First, it exhibits a rekeying cost which is constant and independent of the group size, thus being highly scalable with the number of users. At the same time, memory occupancy and computational overhead are affordable on customary platforms. Second, HISS considers collusion as a first-class attack and recovers the group in such a way that it does not require a total group recovery. Efficiency of collusion recovery gracefully decreases with the attack severity. We prove the correctness of HISS, analytically evaluate its performance and argue that it is deployable on customary platforms. Finally, we show that it is possible to practically contrast or even prevent collusion attacks by properly allocating users to subgroups.

Keywords: security; key management; group communication

Received 8 May 2012; revised 14 September 2012
Handling editor: Albert Levi

1. INTRODUCTION

Group communication has proved to be a suitable and efficient communication model for distributed systems in many different application scenarios, including content distribution, Wireless Sensor Networks (WSNs), teleconferencing, wargaming and others. These application scenarios typically involve a large number of participants, which dynamically join and leave the application, so causing group membership to change frequently. Furthermore, they also require that group communication is protected from unauthorized accesses. This is achieved by restricting the access to group communication to group members only.

Intuitively, this is achieved by distributing a group key to all group members which use it to encrypt (decrypt) messages broadcast (received) within the group. Good cryptographic practices suggest one to periodically refresh the key in order to prevent cryptanalysis. Furthermore, when a new user joins the group, he must not be able to access any group communication prior its joining (backward security). Besides, when a current member leaves the group, he must be prevented from accessing any further group communication (forward security). As a consequence, upon a new user’s joining or a current one’s leaving, the current group key has to be revoked and a new one has to be distributed. This process is known as rekeying, and makes it possible to fulfill the backward and forward security requirements [1].

Rekeying the group upon a new member’s joining is actually trivial. In contrast, rekeying the group upon a current member’s leaving is a far more complex problem. A naïve solution consists in separately rekeying every remaining member, by transmitting that member the new group key encrypted with the member’s secret key. Although very simple, this solution requires $O(n)$ rekeying messages, where $n$ is the group size, and thus scales poorly. Actually, efficient rekeying requires to broadcast the new group key in its encrypted form into the group. However, the current group key cannot be used, because the leaving member is aware of it. A typical approach is to encrypt rekeying messages by means of administrative keys that are not known to the evicted member. Furthermore, the administrative keys known to the evicted member should also be replaced. It follows that
efficient rekeying is a crucial component in any secure group communication system [1–8].

A group member may leave a group for several reasons. For example, because its mission is concluded, its subscription is expired, or it has failed or depleted its energy. Furthermore, a group member may be forced to leave because it has been compromised or it is suspected so. A collusion attack occurs when evicted compromised members share their security material in order to regain access to the group key [1]. A typical collusion attack consists of an adversary capturing a set of initial key material of individual members to a level that allows revealing the group key and, consequently, all encrypted traffic in the network. No group rekeying scheme is exempt from collusion attacks, and different schemes display different levels of resilience to this kind of attack [6, 9–13]. In any case, after a successful collusion attack, group rekeying schemes generally require a total group recovery, i.e. every group member must be re-initialized in a one-to-one fashion. Of course, such a total group recovery has a negative impact on the overall rekeying scheme performance.

In this paper, we propose HISS, a highly scalable rekeying scheme for large-scale, dynamic groups. HISS is centralized and leverages on logical subgrouping, a consolidated conceptual technique that has already been exploited in several existing group rekeying schemes [6, 7, 9–11, 13–15]. HISS partitions group members into non-overlapping logical subgroups that become the units of rekeying and recovery from collusion attacks.

With respect to other rekeying schemes based on subgrouping, HISS features the following two novel contributions. First, unlike other well-known centralized approaches [6, 7, 11, 14], HISS rekeys the system with a number of messages that is small, constant, and independent of the group size, thus resulting in highly scalable and efficient. Specifically, HISS only requires two broadcast messages to rekey the system, so displaying $O(1)$ communication complexity. One broadcast message rekeys the subgroup containing the leaving member, and another one rekeys all the remaining subgroups.

Second, differently from [9, 10, 13, 15], HISS considers collusion attacks as a first-class problem, and provides an integrated recovery protocol that re-establishes security as soon as a collusion attack has been detected. Such protocol gracefully decreases in efficiency with the collusion attack severity, and does not require a total group recovery of the system. In fact, only compromised subgroups are totally recovered, whereas uncompromised ones are efficiently rekeyed by a single broadcast message each. The rational basis for this is that while rekeying accounts for the normal functioning of the system (joining or a leaving), recovery has to be considered an exceptional event. Therefore, according to the well-known Lampson’s recommendations for computer systems design [16], while rekeying must be very efficient, it is sufficient that recovery is able to make progress provided it remains practically sustainable. Finally, we also show that a proper allocation of users to subgroups may practically increase the resilience of HISS to collusion attacks or, even, prevent them altogether.

The rest of this paper is organized as follows. In Section 2, we discuss some related work, and compare HISS with the current state of the art. Section 3 describes the system scenario and the threat model. Section 4 presents the HISS rekeying scheme, which is composed of a suite of three protocols: the leave protocol, the join protocol and the recovery protocol. While the first and the second protocol implement rekeying, the last one supports recovery from collusion attacks. Section 5 presents a security analysis and argues that HISS guarantees forward security and backward security. Section 6 analytically evaluates HISS performance in terms of communication, storage and computing overhead. In this section, we show that while rekeying is highly scalable, the recovery overhead is sustainable on customary platforms (e.g. personal computers or sensor nodes). In Section 7, we discuss how HISS can help us to deal with collusion attacks and argue that allocation policies of users to subgroups can be defined to practically contrast or, even, prevent collusion attacks. Finally, in Section 8, we draw our final conclusions.

2. RELATED WORK

Group key management schemes are typically classified as centralized, decentralized or distributed [1]. All these schemes attempt to provide a balance among security and performance. In a centralized scheme, key management is concentrated in a single entity. Relevant examples include [6, 7, 14]. Typically, a centralized approach makes it possible to minimize storage, computation and communication requirements on both client and server sides. However, the single group key managing entity may become a performance and security bottleneck. Centralized settings are suitable for large-scale, possibly geographically disparate, groups with dynamic membership. Content distribution applications such as pay TV, news or stock information dissemination belong to this category [17].

The decentralized and distributed approaches attempt to overcome these limitations by distributing key management over a set of key managers in the former case, or, even, the users themselves in the latter. Relevant examples of the decentralized approach include [10, 13, 18, 19], whereas examples of distributed approach include [5, 20–23]. Unfortunately, decentralized and distributed approaches tend to be harder to implement than centralized ones, raise security concerns, and are often less scalable. Applications like conferencing and distributed interactive simulation fall under the category of distributed setting. The group sizes in such applications are typically small and justifies the usage of the relatively high end computation required by the group key agreement techniques [17]. Centralized versus distributed or decentralized group key management is still an open debate.

HISS is based on a centralized approach and introduces the following two benefits with respect to the current state of the
art. First, it is highly scalable with the number of users, as it requires a number of rekeying messages that is constant and independent of the group size. At the same time, it is affordable in terms of memory occupancy and computational overhead. Secondly, HISS considers collusion as a first-class attack and provides a specific protocol aimed at recovering the group from collusion attack. In particular, unlike many rekeying schemes, HISS does not require a total group recovery, and its performance gracefully decreases with the collusion attack severity.

In the rest of this section, we provide a comparison between HISS and other well-known rekeying schemes. First, we consider rekeying approaches akin to HISS from an architectural standpoint, and give an analytical comparison of performance. Then, we qualitatively compare HISS to other rekeying schemes that, although different from an architectural point of view, display however certain similarities with it.

As stated above, HISS takes a centralized approach, as LKH [6], Key Graphs [7], LARK [14] and KTR [24]. These systems use a set of ‘administrative keys’ (or other cryptographic material, e.g. tokens) and logically group them in order to provide a scalable rekeying in the case of leaving events. LKH hierarchically organizes administrative keys in a logical tree, so achieving an overhead that is logarithmic in the number of users of the system. Key Graph is essentially a generalization of LKH and logically organizes administrative keys in a graph, so achieving a performance that depends on the specific graph topology but is no better than LKH’s [14]. LARK is based on Key Graph and logically groups administrative keys into a graph, but for different reasons than HISS. In LARK, logical grouping is a tool for application design. An application designer defines a key graph topology reflecting cooperation within the system. The Key Manager receives such a topology and does its best to provide efficient rekeying. An approach similar to LARK is taken by KTR that generalizes LKH to manage multiple subscriptions in content distribution applications and wireless broadcast services [24]. In contrast to LARK and KTR, logical subgrouping in HISS is instead a network management tool aimed at supporting efficient and scalable rekeying. Subgroups have no application meaning and are even transparent to the application level.

More in detail, LKH organizes keys hierarchically into a logical tree where the root contains the group key, the leaves contain users’ private keys, and internal nodes contain administrative keys. If we assume that the group size is \( n \) and the key tree is balanced with arity \( a \), \( a > 1 \), then the (leave) communication overhead is \( a \cdot \log_a n - 1 \), and the storage overhead at the user side is \( (\log_a n) + 1 \). A number of schemes deriving from LKH have been proposed, including [2, 3, 8, 14, 24].

It is evident that the communication overhead of LKH grows logarithmically with \( n \). Instead, the communication overhead of HISS is equal to five and therefore is small, constant, and independent of the group size (Section 6.2). This overhead accounts for the amount of information that a user has to receive and process. Therefore, it results to be particularly important, since, in a centralized key management scheme, users are generally considered computationally less capable than servers.

As to the LKH memory overhead at the user side, it grows logarithmically with \( n \). On the other hand, the HISS memory overhead grows with \( \sqrt{n} \) (Section 6.1). Although LKH is more efficient than HISS from this standpoint, HISS memory overhead is practically affordable in current platforms (Section 6.1).

Finally, LKH becomes vulnerable to a collusion attack when at least \( t_{LKH} = n/2 \) users are captured before they are detected. In contrast, HISS requires \( t_{HISS} = 2 \cdot \sqrt{n} \) users to be captured (see Section 7.1). For any system larger than \( n = 16 \) users, \( t_{LKH} > t_{HISS} \). Hence, LKH results more resistant than HISS to collusion attacks. However, a proper allocation of users to subgroups can make HISS practically resistant to collusion attacks (Section 7). Table 1 resumes performance differences between HISS and the LKH scheme.

<table>
<thead>
<tr>
<th></th>
<th>Communication overhead</th>
<th>User storage overhead</th>
<th>Collusion threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>LKH</td>
<td>( O(\log_a n) )</td>
<td>( O(\log_a n) )</td>
<td>( \frac{n}{2} )</td>
</tr>
<tr>
<td>HISS</td>
<td>( O(1) )</td>
<td>( O(\sqrt{n}) )</td>
<td>( 2 \cdot \sqrt{n} )</td>
</tr>
</tbody>
</table>

As it turns out, HISS is highly scalable, more than LKH, while displaying a practical and affordable overhead and resistance to collusion attacks.

Rekeying schemes such as SECK [9], HySOR [11], SHELL [13] and EBS [15] are architecturally different from HISS but display similarities with it. HySOR makes it possible to trade off the message cost of rekeying with some increased vulnerability to collusion [11]. Also, it provides a range of protocols with LKH at one extreme and LORE at the other, which requires \( O(1) \) messages for rekeying, but in which any two receivers can collide. In contrast to HySOR, the HISS cost of rekeying is always constant, small, and independent of the group size. Furthermore, in LORE two colluding receivers compromise the whole group, whereas in HISS two colluding users compromise only a single subgroup. Finally, HySOR does not suggest any recovery protocol.

Exclusion basis system (EBS) is another scheme that displays some similarities with HISS, as it uses a form of grouping [15]. EBS views the logical structure of the group as a collection of subsets of the group members, and applies combinatorial optimization techniques to the key management problem. In EBS, keys are reused in multiple nodes and only key combinations are unique. It follows that collusion of a few nodes can reveal all the keys employed in the network to the adversary, so causing forward security to be completely broken.
the capture of the entire network, and the consequent need of a total network recovery. Optimal assignment of keys to prevent network capturing is a classical resource allocation problem that is NP-hard [25]. Thus, differently from HISS, in EBS it is possible to mitigate, but not eliminate, collusion using allocation heuristics that reduce the probability of capturing the entire network, but require, for example, the knowledge of nodes’ location while allocating keys.

SECK [9, 10] and SHELL [13] are two EBS-based schemes for WSNs that provide countermeasures against collusion attacks. Precisely, these systems should be classified as decentralized because they physically cluster neighbouring nodes into clusters and use EBS-based key management schemes to administer keys within each cluster. However, a comparison with HISS is interesting, at least at the cluster level, because they use logical grouping of keys, although at the cluster level, and consider a collusion attack in their threat model.

In SECK, the robustness to a collusion attack may be tuned by acting on certain EBS parameters. Unfortunately, making SECK more robust to a collusion attack decreases the communication performance of rekeying, and vice versa. Furthermore, SECK recovery in the case of a successful collusion attack performs better in localized attacks. Also in HISS the robustness to collusion attacks depends on system parameters, namely subgroups and allocation of users to subgroups. However, the impact on performance due to the rekeying protocol is small, constant and independent of those parameters. Furthermore, in the case of localized attacks, such as those described in Section 7.2, a proper definition of subgroups and a proper allocation of users to subgroups make it possible to practically prevent collusion attacks.

SHELL suggests a key assignment approach such that, even though colluding nodes share their keys, an adversary would not be able to access all the keys of the network unless many nodes are compromised. In other words, SHELL attempts to reduce the probability that the entire network is captured. However, if a collusion attack is successful, the entire network is compromised and a total recovery is necessary. SHELL is based on quite a specific collusion model, i.e. compromised nodes have direct communication links. In contrast, HISS is based on a more general collusion model. Actually, HISS does not place any restriction on the communication among captured nodes, and between captured nodes and the adversary. In Section 7.1, we assume that the adversary collects keying material from users possibly selected at random, whereas, in Section 7.2, we assume that an adversary can even physically capture users.

Finally, it is worth mentioning CRMS [12] and the collusion-resistant rekeying architecture proposed by Cheung et al. [26]. CRMS is a matrix-based scheme similar to EBS, aimed at supporting dynamic membership and increasing the robustness to collusion attacks. The scheme proposed by Cheung et al. relies on attribute-based encryption [26]. They refer to the ciphertext-policy attribute-based encryption (CP-ABE) scheme described in [27], and use it to improve the flat table group key management scheme [5, 28], so making it resistant to collusion attacks. However, it is quite challenging to define user attributes in an optimal way, in order to assure efficiency during the rekeying procedure. In addition, the considered attribute-based encryption primitive produces very large ciphertexts, thus resulting in huge broadcast rekeying messages.

Table 2 summarizes the differences between HISS and the other rekeying schemes discussed aforementioned. Specifically, it focuses on performance of the rekeying procedure, and collusion management.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Total Recovery</th>
<th>Recovery Protocol</th>
<th>Rekeying Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>HISS</td>
<td>No</td>
<td>Yes</td>
<td>The amount of rekeying messages is independent of the group size</td>
</tr>
<tr>
<td>HySOR</td>
<td>No</td>
<td>No</td>
<td>The amount of rekeying messages can be constant</td>
</tr>
<tr>
<td>EBS</td>
<td>Yes</td>
<td>Yes</td>
<td>Incomparable</td>
</tr>
<tr>
<td>CRMS</td>
<td>No</td>
<td>Yes</td>
<td>Collusion protection decreases rekeying performance</td>
</tr>
<tr>
<td>SECK</td>
<td>No</td>
<td>Yes</td>
<td>Incomparable</td>
</tr>
<tr>
<td>SHELL</td>
<td>Yes</td>
<td>Yes</td>
<td>Assuring efficiency is a difficult task</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>Ciphertexts are very large in size</td>
</tr>
</tbody>
</table>

3. SYSTEM ARCHITECTURE

We consider a group $G$ of users. A user becomes member of the group by explicitly joining it. As a member of the group, a user may broadcast messages to the other members. Later on, a member may voluntarily leave the group or be forced to leave if compromised or suspected to be so. In order to guarantee the security of group communication, it is generally required that when a user joins the group it is not able to access group communication prior its joining (backward security), and that when a user leaves the group it is prevented from accessing any further group communication (forward security). In order to achieve this, group members share a secret cryptographic key, the group key, which they use to encrypt and decrypt messages that are transmitted and received, respectively, within the group. We denote by $K_G$ the group key of the group $G$. When a new user joins or a current member leaves the group, the current
The group controller, GC.

HISS solves it. Finally, in Section 4.3, we present the rekeying protocols that manage a user’s leaving and joining, as well as the recovery from multiple colluding user captures.

4. THE REKEYING SCHEME

In this section, we describe HISS. In Section 4.1, we provide an informal description of the basic rekeying scheme with particular reference to key revocation and distribution upon a user’s leaving. Then, in Section 4.2, we introduce the problem of collusion attack and give an informal description of how HISS solves it. Finally, in Section 4.3, we present the rekeying protocols that manage a user’s leaving and joining, as well as the recovery from multiple colluding user captures.

4.1. The basic scheme

We assume that the group G is partitioned into a set S of nonempty subgroups, such that every member of G is exactly in one of these subgroups. More formally, let $S \subseteq 2^G$ be a partition of G. Then, $\forall S, S' \in S, S \cap S' = \emptyset$ and $\bigcup_{S \in S} S = G$. We call cognate two users belonging to the same subgroup. Furthermore, we consider the function $\text{SubgroupOf} : G \rightarrow S$ that returns the subgroup of a given user, i.e. $\forall u \in G, \forall S \in S, S = \text{SubgroupOf}(u)$ if $u \in S$. Subgroups are relevant only for key management and have no application-level meaning.

Finally, we consider the function $\text{SubgroupSetOf} : 2^G \rightarrow S$ that given $G' \subseteq G$ returns a set of subgroups $S'$ such that $\forall u \in G' \Rightarrow \text{SubgroupSetOf}(u) \in S'$ and $\forall S \in S', S \cap G \neq \emptyset$.

We assume that every user and every subgroup are associated with random quantities called tokens. We call user tokens and subgroup tokens the tokens associated with users and subgroups, respectively. We denote by $t_u$ the token associated to user $u$, and by $t_S$ the subgroup token associated to subgroup $S$. We also assume that every user a priori shares a user key with the key manager, KM. We denote by $K_u$ the user key of user $u$. Finally, we assume that every subgroup is associated to a subgroup key. We denote by $K_S$ the subgroup key of subgroup $S \in S$. A subgroup key is shared between the Key Manager KM and every user in the subgroup. KM and users keep tokens and keys secret.

The Key Manager maintains subgroups, tokens and keys for all users and subgroups in the system. In particular, the Key Manager records: (i) all user tokens in the User Token Set (UTS); (ii) all subgroup tokens in the Subgroup Token Set (STS); (iii) all user keys in the User Key Set (UKS); and, finally, (iv) all subgroup keys in the Subgroup Key Set (SKS).

Let us consider a user $u$ belonging to a subgroup $S$, i.e. $S = \text{SubgroupOf}(u)$. The user maintains its user key $K_u$ and the subgroup key $K_S$ associated with its subgroup $S$. Furthermore, the user maintains the User Token Set UTS$_u$, namely the set of user tokens associated to its cognate users. More formally,

$$\text{UTS}_u = \{t_v \mid v \in S \land v \neq u\}. \quad (1)$$

Finally, the user $u$ maintains the Subgroup Token Set STS$_S$, namely, the set of subgroup tokens of all subgroups belonging to the absolute complement of $S$ in $S$. More formally,

$$\text{STS}_S = \text{STS}_{\text{SubgroupOf}(u)} = \{t_R \mid R \in S \land R \neq S\}. \quad (2)$$

In order to fix ideas, without any ambition of generality, consider the example in Fig. 2. Figure 2A shows a group $G$ partitioned in three subgroups $S$, $S'$ and $S''$, whereas Fig. 2B shows the data structures maintained by members of $S$ and $S'$. Intuitively, the Key Manager uses tokens to rekey the system in a scalable way as follows. With reference to Fig. 2A, let us...
suppose that user $u \in S$ leaves the group $G$ and, therefore, the current group key $K_G$ has to be revoked and a new one $K_G^+$ redistributed to all members of $G$ but $u$. In order to distribute the new key to all members of $S$ but $u$, the Key Manager uses $t_u$. By construction, all users in $S$ but $u$ know $t_u$ (see Fig. 2B). Therefore, the Key Manager can use $t_u$ to derive the key encryption key, decrypt the received ciphertext and finally obtain $K_G^+$.

In order to distribute the new key to all the other subgroups, the Key Manager uses $t_S$. By construction, this token is known to all members of any subgroup $S'$ different from $S$. Thus, user $u$ does not know $t_S$ (see Fig. 2B). Therefore, the Key Manager can use $t_S$ to generate another key encryption key, use it to encrypt $K_G^+$, and then broadcast the resulting ciphertext into $G$. Any user $w \in S', S' \neq S$, can derive the key encryption key, decrypt the received ciphertext and finally obtain $K_G^+$. If two or more users are captured within a subgroup, two cases need to be considered: (1) non-colluding user captures (e.g. attacks carried out by different adversaries) and (2) colluding user captures. In the former case, it is possible to exploit the basic scheme in order to evict non-colluding users (see Section 4.1). In the latter case, colluding attackers may compromise the whole subgroup. Actually, by joining the user token sets of at least two users it is possible to obtain the whole set of user tokens associated with the subgroup. With reference to Fig. 2A, if users $u$ and $v$ collude, then all user tokens associated to $S$, namely $\text{UTS}_u \cup \text{UTS}_v = \{t_u, t_v, t_z\}$ are compromised. It follows that user tokens cannot be used to rekey the compromised subgroup.

Note that in this case, subgroup tokens can still be used to rekey the other non-compromised subgroups. However, in the most general case, multiple users belonging to two or more subgroups collude. Let us suppose that $u, v, w$ and $x$ collude. Similarly to before, all user tokens associated with subgroups $S$ and $S'$ are compromised. However, with respect to the previous case, the adversary may obtain all the subgroup tokens by joining the Subgroup Token Sets of any two users belonging to different subgroups. With reference to Fig. 2A, if users $u \in S$ and $w \in S'$ collude, then all the subgroup tokens, namely $\text{STS}_S \cup \text{STS}_S' = \{t_z, t_S', t_{S''}\}$ are compromised. It follows that subgroup tokens cannot be used to rekey non-compromised subgroups.

The basic scheme described in Section 4.1 is efficient as it requires just two broadcast messages to rekey the group upon a member’s leaving, and works well provided that captured users are non-colluding. However, in practice collusion attacks can occur and the group key management system solution has to...
take this threat into account. Researchers have pointed out that there exist no sure and efficient way to readily detect a single user capture [10, 29]. Therefore, for a group key management solution to be truly effective in a hostile environment, it must be able to recover from multiple user captures.

Let us suppose that \( r \) subgroups are compromised due to a collusion attack involving multiple users. Let \( C \subseteq S \) be the set of these subgroups. Let \( U \) be the absolute complement of \( C \) in \( S \), i.e. \( U = S \setminus C \). Subgroups in \( U \) are not compromised. Then, every compromised subgroup in \( C \) needs to be totally recovered by unicasting the new keying material to every non-captured user using its respective user key. Furthermore, every non-compromised subgroup in \( U \) is recovered by broadcasting the new keying material to the subgroup using its subgroup key. As it turns out, a collusion attack affects subgroups separately, and does not compromise the entire group. More specifically, a total group recovery is not necessary upon a collusion attack. In contrast, total recovery is necessary only for the subgroups involved in the attack. The other subgroups that are not involved in the attack can be efficiently rekeyed by broadcasting the new keying material. This form of partial recovery provides a form of graceful degradation of performance in terms of the number of recovery messages and cryptographic operations.

4.3. The protocols

In this section, we present the leave, join and recovery protocols in a more detailed way. Each protocol is a master–slave protocol, where the Key Manager is the master, and the users are the slaves. Every user is structured as a collection of message handlers. Each handler is denoted by \( \square \) HANDLER handler name. The execution of a handler is triggered by the reception of the corresponding message, and runs uninterrupted until completion.

We also use the following notation. By \( h() \), we denote a one-way hash function [38]. A one-way hash function \( h() \) is a function that has the following properties: given an input \( x \), it is easy to compute the image \( y = h(x) \), whereas given \( y \) it is computationally unfeasible to compute the preimage \( x \) so that \( y = h(x) \). Furthermore, given an input \( x \) and its image \( y = h(x) \) it is computationally unfeasible to find a second preimage \( z \) such that \( y = h(z) \). Examples of one-way hash functions are SHA-1, SHA-2, SHA-256 [39]. Furthermore, by \( \text{kdf}() \) we denote a key derivation function that is a pseudo-random function that derives one or more cryptographic keys from a secret value [40]. Keyed cryptographic hash functions are popular examples of pseudo-random functions used for key derivation. Finally, by \( |x|_k \), we denote the encryption of quantity \( x \) by means of key \( k \). Concerning this, we assume that: (i) the cipher is computationally secure; (ii) the cryptographic keys are generated by means of a secure random generator, unless otherwise specified; and (iii) key length is adequate to discourage an exhaustive key search [38]. Finally, by \( u \rightarrow v : m \), we denote process \( u \) sending a unicast message \( m \) to process \( v \), whereas by \( u \rightarrow G : m \), we denote process \( u \) sending a broadcast message \( m \) to the group \( G \).

4.3.1. System initialization

Upon system initialization, the Key Manager performs the following actions: (i) randomly generate user tokens and subgroup tokens, as well as user keys and subgroup keys for all users and subgroups in the system; (ii) store tokens and keys into the Key Manager’s User Token Sets, Subgroup Token Sets, User Key Set and Subgroup Key Set, as appropriate (see Section 4.1); and, finally, (iii) initialize every user with the corresponding user and subgroup key as well as the user and subgroup tokens, according to the subgroup the user belongs to (see Section 4.1). User initialization (step iii) is performed through a pre-existing secure, confidential and authentic channel.

4.3.2. Forward security

In order to assure forward security, the Key Manager KM relies on two different protocols, namely the leave protocol and the recovery protocol. The former protocol is used when a single user has to be evicted from the group \( G \). This may occur when the user has completed its mission and thus leaves the group. Alternatively, this may happen when the Intrusion Detection System has detected a single user capture and thus the compromised user has to be evicted from the group. The recovery protocol is used in case the Intrusion Detection System has detected multiple possibly colluding user captures and thus multiple users have to be evicted at the same time.

More specifically, upon receiving the \( \text{rekey}(G_c, \text{leave}) \) call from the Group Membership Service GMS, the Key Manager KM determines whether to trigger the leave protocol or the recovery protocol, according to the following steps:

1. If \( G_c \) contains a single user \( u \), then KM triggers the leave protocol specifying \( u \) as argument. Otherwise,
2. If \( G_c \) contains \( t \) users, \( t > 1 \), then
   a. If the \( t \) users belong to different subgroups, i.e. there are no couples of cognate users, then for each user \( u \in G_c \), KM triggers an instance of the leave protocol specifying \( u \). Otherwise,
   b. If a number of users in \( G_c \) are cognate, i.e. one or more subgroups have been compromised, then KM triggers the recovery protocol specifying the set \( G_c \) as argument.

It is worthwhile to note that multiple user captures require the execution of the recovery protocol if and only if the compromised users are colluding. Actually, the presence of multiple user captures does not imply in itself that they are colluding too. If compromised users are not colluding, they can be efficiently evicted from the system by means of the leave protocol. In general, determining whether two or more compromised users are also colluding may be a difficult task that strictly depends on the specific application...
The Key Manager KM revokes the current group key and basic mechanisms for any intrusion management policy. Protocols in the two respective relevant situations, namely upon and recovery policy, and present the leave and recovery detectes multiple user captures.

In this paper, we abstract away from both the application scenario and the corresponding intrusion detection technique and recovery policy, and present the leave and recovery protocols in the two respective situations, namely upon a user’s leaving, or being evicted, and upon evicting a set of colluding captured users. These two protocols constitute the basic mechanisms for any intrusion management policy.

The leave protocol. Let us suppose that a user $u$, belonging to the subgroup $S$, leaves, or is forced to leave, the group $G$. The Key Manager KM revokes the current group key $K_G$ and distributes a new one $K_G^+$ according to the leave protocol.

**Key Manager KM**

1. KM generates (i) a new group key $K_G^+$; (ii) a new subgroup key $K_S^+$; (iii) a key encryption key $KEK_u$, $KEK_u \leftarrow kdf(t_u)$, to rekey the subgroup $S$; and, finally, (iv) a key encryption key $KEK_S$, $KEK_S \leftarrow kdf(t_S)$, to rekey all the other subgroups. Then,

2. KM broadcasts the following rekeying messages:

   $M_1 : KM \rightarrow S : u, \{K_G^+, K_S^+\}_{KEK_u}$,

   $M_2 : KM \rightarrow G \setminus S : S, \{K_S^+\}_{KEK_S}$.

Finally,

3. KM installs $K_G^+$ as the current group key, $K_G \leftarrow K_G^+$; installs $K_S^+$ as the current subgroup key of the subgroup $S$, $K_S \leftarrow K_S^+$; removes $t_u$ from UTS, $UTS \leftarrow UTS \setminus \{t_u\}$, and updates the user tokens related to the remaining users in $S$, $\forall v \in UTS, v \in S, t_v \leftarrow h(t_v\|K_G^+)$; and, finally, updates all the subgroup tokens, $\forall t \in STS, t \leftarrow h(t\|K_G^+)$. 

**User $u$**

- **Handler LH1.** Upon receiving message $M_1$, user $v$, $v \in S$, $v \neq u$, performs the following actions.

  1. User $v$ computes the key encryption key $KEK_u$, $KEK_u \leftarrow kdf(t_u)$, and uses it to retrieve the new group key NEXTGKEY and the new subgroup key $K_S^+$. Then,

  2. User $v$ installs $K_G^+$ as the current group key, $K_G \leftarrow K_G^+$; installs $K_S^+$ as the current subgroup key of the subgroup $S$, $K_S \leftarrow K_S^+$; removes $t_u$ from $UTS_v$, $UTS_v \leftarrow UTS_v \setminus \{t_u\}$, and updates the remaining user tokens, $\forall t \in UTS_v, t \leftarrow h(t\|K_G^+)$; and, finally, updates its own Subgroup Token Set $STS_v$, $\forall t \in STS_v, t \leftarrow h(t\|K_G^+)$. 

**Handler LH2.** Upon receiving message $M_2$, user $v$ in $S'$, $S' \in S$, $S' \neq S$, performs the following actions.

1. User $v$ computes the key encryption key $KEK_S$, $KEK_S \leftarrow kdf(t_S)$, and uses it to retrieve the new group key $K_G^+$. Then,

2. User $v$ installs $K_G^+$ as the current group key, $K_G \leftarrow K_G^+$, and updates its Subgroup Token Set, $\forall t \in STS_v, t \leftarrow h(t\|K_G^+)$. 

It is worth noting that, in any execution of the leave protocol, a user $v$ receives either message $M_1$ or message $M_2$. Since both the Key Manager and any user’s handler have a fixed number of steps, the protocol time complexity is constant.

The recovery protocol. Let $G_c$ be the set of compromised users. KM revokes the current group key $K_G$ and distributes a new one $K_G^+$ according to the recovery protocol.

**Key Manager KM**

1. Initially, KM builds $C = \text{SubgroupSetOf}(G_c)$ and $U = S \setminus C$. Then, $\forall u \in G_c$, KM removes $u$ from $G$ and from $\text{SubgroupOf}(u)$. Then, KM randomly generates (i) a new group key $K_G^+$; (ii) a new subgroup key $K_S^+$ for each compromised subgroup $S \in C$; (iii) a new user token $t_u$ for each non-compromised user $u$ belonging to a compromised subgroup $S$, i.e., $u \notin G_c$, $u \in S$, $S \in C$; and, finally, (iv) a new subgroup token $t_{S'}$ for every subgroup $S \in S$. Then,

2. $\forall S \in C, \forall u \in S$, KM sends $u$ a unicast rekeying message $M_u$

   $M_u : KM \rightarrow u : \{K_G^+, K_S^+, UTS_u^+, STS_{u'}^+\}_{KEK_u}$

   carrying the new group key $K_G^+$, the new subgroup key $K_S^+$, the new User Token Set $UTS_u^+$ (see Equation (1)) and the new Subgroup Token Set $STS_{u'}^+$ (see Equation (2)) containing, respectively, the new user tokens and the new subgroup tokens built at step (1).

3. Then, $\forall S \in U$, KM broadcasts a rekeying message $M_S$

   $M_S : KM \rightarrow S : \{K_G^+, STS_{S'}^+\}_{KEK_S}$

   carrying the new group key $K_G^+$ and the new Subgroup Token Set $STS_{S'}^+$ (Equation (2)) which contains the subgroup tokens built at step (1). Finally,

4. KM installs $K_G^+$ as the current group key, $K_G \leftarrow K_G^+$; installs the new subgroup keys as the current subgroup keys of each colluding subgroup belonging to $C$; and, finally, updates its own User Token Set UTS and Subgroup Token Set STS with the user tokens and subgroup tokens generated at step (1).

**User $u$**

- **Handler RH1.** Upon receiving the unicast rekeying message $M_u$, non-compromised user $u$ belonging to a compromised subgroup $S \in C$ performs the following actions.

1. User $u$ uses the user key $K_u$ shared with KM to decrypt message $M_u$ and retrieve the new group key $K_G^+$, the new subgroup key $K_S^+$, the new User Token Set $UTS_u^+$ and the new Subgroup Token Set $STS_{u'}^+$. Then,
Also, involved users execute their own rekeying operations the current group key provided by the Key Manager KM. By doing so, KM revokes becoming a member of the subgroup according to the join operation. Having access to old secured messages.

Before a user can become a new member of the group, the Key Manager has to refresh the network security material in order to unicast rekeying messages only to non-compromised users in the whole group G is not entirely compromised. This implies that, in order to restore secure communications, it is not necessary to totally recover it by uncasting rekeying messages to every non-compromised users remained in the group G. In fact, it is necessary to unicast rekeying messages only to non-compromised users in compromised subgroups, i.e. subgroups in C. As subgroup keys associated to non-compromised subgroups, i.e. subgroups in U, are not compromised, then we can use each one of them to efficiently rekey its respective subgroup by means of a single rekeying message.

4.3.3. Backward security

Before a user can become a new member of the group G, the Key Manager has to refresh the network security material in order to assure backward security also. More specifically, a new user must not be able to gain knowledge of the past group activity that took place before its join, that is, it must be prevented from having access to old secured messages.

Let us suppose that a user u wants to join the group G and become a member of the subgroup S ∈ S. First, u invokes the join(u) operation provided by the Group Membership Manager GMS. Then, GMS invokes the rekey(u, join) operation provided by the Key Manager KM. By doing so, KM revokes the current group key KG and distributes a new group key KG+ according to the join protocol.

Key Manager KM

(1) KM generates a new group key KG+, a new subgroup key KS+, and a new user token tu. Then,

(2) KM broadcasts the following rekeying messages:

\[ M_1 : KM \rightarrow G : \{K^+\}_G, \]
\[ M_2 : KM \rightarrow S : \{K^+_S, t_u\}_S. \]

Finally,

(3) KM installs KG+ as the current group key, KG ← KG+; installs KS+ as the current subgroup key of the subgroup S, KS ← KS+ and, finally, adds tu to UTS, UTS ← UTS ∪ {tu}.

User v

Handler JH1. Upon receiving the rekeying message M1, user v ∈ G performs the following actions.

(1) User v uses KG to retrieve the new group key KG+

(2) user v installs KG as the current group key, KG ← KG+.

Handler JH2. Upon receiving the rekeying message M2, user v ∈ S performs the following actions.

(1) User v uses KS to retrieve the new subgroup key KS+

(2) user v installs KS as the current subgroup key of the subgroup S, KS ← KS+.

Once the join protocol has been completed, user u joins the group G and the subgroup S, according to the following steps.

(1) KM provides user u with the new group key KG+, the new subgroup key KS+, the (m – 1) user tokens associated to its subgroup cognates, and the (p – 1) subgroup tokens sTS, S′ ∉ S, through a pre-existing secure channel. Then,

(2) user u installs KG as the current group key, KG ← KG+; installs KS as the current subgroup key of the subgroup S, KS ← KS+; and, finally, builds its own User Token Set UTSu (see Equation (1)) and Subgroup Token Set STS (see Equation (2)) with the user tokens and subgroup tokens received at step (1).

As stated in step (1), the keying material is provided to the new user u by the Key Manager through a pre-existing secure channel. As a consequence, authentication and confidentiality are both assured. Possible implementations include an a priori shared cryptographic key or out-of-band means.

5. SECURITY ANALYSIS

In this section, we argue that the leave, recovery and join protocols guarantee the forward and backward security requirements.

Theorem 5.1. The leave protocol guarantees forward security.
Proof. Given the assumptions on the cipher strength and key length, the proof consists in showing that all group users but
$u \in S$ can derive the next group key $K_G^+$ from an execution instance of the leave protocol, and that user $u$ cannot derive it
nor any future group key.

Thanks to HANDLER LH1, all users in $S$ but $u$ retrieves the
new group key $K_G^+$. Then, thanks to HANDLER LH2, every user
$v \in S', \forall S' \subseteq S, S' \neq S$ retrieves $K_G^+$ as well. Thus, user $u$
will be evicted from the group $G$, and all other users hold the new
group key $K_G^+$.

Furthermore, tokens are updated by KM and users in $S$ by
means of the new group key $K_G^+$. As it does not hold such key,
user $u$ cannot compute these new tokens. Since we assume that
the token length is large enough to discourage a token exhaustive
attack, user $u$ cannot derive $K_G^+$, nor any future group key.
Therefore, forward security is guaranteed.

Theorem 5.2. The recovery protocol guarantees forward
security.

Proof. Given the assumptions on the cipher strength and key
length, the proof consists in showing that all group users but
colluding ones can derive the next group key $K_G^+$ from an
execution instance of the recovery protocol, and that colluding
users derive neither it nor any future group key.

Thanks to HANDLER RH1, every non-colluding user
belonging to a compromised subgroup retrieves the new group
key $K_G^+$. Then, thanks to HANDLER RH2, every user belonging to
a non-compromised subgroup retrieves the new group key $K_G^+$
as well. It follows that all non-colluding users are successfully
rekeyed.

Furthermore, colluding users do not hold cryptographic keys
used to protect rekeying messages. Thus, they are not able to
retrieve the new security material. Since we assume that the key
length is large enough to discourage a key exhaustive attack,
colluding users cannot derive $K_G^+$, nor any future group key.
Therefore, forward security is guaranteed.

Theorem 5.3. HISS guarantees forward security.

Proof. The proof descends directly from Theorem 5.1, in the
case of a single user’s leaving, and Theorem 5.2, in the case of
colluding users’ leaving.

Theorem 5.4. HISS guarantees backward security.

Proof. First, the join protocol provides current users with new
keys. Then, once current users have been rekeyed, the joining
user $u$ is provided with the new keys as well, i.e. $K_G^+$ and $K_S^+$. As a consequence, user $u$ never has knowledge of security
material used before its join, and thus is not able to access old
communications. Therefore, backward security is assured.

5.1. On rekeying message authenticity

Authenticity of rekeying messages specified in Sections 4.3.2
and 4.3.3 must be guaranteed. Otherwise, an adversary could
modify in-transit rekeying messages or inject fake ones, so
completely breaking the rekeying protocol. In this section, we
discuss techniques that could be used in HISS to assure the
authenticity of rekeying messages. The choice of one of them
is tightly related to the current application scenario, the network
technology, as well as the hardware capabilities of the network
devices.

Digital signatures are a typical and widely adopted
solution for providing rekeying messages authentication [6, 7].

Digital signature are quite onerous from a computation
and communication point of view [38]. The communication
overhead derives from the increased size of packets due
to the appending of the digital signature to the message
itself. In RSA-1024, the increment of a message size is 128
bytes. Since rekeying messages are ‘short’, a digital signature
may even be larger than the information it protects. Elliptic
Curve Cryptography (ECC) ameliorates this situation [41]. For
instance, ECC-160 digital signature is, roughly, an order of
magnitude faster than RSA-1024 and increases a message by
only 40 bytes while delivering the same security level.

While digital signatures are adequate for conventional
distributed applications (e.g. teleconferencing or content
distribution), they may result practically infeasible for resource-
constrained computing platform such sensor nodes in WSNs.
Here, even ECC-based digital signatures especially conceived
for sensor nodes may result too computing and energy
demanding [42, 43]. In this case, authentication techniques
more efficient than digital signatures have to be exploited.
For instance, group rekeying schemes for WSNs such as
LARK [14], S2RP [44, 45], µTESLA [46] and LEAP++ [47]
use hash-chains, an authentication mechanism deriving from
Lamport’s one-time password [48]. The advantage of an hash-
chain is that the current element in the chain can be efficiently
authenticated by computing its hash and verifying that the
result is equal to the previous element in the chain. Therefore,
it is sufficient to distribute the head of the chain in an
authenticated way, e.g. off-line or through a predefined point-
to-point authenticated channel, so that all the remaining items
can be automatically and efficiently authenticated.

In HISS, hash-chains may be used to authenticate both
the new group key and the new subgroup key in the leave
protocol messages $M_1$ and $M_2$, and the new group key
and the new subgroup tokens in the recovery protocol message
$M_s$. Therefore, the authentication of these quantities would
require just the execution of a hash function. Furthermore, these
quantities are ‘self-authenticating’, so avoiding the need for an
MAC or a digital signature which would increase the rekeying
and recovery message size. Finally, it is worthwhile to notice
that the contents of the recovery protocol message $M_u$ does not
require any special technique to be authenticated, neither the
digital signature nor the hash-chains, because it is encrypted by means of the user key.

Using hash-chains has cons too. In fact, the previous items in the chain can be computed from the current one by repeatedly computing an hash function. This implies that when a user joins the group, he is able to compute all the previous group keys. It follows that the backward security requirements does not hold anymore. However, backward security is not so crucial for WSN applications [14, 49]. In fact, they actually deal most with monitoring and control, where integrity is a top priority. In this application domain, supporting the backward security requirement is not critical from an integrity stand point. Actually, the adversary may only attempt to inject messages by using ‘past’ group keys, but they are supposed to be discarded by the monitoring and control algorithm.

6. PERFORMANCE EVALUATION

In this section, we evaluate the rekeying scheme performance with respect to the following factors: communication overhead, storage overhead and computing overhead. In this evaluation, we abstract away from the specific design and implementation choices (e.g. the cryptographic suite or the network technology). This means that we evaluate the storage overhead and the communication overhead as the number of information items that protocol actors have to store and broadcast. Similarly, we evaluate the computing overhead as the number of cryptographic operations performed during protocol instances, i.e. the number of encryptions, decryptions, and hash function executions. In any case, we perform a worst-case analysis, that is we analyse the largest overhead implied by a given protocol execution instance.

For simplicity, but without lack of generality, we assume that the system is organized in $p$ homogeneous subgroups containing $m$ users each. It follows that the group $G$ is composed of $n = p \cdot m$ users. We point out that HISS supports heterogeneous subgrouping policies, according to which different subgroups may contain different numbers of users. However, an homogeneous subgrouping policy allows us to perform an analytical performance analysis without any significant lack of generality.

While the communication and computing overhead is protocol-specific, the storage overhead is common to all protocols. For this reason, we analyse the storage overhead in a separate section (Section 6.1), and devote one section for the communication and computing overhead of each protocol (Sections 6.2–6.4).

In order to fix ideas, we also discuss the respective overheads of two paradigmatic applications, namely content distribution applications (e.g. Pay-TV) and WSNs applications. Such a discussion is contextual to every overhead type and thus distributed over Sections 6.1–6.4. As to the content distribution application, we consider a group composed of $2^{20}$ users, each one equipped with a commodity Personal Computer (PC), and interconnected by the Internet. As to the WSN application, we consider a group composed of 1024 sensor nodes (e.g. Mica2 or TmoteSky) interconnected by an IEEE 802.15.4 wireless network. Every sensor node is battery operated and equipped with a microcontroller, a small amount of memory.

Of course, these applications are not exhaustive of the full range of possible applications in which HISS can be used. However, they represent two extremes of this range and therefore allow us to give a concrete insight of the high scalability and practical sustainability of HISS.

6.1. Storage overhead

As to the storage overhead, the Key Manager KM stores the current group key $K_G$, all the $p$ current subgroup keys, all the $n$ user keys, all the $n$ user tokens, and all the $p$ subgroup tokens. Thus, the storage overhead for KM is $O_{s,km} = (2 \cdot p + 2 \cdot n + 1)$. On the other hand, every user $u \in S$, stores: (i) its user key $K_u$; (ii) the current group key $K_G$; (iii) the current subgroup key $K_G$; (iv) the $(m - 1)$ user tokens associated to its cognate users; and, finally, (v) the $(p - 1)$ subgroup tokens associated to all the subgroups different from $S$. Thus, the storage overhead for every user is $O_{s,u} = (p + m + 1)$.

Storage overhead $O_{s,km}$ at the Key Manager grows linearly with $n$. However, this is not a problem in practice because the Key Manager has plentiful of resources. The key point is that the HISS memory overhead is affordable at the user side. From this standpoint, it is important to notice that for $p \gg 1$ or $m \gg 1$, $O_{s,u} \simeq p + m$. Therefore, under the assumption that users are uniformly distributed in $p$ subgroups of $m$ members each, then the minimum storage overhead $O_{s,u}^{(\min)} = 2 \cdot \sqrt{n}$ for $p = m = \sqrt{n}$. Hereafter, we will assume this distribution of users to subgroups.

In the content distribution application, the minimum storage overhead at the user side is $O_{s,u}^{(\min)} = 2048$. If we consider that secrets have a size equal to 16 bytes, the same as an AES key [50], the memory overhead in bytes is 32768. As a PC is nowadays equipped with a few GBytes of RAM, the HISS memory overhead is practically negligible even in this case.

In the WSN application, the minimum storage overhead is $O_{s,u}^{(\min)} = 64$. If we assume that secrets (keys and tokens) have a size equal to 80 bits, then the storage overhead is 640 bytes. The rationale basis of this choice is that 80 bits is the size of keys of Skipjack, a secure and efficient cipher for WSN applications [51–53]. If the sensor nodes belong to the TmoteSky class, they are equipped with 48 kB of memory [54], and thus the memory overhead is the 1.30% of the total sensor node’s memory. If, instead, sensor nodes belong to the Mica2 class, they have 128 kB of memory, and thus the memory overhead becomes the 0.49% of the memory. It follows that the storage overhead of HISS is practically negligible even in resource-poor devices such as sensor nodes.
6.2. The leave overhead

Let us consider the leave protocol (see Section 4.3.2), and a user \( u \in S \) which leaves the group \( G \). The communication overhead accounts for messages M1 and M2. Message M1 introduces a communication overhead equal to three, as it conveys the identifier of the leaving user, the next group key \( K^+_{G} \), and the new subgroup key \( K^+_{S} \). Message M2 introduces a communication overhead equal to two, as it conveys the identifier of the subgroup \( S \) and the next group key \( K^+_{G} \) in its encrypted form. It follows that the total communication overhead is constant and equal to five, i.e. \( O^{(l)} = 5 \).

From these initial considerations, we can conclude that HISS provides a small and constant communication overhead that is independent of the group size and the adopted subgrouping strategy. This property is particularly important because it makes HISS highly scalable. Of course, in a real implementation setting, the possibility of exploiting this property strictly depends on the broadcast communication service provided by the underlying network technology.

As to the computing overhead, the worst case regards the operations performed by a user \( u \in S, v \neq u \). In particular, such a user is required to: (i) compute the key encryption key \( KEK_u \) by means of \( kdf() \); (ii) decrypt the message M1 to retrieve \( K^+_{G} \) and \( K^+_{S} \); (iii) update its own User Token Set UTS by executing \((m - 2)\) hash functions, and its own Subgroup Token Set STS by executing \((p - 1)\) hash functions. Thus, in the worst case, a user performs one decryption and \((p + m - 2)\) hash function executions.

On the other hand, the Key Manager has to: (i) compute the new group key \( K^+_{G} \) and the new subgroup key \( K^+_{S} \), as well as the two key encryption keys \( KEK_u \) and \( KEK_S \), by means of \( kdf() \); (ii) encrypt messages M1 and M2; and, finally, (iii) update its own User Token Set UTS by executing \((m - 1)\) hash functions, and its own Subgroup Token Set STS by executing \( p \) hash functions. Thus, the Key Manager has to perform 2 encryptions and \((p + m + 3)\) hash function executions.

As it turns out, the computing overheads of the Key Manager and users, respectively, have the same order of magnitude. Once again, since the Key Manager is plentiful of resources, the practical viability of HISS from the computing standpoint depends on the users. In the content distribution application, users are required to perform at most 1 decryption and 2046 hash function executions, which is practically negligible for modern PCs. In the WSN application, users are required to perform at most 1 decryption and 62 hash function executions, which is practically affordable for sensor nodes.

6.3. The recovery overhead

Let us consider the recovery protocol (see Section 4.3.2), with \( C \) compromised subgroups and \((p - C)\) non-compromised subgroups. For the sake of simplicity, but without any lack of generality, let us assume that every compromised subgroup contains \( c \) colluding captured users. It follows that the overall captured users are \((c \cdot C)\), whereas non-captured users are \((n - c \cdot C)\).

The communication overhead accounts for messages of type \( M_u \) and \( M_s \). For every compromised subgroup, the Key Manager sends a message of type \( M_u \) to every non-captured node in the subgroup. The size \( \|M_u\| \) of a message of type \( M_u \) is \( \|M_u\| = p + m - c \), as the message conveys the new group key \( K^+_{G} \), the new subgroup key \( K^+_{S} \), the new User Token Set UTS, and the new Subgroup Token Set STS. As the number of compromised subgroups is \( C \), and the number of non-captured users in every compromised subgroup is \( m - c \), the total communication overhead due to messages of type \( M_u \) is \( C \cdot (m - c) \cdot (p + m - c) \).

A message of type \( M_S \) is broadcast to every non-compromised subgroup \( S \in \mathcal{U} \). Thus, the number of these messages is \((p - C)\). The size \( \|M_S\| \) of a message of type \( M_S \) is \( \|M_S\| = p \), as it conveys the new group key \( K^+_{G} \) and the new Subgroup Token Set STS. Therefore, \( M_S \) messages introduce a total overhead equal to \( p \cdot (p - C) \). Consequently, the total communication overhead of the recovery protocol is equal to \( O^{(r)} = [C \cdot (m - c) \cdot (p + m - c)] + p \cdot (p - C) \).

If we reasonably assume that (i) each subgroup includes a non-negligible number of members, i.e. \( m \gg 1 \); (ii) a few users per subgroup are captured, i.e. \( m \gg c \); (iii) a few subgroups are compromised, i.e. \( p \gg C \); and (iv) \( p = m = \sqrt{n} \), for storage optimization, then \( M_u, M_S \) and \( O^{(r)} \) can be approximated as follows: \( \|M_u\| \simeq 2 \cdot \sqrt{n} \), \( \|M_S\| \simeq \sqrt{n} \) and \( O^{(r)} \simeq (2C + 1) \cdot n \). As it turns out, the total communication overhead linearly grows with \( C \), so displaying a smooth degradation behavior.

The total communication overhead also grows linearly in \( n \). However, an execution of the recovery protocol has to be considered an exceptional event with respect to an execution of the leave protocol, that is instead a normal event. So, according to the well-known Lampson’s recommendations for computer systems design [16], while the leave protocol must be efficient and scalable, the recovery protocol must be able to make some progress. However, it is crucial that the recovery protocol is sustainable. In the rest of this section, we argue that this is indeed the case.

In the content distribution application, we have \( \|M_u\| = 32 \) kB, \( \|M_S\| = 16 \) kB and \( O^{(r)} = 48 \) MB, an amount of traffic that is fully sustainable in a conventional network of commodity PCs.

In the WSN application, we have \( \|M_u\| = 640 \) bytes, \( \|M_S\| = 320 \) bytes and \( O^{(r)} = 30 \) kB. Let us consider the IEEE 802.15.4 communication protocol [55], where unsecured frames have a payload whose size can be up to 102 bytes. Thus, every message \( M_u \) requires seven frames, whereas the transmission of the whole \( O^{(r)} \) overhead requires 302 frames. In an implementation of IEEE 802.15.4, the effective data rate (i.e. excluding headers, CRCs, and control packets) is about 8.4 Kbps (out of 250 Kbps), thus the transmission of every message \( M_u \) requires about 610 ms (per-hop), whereas the
transmission of the whole $O^{(r)}_G$ overhead requires about 29.25 s (per-hop). It follows that also in this case the communication overhead is sustainable. However, it is worthwhile to note that IEEE 802.15.4 is conducive to provide better performance. For instance, Latrè et al. [56] showed that a throughput of about 140 Kbps can be achieved in IEEE 802.15.4, even if Acknowledgment frames are transmitted. In such a case, the transmission of a message $M_u$ requires about 36.57 ms (per-hop), whereas the transmission of the whole $O^{(r)}_G$ overhead requires 1.75 s (per-hop).

As to the computing overhead, each user performs only one decryption, in order to decrypt either a message $M_u$ or a message $M_S$. On the other hand, the Key Manager has to: (i) compute $K^{+}_G$, $(p - C)$ new subgroup keys, $C \cdot (m - c)$ new user tokens, and $p$ new subgroup tokens by means of kdf; (ii) encrypt $C \cdot (m - c)$ messages $M_u$ and $(p - C)$ messages $M_S$. Thus, the Key Manager has to perform $p + C \cdot (m - c - 1)$ encryptions and $2 \cdot p + C \cdot (m - c - 1) + 1$ hash functions.

The computing overhead at the Key Manager side is generally not a problem because the Key Manager may be implemented on a high-performance server platform. Therefore, the actual sustainability from the computing standpoint practically depends on the user side.

In the content distribution application, messages $M_u$ and $M_S$ are 32 and 16 kB in size, respectively. Decrypting these amounts of data by means of a symmetric cipher does not constitute a problem on any commodity PC.

In contrast, more attention must be paid in the WSN application scenario, where users are sensor nodes with constrained computing capabilities. In such a case, messages $M_u$ and $M_S$ are 640 bytes and 320 bytes in size, respectively. On a Mica2 node, the performance of a software-version of Skipjack is 25 µs per encrypted/decrypted byte [53], whereas, on a TmoteSky node, it is 77 µs [14]. Hence, decrypting a message $M_u$ or a message $M_S$ takes 16 and 8 ms, respectively, on a Mica2 node, and 49.28 and 24.64 ms, respectively, on a TmoteSky node. This makes the HISS recovery protocol affordable from the computing standpoint too.

6.4. The join overhead

Let us consider the join protocol (Section 4.3.3), and a user $u$ which wants to join the group $G$ and become a member of the subgroup $S$. An execution instance of the protocol requires a constant number of rekeying messages, namely (i) a broadcast message providing the new group key $K^{+}_G$; and (ii) a broadcast message providing the new subgroup key $K^{+}_S$ and the user token $t_u$ to the members of $S$. Thus, the total communication overhead amounts to three, i.e. $O^{(1)}_G = 3$.

As to the computing overhead, the worst case regards the operations performed by the members of the subgroup $S$. That is, each one of them is required to perform two decryptions, in order to retrieve $K^{+}_G$ and $K^{+}_S$ and $t_u$ from messages M1 and M2, respectively. On the other hand, the Key Manager has to:

(i) compute the new group key $K^{+}_G$, the new subgroup key $K^{+}_S$, and the user token $t_u$ and (ii) encrypt messages M1 and M2.

Thus, the Key Manager has to perform 2 encryptions and 3 hash function executions.

7. COLLUSION MANAGEMENT

So far, we have shown that if two captured users belonging to the same subgroup collude, then the whole subgroup is compromised. However, we have also shown that recovering a compromised subgroup requires only to totally recover such a subgroup rather than the whole group. The remaining uncompromised subgroups can then be efficiently rekeyed by means of a single broadcast message each.

In this section, we argue that a proper allocation of users to subgroups can make a successful collusion attack practically unfeasible. As we consider a form of practical security, we cannot abstract away from the application scenario and the hypothesized adversary strength. So, given an application scenario, we have first to assess the adversary strength, and then devise an allocation strategy that is consistent with the application constraints and requirements, and that makes a collusion attack practically unfeasible for the alleged adversary. This is a common process in security management [57].

In any case, the chosen allocation strategy has to satisfy the general principle that, for an adversary, it must be practically unfeasible to (i) determine the subgroups membership or (ii) compromise two or more users belonging to the same subgroup.

In the former case, the only strategy for the adversary consists in exhaustively attacking users in the attempt to compromise two users belonging to the same subgroup. The system is practically secure if the resources required to carry out this exhaustive attack strategy exceed the resources of the hypothesized adversary. For instance, if we consider the Internet and its magnitude, determining subgroups and their membership through traffic analysis is a practically unfeasible task.

In the latter case, subgroup membership is known to the adversary, who can thus target the attack to well-defined users. Therefore, the system is secure if the resources required to attack, physically or logically, two or more users belonging to the same subgroup exceed the resources of the adversary. In this case, off-the-shelf protection measures and security engineering best-practices can be employed [58, 59].

In order to fix ideas, we refer again to the application scenarios that so far have constituted the leading examples of this paper, namely large-scale content distribution and WSNs applications (see Section 6). In particular, for each application scenario, we will discuss a possible instantiation of the allocation strategy principle, that depends on the specific application requirements and constraints as well as the related hypothesized adversary.
In both cases, we consider an homogeneous allocation of users to subgroups, with $p$ subgroups containing $m$ members each, and a total group size $n = p \cdot m$. Once again, it is arguable that, although not exhaustive, they, however, constitute two meaningful examples to get a concrete insight of the HISS flexibility in collusion management.

7.1 Large-scale content distribution

Given the large-scale nature of this distributed application, we reasonably assume that the adversary is not able to observe the whole network traffic and analyse it. It follows that it is practically unfeasible for the adversary to deduce subgroups membership and thus perform an adaptive attack. Therefore, if users are randomly allocated to subgroups, the only possible strategy for the adversary is to compromise users at random as well.

As a measure of resistance to collusion attack, we consider the probability $P(t)$ that the adversary has compromised at least one subgroup after $t$, $1 < t < p$, users have been captured. Of course, for $t = p$, at least two captured users belong to the same subgroup.

Initially, we compute the probability $Q(t)$ of the complementary event, i.e. the probability that no subgroups have been compromised after $t$ captures. By definition $P(t) = 1 - Q(t)$. In Appendix 1, we show that $Q(t)$ may be expressed as follows:

$$Q(t) = \prod_{i=1}^{t-1} \frac{n - i \times (m - 1)}{n}.$$ 

Let $\alpha = (m - 1)/n$. If we reasonably assume that $n \gg m$, then $\alpha \ll 1$, and thus it is possible to approximate $Q(t)$ at the first order, i.e. $Q(t) \simeq 1 - \sum_{i=1}^{t-1} i \times \alpha = 1 - \alpha \times t(t - 1)/2$. It follows that $P(t) \simeq t(t - 1)/2 \times \alpha$. If, hopefully, $t \gg 1$, then $P(t) \simeq 1/2 \times \alpha \times t^2$. In the case $m \gg 1$, then we can approximate $\alpha$ as $\alpha \simeq m/n$ and we obtain $P(t) = 1/2 \times m/n \times t^2$. Thus, if we wish that $P(t) < \epsilon$, for an arbitrarily small $\epsilon$, then we obtain $m/n < 2\epsilon/t^2$.

If we consider $n = 2^{20}$ and require that after $t = 100$ user captures the probability of having at least one compromised subgroup is no greater than $\epsilon = 0.5$, then a possible choice consists in $p \geq 10^4$ subgroups, containing at most $m \leq 100$ members each. On the other hand, if we wish to minimize the storage overhead and set $m = \sqrt{n}$, then an adversary has to capture $t = 32$ users in order to have a probability $\epsilon = 0.5$ of compromising a subgroup. As it turns out, HISS makes it possible to trade off security against resource consumption, while keeping rekeying scalability high.

This allocation strategy can be complemented by other countermeasures. As a possible countermeasure, we may consider a reactive one, where the Intrusion Detection System (IDS) detects the colluding captured users. Of course, the larger the $t$ the larger the time at hand for the IDS, and the more numerous the adversary’s traces. Another possible countermeasure might be preventive and rely upon secure hardware [60]. For instance, in Pay-TV applications, a subscriber holds a smart card containing credentials and keys to exploit the subscribed service. Then, we could add the security material necessary for HISS to the smart card, especially the User Token Set. By doing so, as smart cards can be considered tamper-resistant to some practical extent [61], it would be no longer practically possible to jeopardize the HISS rekeying procedure, and collusion would not be an actual problem anymore.

7.2 WSN applications

In this application scenario, we could exploit the random allocation strategy discussed in Section 7.1. This strategy would be useful when, for example, sensors are deployed randomly. According to this strategy, if we consider a probability $\epsilon = 0.5$ of compromising at least one subgroup, then at least $t = 8$ sensor nodes captures are necessary in a WSN organized in $p = 64$ subgroups of $m = 16$ members each. Alternatively, at least $t = 16$ sensor nodes captures are necessary for $p = 256$ subgroups of $m = 4$ members each. If these numbers of nodes captures are too small with respect to the alleged adversary strength, then one can devise more application-specific allocation policies that better contrast a collusion attack performed by that specific adversary.

WSNs are often employed in monitoring, surveillance and control of buildings, plants or critical infrastructures. In these application scenarios, we can assume that an adversary is able to physically break into a given area, and capture ‘neighbouring’ sensor nodes, possibly all, that are deployed in that area. Once a sensor node has been captured, all the secrets it holds become known to the adversary. The adversary can thus gather secrets possibly from all sensor nodes in that area, and possibly collect enough information to be able to gain access to the group key. It has been experimentally shown that an equipped and experienced adversary is able to obtain copies of all the memory and data of, for instance, a Mica2 mote in tens of seconds, or minutes, after a node is captured [62].

On the other hand, due to the reduced amount of services provided by sensor nodes, it is believed that a WSN presents a number of vulnerabilities which is smaller with respect to computers typically connected to the Internet. Since there are less functionalities and less complex code, there are less software bugs too. Furthermore, as sensor nodes are resource-constrained, programmers spend more time per line of code in sensor network applications, than in applications for regular computer networks [49].

Thus, it is the size of the area the adversary can physically compromise that gives a practical measure of the adversary’s strength. Such a ‘critical’ value is typically defined at pre-deployment after a threat analysis [57]. It follows that the problem that has to be solved consists in devising an allocation and deployment policy such that any area not larger than
the critical size contains sensor nodes belonging to different subgroups with high probability.

It is well known that devising an optimal allocation policy is in general an NP-hard problem [13, 25]. Therefore, we have to resort to heuristics for the assignment of users to subgroups that reduce, or even practically eliminate, the probability of successful collusion attacks. Such heuristics depend on both the adversary model and application-specific requirements and constraints. In this section, we discuss a real case-study, the ‘Highly Automated Airfield’ (HAA) reference application of the PLANET European integrated project [63].

The HAA application provides the support for automated airfield operations involving cooperating unmanned air vehicles (UAVs). In particular, by means of a WSN, the HAA application intends to monitor the runways of an airport in order to permit automated take-offs and landings of UAVs. The HAA application considers a WSN composed of \( n \) sensor nodes, evenly distributed on both sides of every runway to monitor both the runway occupancy and the presence of obstacles. As the airport is provided with a physical land surveillance system, we assume that an adversary can compromise a portion \( L \) of a runway, possibly spanning both sides (see Fig. 3), and thus capture the \( w \) ‘neighbouring’ sensors deployed therein. The adversary cannot move from one runway to another without being caught.

In order to avoid collusion attacks, we organize sensor nodes into \( p = 2 \cdot w \) different subgroups with \( m = n / (2 \cdot w) \) members each, and cyclically allocate sensor nodes to subgroups. Figure 3 shows an example for \( w = 5 \). Since the adversary is not able to compromise a number of sensor nodes larger than \( w \), nor to move from one runway to another, then he cannot successfully perform a collusion attack. It is worthwhile to note that the application-specific constraints and requirements have allowed us to reduce the allocation problem to a single-dimension problem, and thus devise an effective allocation strategy that prevents collusion attack under the considered threat model.

Table 3 reports the storage overhead in the TmoteSky and Mica2 platforms, for different values of \( w \). The column ‘Memory’ reports the amount of storage necessary for the sensor node to store the HISS keys and tokens (see Section 6). The columns ‘TmoteSky’ and ‘Mica2’ specify, respectively, the percentual impact of this amount of storage on the total memory available on the platform. It follows that the proposed allocation policy has a practically affordable storage overhead. Actually, in the most extreme case of tolerating the capture of \( w = 256 \) sensor nodes, the overhead is around 10% on the TmoteSky platform and 4% on the Mica2 platform.

### Table 3. Highly Automated Airfield: storage overhead.

<table>
<thead>
<tr>
<th>( w )</th>
<th>Memory (bytes)</th>
<th>TmoteSky (%)</th>
<th>Mica2 (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>800</td>
<td>1.66</td>
<td>0.62</td>
</tr>
<tr>
<td>64</td>
<td>1360</td>
<td>2.83</td>
<td>1.06</td>
</tr>
<tr>
<td>128</td>
<td>2600</td>
<td>5.41</td>
<td>2.03</td>
</tr>
<tr>
<td>256</td>
<td>5140</td>
<td>10.70</td>
<td>4.01</td>
</tr>
</tbody>
</table>

In order to avoid collusion attacks, we organize sensor nodes into \( p = 2 \cdot w \) different subgroups with \( m = n / (2 \cdot w) \) members each, and cyclically allocate sensor nodes to subgroups. Figure 3 shows an example for \( w = 5 \). Since the adversary is not able to compromise a number of sensor nodes larger than \( w \), nor to move from one runway to another, then he cannot successfully perform a collusion attack. It is worthwhile to note that the application-specific constraints and requirements have allowed us to reduce the allocation problem to a single-dimension problem, and thus devise an effective allocation strategy that prevents collusion attack under the considered threat model.

The HAA application provides the support for automated airfield operations involving cooperating unmanned air vehicles (UAVs). In particular, by means of a WSN, the HAA application intends to monitor the runways of an airport in order to permit automated take-offs and landings of UAVs. The HAA application considers a WSN composed of \( n \) sensor nodes, evenly distributed on both sides of every runway to monitor both the runway occupancy and the presence of obstacles. As the airport is provided with a physical land surveillance system, we assume that an adversary can compromise a portion \( L \) of a runway, possibly spanning both sides (see Fig. 3), and thus capture the \( w \) ‘neighbouring’ sensors deployed therein. The adversary cannot move from one runway to another without being caught.

In order to avoid collusion attacks, we organize sensor nodes into \( p = 2 \cdot w \) different subgroups with \( m = n / (2 \cdot w) \) members each, and cyclically allocate sensor nodes to subgroups. Figure 3 shows an example for \( w = 5 \). Since the adversary is not able to compromise a number of sensor nodes larger than \( w \), nor to move from one runway to another, then he cannot successfully perform a collusion attack. It is worthwhile to note that the application-specific constraints and requirements have allowed us to reduce the allocation problem to a single-dimension problem, and thus devise an effective allocation strategy that prevents collusion attack under the considered threat model.

Table 3 reports the storage overhead in the TmoteSky and Mica2 platforms, for different values of \( w \). The column ‘Memory’ reports the amount of storage necessary for the sensor node to store the HISS keys and tokens (see Section 6). The columns ‘TmoteSky’ and ‘Mica2’ specify, respectively, the percentual impact of this amount of storage on the total memory available on the platform. It follows that the proposed allocation policy has a practically affordable storage overhead. Actually, in the most extreme case of tolerating the capture of \( w = 256 \) sensor nodes, the overhead is around 10% on the TmoteSky platform and 4% on the Mica2 platform.

### 8. CONCLUSIONS

In this paper, we have presented HISS, a highly scalable scheme for group rekeying based on logical subgrouping. We believe that HISS has the following merits.

(i) HISS allows for securely and efficiently performing group rekeying upon a user join or leave, thus assuring both backward and forward security. In order to do that, HISS requires a number of rekeying messages that is small, constant and independent of the group size.

(ii) In terms of memory occupancy, HISS requires every user to store \( O(\sqrt{n}) \) secrets, which is an affordable storage overhead in the most practical cases even encompassing resource-scarce devices such as sensor nodes.

(iii) HISS provides a recovery protocol aimed at restoring group security upon a collusion attack. The recovery protocol does not require to re-initialize the group, it is affordable on customary platforms, and displays a form of graceful degradation, namely, the protocol
communication performance degrades with the number of compromised subgroups.

(iv) HISS makes it possible to define policies of allocation of users to subgroups that practically contrast or even prevent successful instances of collusion attacks. We have shown two practical examples of these policies.

ACKNOWLEDGEMENTS

We thank the anonymous referees for having helped us to improve our work with their precious comments and recommendations.

FUNDING

This work was supported by EU FP7 Network of Excellence CONET [grant number FP7-224053]; and EU FP7 Integrated Project PLANET [grant number FP7-257649].

REFERENCES


THE COMPUTER JOURNAL, 2012


APPENDIX 1. ON DETERMINING $Q(T)$

Let us suppose that initially the adversary has compromised a given user, namely $u$. Hence, $t = 1$. Then, the adversary compromises another user, namely $v$. Hence, $t = 2$. No subgroup has been colluded if $v$ belongs to a different subgroup than $u$. This occurs with probability $Q(2) = (n - (m - 1))/n$.

Then, the adversary compromises another user, namely $z$. Hence, $t = 3$. No subgroup has been compromised if $z$ belongs to a different subgroup than $u$ and $v$. This occurs with probability $Q(3) = (n - 2 \times (m - 1))/n \times (n - (m - 1))/n$.

By repeating this reasoning, after $t$ captures, the probability that no subgroup has been compromised is

$$Q(t) = \frac{n - (t - 1) \times (m - 1)}{n} \times \cdots \times \frac{n - (m - 1)}{n},$$

i.e. $Q(t) = \prod_{i=1}^{t-1} (n - i \times (m - 1))/n$. 


