Technical Report #319, ISBN 82-7368-272-2
Department of Informatics, University of Oslo
February 2005

SKiMPy: A Simple Key Management Protocol for MANETs in Emergency and Rescue Operations

Matija Pužar[1], Jon Andersson[2], Thomas Plagemann[1], Yves Roudier[3]

[1] Department of Informatics, University of Oslo
{matija, plageman}@ifi.uio.no
[2] Thales Communications
jon.andersson@no.thalesgroup.com
[3] Institut Eurécom
yves.roudier@eurecom.fr

ABSTRACT

Mobile ad-hoc networks (MANETs) provide the technical platform for efficient information sharing in emergency and rescue operations. Some the data present on the scene is highly confidential and requires protection. However, one of the main threats to a network is insertion of false data or alteration of existing ones, which could easily lead to network disruption and, ultimately, cause loss of human lives. This paper presents a simple and efficient key management protocol, called SKiMPy. The protocol allows devices carried by the rescue personnel to establish a symmetric shared key, to be used primarily to perform message signatures. The protocol is designed and optimized having in mind the high dynamicity present in such a scenario. Two different implementations were made, first as a standalone application and later as a plugin for the Optimized Link State Routing Protocol (OLSR). We present the evaluation results for both implementations and, in addition, for the latter one we describe in detail the emulation platform developed to test and evaluate this and other MANET protocols.

1. INTRODUCTION

Efficient collaboration between rescue personnel from various organizations is a mission critical key element for a successful operation in emergency and rescue situations. There are two central requirements for efficient collaboration, the incentive to collaborate, which is naturally given for rescue personnel, and the ability to efficiently communicate and share information. Mobile ad-hoc networks (MANETs) could provide the technical platform for efficient information sharing in such scenarios, assuming that all rescue personnel is carrying and using mobile computing devices with wireless network interfaces.

Wireless communication is by nature more susceptible to eavesdropping, compared to the other media. In most cases, the data involved should not be available to third parties, such as malicious persons, like arsonists, or curious journalists who might make confidential data public. Another requirement is to prevent third parties from inducing false data. At the application layer this might for example lead to wrong management decisions. At the network layer it has been shown that a very few percent of misbehaving nodes could easily lead to network disruption and partitioning [12]. In both cases, efficiency of the rescue operation will be drastically reduced and might ultimately cause loss of human lives. In order to prevent such a disaster, all data traffic should be properly signed, allowing only authorized nodes the possibility to perform authentication and integrity checking. Given that the devices carried by the rescue personnel will mostly have scarce resources, asymmetric cryptography is too costly to be used the whole time.

This paper describes a simple key management protocol, called SKiMPy, used to establish a symmetric shared key between the rescue personnel's devices. This approach provides the means to establish a secure network infrastructure between authorized nodes, while keeping out unauthorized ones. In addition, at the application layer it may be decided whether the established shared key will be used to encrypt data as well. SKiMPy is designed and optimized for highly dynamic ad-hoc networks and it is completely autonomous, requiring no user interaction at all. This is also an important factor, having in mind that rescue personnel does not have time to think or care about details of lower layers of the network infrastructure.

To facilitate the development and testing of the protocol, we developed an emulation test bed which we here describe in detail in this paper. Using an emulator instead of a simulator, allows us to test and evaluate protocols by means of real processes, without the need for rewriting code when the applications are transferred to genuine wireless devices.

The main contribution of this paper is a key management protocol for emergency and rescue operations and all other application domains where it is possible for the nodes to authenticate each other without need for contacting a centralized server. Another contribution is a flexible, Linux based emulation environment that can emulate mobility traces generated for ns-2.

The organization of the paper is as follows. In Section 2 we present related work, followed by a detailed description of our protocol in Section 3. Section 4 describes the emulation test bed. In Sections 5 and 6 we show two different implementations of the protocol and results of their evaluations. Finally, conclusion and future work are presented in Section 7.

2. RELATED WORK

Key management protocols can be roughly divided into three categories [5].

The first one relies on a fixed infrastructure and servers that are always reachable. Since we never know where accidents will happen, and we should expect them to happen at worse possible places, we cannot rely on the fact that fixed infrastructure will be present.

The next category comprises contributory key agreement protocols, not suited for our scenario for several reasons. Such protocols ([1], [3], [7], [23], [24], to name a few) are based on Diffie-Hellman two-party key exchange [8] where all the nodes give their contribution to the final shared key, causing rekeying every time a new node joins or an existing node leaves the group. In an emergency and rescue operation, we can expect nodes to pop up and disappear all the time, often causing network partitioning and merging. Therefore, using contributory protocols would cause a lot of computational and bandwidth costs which cannot be afforded. Besides, most of these protocols rely on some kind of hierarchy (chain, binary tree, etc.) and a group manager to deploy and maintain shared keys. In a highly dynamic scenario this approach would be quite ineffective. Another reason why such protocols are not suited for us, is that in order for the nodes to be able to exchange keys, a fully working routing infrastructure has to be established prior to that. Since the routing protocol is one of the main things we need to protect, this is a major drawback.

The last category are protocols based on key pre-distribution. The main characteristic of such protocols is that a group of nodes can compute a shared key out of pre-distributed sets of keys present on each node. These sets of keys are either given by a trusted entity before the nodes come to the scene [2], [17], or chosen and managed by the nodes themselves, as it is done in DKPS [5]. Although DKPS seems to be closest to SKiMPy in some aspects, there is a major difference. DKPS relies on the notion of distributed trust, whereas we make use of pre-installed certificates to perform authentication of nodes, explained in detail in the next section.

3. BASIC SKiMPy ELEMENTS

Our protocol was primarily designed for emergency and rescue operations, which we have studied in-depth in our project. In such scenarios, we must expect a very dynamic and unpredictable scenario. New nodes frequently appear and others disappear causing frequent network parti-

tioning and merging. Nodes carried by the rescue personnel will be relatively small and as such scarce on resources, so the computational and bandwidth costs of the key management protocol must be as low as possible. There is no point of establishing a secure network that afterwards cannot provide sufficient resources for the applications to be useful.

3.1 Authentication

An important characteristic of an emergency and rescue scenario is that the organizations involved (police, fire department, paramedics, etc.) are often well structured, public entities. Some of them might have sensitive data on the scene, like medical or police records, that are highly confidential and should remain such. Before the rescue personnel of the different organizations comes to the rescue scene, all devices are prepared for their tasks. One task in the preparation phase, which we call *a priori* phase, is the installation of valid certificates. The certificates are signed by a commonly trusted authority, such as the ministry of internal affairs, ministry of defense, etc., on the top of the trust chain. This gives nodes possibility to authenticate each other without need for contacting a third party. Therefore, there is no need for a fully self-organized public-key management system that does not rely on trusted authorities, as it is presented by Capkun et al. [4]. The advantages are twofold: first, the data in the network is more secure. This is a central requirement for our application domain, because highly sensitive data might be present in the air. Second, establishing trust and agreeing on a shared key is much more efficient, i.e., faster and less resources are consumed, because we preinstall certificates on all devices during the *a priori* phase.

So far, we have kept the question whether the certificates on the nodes will identify devices, or actually users handling them (who would then present the certificate to the device by means of a token, i.e. smartcard), open. The decision for this question does not impact the key management in SKiMPy, but it impacts the way how lost and stolen nodes are handled, i.e., revoking certificates and blacklisting of such nodes. We explain this later in Section 3.5.

Using asymmetric cryptography is expensive in many terms. With regards to the bandwidth, certificates could become large messages if several levels in the certificate chain are present, and as such their presence in the air should be reduced to a minimum. The authentication process using asymmetric cryptography is also computationally very expensive, which in our case becomes a problem with regards to battery power. In order to save resources as much as possible, our protocol makes the nodes learn about their neighborhood before acting, thus reducing the number of performed authentications. This is possible due to the fact that all nodes directly trust the same certificate authority and, therefore, if a node has been successfully authenticated before and has received the shared secret, we implicitly trust it.

3.2 Choosing Keys

During the initialization phase, each node generates a random key with a random ID number. It is the task of SKiMPy to make sure that all the nodes agree on a shared key. This shared key is always selected from nodes' initial keys. To achieve this, we introduce the notions of "better" and "worse" keys, together with the relation ">" representing "better than". There are several possible schemes for deciding which of the keys is *better* or *worse* and all schemes can be equally valid, as long as they cannot cause key exchange loops, are unambiguous and possibly transitive: *(A > B* and *B > C) => A > C*. The necessary control information, which depends on the scheme chosen, is always sent with the message signature.

We will briefly describe two examples and their advantages and drawbacks.

The first scheme is quite straightforward, the key having a higher or lower ID number, timestamp or a similar parameter, is considered to be better. The advantage of this scheme is that it is unambiguous, transitive and easy to implement. In addition, it can be "tweaked" in a way that would prevent a single node to cause rekeying of an already established network cell. For example, if the scheme defines that the lower ID number means a better key, the highest bit of the ID number can be always set to "1" when the node is turned on, and cleared once two nodes merge.

Assuming that nodes in a certain area will in most cases pop up independently, this simple and yet efficient method might prevent a lot of unnecessary rekeying traffic. If we use the keys' timestamps instead of the ID numbers, choosing a lower timestamp could imply that the key is older and that more nodes have it already. This assumption might not necessarily be true, especially if the key creator's clock was heavily out of sync, but it is worth considering. One major drawback of this scheme is that a small cell (consisting of, for example, 2 nodes) could easily cause rekeying of a much bigger cell (having, for example, 100 nodes), which would be a waste of resources.

The second scheme takes care of this problem by tracking the number of nodes in each network cell. The simple rule for this scheme is to always rekey the smaller cell, i.e. the one with the lower number of nodes, thus minimizing resource consumption for the necessary rekeying. The approximate number of nodes can be either retrieved from the routing protocol state information (if, for example, the OLSR routing protocol [6] is used) or maintained at a higher protocol layer, as it is done in our project. Even though this scheme is tempting, there might be a small problem which would make it impractical. If not all of the nodes have exactly the same information (which is to be expected in a dynamic scenario), and for some obscure reason we have two separate merging processes between same two cells, happening at the same time, a key exchange loop may occur. One approach to this problem is to adjust in each node the state information of the number of nodes in its cell, always increasing it when new nodes join, but never decreasing it upon partitioning of the cell.

An in-depth study of these two schemes and their variations is subject to ongoing and future work.

3.3 Key Distribution

Once a node gets a new key as a result of network merging, the key should be deployed within its previous network cell. There are several ways to achieve this:

- *Proactively* - each node receiving the key immediately forwards it to the others. This approach would generate a lot of unnecessary network traffic and is not recommended.
- *Proactively, using Multipoint Relays (MPRs)* - MPRs are selected nodes within the OLSR routing protocol used to broadcast messages into the whole network cell, so that the number of redundant retransmissions is reduced.
- *Reactively* - when a node receives a key, it does nothing. Only after detecting a message sent by a neighbor and signed with the old key, the node sends the key further. This approach uses less resources, but it takes more time for the whole cell to get a stable key.
- *Combination* - the first node getting the new key (that is, the node which performed the merge) immediately forwards the key to its one-hop neighbors, since it knows that no other node in its previous cell has it yet. The other nodes do not distribute it right away, but rather when (if) they notice that a certain node was "left behind".

In any of the given cases, the new key is encrypted using the old one before sending, giving all the other nodes the possibility to immediately start using it. The old key is then saved for a certain amount of time, for possible latecomers. This can be done because in this particular case the rekeying was not performed explicitly for the purpose of preventing traffic analysis attacks.

In our implementations, described in Sections 5 and 6, we use the *combination* approach.

3.4 Key Update

When created, each key has a companion key (called *update key*) used to periodically update it. The update key is never used on traffic that goes onto the network and therefore it is prone to traffic-analysis attacks. Once the time comes to change the key, the new key can be computed using one-way hash functions such as SHA-1 [9] or MD5 [19], ensuring backward secrecy in the case the key gets broken at some stage.

In addition to the ID of the key used to sign it, a message contains also the update-number saying how many times the key on the sender-node has been updated. That way, the receiver can easily compute the new key if it notices a mismatch, which could happen since we can't

expect all the nodes to perform the update at exactly the same time.

3.5 Exclusion of Nodes

Internal attacks, that is, attacks from the insiders, are in general much more difficult to cope with than the external ones [15]. Once authenticated, a node is a fully trusted member of the network. This poses the evident problem of how to exclude such a node once the device has been lost or, even worse, stolen by a malicious person. The following measures should be taken in order to ensure that such a node stays out of the network.

First, its certificate should be revoked, preventing the node from re-authenticating later at some stage. Since there is no central authority, there should be a way to decide which node or person can perform the task of revoking certificates. If the certificates contain also additional attributes such as rank or role of the persons (assuming that the certificates do in fact represent persons, not devices), it can be decided that only certain roles/ranks (such as *leader*) can perform revocation and blacklisting. In theory, the leaders' devices might also be stolen, but in practice they should normally be physically well protected. It is important to ensure that the compromised node itself does not revoke and blacklist legitimate ones or, even worse, the whole network.

Next, the node's IP address should be put on a common blacklist, assuming that IP addresses are bound to the certificates (as presented in e.g. [18]) and, thus, cannot be changed. Traffic coming from blacklisted nodes must be discarded at the lowest possible layer and, in case legally signed traffic coming from a blacklisted node is detected, the compromised key must be removed and re-authentication and rekeying should be done immediately.

3.6 SKiMPy Phases and Messages

The protocol consists of three phases:

I. Neighborhood Discovery

II. Batching

III. Key Exchange

During phase I, a node listens to all traffic sent by its immediate neighbors. If it detects a node using a *worse* key, it will send an *Authentication Request* message to it, saying it is willing to pass on its key. Upon receiving such a message, the other node enters the phase II, waiting for possible other authentication requests before sending a response. This waiting period is used for optimization - a node will only perform authentication with the *best* of all neighbors. All the other keys will, due to the transitiveness property of the *better than* relation, eventually get overruled and therefore there is no point in getting them in forehand. After the node has chosen its peer, it sends an *Authentication Response* after which its peer initializes the actual authentication procedure, that is, exchange of certificates, establishing a secure tunnel, and finally transfer of the key. The reason for having such a handshake procedure is to ensure that the nodes can indeed communicate, because in some standards, such as 802.11b [14], traffic like broadcast messages can be sent on a lower transmitting rate with larger transmission range than data messages. Thus, broadcast messages might reach a remote node and trigger a key exchange, even though the nodes cannot directly exchange data packets.

The protocol is built on the following messages:

- *Authentication Request* (AUTH_REQ): sent by a node after it detects traffic from a node having a key that is *worse* than its own one. The message is used to inform the remote node that the sending node is willing to transfer its key.
- *Authentication Response* (AUTH_RESP): sent by a node, as a result of a received AUTH_REQ message. The message is used to inform the remote party that the sender is willing to perform the authentication and receive the remote and *better* key.

The authentication procedure prior to the key exchange includes establishment of a secure tunnel through which the key will be securely transferred. Furthermore, SKiMPy makes use of the existing traffic in the network to trigger key exchange. In this example, periodic routing beacons (HELLO) are used, as they are sent by proactive routing protocols.

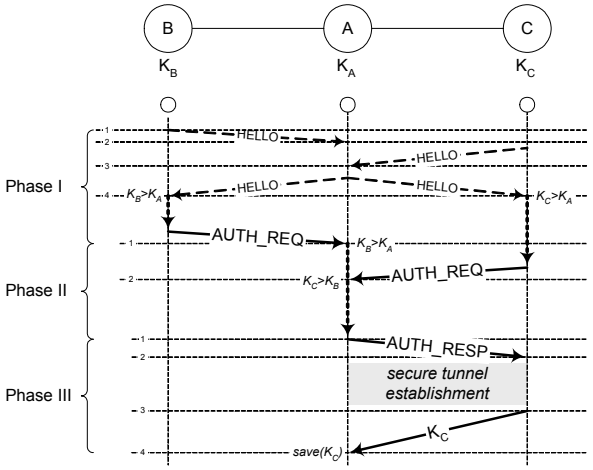

Figure 1. Message Flow Diagram

Figure 1 shows an example of the key exchange between three nodes (**A**, **B** and **C**) and indicates the different phases of the key exchange for node **A**. Node **A** enters phase I when turned on. Nodes **B** and **C** do not directly hear each other's traffic and are only able to communicate through node **A**, once the shared key is fully deployed.

The initial states of the three nodes are as follows: **A** has the key K_A, **B** has K_B and **C** has K_C. In this example, K_C is the *best* key, whereas K_A is the *worst* key.

Phase I:

1. Node **A** is turned on. All nodes send periodic HELLO messages which are part of the routing protocol.
2. **A** receives a HELLO message from **B**, notices a key mismatch, but ignores it because K_A is *worse* than K_B.
3. **A** receives HELLO from **C**, notices a key mismatch, but ignores it because K_A is *worse* than K_C.
4. **B** and **C** receive HELLO from **A**, they both notice they have a *better* key than K_A, and after a random time delay (to prevent traffic collisions), send an AUTH_REQ message to **A.**

Phase II:

1. **A** receives AUTH_REQ from **B** notices that **B** has a *better* key and schedules authentication with **B**. The authentication is to be performed after a certain waiting period, in order to hear if some of the neighbors has an even *better* key.
2. **A** receives AUTH_REQ from **C** as well, sees that **C** has a key *better* than K_B, and therefore decides to perform authentication with **C** instead.

Phase III:

1. **A** sends an AUTH_RESP message to **C**, telling it is ready for the authentication process
2. **C** initiates the authentication procedure with **A**, they exchange and verify certificates; the secure tunnel is established.
3. **C** sends its key K_C to **A** through the secure tunnel.
4. **A** receives the key and saves it locally; the old key K_A is saved in the key repository for eventual later use; **A** sends the new key further, encrypted with K_A.

In the next round, that is, after it hears traffic from node **B** signed with K_B, node **A** will use the same procedure to deliver the new key K_C to node **B**, hence establishing a common shared key in the whole cell.

There are two important parameters which influence the performance of the protocol and therefore have to be chosen carefully. The delays used before sending AUTH_REQ are random, to minimize the possibility of collisions in the case when more nodes react to the same message. On the other hand, the delay from the moment a node receives AUTH_REQ to the moment it chooses to answer with AUTH_RESP is a fixed interval and should be tuned so that it manages to hear as many neighbors as possible within a reasonable time limit. By this, all nodes that have been heard during the waiting period can be efficiently handled in the same batch.

4. EMULATION ENVIRONMENT

To facilitate development of this and other protocols, it was important to carefully choose a simulation tool or emulation test bed. Simulators, where GloMoSim [25] and ns-2 [20] seem to be the best candidates, have long been used in the ad-hoc field and make it possible to experience very diverse communication situations and a large scale of deployment. On the other hand,

emulators such as JEmu [10], EMWIN [27] or MobiEmu [26] might be a better option, in particular because they may run protocols in a more realistic manner and unveil issues inaccessible to simulation. Since in an emulation environment real processes are used, this approach facilitates porting of code to genuine wireless devices. Therefore, we decided to choose the emulation approach.

As the first test bed, we used several PCs equipped with Intel IXP network processors with multiple Ethernet ports, where each port acted as a single wireless node. The ports were connected either directly, to emulate one-hop communication, or using hubs to emulate a wireless cell. Although not entirely realistic, this was a good starting point to develop, test and evaluate the protocol. With time, this test bed posed limitations which made it impractical for further development. Limited number of network cards and the fact that cables had to be switched to make topology changes made us reflect on another solution.

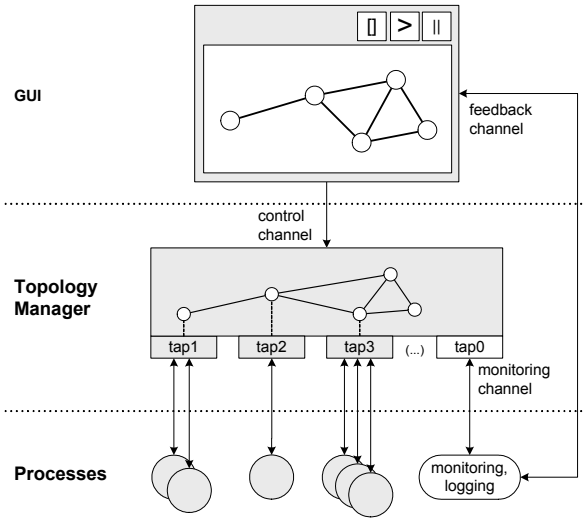

Figure 2. Emulation test bed

At the next stage, we developed a simpler and more effective emulation test bed (shown on Figure 2) based on virtual Ethernet network devices (TAP) available in the Linux kernel. These devices provide low level support for Ethernet tunneling. Every frame received on a TAP interface is available to the user-level application maintaining it, and every frame the application generates is sent to the interface and further to processes hooked to it. In our implementation, a process, called *topology manager*, creates a certain number of TAP interfaces. The topology manager gets all messages sent to its TAP interfaces and can decide to forward them to others, according to the topology information it has at a certain moment. One interface is reserved as the *monitoring channel*, having connection to all other TAP interfaces, independent from the topology, and thus allowing for analysis of the network traffic. The monitoring channel allows us also to induce traffic into the network if needed.

On top of this basic emulated network infrastructure, processes hook to the TAP interfaces using standard sockets with the socket option SO_BINDTODEVICE. This option ensures that a process will listen and send only to the specified interface, and thus not interfere with traffic addressed to some other process. Since this infrastructure emulates the link layer of MANETs, we first establish an IP infrastructure by means of routing daemon processes hooked to the TAP interfaces. Afterwards, other processes can hook to the same interfaces and use the established IP infrastructure. This way, the emulation test bed can be also used to implement and test, for example, middleware and application layer protocols.

The graphical user interface of the test bed is based on previously mentioned MobiEmu. It is a Tcl/Tk script, independent from the topology manager and can even run on a separate machine. The GUI shows the current position of nodes, their transmission ranges and links between nodes that can directly communicate with each other. Topology and node movement data are acquired from standard ns-2 scenario files, created by, for example, ns-2's *setdest* program. Information about topology changes is sent to the topology manager through the control channel, in form of UDP packets.

In the current implementation of the emulator, we still lack some characteristics typical for wireless networks, such as collisions in the air, hidden terminals, obstacles, etc. We are looking into a way to include those as well, in order to give us an even more realistic picture.

5. STANDALONE SKiMPy: EARLY IMPLEMENTATION

As the first implementation, the key management protocol was a self-standing application generating dummy routing messages in order to trigger key exchange. The signatures were inserted into a standard IP Authentication Header [16], which made it easy to analyze and debug packets. We measured the number of certificates and key management messages exchanged, and compared these figures to the number of routing messages needed from the moment when the nodes were turned on, up to the moment when a stable shared key was established.

Different variants of the protocol have been designed and applied, varying in the way the authentication and key distribution were done.

The first variant we designed was simple, causing an exchange of certificates and keys as soon as a node detected key inconsistency. Although the protocol proved to be working, this approach was clearly far from being optimal.

The next variant was a more intelligent one, analyzing the neighborhood before initiating the authentication process, as described in Section 3.6.

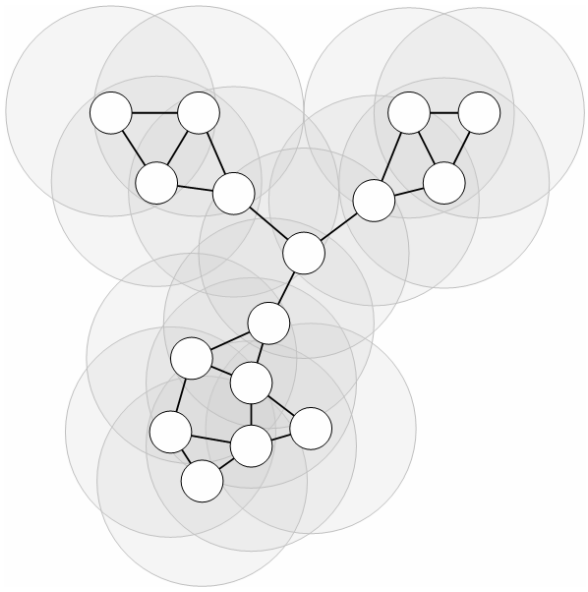

Figure 3. First evaluation scenario: 3 cells, connected by one node in the middle

Protocol Evaluation

To compare the two variants of the protocol, we have created a simple scenario (shown on Figure 3) on our first test bed. There were 16 nodes grouped into 3 cells, having one common node in the middle that would perform the merge. All the nodes were turned on simultaneously and started sending HELLO messages every 2 seconds. The criterion for choosing the *best* key was having the lowest ID-number.

We measured the number of certificates and key management messages exchanged, and compared these figures to the number of HELLO messages needed from the moment when the nodes were turned on, up to the moment when a stable shared key was established in the whole network. 100 independent measurements were performed for each protocol, each time starting with new random key ID numbers.

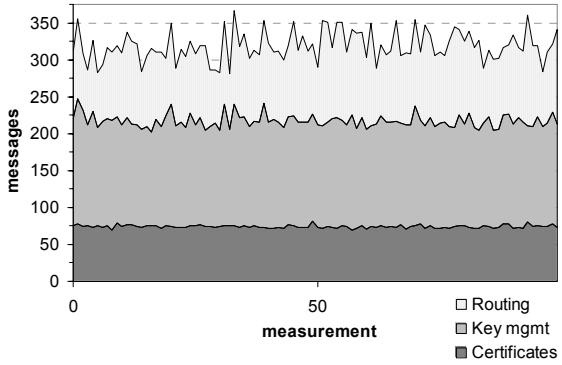

Figure 4. Traffic analysis of the first protocol implementation

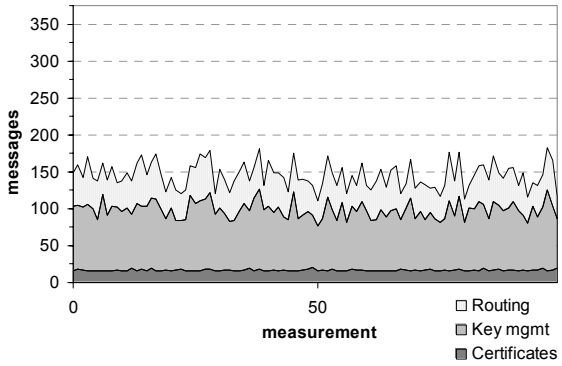

Figure 5. Results for the same scenario, after introducing neighborhood awareness

Figures 4 and 5 show the remarkable improvement achieved after introducing neighborhood awareness, approximately halving the total number of messages and, proportionally, the time needed to reach a stable state (getting to an average of 6 seconds). Moreover, the number of messages carrying certificates, whose size is much larger than other key management messages, has been reduced to approximately 23% of the initial number. Since the focus was being put on the key management protocol and not on the authentication itself, the authentication was considered to be done after the exchange of certificates. Therefore, the results shown here are only an approximation, and might be slightly different when an actual authentication algorithm is used.

6. OLSR-INTEGRATED SKiMPy: CURRENT IMPLEMENTATION

Optimized Link State Routing Protocol (OLSR) [6] is a proactive routing protocol for ad-hoc networks which is one of the candidates to be used in our solution for the emergency and rescue operations. The olsr.org OLSR daemon [22] is the implementation we decided to test, since it is portable and expandable by means of loadable plugins. One example of such a plugin, present in the main distribution, is the Secure OLSR plugin [11]. The plugin is used to add signature messages to OLSR traffic, only allowing nodes that possess the correct shared (pre-installed) key to be part of the OLSR routing domain. One important functionality this plugin lacks is a key management protocol. Although SKiMPy is mainly designed to protect all traffic and not only routing, it is still a good opportunity to test and analyze it in a realistic environment with a real routing protocol.

The key management protocol has been coded directly into the security plugin, although the plans are to make it as a separate one, if possible. To facilitate the implementation, X.509 certificates [13] and OpenSSL [21] are used to perform node authentication.

Protocol Evaluation

To get a visual understanding on what is going on in the network and how keys are distributed, we use the monitoring channel of the emulator to analyze the keys used by each of the routing daemons. ID numbers of the keys are converted to 24-bit RGB color codes and sent as feedback to the GUI, which then colors the nodes on the screen accordingly. That way we got an easy and yet effective way to see in real time how the protocol works.

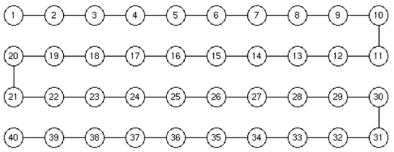

Figure 6. Example of a chain scenario

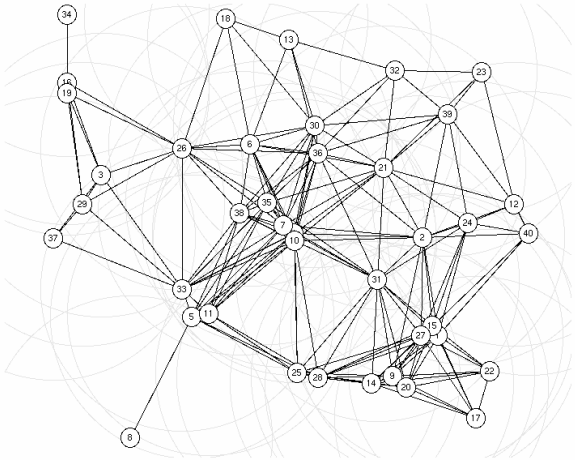

Figure 7. Example of a mesh scenario

In order to test performance and scalability the protocol, we have made measurements from 10 to 100 nodes, with two very different kinds of scenario - chain and mesh. Figures 6 and 7 show example screenshots taken from the GUI, representing the two different scenarios.

In a chain scenario, all nodes are lined up in a single chain and the distance between all nodes in the chain is such that only the direct neighbors can communicate in a single hop with each other. We consider this to be the worst case scenario. Given that all the nodes have to perform authentication with both their neighbors, this leaves no place for optimization, i.e. batching during the waiting period, as described in Section 3.6.

In a mesh scenario, however, nodes have multiple, randomly scattered neighbors, as it is natural in ad-hoc networks. Having multiple neighbors allows the protocol to exploit the batching phase, reducing traffic and resource consumption.

Ten independent runs were performed for each number of nodes and each scenario. All the nodes were started simultaneously (which we assume is the worst case for our protocol), with a random key and key ID. The scheme for choosing the key was the same, i.e., a key with a lower ID is *better*. The scheme for key distribution was a combination of the proactive and reactive schemes, described in Section 3.3.

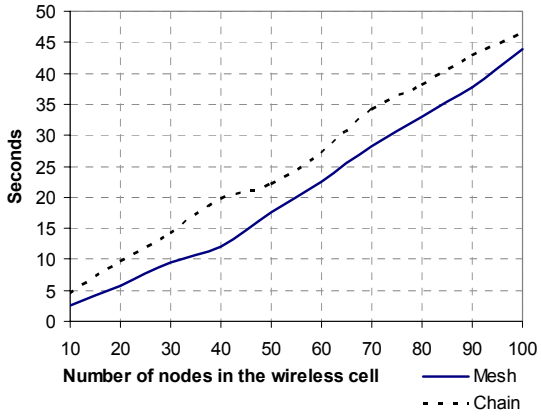

Figure 8. Time needed to achieve a stable shared key

One important fact that the results on Figure 8 immediately show is that the protocol scales linearly with linear increase of the number of nodes and physical network area accordingly (thus giving the same density of nodes). We also proved that having multiple neighbors does in fact lower the time necessary to reach a stable state. This scenario gives less deviation as well, which is understandable since in the case of chain there is more fluctuation of keys, nicely seen in the GUI.

In an additional test, we tried to see the scalability of the protocol when the size of the physical area remains constant while the number of nodes increases. In this case there is much more network traffic involved, as well as computation, making the test-machine the bottleneck. Therefore, we were only able to test up to 50 nodes, and the results for these measurements have shown minimal differences in the performance, compared to the results when the density of the nodes was constant.

7. CONCLUSION

In this paper, we presented a simple and efficient key management protocol, called SKiMPy, developed and optimized especially for highly dynamic ad-hoc networks. The protocol relies on the fact that there will be an *a priori* phase of rescue and emergency operations, within which certificates will be deployed on rescue personnel's devices. Pre-installed certificates are necessary due to the fact that highly sensitive data may be exchanged between the rescue personnel. The certificates make it possible for the nodes to authenticate each other without need for a third party present on the scene and, to the best of our knowledge, SKiMPy is the only protocol using such an authentication scheme.

We described two different implementations of the protocol, together with evaluation results. The results show that SKiMPy performs very well and it scales linearly with the number of nodes. As part of further work we will analyze more in-depth different key selection and distribution schemes, authentication protocols, and fine tuning of certain protocol parameters (such as the delays described in Section 3.6). Open issues like exclusion of compromised nodes, duplicate key ID numbers, denial of service attacks, etc. are also subject of further investigation.

In addition to the protocol itself, we presented the emulation test bed developed to test and evaluate this and other protocols. Emulation allows us to run the protocols real time, by real processes that can afterwards easily be deployed on genuine wireless devices. We will continue to make the emulation environment as realistic as possible, trying to introduce typical issues present in real wireless networks, such as collisions in the air, hidden terminals, obstacles, etc.

8. REFERENCES

[1] Alves-Foss, J., "An Efficient Secure Authenticated Group Key Exchange Algorithm for Large And Dynamic Groups", Proceedings of the 23rd National Information Systems Security Conference, pages 254-266, October 2000

[2] Blom, R., "An Optimal Class of Symmetric Key Generation System", Advances in Cryptology - Eurocrypt'84, LNCS vol. 209, p. 335-338, 1985.

[3] Bresson, E., Chevassut, O., Pointcheval, D., "Provably Authenticated Group Diffie-Hellman Key Exchange - The Dynamic Case (Extended Abstract)", Advances in Cryptology - Proceedings of AsiaCrypt 2001, pages 290-309. LNCS, Vol. 2248, 2001

[4] Capkun, S., Buttyán, L., Hubaux, J.-P., "Self-Organized Public-Key Management for Mobile Ad Hoc Networks", IEEE Transactions on Mobile Computing, Vol. 2, No. 1, Jan-Mar 2003

[5] Chan, Aldar C-F., "Distributed Symmetric Key Management for Mobile Ad hoc Networks", IEEE Infocom 2004, Hong Kong, March 2004

[6] Clausen T., Jacquet P., "Optimized Link State Routing Protocol (OLSR)", RFC 3626, October 2003

[7] Di Pietro, R., Mancini, L., Jajodia, S., "Efficient and Secure Keys Management for Wireless Mobile Communications", Proceedings of the second ACM international workshop on Principles of mobile computing, pages 66-73, ACM Press, 2002

[8] Diffie, W., Hellman, M., "New directions in cryptography", IEEE Transactions on Information Theory, 22(6):644-652, Nov. 1976

[9] Federal Information Processing Standard, Publication 180-1. Secure Hash Standard (SHA-1), April 1995

[10] Flynn, J., Tewari, H., O'Mahony, D. "A Wireless Network Emulator for Mobile Ad Hoc Networks", Proceedings of the Communication Networks and Distributed Systems Modeling and Simulation Conference 2002, San Antonio, Texas, 2002.

[11] Hafslund A., Tønnesen A., Rotvik J. B., Andersson J., Kure Ø., "Secure Extension to the OLSR protocol", OLSR Interop Workshop, San Diego, August 2004

[12] Hollick, M., Schmitt, J., Seipl, C., Steinmetz, R., "On the Effect of Node Misbehavior in Ad Hoc Networks", Proceedings of IEEE International Conference on Communications, ICC'04, Paris, France, volume 6, pages 3759-3763. IEEE, June 2004.

[13] Housley, R., Ford, W., Polk, W. and D. Solo, "Internet X.509 Public Key Infrastructure", RFC 2459, January 1999.

[14] IEEE, "IEEE Std. 802.11b-1999 (R2003)", http://standards.ieee.org/getieee802/download/802.11b-1999.pdf

[15] Kärpijoki, V., "Security in Ad Hoc Networks", Tik-110.501, Seminar on Network Security, HUT TML 2000

[16] Kent, S., Atkinson, R., "IP Authentication Header", RFC 2402, November 1998

[17] Matsumoto, T., Imai, H., "On the key predistribution systems: A practical solution to the key distribution problem", Advances in Cryptology - Crypto'87, LNCS vol. 293, p. 185-193, 1988.

[18] Montenegro, G., Castelluccia, C., "Statistically Unique and Cryptographically Verifiable (SUCV) Identifiers and Addresses", NDSS'02, February 2002

[19] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992.

[20] The Network Simulator - ns-2, http://www.isi.edu/nsnam/ns/

[21] The OpenSSL project, http://www.openssl.org/

[22] Tønnesen A., "Implementing and extending the Optimized Link State Routing protocol", http://www.olsr.org/, August 2004

[23] Wallner, D., Harder, E., Agee, R., "Key management for Multicast: issues and architecture", RFC 2627, June 1999

[24] Wong, C., Gouda, M. and S. Lam, "Secure Group Communications Using Key Graphs", Technical Report TR 97-23, Department of Computer Sciences, The University of Texas at Austin, November 1998

[25] Zeng, X., Bagrodia, R., Gerla, M., "GloMoSim: a Library for the Parallel Network Simulation Environment", Proceedings of the 12th Workshop on Parallel and Distributed Systems, 1998

[26] Zhang, Y., Li, W., "An Integrated Environment for Testing Mobile Ad-Hoc Networks", Proceedings of the 3rd ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc 2002), Lausanne, Switzerland, June 2002.

[27] Zheng P., Ni, L. M., "EMWIN: emulating a mobile wireless network using a wired network", Proceedings of the 5th ACM international workshop on Wireless mobile multimedia. ACM Press, 2002, pp. 64–71