LiSP: Lightweight Security Protocols for Wireless Sensor Networks

by

Taejoon Park

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical Engineering: Systems)
in The University of Michigan
2005

Doctoral Committee:

Professor Kang G. Shin, Chair
Professor Stephane Lafortune
Associate Professor Brian D. Noble
Assistant Professor Mingyan Liu

To my family, with love and thanks

ACKNOWLEDGMENTS

First of all, I would like to respectfully express my sincere gratitude to my advisor, Professor Kang G. Shin, for his guidance, insightful suggestions, invaluable comments, and constant encouragement and optimism that not only helped me overcome frustrations and difficulties throughout my thesis research, but also will influence my future career in pursuit of excellence. I was really fortunate to complete my thesis under his supervision as well as to be part of his talented research group. I'm also indebted to the other members of my thesis committee, Professors Stephane Lafortune, Brian Noble and Mingyan Liu, for their advice and comments.

I'm grateful to all of the members of Real-Time Computing Laboratory for their social and technical assistance, including Dr. Dan Kiskis, Abhijit Bose, Katharine Chang, Zhigang Chen, Min-Gyu Cho, Mohamed El Gendy, Hyoil Kim, Kyu-Han Kim, Songkuk Kim, Matt Knysz, Jai-Jin Lim, Daji Qiao, Saurabh Tyagi, Haining Wang, Jian Wu, and Jisoo Yang. I also would like to extend my thanks to my colleagues in the Department of EECS, especially, Sangtae Ahn, Hyunseok Chang, Junho Choi, Suhan Choi, Dongsook Kim, Nam Sung Kim, Hyunseok Lee, Jinsol Lee, Ilju Na, and Hosung Song, for their friendship and help. Special thanks should go to my family who always showed me their unconditional love, faith and support.

Finally, I would like to thankfully acknowledge the financial support of the US Office of Naval Research and the US Naval Research Laboratory under Grant No. N00014-04-10726, of the US National Science Foundation under Grant CCR-0329629, of the US

Defense Advanced Research Projects Agency under contract F33615-02-C-4031 administered by the Air Force Research Laboratory, and of Cisco Corporation.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

CHAPTER I

INTRODUCTION

1.1 Motivation

An increasing number of safety- and security-critical applications, such as situation monitoring and facility surveillance, rely on a network of small, inexpensive, battery-powered sensor devices that have limited energy supplies, storage, computation, and communication capacities. These sensor networks can be used for various applications such as safeguarding of, and early warning systems for, the physical infrastructure that includes buildings, transportation systems, water supply systems, waste treatment systems, power generation and transmission, and communication systems. The success of these applications hinges on their own *security*; they must protect themselves by preventing and/or tolerating critical attacks from malicious adversaries. However, despite its importance, it is challenging to achieve high-level security throughout the lifetime of sensor networks due mainly to the operational issues and requirements unique to sensor networks, such as energy-efficiency in terms of prolonging the lifetime of sensor devices as much as possible, scalability to a large number (thousands to millions) of nodes, and survivability even in a harsh, unattended environment.

With rapid advances in device technology, the processing capability of embedded systems has been improving at an exponential rate. However, this improvement in com-

puting performance comes with a rapid increase in complexity and power consumption. By contrast, the battery and energy storage technologies have been improving at a much slower pace, failing to meet the increasing energy demands of emerging embedded systems. *Energy-efficiency* is, therefore, critical to all portable, embedded computing devices. Specifically, in sensor networks where it is often very difficult, and sometimes impossible, to change or recharge batteries for devices after their deployment, energy-efficiency is one of the most important requirements.

To address the challenges of both security and energy-efficiency, this thesis presents ***Li****ghtweight* ***S****ecurity* ***P****rotocols* (LiSP) for a network of resource-limited embedded sensors. We have taken an approach to building LiSP as a set of closely-coupled security protocols to meet the following design objectives: the protocols must be

- *lightweight* so as to prolong the network lifetime significantly, which requires the use of computationally-efficient ciphers such as symmetric-key algorithms and cryptographic hash functions;
- *cooperative* in the sense of achieving high-level security via mutual collaboration & cooperation among sensor nodes as well as with other protocols, thereby disfavoring resource-demanding ciphers;
- *attack-tolerant* to enable the network to gracefully tolerate attacks and device compromises as well as heal itself by detecting, identifying, and removing the sources of attacks;
- *flexible* enough to make a tradeoff between security and energy consumption;
- *compatible* with existing security mechanisms and services; and
- *scalable* to the rapidly growing network size.

1.2 Overview of Sensor Networks

1.2.1 Network Architecture

For cost and size reasons, sensor devices are designed to minimize resource requirements, e.g., Motes [33, 46] feature an 8-bit CPU running at 4 MHz, 128 KB of program memory, 4 KB of RAM and 512 KB of serial flash memory powered by two AA batteries (2850 mAh each). That is, sensors are usually built with limited processing, communication and memory capabilities in order to prolong their lifetime with the limited energy budget. This disfavors use of advanced (thus resource-demanding) cryptography, such as public-key algorithms.

Sensor networks are deployed for data acquisition for various applications including: (1) pursuit-evasion game (PEG) [45, 109], in which a group of pursuers track and capture moving evaders based on the information collected and processed by the sensor network; (2) common reference grids, in which the sensor network collects and maintains information for a positioning service or a distributed directory service to locate critical services; (3) shooter localization [35], in which sensors detect the acoustic shock wave of a bullet as it travels through the air; and (4) habitat and environmental monitoring [67], in which sensors are deployed to collect data without incurring disturbance effects (e.g., by humans).

A sensor network is usually built with a large number (thousands or even millions) of sensor nodes, each capable of, for example, reading temperature or detecting (part of) an object moving nearby. Moreover, the sensor network is usually deployed in a hostile/harsh environment, and removal (due to device failures or depletion of battery energy) and addition of sensor nodes are not uncommon. Sensors collaborate and coordinate with one another to achieve a higher-level sensing task, e.g., measuring and reporting, with accuracy, the characteristics of a moving object, such as the speed and direction of its movement.

1.2.2 Communication Models

The communication models for sensor networks are *cluster-based* or *peer-to-peer*. The cluster-based model typically appears in a tiered architecture [21, 115], where multiple clusters are formed statically and/or dynamically, and a cluster-head—that is more capable than the usual member devices—manages and controls operations inside each cluster. Cluster-heads aggregate sensed data within their own cluster (intra-cluster communication) as well as disseminating/relaying aggregated data among themselves (inter-cluster communication).

Many emerging applications and services rely more on the peer-to-peer model: each sensor communicates directly with any of the other sensors without relying on dedicated devices. The main challenge associated with a sensor network is the large volume of data to be collected and processed over the entire network. To address this challenge, researchers have proposed efficient ways of storing and extracting relevant data from the network based on the peer-to-peer model. The authors of [44] proposed data to be named and communication abstractions to refer to these names rather than sensor IDs. In a data-centric storage [90], the sensor network stores and looks up relevant data by name, i.e., it hashes the data into geographic coordinates (name) using a Geographic Hash Table (GHT) and stores data at the sensor geographically closest to the hashed coordinates. The two-tier data dissemination in [126] provides, based on a grid structure, a data-delivery mechanism to mobile data-sinks. The locations of mobile data-sinks can be looked up through the application of location management schemes [62, 80, 124], in which each mobile node chooses a small subset of sensor nodes and periodically updates them with its location information, thus allowing data sources to query these sensors for the locations of data-sinks.

Both communication models call for transactions between remote nodes because data sinks can be far away from the data source, and in such a case, the data delivery should be done via inter-cluster communications or data storage/lookup/dissemination services. The need for long-distance communications will continue to increase as new applications/services are expected to aggressively exploit the large-scale, distributed nature of sensor networks; otherwise, the deployment of, and internetworking among, a large number of sensors wouldn't be necessary.

1.2.3 Localization Algorithms

There exist many applications that require each sensor node to be location-aware; for example, each sensor node must be uniquely identified by its location estimate for geographic routing [52, 55] in which a source or an intermediate sensor node forwards a packet to one of its neighbors closest to the packet's destination. As such, *localization* — assigning locations to sensors consistently with measured or estimated distances — is one of the core services of sensor networks.

Techniques for estimating the distance between a pair of communicating nodes are typically based on: (1) received signal strength (RSS) that can be translated into a distance estimate; (2) time of arrival (TOA) and time difference of arrival (TDOA) that use the signal propagation time, and (3) angle of arrival (AOA) that estimates the relative angle between nodes. These techniques are then combined with various signalling methods, e.g., based on RF, ultrasound or infrared signals [14, 128]. Direct RSS-to-distance conversion, currently supported by Motes [33], becomes inaccurate as the distance increases due mainly to nonuniform signal propagation characteristics and fading/interference effects. To mitigate this estimation error, one may apply averaging or smoothing as in [97, 118].

Besides these ranging techniques, range-free schemes have also been proposed to pro-

vide cost-effective, coarse-grained localization. In [16], the location of a sensor is determined as the center of all the anchors it hears. In [43], each sensor forms virtual triangular regions among the anchors of interest, determines in which regions it resides based on its neighbors' measurements, and finally calculates the overlapping area between these regions. These schemes typically require very high anchor density and long anchors' radio ranges as each sensor has to hear from as many anchors as possible.

It is preferable to provide the localization capability even when there are only a very small number of anchors in the network. In the hop-count-based localization [76], each sensor determines the minimum hop-counts to anchors by running a distance vector algorithm, and computes physical distances by multiplying them to the average per-hop distance. This scheme, unfortunately, yields poor localization accuracy. Approaches like [49, 83, 88] employ mobile anchors to meet the requirements of both low anchor density and high accuracy of distance estimation, but suffers a large latency.

The highest localization accuracy (in terms of minimizing the difference between assigned and real locations) can be achieved by utilizing *multidimensional scaling* (MDS), widely used in mathematical psychology, economics, sociology and machine learning communities for modeling proximity relations. MDS-MAP [100] constructs network-wide connectivity information in the form of a matrix of all possible distance estimates (in hop-counts), and then applies MDS to derive sensors' locations that fit well those estimated distances. However, it must rely on a central processing node that collects all distance estimates and computes location assignments, significantly degrading scalability due mainly to the resulting high communication overhead.

The MDS technique is flexible enough to find consistent location assignments even when a limited set of distance estimates are available. In such a case, one may apply various *iterative MDS* techniques, e.g., those reported in [8]. Ji and Zha [53] took this

approach to develop a distributed localization scheme by applying MDS iteratively to build a local map of locations for each group of adjacent sensors and then combining these maps together to obtain a global location map. This scheme requires the computation of eigendecomposition per iteration that takes $\mathcal{O}(n^3)$ time for a group of n sensors. Costa *et al.* [29, 28] also developed a distributed, iterative MDS scheme that (i) relies solely on distance measurements between neighboring sensors and (ii) is computationally less demanding than that in [53].

1.3 Security Attacks

Sensor networks are vulnerable to various security attacks, especially because they are deployed in an unattended, hostile environment. For instance, an adversary with a compatible radio receiver/transmitter can easily eavesdrop ongoing communication sessions, inject or modify packets, jam the surrounding area, and even locate specific sensors or hot spots. Possible types of adversaries can be classified, in the order of increasing strength, as: (1) passive attackers, only eavesdropping conversations in the network; (2) active attackers, possessing no cryptographic keys but capable of injecting packets into the network; and (3) active attackers, having all keys of multiple compromised sensors. The last type of attacks is considered as *insider* attacks, while the first two as *outsider* attacks.

Attacks on the sensor network can be classified as:

- *physical attacks* on sensor devices, e.g., destroying, analyzing, reprogramming and/or cloning sensors;
- *service disruption attacks* on routing, localization and clock synchronization;
- *data attacks*, e.g., traffic capture, replaying and spoofing;

- *resource-consumption and denial-of-service (DoS) attacks* that diminish or exhaust the sensors' capacity/energy to perform its normal function; and

- *sybil attacks* [34] by which a single malicious sensor device claims/presents multiple sensor IDs (locations) to control a substantial fraction of the ID space which, in turn, makes it easier to mount other attacks, such as disruption of routing services [54].

An adversary's attempt to disrupt, subvert or destroy the sensor network belongs to a broad category of the denial-of-service (DoS) attack that diminishes or eliminates the network's capacity to perform its normal function. The authors of [122] summarized plausible tools for DoS attacks as: (i) jamming that interferes with the operating radio frequencies; (ii) collisions that are induced on ongoing packet transmissions; (iii) exhaustion that forces the link layer to repeat the same packet transmission; and (iv) vulnerability of existing protocols.

The adversary may also disrupt the integrity/availability of localization service. Possible attacks [18, 49, 64] on the localization service include:

- sensor displacement or removal;

- distance enlargement/reduction via jamming, adjustment of transmission power, or placement of obstacles interfering with direct paths;

- announcement of false locations, distances or hop-counts;

- message modification or replaying;

- wormhole attacks that create hidden links between remote (compromised) sensors to be used for replaying messages or altering distance measurements or hop-counts; and

- deployment of bogus anchors that propagate false reference location information.

These localization-specific attacks try to propagate wrong information about locations of, or distances to, the sensors (or anchors) under the adversary's control in an attempt to disrupt the localization service.

One of the serious attacks to the sensor networks deployed in an unattended environment is physical tampering with sensors. An adversary can easily (i) capture one or more sensors, (ii) scrutinize/reverse-engineer/alter the program and/or master-secret in the sensor, and (iii) create/deploy (multiple clones of) manipulated sensors. A small number of sensors compromised by physical attacks may serve as zombies for many serious attacks, such as initiating DoS or sybil attacks and sabotaging certain services of the sensor network, which will, in turn, facilitate the subversion of the entire network. The hardware tamper-resistance techniques [3, 19] can be used as a device-level countermeasure against this type of attacks.

1.4 Related Work

1.4.1 Cryptography for Sensor Networks

Public-key algorithms like the Diffie-Hellman (DH) protocol have been widely used for the development of various key establishment protocols [4, 20] that derive a common key among nodes. However, they are unsuitable for sensor networks, because of their large energy demands, let alone the requirement of exchanging public-key certificates. In particular, existing implementations of the DH protocol on sensor nodes [68, 106] consume 1.19 $\sim$ 12.64 [J] per operation, which is too much to be usable in devices with a limited energy budget, e.g., 61,560 [J] in Motes powered by two AA batteries. By contrast, symmetric-key ciphers and cryptographic hash functions use significantly less energy, e.g., 0.115 [mJ] in TinySec. It is, therefore, desirable to set up keys based solely on symmetric-key ciphers.

A sensor device, therefore, cannot use public-key algorithms due mainly to its severe resource constraints. The symmetric-key ciphers and cryptographic hash functions, which are orders-of-magnitude cheaper and faster, would be a better choice for sensor nodes. Moreover, data packets in sensor networks are generally small. A desirable property in this environment is that the size of the ciphertext should be the same as that of the plaintext. These requirements suggest the use of a stream cipher as the underlying encryption engine. For example, SPINS [86] realizes the stream cipher by running the RC5 block cipher in the counter (CTR) mode, the authors of [17] use RC5 in the output feedback (OFB) mode, and the IEEE 802.11 Wired Equivalent Privacy (WEP) [50] uses the RC4 stream cipher.

However, it is well-known that stream ciphers are vulnerable to keystream reuse.[1] This weakness allows attacks against stream ciphers that succeed irrespective of the symmetric-key size. For example, WEP prefixes each encrypted packet with a per-packet initialization vector (IV), but, due to the limited IV space (24 bits), it is vulnerable to a number of practical attacks as reported in [13, 69, 111]. To remove the keystream-reuse problem, SPINS forces both communicating parties to maintain IV separately, instead of including IV in data packets, while updating the key after IV wraps around. Unfortunately, this design choice creates the following new problems.

- Lossy wireless links may cause IVs loss of synchronization, and in such a case, communication will remain disabled until IVs are resynchronized.
- It cannot protect the network against replay attacks and incurs an additional overhead of maintaining IV states of the other sensor devices.
- The re-keying overhead increases rapidly as the network size grows.

[1] The keystream is generated as a function of the symmetric key and the initialization vector (IV), and is XORed with the plaintext to produce the ciphertext.

Problems with the schemes in WEP and SPINS emanate from the fact that they solely control IVs without refreshing the key, or use implicit IVs and triggered re-keying. It is, therefore, important to refresh the symmetric key as often as needed while keeping small-length IVs, not only to remove keystream collisions but also to improve performance.

1.4.2 Key Management/Sharing Schemes

The cluster-based key management [7, 19] is concerned with (periodic) distribution and refreshment of a shared cluster key by the cluster-head acting as a key server within the cluster. Although this scheme performs well for local transactions, it still has problems; for example, each cluster-head (even though better-equipped and better-protected than normal sensor nodes) is a single point of failure, implying that if compromised, it may break the cluster's security. Moreover, an efficient mechanism for securing inter-cluster communications must be provided to deal with transactions between remote nodes residing in different clusters. Note that using a globally-shared key for all clusters makes the entire network vulnerable to a single sensor compromise.

Key pre-deployment schemes [19, 22, 37] statically set up pairwise shared keys based on keys loaded into sensor devices prior to their deployment. In the probabilistic key sharing [37], each sensor is preloaded with multiple (a couple of hundreds) keys randomly chosen from a large pool of keys, and hence, a pairwise key is established between a pair of neighboring sensors if a key happens to be common to both sensors. However, the pairwise keying performs poorly for communications over multiple-hop paths, since it requires transcoding (decryption followed by re-encryption) for each and every hop, thereby significantly risking the security as any malicious sensor node on the path may take control of the communication as well as increasing sensors' workloads (as routers) and the packet-delivery latency. Hence, it is preferable to minimize the number of transcodings

per communication for both security and performance reasons.

Attempts were also made to combine several key sharing schemes. For example, in [133], each node simultaneously maintains an individual key, a pairwise key and a cluster key to support in-network processing. However, it still lacks support for long-distance communications.

1.4.3 Countermeasures against Attacks on Localization

Determining sensors' locations in an untrusted environment is a challenging, but important, problem that has not yet been fully studied. Like other security applications, one may want to authenticate all the messages to protect the network against attacks (targeting at data traffic). For this purpose, as discussed in [49], one may attempt to use digital signatures or μTesla [86] together with key pre-deployment schemes [37, 22]. However, the former suffers high computational overhead while the latter suffers a large authentication latency, and, more importantly, many of the above-mentioned localization-targeted attacks are non-cryptographic in nature, making these authentication-based solutions highly unlikely to succeed.

The method proposed by Lazos and Poovendran [61] is conceptually similar to that in [43] in that each sensor hears directly from multiple anchors, identifies a region it resides in, and determines its location as the center of the region. In this scheme, the security against chosen attacks is preserved if the anchors are trusted and cannot be compromised by the adversary. However, its main drawback is the requirement of a large number of *specialized* anchors equipped with directional/sectored antennae and capable of high power transmission.

Recently, statistical approaches have been proposed [64, 65]. In [64], the authors presented an attack-tolerance mechanism for triangulation-based localization, in which each

sensor applies the least median squares algorithm on the distance estimates to anchors in order to mitigate the effect of attacks. The authors of [65] also use a collection of anchors' reference locations associated with estimated distances, and apply the mean square error criterion to identify and discard malicious location references. Unfortunately, these methods invite attacks from relaying sensors and require a significant amount of redundant location/distance information from anchors, incurring a high network overhead to achieve a reasonable degree of robustness against attacks. These drawbacks mainly come from the fact that they do not fully extract/utilize the available information and ignore the relationship/correlation among sensors' locations.

Although not directly applicable to localization, the authors of [96, 117] developed algorithms to verify the distance or location claims of a node, e.g., to make sure the node to be within a certain region. They rely on a challenge-response protocol that measures the round-trip time between a verifier and the node, and then translate the elapsed time into distance. Another way [18] is to check if the node's location falls within a triangular region formed by three trusted verifiers. These algorithms use centralized trusted servers, and hence, can be used as local defense mechanisms against distance-reduction attacks, but not as global, general-purpose solutions.

1.4.4 Tamper-Proofing Techniques

A number of approaches have been proposed to generate tamper-resistant programs without any hardware support. Most of them are intended for environments equipped with sufficient computation power. Code obfuscation [26, 113, 114, 123] converts the executable code into an unintelligible form that makes analysis/modification difficult. However, the level of difficulty to tamper with gets substantially lower as the program becomes smaller, and hence, it cannot protect against determined attackers. Furthermore, as Barak

et al. [6] showed, obfuscating programs while preserving its functionality is theoretically impossible. Result checking [10, 36, 116] examines the validity of intermediate results produced by the program, but it is inappropriate for use in battery-powered devices because it continuously incurs verification overhead. Aucsmith [5] proposed to store the encrypted executable and decrypt it before execution. However, this scheme suffers from a very high decryption/re-encryption overhead, and the security of self-decrypting programs can be easily broken unless the decryption routines are protected from reverse-engineering, e.g., by hardware. Self-checking techniques [5, 23, 48] aim to detect changes in the program and take appropriate actions against those changes, as the program is running. To this end, they use embedded codes (e.g., testers [48] or guards [23]) to compute a hash value on the program and compare it with the correct value. However, similarly to the self-decryption techniques, they become defenseless once the hash computation code and/or the hash value has been identified/analyzed. In summary, all of these approaches are not suitable for resource-limited devices with small programs and slow CPUs.

Besides protection of "stationary" software, a number of researchers studied the tamper-proofing of mobile software agents. Wilhelm *et al.* [119] proposed a technique based on the tamper-resistant hardware, but the severe resource constraints in each sensor device preclude the use of hardware-based protection. Execution tracing [110] attempts to detect unauthorized modifications of a mobile agent through the faithful recording of the agent's behavior during its execution. However, this approach is inappropriate for resource-limited sensor devices due to the size and number of logs that need to be retained. Blackbox security [47] scrambles the code in such a way that no one can gain complete understanding of its function for a certain time interval, but it cannot protect against active attacks, e.g., denying the execution or returning incorrect results. Sander and Tschudin [95] proposed the concept of computing with encrypted functions, by which mobile agents can safely

compute cryptographic primitives in an untrusted computing environment. However, they did not provide a general scheme for creating mobile agents that encode arbitrary functions. Kotzanikolaou *et al.* [58] realized the idea in [95] by applying the RSA public-key algorithm to the mobile agents dealing with a small amount of data. Unfortunately, this scheme becomes very inefficient as the size of data to be processed increases. Also, sensors do not have enough resources to support public-key algorithms.

While most existing tamper-proofing solutions attempted to realize tamper-resistance within the program itself, our approach differs from them in that it relies on external servers to examine the program and check if it is identical to the original one. Our approach is well suited to sensor networks, because examination of a small sensor program will be fast and occurs only infrequently, and it relies on computationally-efficient hashing algorithms.

Kennell and Jamieson [56] presented a software-based scheme to verify the genuinity of a remote computer system. They send the checksum code to the remote system, compute a hash via randomized memory access, and use timing to determine the system's genuinity. The key to their scheme is the randomized memory access that triggers more page faults and cache misses on a virtual memory system of the compromised machine, leading to a severe slowdown in hash computation. However, their approach is not suitable for sensor devices that do not have virtual memory support. Seshadri *et al.* [98] proposed a software-based attestation technique that verifies memory contents of embedded devices. They also used randomized memory traversal to force an attacker (who altered the memory) to check if the current memory access is made to a modified location, causing a detectable increase in the hash calculation time. However, this scheme is inefficient as it incurs much more memory accesses than sequential scanning of the program, without guaranteeing 100 % detection of memory modifications. Moreover, the (random) communication latency in a networked environment may significantly reduce the detectability of

this scheme. Our approach is different from that of [98] in that it accesses each memory location exactly once and allows for byte-oriented processing of program contents, resulting in much faster and accurate verification.

1.5 Main Contributions

The focus of this thesis is "energy-aware security" based on cooperative interactions among individually-proposed, mutually-complementary security solutions. The main contributions of this thesis are as follows.

Group-based key management: we make a tradeoff between security and resource consumption via the novel re-keying mechanism that offers (1) efficient key broadcasting without requiring retransmission/ACKs, (2) authentication for each key-disclosure without incurring additional overhead, (3) the ability to detect/recover lost keys, (4) seamless key refreshment without disrupting ongoing data encryption/decryption, and (5) robustness to inter-node clock skews. Furthermore, these benefits are preserved in conventional contention-based medium access control protocols that do not support reliable broadcast.

Secure routing based on distributed key sharing: we propose a novel *distributed key sharing* scheme, in which each participating sensor node shares its unique keys with a small number of other sensor nodes—called *distributed key servers* (DKSs)—chosen according to their geographic distance and communication direction. Using DKSs, we develop two secure routing protocols: (1) *secure geographic forwarding* that delivers packets via a chain of DKS lookups, each secured with its own key and forwarded geographically; and (2) *key establishment* that creates a secure session between two remote sensor nodes based solely on symmetric-ciphers.

Attack-tolerant localization: we present an attack-tolerant localization protocol, under which sensors cooperatively safeguard the localization service. By exploiting the high *spatio-temporal correlation* existing between adjacent nodes, we realize (1) adaptive management of a profile for normal localization behavior, and (2) distributed detection of false locations advertised by attackers by comparing them against the profile of normal behavior.

Program-integrity verification: as a countermeasure against physical attacks, we develop a soft tamper-proofing technique that verifies integrity of the program residing in each sensor device, whenever the device joins the network or has experienced a long service blockage. The verification is based on the novel *randomized hash function* tailored to low-cost CPUs, by which the algorithm for hash computation on the program can be randomly generated whenever the program needs to be verified. By realizing this randomized hash function, we successfully (1) prevent manipulation/reverse-engineering/reprogramming of sensors unless the attacker modifies the sensor hardware (e.g., attaching more memory); (2) provide purely software-based protection; and (3) achieve infrequent triggering of verification, thus incurring minimal intrusiveness into normal sensor functions.

As shown in Figure 1.1, a complete LiSP framework is built on top of the above-mentioned security solutions that closely interact with one another. The core building blocks of LiSP are described below.

- *Two key management/sharing schemes* deal with efficient distribution/sharing/renewal of cryptographic keys. They are mutually complementary: the group-based scheme (in Chapter II) is tailored to localized, cluster-based communications, while the distributed key sharing (in Chapter III) achieves extremely lightweight protection for

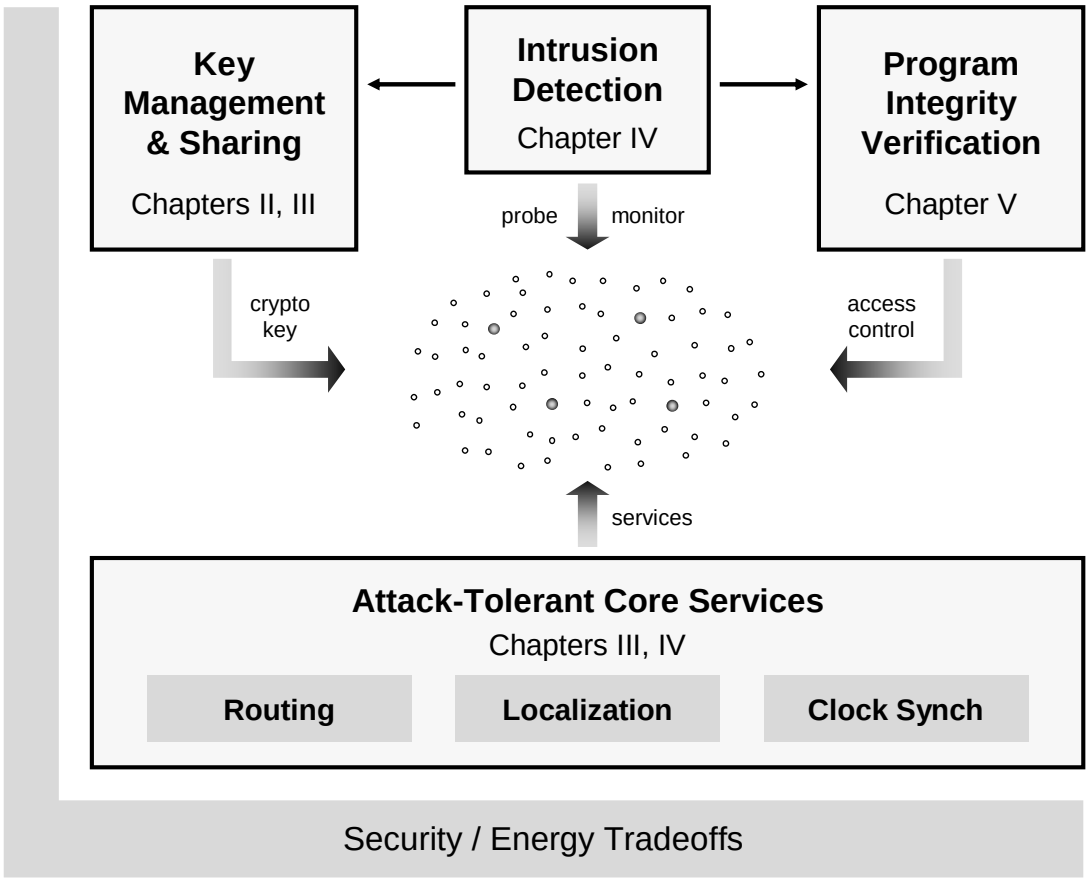


Figure 1.1: The LiSP architecture.

communications between distant sensor nodes.

- *Attack-tolerance mechanisms* make it possible for the sensors' core services, such as routing, localization, and clock synchronization, to gracefully tolerate attacks from compromised/malicious nodes. The two secure routing protocols based on distributed key sharing (in Chapter III) constitute an attack-tolerant routing service, while the statistical method exploiting spatio-temporal correlation (in Chapter IV) realizes an *anomaly-based intrusion detection system* tailored to the service of interest. The latter monitors (or probes) network activities to uncover misbehaving or compromised nodes.

- *The program-integrity verification* (in Chapter V) serves as a strong *access control* mechanism against compromised sensors (with malicious program codes), under which a sensor joining the network or suspected of having been compromised must register itself to the server via verification of its program.

Finally, the tradeoff between security and energy consumption is supported by all the building blocks of LiSP.

1.6 Thesis Outline

The rest of this thesis describes the components of LiSP in detail. Chapter II presents a *Group-based Key Management Protocol* (GKMP) for heterogeneous sensor networks. This chapter describes the procedures for initial setup, re-keying and authentication/recovery of lost keys, message encryption/decryption, and inter-group communications. It then evaluates and discusses the performance of the proposed protocol using the Markov chain analysis as well as in-depth simulation.

Chapter III proposes a *distributed* way of sharing keys in large-scale sensor networks,

and, based on the thus-shared keys, develops two protocols for secure routing: a *secure geographic forwarding protocol* (SGFP) and a *temporal-key establishment protocol* (TKEP). After describing these proposed protocols in detail, this chapter analyzes, evaluates, and discusses their security and energy-efficiency.

Chapter IV deals with the problem of attack-tolerance in the design of localization service, and presents an attack-tolerant localization protocol, called *Verification for Iterative Localization* (VeIL), that is essentially an anomaly-based intrusion detection system. This chapter first describes the two building blocks of VeIL, the profile manager and the attack detector, then analyzes how VeIL defends itself against many critical attacks, and finally, evaluates its attack-detection capability through simulation.

Chapter V proposes a soft tamper-proofing technique, called *Program-Integrity Verification* (PIV), that verifies the integrity of the program residing in each sensor device. It first presents the randomized hash function, and then discusses all aspects of PIV including the security framework, the PIV architecture, the pre-deployment phase of sensors, the state transition diagram for sensors, and the verification protocol. It finally analyzes and evaluates the security and efficiency of PIV.

Conclusions are drawn in Chapter VI. This chapter also discusses possible future research directions.

CHAPTER II

GROUP-BASED KEY MANAGEMENT

2.1 Introduction

Sensor networks must meet several operational challenges, such as energy-efficiency in terms of maximizing the lifetime of sensor networks; scalability to a large number of nodes; survivability in certain environments where sensors are subject to compromise, capture and manipulation by adversaries; support for dynamic addition/removal of sensors (to expand the network coverage area or replace faulty/subverted nodes); and robustness to spontaneous interferences, collisions and packet losses. Moreover, they are vulnerable to many serious security attacks as described in Section 1.3. These challenges, along with the severe resource constraints in each sensor node, limit both the security and the performance of a sensor network. Confidentiality, data integrity and authentication services must be supported to prevent adversaries from compromising the sensor network, but advanced cryptography cannot be used by resource-poor sensor nodes. It is, therefore, important to make a good tradeoff between the levels of security and resource consumption.

The security for sensor networks hinges on a *group communication model*: authorized sensors in the network share a symmetric key that is used to encrypt communication messages; new sensors *join* the network after their deployment; and the compromised sensors are forced to *leave* the network. In this model, forward and back-

ward confidentiality[1] [112] should be provided via re-keying, in which the shared key is changed/redistributed whenever a sensor joins or leaves the network. For its proper operation, re-keying must be protected by the following mechanisms. First, the master secret, used for securing the shared-key updates, must be pre-deployed to each sensor using tamper-proofing techniques (Chapter V) or a ring of keys [37]. Second, intrusion detection systems [9, 51, 60, 73, 129, 130] must be used as sensors are likely to be compromised because (i) use of tamper-proofing techniques is limited by the low-cost requirement for sensor devices, and (ii) only computationally-inexpensive cryptography can be employed.

For scalability, the entire network is typically divided into multiple groups, each with its own symmetric key [74] or with a key shared among all groups [99]. Carman *et al.* [19] conducted a broad survey of group-determination algorithms and the associated group re-keying protocols. Group re-keying protocols can be either *reactive* or *periodic*. The reactive protocol [24, 41, 112, 120] renews the key upon a member's join/leave. This approach, however, does not attempt to reduce the frequency of re-keying that causes high re-keying overhead in large and/or dynamic groups. By contrast, the periodic protocol [63, 94, 99] refreshes keys periodically to decouple the frequency of re-keying from the group size and dynamics, and hence, scales well to large groups. Yang *et al.* [125] have shown that periodic re-keying reduces both processing and communication overheads of the key-server[2] and improves the scalability and performance over the reactive re-keying. Moreover, severe resource constraints in each sensor node and the requirement for a large number of sensors in the network make it necessary to limit the frequency of re-keying so as to reduce its overhead. In such a case, periodic re-keying might be preferable to reactive protocols [7, 99].

[1] New members joining the network should not be able to access the packets transmitted before their joining and those having left the network should not be able to access the packets communicated after their departure.

[2] The key-server is responsible for distributing a new key within its group.

The re-keying must ensure reliable distribution of keys. The Time Division Multiple Access (TDMA) protocol can provide a reliability service to the key-management layer. However, Ye *et al.* [127] argued that TDMA is unsuitable for sensor networks as it is difficult to manage dynamic groups and control inter-group communications and interferences. Most protocols used/proposed for sensor networks [101, 121, 127] are essentially the Carrier Sense Multiple Access (CSMA) protocol. To achieve reliable key distribution in the CSMA protocol, one may perform multiple unicasts with handshakes (RTS/CTS/Data/ACK), but this suffers from high latency and excessive control traffic. Broadcasting keys eliminates these problems, but, since the CSMA protocol does not provide any means of recovering lost frames, the broadcast reliability is degraded due to the increased probability of frame losses as a result of frame collisions. Several protocols [77, 104, 105, 108] have been proposed to improve the CSMA's broadcast reliability. Unfortunately, they still introduce significant control traffic, without guaranteeing 100% reliability. We, therefore, need a key-management protocol that reliably coordinates the key-distribution service.

In this chapter, we propose a *Group-based Key Management Protocol* (GKMP) that is equipped with key renewability and makes a tradeoff between security and resource consumption. The heart of GKMP is a novel re-keying protocol that (1) periodically renews the shared key to solve the keystream-reuse problem and maximize scalability/energy-efficiency. and (2) supports reliable key-distribution. The re-keying protocol has the following salient features:

- Efficient key broadcasting without retransmission/ACKs;
- Implicit authentication for new keys without incurring additional overhead;
- Ability of detecting/recovering lost keys;

- Seamless key refreshment without disrupting ongoing data transmission; and
- Robustness to clock skews among nodes.

These features make GKMP very flexible in that it only requires very loose time synchronization, and does not stress the underlying network/link layers, i.e., not requiring reliable broadcast at the link-layer. GKMP is also energy-efficient and robust to DoS attacks, since it does not require any retransmissions or other control packets. To our best knowledge, there is no previous work that effectively handles all of these issues.

We propose a joint authentication and recovery algorithm for re-keying, in which the key-server periodically broadcasts a new key well before its use for encryption/decryption, and a client node first authenticates the received key and then recovers all previously-missed keys, if any. The proposed algorithm relies on the unique properties of the cryptographic one-way function. It is efficient in that each node buffers keys only, as compared to TESLA [85] which buffers all the received data packets until the node receives an error-free key. GKMP also uses double-buffering of keys for their seamless, robust refreshment: while the key in one slot is being used for data encryption/decryption, the next key will be written to the other slot.

The rest of the chapter is organized as follows. Section 2.2 presents the network and threat models. Section 2.3 describes the details of GKMP. Section 2.4 analyzes the theoretic performance of GKMP while Section 2.5 presents the results of our performance evaluation. Finally, the chapter concludes with Section 2.6.

2.2 System Architecture

2.2.1 The Network Model

PEG-like applications are realized on two wireless networks, one for connecting sensors and the other for connecting pursuers, as shown in Figure 2.1. The sensor network

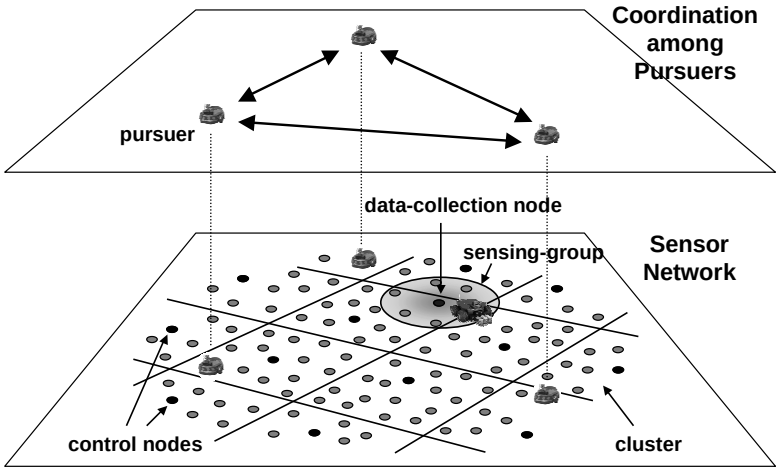


Figure 2.1: The two-tier architecture for heterogeneous sensor networks.

typically covers a wide area, requiring thousands or even millions of sensors, each of which is capable of detecting (part of) an object moving nearby. On top of the sensor network, a separate wireless network of pursuers is formed, for example, to build a terrain map and cooperate with one another to capture/kill evaders based on the information collected from sensors.

Usually, there exists significant heterogeneity between sensors and pursuers. Sensors typically have limited battery energy, computation, memory and communication capabilities. In contrast, pursuers, such as Unmanned Aerial Vehicles (UAV) and Unmanned Ground Vehicles (UGV), do not have such resource limitations. Each pursuer is equipped with the same radio receiver/transmitter as sensors, as well as a more powerful RF interface to communicate with other pursuers.

The sensor network includes (i) *data-collection* nodes, which collect/store sensed data, and process and make it available to pursuers, and/or (ii) *control* nodes, which coordinate (multi-hop) routing among sensors and broadcast commands to sensors. Clusters in this two-tier routing system rely on control nodes, called *cluster-heads*, for managing cluster topology, routing information, pursuers' locations, etc. Clusters may be statically formed

according to geographic grids. Sensing groups, which are formed around evaders to aggregate/disseminate sensor data about evaders, dynamically elect data-collection nodes, as in [12, 38, 66, 126]. In light of the group communication model, we will henceforth refer to each cluster or sensing group as a *group*, and each control or data-collection node as a *group-head* (GH).

Figure 2.1 illustrates the role of GHs in the sensor network: (i) each GH collects data about evaders, and (ii) all GHs cooperate in sending/receiving data to/from the pursuer (inter-group communication) as well as communicating with sensors within their own group (intra-group communication). So, the communication between a sensor and the pursuer is made in three steps: (1) a source s sends data to its GH h_1; (2) h_1 relays the data to another GH h_2 that knows the location of the receiver d; and finally, (3) h_2 forwards the data to d.

There exist two types of intra-group communication, one from GH to sensors and the other from a sensor to GH. GH either unicasts specific commands to a sensor or broadcasts control packets, such as beacons, queries and requests, to all of its sensors, while each sensor unicasts data to its GH. Since sensors are assumed immobile, it suffices for them to use a table-driven routing protocol, under which each GH acts as a coordinator, maintaining the routing topology, and each sensor within a cluster stores only one entry, the next-hop information, in its routing table to reach its GH.

2.2.2 The Threat Model

Possible security attacks we assume are very general: an attacker can eavesdrop, forge, modify, and delete any information. It can also mount off-line dictionary attacks for future break-ins, man-in-the-middle attacks, replay attacks, resource-consumption (or DoS) attacks, sybil attacks, wormhole attacks, and so on. We also assume that an attacker can

take over any sensor node within the network, because perfect tamper-proofing is too expensive to be built into low-cost sensor devices. It is, therefore, reasonable to assume that any secret can be securely preserved from attackers only for a certain period of time.

Compromising a single node means that all nodes within its communication range can be blocked/denied from receiving and/or sending/relaying any information. So, we must minimize the effects of a compromised node on the rest of the network, i.e., the single compromised node should not be allowed to enable subversion of the entire network.

2.3 The Proposed Protocol

2.3.1 Overview

GKMP aims to offer a lightweight security solution for a large-scale network of resource-limited sensor devices. For scalability to a large number of sensors, GKMP decomposes the entire network into clusters and/or sensing groups and selects a GH for each of them, as described in Section 2.2.[3] GKMP addresses the following two main questions associated with the device's resource constraints:

Q1. How to combine security with other services, such as routing, sensor data aggregation/dissemination, and location services?

Q2. How to make a tradeoff between security and resource consumption?

To address the first question, GKMP introduces the notion of *key-server* (KS), which controls the security of a group. For a sensor network that consists of multiple groups, GKMP designates GHs as KSs. The wireless networks for connecting pursuers would also be partitioned into groups, each of which elects the KS among its members. So, without loss of generality, we can assume the existence of one KS per group. GKMP also uses

[3] Each group (cluster) will be reasonably sized. Accordingly, the larger the network gets, the more groups (clusters) GKMP creates.

a Key-Server for the Network (KSN) that coordinates KSs in re-keying for inter-group communications.

For the security tradeoff, GKMP (i) uses a stream cipher for its cheap and fast processing, and (ii) supports periodic renewal of keys with inexpensive cryptographic hash algorithms. It is reliable, and works well with the conventional CSMA protocols that do not support reliable broadcast. Moreover, GKMP requires only very loose time-synchronization among group members.

GKMP achieves the following goals in protecting security-critical information from attackers.

- *Confidentiality*: keeps data from being eavesdropped, and ensures that an attacker will not acquire any information about the plaintext, even if it sees multiple encrypted versions of the same plaintext.
- *Data integrity*: prevents tampering with the transmitted data.
- *Access control*: protects and controls access to the network.
- *Availability*: protects the network from interruptions in service.
- *Key renewability and revocability*: protects the network from compromised nodes, if any.

2.3.2 The GKMP Architecture

For key renewability, GKMP implements a key hierarchy as shown in Figure 2.2. This hierarchy defines two keys: (i) a *temporal key* (TK) for encrypting/decrypting data packets; and (ii) a sensor-specific *master key* (MK) that is used by KS to unicast TK to an individual sensor. Under the symmetric-key cryptography, a TK should be shared by all group members (for intra-group communications) and refreshed periodically to ensure forward and

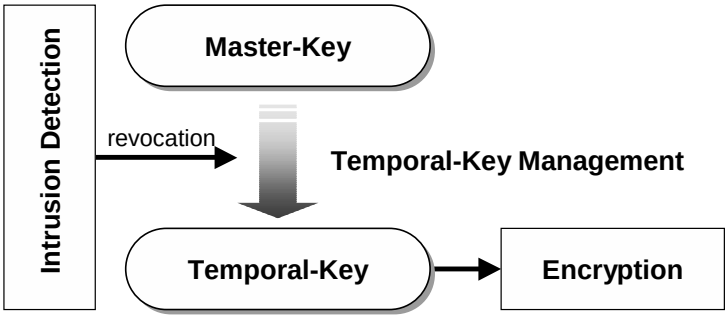


Figure 2.2: The key hierarchy for GKMP

backward confidentiality as well as elimination of keystream collisions. Using its group-based architecture, GKMP achieves scalable and distributed re-keying, since membership changes[4] in a group do not affect the other groups in the network.

The KS executes *entity authentication*[5] with a new sensor joining the group, and if successful, grants a membership to the sensor by storing the sensor's MK in its database and then transmitting the current TK. MKs for sensors will be stored in tamper-resistant hardware, but we assume limited tamper-resistance built in low-cost sensor devices. This means that an attacker may access MKs of the subverted sensors.

As shown in Figures 2.2 and 2.3, there are two main components associated with the key hierarchy: intrusion detection which probes/monitors network activities to uncover compromised nodes, and TK management which protects network traffic from attacks by re-keying TK periodically. Since it is almost impossible to safeguard the network against all possible attacks, it is important to introduce a second line of defense, i.e., GKMP uses an intrusion detection system (IDS) [9, 51, 60, 73, 129, 130] to probe/monitor for anomalies in the network. Since each GH that serves as KS, is a traffic concentration point of the corresponding group, it will be equipped with an IDS to monitor the ongoing

[4]The group membership changes if a new sensor joins the group or if an existing member leaves the group. The latter event occurs when the member is compromised.

[5]The entity authentication between two nodes verifies each other's identity/authenticity. It typically relies on trusted third parties such as distributed certificate authorities [57, 131, 132].

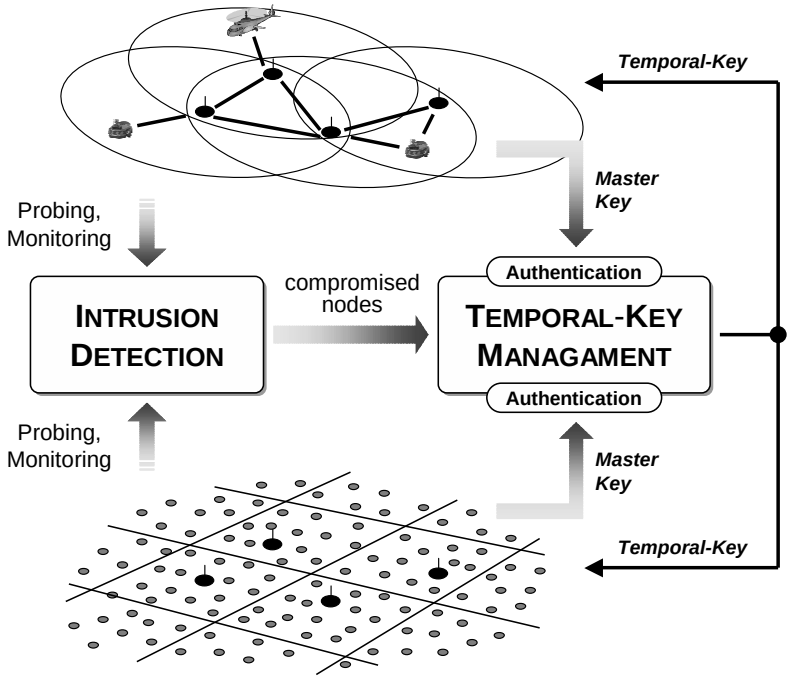


Figure 2.3: The GKMP architecture

traffic. However, in sensor networks, all useful information is local at some point of time, to the group(s) near the evader. Due to this distinct feature, a single point of failure at the KS might cripple the entire network. To avoid this problem, the IDS is organized hierarchically: each KS is in charge of monitoring sensors within a group, while a more powerful IDS running, for example, on pursuers, watch out for possible KS compromises. In our framework depicted in Figure 2.1, both KSs and pursuers are more capable and have more resources than usual sensors, and hence, the IDS's resource consumption, unless it is excessive, should not be an issue.

Once a compromised sensor is identified by the KS, the TK manager disables the sensor and renews the TK in the next update cycle. If a KS is found to have been compromised, GKMP either (i) elects a new KS for the group, or (ii) redistributes member sensors to the neighboring groups.

The TK manager, running on the KS, renews TK for the group. Due to the severe

resource limitation in each sensor device, TK re-keying should be lightweight and conserve resources as much as possible. Our approach to meeting this requirement is to renew TK periodically using (not necessarily reliable) broadcast, instead of using triggered and unicast/retransmission-based renewal. Periodic re-keying of the TK is crucial to counter keystream-reuse attacks and improve scalability/energy-efficiency of group re-keying. The proposed TK management has the following salient properties.

- Efficient TK broadcasting without relying on retransmissions/ACKs;
- Implicit authentication of TKs;
- Fault-tolerance by recovering lost TKs;
- Seamless TK re-keying without disrupting ongoing data transmissions; and
- Robustness to inter-node clock skews.

These properties of GKMP have yielded high-performance TK re-keying: it minimizes the number of control packets generated in the network and reduces the size of each control packet. The TK management will be detailed next.

2.3.3 TK Management

The challenges in TK management are how to enable all nodes to (i) acquire a new TK efficiently, securely and reliably, and (ii) switch to the new TK without disrupting the ongoing data transmission. Note that, with the symmetric-key ciphers, it is difficult to refresh TK seamlessly, as it requires the same key to be possessed by both communicating parties. To address the first challenge, TK distribution must be secure and fault-tolerant: the “secure” part relates to confidentiality and authenticity of TKs, and the “fault-tolerant” part means the ability to restore lost TKs. The second challenge requires seamlessness and weak internode time-synchronization.

The proposed TK management meets the above challenges/requirements while incurring low overhead. The main ideas of the proposed protocol are to: (1) generate a sequence of TKs by utilizing the cryptographic one-way function, similarly to that of S/KEY [40]; (2) distribute each TK well before it is used for encryption/decryption; (3) perform TK buffering in all sensors in the group; and (iv) verify the authenticity of the received TK and detect/recover missing TKs using the other stored TKs.

To ensure secure TK distribution, the KS initiates TK management by encrypting, authenticating and transmitting a control packet that includes the length t of the key-buffer (for TKs), an initial TK, and the TK-refreshment interval, $T_{refresh}$.[6] Then, once every $T_{refresh}$, the KS encrypts and broadcasts a control packet that contains a future TK. Note that the latter does not include a message authentication code (MAC) for TK. Thanks to the cryptographic one-way property of TK sequence, receivers can determine whether or not the received TK belongs to the same key sequence as those stored in the buffer, thus verifying the TK's authenticity. This procedure, called *implicit authentication*, reduces the size of control packet significantly, because the size of MAC (e.g., 128-bit in MD4) is as large as that of fields to be protected.

TK refreshment must tolerate TK losses caused by a noisy channel. A retransmission–based reliability mechanism cannot be used because it will generate too many control packets and/or result in very high latencies. It would be more efficient and more desirable if nodes could automatically restore TKs, rather than asking the KS for retransmission of the lost TKs. Thus, we need a lightweight mechanism to detect the loss of TKs and restore up to t lost TKs in a cryptographically-secure way. GKMP achieves this based on a one-way key sequence.

[6]The larger t, the more fault-tolerant GKMP becomes at the expense of larger key-buffer space. $T_{refresh}$ is a design parameter for making a tradeoff: the shorter $T_{refresh}$, the higher overhead and the smaller re-keying latency. $T_{refresh}$ should be shorter than the interval that ensures collision-free keystreams at sensors' maximum packet-generation rate.

The last two requirements — seamlessness and weak/loose time-synchronization — are met by equipping each node with two key-slots to be used concurrently. While the TK in one key-slot is being used for data encryption, the next TK is written into the other key-slot. Then, at the midpoint of the refresh interval, the node switches the active key-slot to the one with the new TK. In summary, TK management stores/utilizes $t + 2$ TKs, i.e., t TKs in the key-buffer are for authentication and recovery of lost TKs and 2 TKs in the key-slots for encryption/decryption.

Control Packets

GKMP uses three different control packets: `InitKey`, `UpdateKey`, and `RequestKey`. `InitKey` is used by the KS to initiate TK refreshment, and contains, t, the number of lost TKs that can be recovered; an initial TK; $T_{refresh}$, TK refreshment interval; and MAC. The KS unicasts this packet to each group member whenever it wishes to (re)configure TK management with a given set of parameters. `UpdateKey` is used by the KS to periodically broadcast the next TK in the key-sequence, and contains a new TK. `RequestKey` is used by individual nodes to explicitly request the current TK in the key-sequence. This packet is generated when a node failed to receive TKs over t key-refresh intervals.

`UpdateKey` is broadcast to all group members, while `InitKey` and `RequestKey` are unicast between KS and an individual member. Therefore, we use notation `InitKey`(m) and `RequestKey`(m) to specify the unicast between KS and node m. `InitKey`(m) uses node m's master secret, MK_m, for encryption and message authentication, where MK_m is shared only between the KS and node m via entity authentication. By contrast, `UpdateKey` uses currently active TK for encryption.

Initial Setup

The KS pre-computes a one-way sequence of keys, $\{ TK_i \mid i = 1, \cdots, n \}$, where n is chosen to be reasonably large, e.g., 100. It picks the last key, TK_n, randomly and computes the entire key sequence using the cryptographic one-way function H, where each TK_i is derived as $TK_i = H(TK_{i+1})$, $i < n$, or equivalently, $TK_i = H^{n-i}(TK_n)$, $H^j(x) = H^{j-1}(H(x))$ and $H^0(x) = x$. Then, at time t_{start}, the KS starts TK management by unicasting to every node m in its group, `InitKey` along with t, TK_{t+2}, $T_{refresh}$ and MAC:[7]

$$KS \rightarrow m: \; E_{MK_m}(\, t \,|\, TK_{t+2} \,|\, T_{refresh} \,) \mid MAC(\, t \,|\, TK_{t+2} \,|\, T_{refresh} \,),$$

where $E_K(x)$ is the encryption of x with key K, and $MAC(y)$ generates a message digest for y using the cryptographic hash function.

On receiving the `InitKey`(k) packet, node k (i) clears all previous TKs; (ii) allocates a key-buffer of length t (`kb[t]`,$\cdots$,`kb[1]`), and two key-slots; (iii) computes keys, $TK_{t+1}, \cdots, TK_1$, from TK_{t+2}; (iv) stores $\{ TK_{t+2}, \cdots, TK_3 \}$ and $\{ TK_2, TK_1 \}$ in the key-buffer and key-slots, respectively; (v) activates TK_1 for data encryption; and finally (vi) sets `ReKeyingTimer` that expires after $T_{refresh}/2$. When the timer expires, the node (1) switches the active key to TK_2, thus making the key-slot (used to store TK_1) available for the next encryption key TK_3, and (2) sets `ReKeyingTimer` to expire after $T_{refresh}$ for future key switching. Figure 2.4 shows how the node copies TKs into the key-buffer and key-slots, and switches the active-key after receiving TK_{t+2}.

[7] The initial TK is not TK_1 but TK_{t+2}. t and $T_{refresh}$ are network-wide parameters shared by all groups. Since receivers cannot recover from `InitKey` loss, the KS should use external reliability services like retransmissions and handshakes (RTS/CTS/Data/ACK).

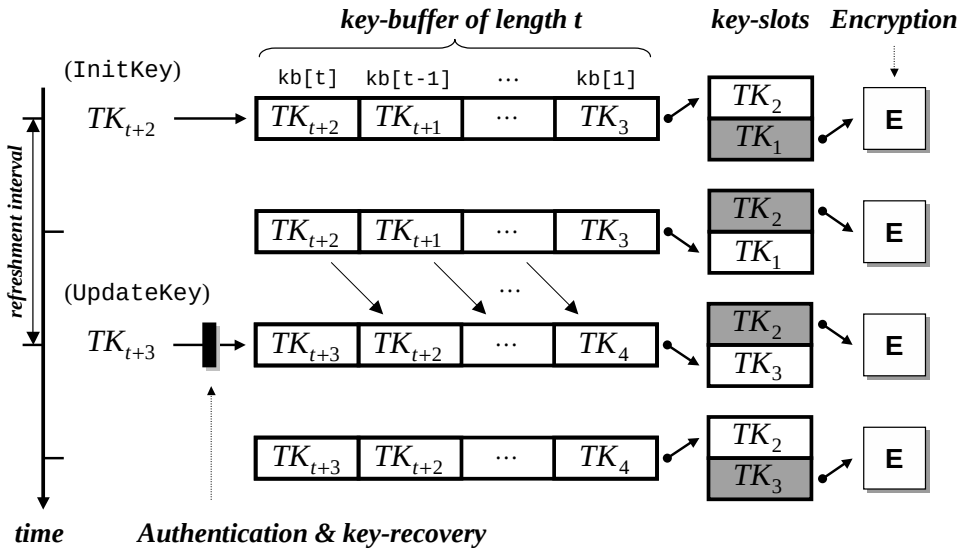


Figure 2.4: TK Management: initial setup and re-keying

Re-keying

After the initial setup, the KS periodically discloses TKs, starting with TK_{t+3} to all nodes in the group. That is, at time $t_{start} + i \cdot T_{refresh}$, the KS broadcasts `UpdateKey` packets containing TK_{i+t+2}, $i = 1, \cdots, n-t-2$:

$$\text{KS} \Rightarrow \text{ group : } E_{TK_{i+1}}(TK_{i+t+2}),$$

where TK_{i+1} is the active encryption key at the time when `UpdateKey` is broadcast.[8]

Upon receiving the `UpdateKey` packet, a node processes it as follows. If it had received the same packet previously or the packet is not from its own KS, the packet is discarded. Otherwise, it re-broadcasts the packet to all of its neighbors and

1. shifts the stored TKs, i.e., `kb[1]` to the inactive key-slot and `kb[i]` to `kb[i-1]`, for $2 \leq i \leq t$;

[8]The group members may reserve an IV value for `UpdateKey` packets to protect the `UpdateKey` packet from keystream collisions.

2. executes TK authentication and recovery on the received TK, as described in Section 2.3.3; and

3. if successful, copies the received TK to `kb[t]` else discards TK.

Whenever `ReKeyingTimer` expires, the node (i) switches the active-key to the TK in the other key-slot, and (ii) sets `ReKeyingTimer` to expire after $T_{refresh}$ elapses. Figure 2.4 illustrates how the key-buffer and key-slots are updated after reception of TK_{t+3} and expiration of `ReKeyingTimer`.

Authentication and Recovery of Lost TKs

After receiving the `UpdateKey` packet, each node verifies if the received TK is authentic, and, if so, recovers lost TKs, if any, using the received TK. To recover from TK losses, the key-buffer stores $\leq t$ TKs. Let $TK_r^\dagger, \cdots, TK_1^\dagger$ denote r ($\leq t$) TKs in the key-buffer $\texttt{kb}[\texttt{r}], \cdots, \texttt{kb}[1]$, respectively, and TK_k is the received TK. Then, there are $e = t - r$ empty slots in the key-buffer. It follows from the property of the one-way key sequence that $H(TK_r^\dagger) = TK_{r-1}^\dagger, \cdots, H(TK_2^\dagger) = TK_1^\dagger$. Since TK_k belongs to the same one-way key sequence, it should meet the following two conditions.

- **Authenticity condition**: TK_k is authentic if $H^{e+1}(TK_k) = TK_r^\dagger$, where $r = t - e$.
- **Fault-tolerance condition**: if $1 \leq e \leq t$, there are e lost TKs, $\{TK_{t-i+1}^\dagger = H^i(TK_k) \mid 1 \leq i \leq e\}$.

TK authentication and recovery uses these two conditions, and works as follows:

- Compute e and $\{ H^i(TK_k) \mid i = 1, \cdots, e+1 \}$.
- If $H^{e+1}(TK_k) \neq TK_r^\dagger$, discard TK_k.
- Otherwise, if $e \geq 1$ then copy $\{H^i(TK_k) \mid i = 1, \cdots, e\}$ to the key-buffer.

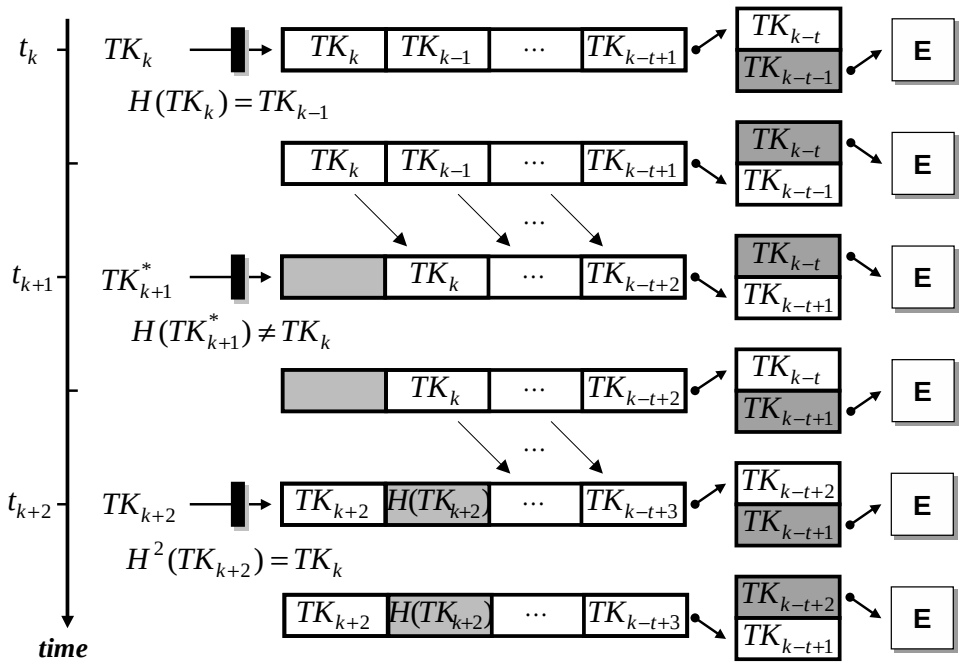


Figure 2.5: TK Management: authentication and recovery of lost TKs

Figure 2.5 illustrates how authentication and key-recovery works. On receiving TK_k, the receiver computes $e = 0$ and $H(TK_k) = TK_{k-1}$, and hence, TK_k is authentic and there are no TK losses. At time t_{k+1}, the node receives $TK^*_{k+1} (\neq TK_{k+1})$, verifies that $H(TK^*_{k+1}) \neq TK_k$ ($e = 0$), and drops TK^*_{k+1}. The key-buffer thus stores $(t-1)$ TKs (or $e = 1$). At a later time t_{k+2}, the node receives TK_{k+2} and verifies that the received TK is the correct key by computing $H^2(TK_{k+2}) = TK_k$, and recovers $TK_{k+1} = H(TK_{k+2})$. Likewise, other TK arrivals will be processed.

GKMP can also detect and correct the situation where a receiver misses TK-disclosures. Consider the case when the node failed to receive TK at time t_{k+1} in Figure 2.5. This can be handled by the `ReKeyingTimer` event triggered at time $t_{k+1} + T_{refresh}/2$: the event handler checks if TKs in the key-buffer has been right-shifted since the last `ReKeyingTimer` event, and, if not, shifts `kb[1]` to the inactive key-slot and `kb[i]` to `kb[i-1]`, for $2 \leq i \leq t$.

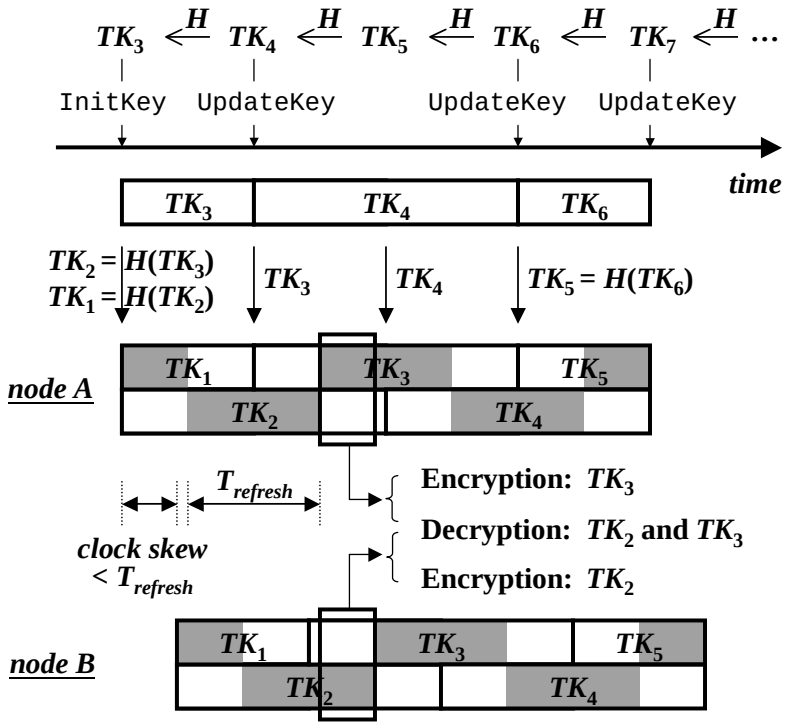


Figure 2.6: TK Management ($t = 1$): robustness to clock skews

Use of the one-way key sequence for recovery of the lost keys in GKMP is similar to that of TESLA. However, the two protocols differ significantly in the way the one-way key sequence is applied. TESLA buffers all of the received packets until it receives an error-free key, and hence, if it misses several key disclosures, TESLA suffers from high latency and large buffer size. In contrast, GKMP buffers only $t + 2$ TKs, so the buffer size is small and fixed. Moreover, the KS distributes TKs well before its use for data encryption, and thus, the missed TKs will not disrupt the ongoing data transmission.

Robustness to Clock Skews

The proposed TK management is robust to clock skews among group members. As an example, Figure 2.6 illustrates, in the time domain, key-slots of two nodes, *A* and *B*, when there exists a clock skew between the two nodes. Thanks to the authentication

and key-recovery, loss of up to t TK losses will not affect the key-slot activation. Let c_j be the mapping from clock time to real time at node j, where $c_j(T) = t$ means that at clock time T, the real time is t. Then, the clock skew between A and B is given by $\delta = |c_A(T) - c_B(T)|$. The figure shows that seamless TK re-keying is preserved, if $\delta < T_{refresh}/2$. During the marked period in Figure 2.6, A uses TK_3 for encryption, while B still uses TK_2 due to the clock skew between A and B. However, since both A and B have the same decryption key pair, $\{ TK_2, TK_3 \}$, they can communicate with each other during this period. In general, GKMP can sustain the worst-case clock skew of $T_{refresh}/2$, i.e., $\max\{ |c_A(T) - c_B(T)| : \forall A, B \} < T_{refresh}/2$.

Moreover, TK distribution from KS to group members also tolerates clock skews of up to $T_{refresh}/2$. That is, TK_k will be processed correctly if it arrives at the node during the time interval $[t_k - T_{refresh}/4,\ t_k + T_{refresh}/4]$, where t_k is the scheduled time when TK_k is disclosed.

Reconfiguration

The KS will reconfigure the TK management at the time of next re-keying, if (1) existing group members have been compromised; (2) all n TKs have been disclosed; (3) a new node has joined the group; or (4) a member has explicitly requested TK, because it missed more than t TK-disclosures. The first two events force all group members to be reconfigured, whereas the third and fourth events allow reconfiguration of the requesting node only. The required actions for each event are summarized as follows.

1. The KS revokes compromised nodes, and if TK_{k-1} has been disclosed previously, discloses TK_{k+t+2}, instead of TK_k, using `InitKey`. This makes all previous TK-disclosures (up to TK_{k-1}) futile.

2. KS computes a new key-sequence $\{ TK'_i \mid i = 1, \cdots, n \}$, and unicasts `InitKey`

with TK'_{t+2} to all members.

3. The KS performs entity authentication with the new node, and if successful, sends the current configuration via an `InitKey` packet.

4. The KS sends the requesting node an `InitKey` packet containing the current configuration.

Tradeoffs

The performance of the proposed re-keying scheme in terms of communication overhead, depends on the size of the group. As the group size increases, each group will have more chances of getting compromised per TK-disclosure, hence increasing the rate of reconfigurations. Since GKMP achieves significant performance gain over conventional re-keying by broadcasting TK-disclosures with no retransmissions, more frequent reconfigurations effectively mean poorer performance. In contrast, larger groups will have more efficient broadcasting of a single TK-disclosure, improving the performance. Therefore, we can make a tradeoff between these two, and there exists an optimal group size that maximizes the overall performance.

2.3.4 Message Encryption/Decryption

For intra-group communications, GKMP encrypts messages with the stream cipher and the currently-active TK. Since each node has two (even and odd) key-slots, GKMP uses a 1-bit $keyID$ to tell which of the two TKs to use. GKMP also includes an ID of the sender and a per-packet IV to counter the keystream-reuse problem.

Figure 2.7 shows the encryption/decryption and authentication of messages from the sender, s, to the destination, d. The message transmission at the sender side proceeds as follows. First, based on the TK referenced by $keyID$, $nodeID$ of the sender s, and the

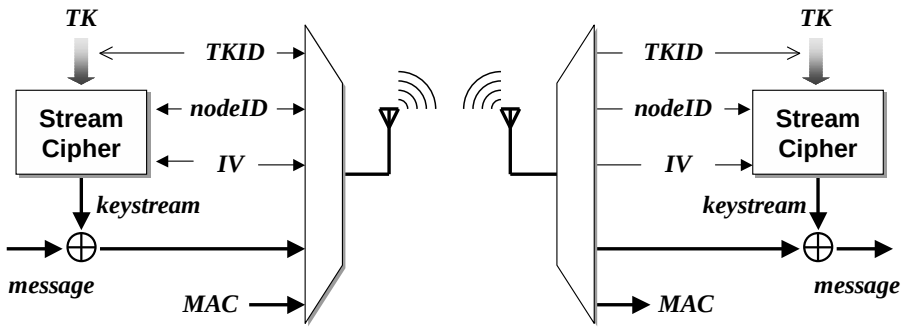


Figure 2.7: Message transmission and reception in GKMP

current IV, s generates a keystream, $keystream(TK, nodeID, IV)$, and then XORs the keystream with plaintext, P, to build a ciphertext, C. Second, s computes a MAC, mac, to protect $keyID$, $nodeID$, IV and P, where $mac = MAC(keyID|nodeID|IV|P)$. Finally, s transmits to d the following information over the wireless link:

$$s \rightarrow d: \quad keyID \,|\, nodeID \,|\, IV \,|\, C \,|\, mac.$$

The data message/packet reception at the destination simply reverses the above process. First, the keystream, $keystream(TK, nodeID, IV)$, is re-generated using the TK pointed to by $keyID$, and then XORed with C to recover the plaintext P'. Second, d checks the integrity of P' by computing $mac' = MAC(keyID|nodeID|IV|P')$ and comparing mac' with the received mac. Only a match will lead to the acceptance of P'.

GKMP ensures that keystreams will never be reused, with the following operations. First, a sender blends its own $nodeID$ into the generation of the keystream to ensure that the various parties sharing TK use different keystreams. Second, a sender increments its own IV by 1 for each message it transmits to avoid any repetition of the keystream. Finally, the KS updates TK at an appropriately-chosen interval, $T_{refresh}$, guaranteeing that none of its members (including itself) starts to reuse IV. The length of the IV field can be made small, thanks to TK re-keying.

The way GKMP manages TK and IVs differs significantly from that of SPINS. In SPINS, IVs are not included in messages/packets, but maintained internally by two communicating peers. SPINS also updates TK whenever IV wraps around. As a result, SPINS has shorter packets than GKMP by the length of the IV field (plus $keyID$). However, SPINS performs poorer than GKMP in the network of a large number of sensors for the following reasons. First, SPINS incurs overheads of (triggered) TK re-keying and IV resynchronization, which increases rapidly as the network size grows. Second, SPINS requires each sensor to allocate memory for maintaining IV states of all other sensors, which also increases with the network size. By contrast, GKMP can control the overhead of TK re-keying regardless of the network size, and reduces the generation of control packets, improving the performance in contention-based networks.

2.3.5 Inter-Group Communication

Under GKMP, the entire network is divided into multiple groups, each with a KS. This architecture is scalable in that compromises in one group do not affect the other groups. It also retains high-performance re-keying, since each reconfiguration is confined to a single group, while groups without any compromised node keep broadcasting TKs. This means that TKs for intra-group communications are independently managed by KSs.

For inter-group communications, KSs should coordinate with one another under the control of KSN as follows. First, all KSs agree in advance on n, t and $T_{refresh}$, by receiving them from KSN. Also, the time to initiate TK management is loosely synchronized with a clock skew of less than $T_{refresh}/2$. Second, KSs and KSN use a key-agreement algorithm such as those in [102] and [99] to agree on the initial seed TK_n for the key-sequence, thus ensuring all KSs to have the same key-sequence. Third, for inter-group traffic, the KS prefixes to the encrypted payload the position of the encryption key in the key-sequence.

```
constants: n, t, T_refresh;

initial setup:
    compute {TK_i | i = 1, ··· , n};
    k = t + 2;
    for all m,
        unicast InitKey(m) to node m;

for every T_refresh:
    k ++;
    broadcast UpdateKey containing E_{TK_{k-t-1}}(TK_k);

if member(s) compromised:
    k += t + 2;
    for all m,
        unicast InitKey(m) to node m;

if k == n:
    do initial setup;

if a new node m joins the group or RequestKey(m) received:
    if m is a new node,
        do entity authentication with m;
    unicast InitKey(m) to node m;
```

Figure 2.8: The pseudocode for the key-server

2.3.6 Realization of GKMP

We now describe how to realize the proposed TK management protocol. Its implementation is comprised of the server-side and client-side programs. Both programs require external modules, such as the intrusion detection system, the entity authentication protocol, and the cryptographic one-way function.

The pseudocode for KS is given in Figure 2.8. After initialization, the KS periodically broadcasts a new TK to all members. It also reconfigures the group security in case of addition/compromise of a node, exhaustion of all TKs, or an explicit request from a member node.

```
if InitKey received:
    Decrypt InitKey to get TK_{t+2};
    Compute TK_{t+1}, ··· , TK_1 from TK_{t+2};
    Copy TK_{t+2}, ··· , TK_1 to key-buffer/key-slots;
    Activate TK_1 for encryption;
    Set e = 0 and TK^† = TK_{t+2};
    Set ReKeyingTimer to T_{refresh}/2;

if UpdateKey received and if not seen before:
    Decrypt UpdateKey to get TK_k;
    Right-shift the key-buffer/key-slot;
    if H^{e+1}(TK_k) != TK^†,
        e++;
    else,
        if e ≥ 1,
            Copy H(TK_k), ··· , H^e(TK_k) to key-buffer;
            e = 0;
        Copy TK_k to key-buffer, TK^† = TK_k;

if ReKeyingTimer triggered:
    if key-buffer not right-shifted,
        Right-shift the key-buffer/key-slot;
        e++;
    Swap active & inactive TKs in key-slots;
    Set ReKeyingTimer to T_{refresh};

if e == t:
    Unicast RequestKey to KS;
```

Figure 2.9: The pseudocode for the client

Figure 2.9 gives the pseudocode for client nodes.[9] The client nodes are *stateless* so that the client-side TK re-keying can be done at a very low cost. Each client maintains two internal variables, e and $TK^{\dagger}$. e keeps track of the number of TK-disclosures that the node failed to receive correctly, and $TK^{\dagger}$ points to the most recent TK in the key-buffer. The right-shift operation is done as illustrated in Figures 2.4 and 2.5.

2.4 Performance Analysis

We derive, through the Markov chain analysis, the computation overhead of client nodes and the communication overhead between the KS and the client. We assume that each occurrence of TK loss or failure is random and mutually independent. We also assume that if the key-buffer of a node becomes empty, it finishes the operation, the request/reception of a current TK (via `RequestKey`), within $T_{refresh}$. These are reasonable assumptions in that $T_{refresh}$ is typically large enough to uncorrelate them. In this section, we first derive a closed-form expression for steady-state distributions for key-buffer states, and then derive computation and communication overheads.

2.4.1 Steady-State Distributions

We model the state of each node with a 2-dimensional Markov chain, as shown in Figure 2.10. Each state (i, j) represents that there were i TK losses and j TK failures, and hence, there are $(i + j)$ empty slots in the key-buffer. The state transition is triggered by three events: a TK loss, a TK authentication failure, and a successful TK reception. As the node misses a TK, the state will be transitioned horizontally, whereas the authentication failure of the received TK will cause a vertical state transition. Let $p_L = \Pr\{TK \text{ is lost}\}$, $p_F = \Pr\{TK \text{ authentication fails} \mid TK \text{ is received}\}$ and $p_S = 1 - p_L - p_F$. Also, let $p(i, j)$ denote the steady-state probability of state (i, j), and $p_e(k)$ the probability that there were

[9] The KS should also perform these tasks for proper communication with other clients.

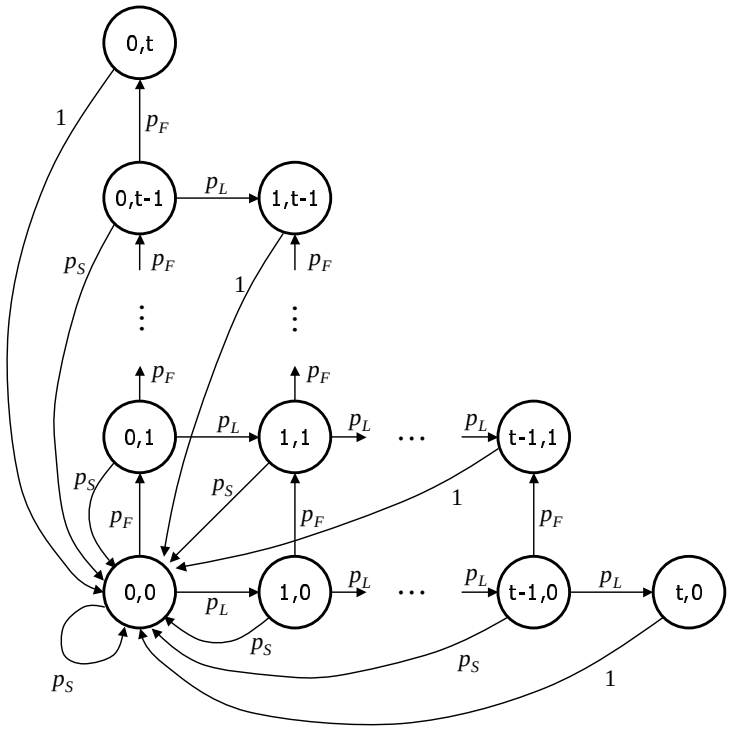


Figure 2.10: The state transition diagram for GKMP

exactly k empty slots. Then, we have

$$p_e(k) = \sum_{i+j=k} p(i,j), \quad k = 0, \cdots, t, \tag{2.1}$$

and

$$\sum_{k=0}^{t} p_e(k) = 1. \tag{2.2}$$

We derive steady-state distributions, $p(i,j)$ and $p_e(k)$, $\forall i, j, k$, as follows. From the global balance equations, we get

$$p(i,j) = \begin{cases} p_F \cdot p(i,j-1) & i = 0, j > 0, \\ p_L \cdot p(i-1,j) & i > 0, j = 0, \\ p_F \cdot p(i,j-1) + p_L \cdot p(i-1,j) & i > 0, j > 0, \\ p^*/(p_F + p_L) & i = j = 0, \end{cases} \tag{2.3}$$

where

$$p^* = \sum_{i'+j'<t,(i',j')\neq(0,0)} p_S \cdot p(i',j') + \sum_{i'+j'=t} p(i',j'). \tag{2.4}$$

Representing $p(i,j)$ with respect to $p(0,0)$ yields

$$p(i,j) = \binom{i+j}{i} \cdot p_L^i p_F^j \cdot p(0,0), \quad (i,j) \neq (0,0). \tag{2.5}$$

From Eqs. (2.1) and (2.5), we have

$$p_e(k) = (p_L + p_F)^k \cdot p(0,0), \quad k = 0, \cdots, t. \tag{2.6}$$

Then, from Eqs. (2.2) and (2.6), $p(0,0)$ is given by

$$p(0,0) = \frac{1-(p_L+p_F)}{1-(p_L+p_F)^{t+1}}. \tag{2.7}$$

2.4.2 Computational Overhead

We derive the expected number of hash computations per TK-disclosure. Let N_{client} denote the number of hash computations per TK-disclosure. If there is exactly k $(< t)$ empty slots, N_{client} is given by

$$N_{client} = \begin{cases} 0 & TK \text{ is lost,} \\ k+1 & TK \text{ authentication fails,} \\ k+1 & TK \text{ authentication succeeds.} \end{cases} \tag{2.8}$$

When all t slots are empty, if the node encounters a TK loss or failure, it must explicitly request the next TK via `RequestKey`, and then do $(t+1)$ additional hash computations using the received TK. Therefore,

$$N_{client} = \begin{cases} t+1 & TK \text{ is lost,} \\ 2t+2 & TK \text{ authentication fails,} \\ t+1 & TK \text{ authentication succeeds.} \end{cases} \tag{2.9}$$

The expected N_{client} when there are k empty slots, $E[N_{client} \mid k \text{ empty}]$, is thus derived as

$$E[N_{client} \mid k \text{ empty}] = \begin{cases} (k+1) \cdot (1-p_L) & k < t \\ (t+1) \cdot (1-p_L) + (t+1) \cdot (p_L + p_F) & k = t \end{cases} \tag{2.10}$$

Finally, $E[N_{client}]$ is given by

$$\begin{aligned} E[N_{client}] &= \sum_{k=0}^{t} E[N_{client} \mid k \text{ empty}] \cdot p_e(k) \qquad (2.11) \\ &= \sum_{k=0}^{t} (k+1)(1-p_L)(p_L+p_F)^k p(0,0) + (t+1)(p_L+p_F)^{t+1} p(0,0). \end{aligned}$$

2.4.3 Communication Overhead

We finally derive the communication cost of a client per TK-disclosure. Let C_{init} and C_{update} denote communication costs for transmitting `InitKey` and `UpdateKey` packets, respectively. Also, let $\alpha = C_{init}/C_{update}$. Then, $\alpha > 1$, because the `InitKey` packet consumes more bandwidth/resources. The KS will transmit the `InitKey` packet (1) once every n TK-disclosures; and (2) if all t slots in the key-buffer become empty, else the `UpdateKey` packet is broadcast. Therefore, the expected communication cost of a client is

$$C_{init} \cdot \left[\frac{1}{n} + p_e(t) \right] + C_{update} \cdot \sum_{k=0}^{t-1} p_e(k).$$

The communication cost normalized by C_{update} is given by

$$\begin{aligned} C_{comm} &= \alpha \cdot \left[\frac{1}{n} + p_e(t) \right] + \sum_{k=0}^{t-1} p_e(k) \qquad (2.12) \\ &= \alpha \cdot \left[\frac{1}{n} + (p_L+p_F)^t p(0,0) \right] + \sum_{k=0}^{t-1} (p_L+p_F)^k p(0,0). \qquad (2.13) \end{aligned}$$

2.5 Performance Evaluation

We are interested in evaluating the resource consumption of the proposed TK management and demonstrating its applicability to the resource-constrained sensor devices. As

Table 2.1: Computational overhead

	Key-Server	Client
Initial setup	$n \cdot C_H$	$(t+1) \cdot C_H$
Re-keying	–	$(e+1) \cdot C_H$

mentioned earlier, the IDS's resource consumption is not a concern, as it runs on a platform with enough resources. To evaluate the performance of TK management, we first quantify (1) the overheads (in both computation and communication) a node pays to renew TKs, and (2) the performance gain the node makes by adding reliability within GKMP. Then, based on the evaluation results, we analyze how GKMP defends itself against various attacks. In this section, we present computation and communication overheads, efficiency of the built-in reliability mechanism, and analyze the security achieved with GKMP.

2.5.1 Computational Overhead

We evaluate the computational overhead of GKMP per group, demonstrating its robustness to losses of, and attacks on, TKs. Since hash computation is the most resource-consuming operation (as compared to data copying/moving), we only consider the cost of computing the cryptographic hash function, H. Let C_H denote the cost of computing a single hash function. Then, computational costs for the KS and the client are given in Table 2.1.

We want to evaluate the average number of hash computations per TK-disclosure. First, the KS, on average, computes $N_{KS} = \frac{n}{n-t-1} < 1$. Since $N_{KS} \approx 1$, if $n \gg t$, the KS performs approximately one hash computation per TK-disclosure. Second, the expected number of hash computations per TK-disclosure, $E[N_{client}]$, of a client is derived in Section 2.4, under the assumption that each occurrence of TK loss or failure

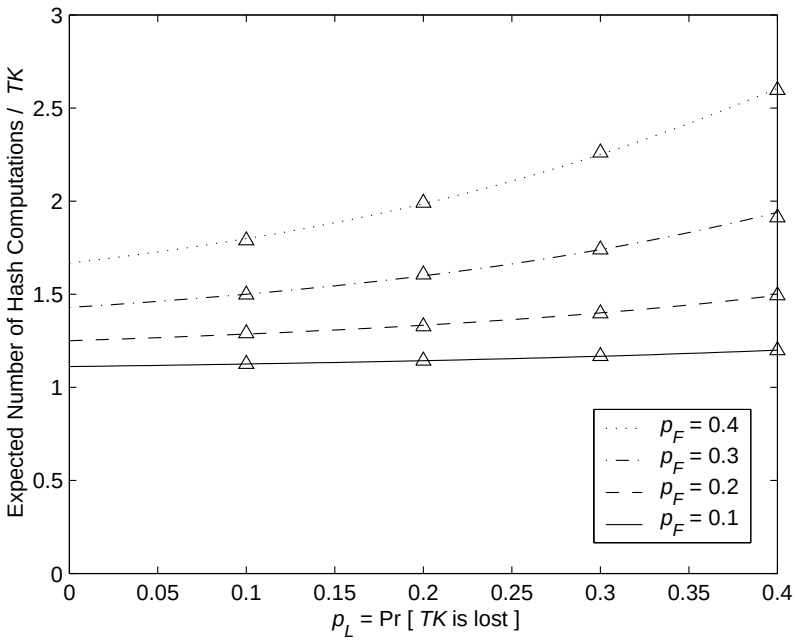


Figure 2.11: The expected number of hash computations per TK-disclosure at the client node vs. p_L: p_F = 0.1 $\sim$ 0.4, $t = 10$

is random and mutually independent. Given fixed p_L (= Pr$\{\, TK$ is lost $\}$) and p_F (= Pr$\{\, TK$ authentication fails $|\, TK$ is received $\}$), from Eq. (2.11), we have

$$E[N_{client}] = \sum_{k=0}^{t}(k+1)(1-p_L)(p_L+p_F)^k p(0,0) + (t+1)(p_L+p_F)^{t+1}p(0,0), \quad (2.14)$$

where

$$p(0,0) = \frac{1-(p_L+p_F)}{1-(p_L+p_F)^{t+1}}. \quad (2.15)$$

p_L reflects the channel condition: a higher p_L represents a highly-lossy wireless channel. By contrast, p_F is mostly affected by the adversary's attempts to manipulate TKs, leading to a DoS attack. So, $E[N_{client}]$ must be small even in case of a high p_F.

Figure 2.11 plots $E[N_{client}]$ as a function of p_L, while varying p_F from 0.1 to 0.4. Similarly, Figure 2.12 plots $E[N_{client}]$ as a function of p_F, while varying p_L from 0.1 to 0.3. The key-buffer length, t, is set to 10. The points marked with '$\triangle$' in the figures are the simulated numbers of the average hash computations for 100,000 TK-disclosures. The

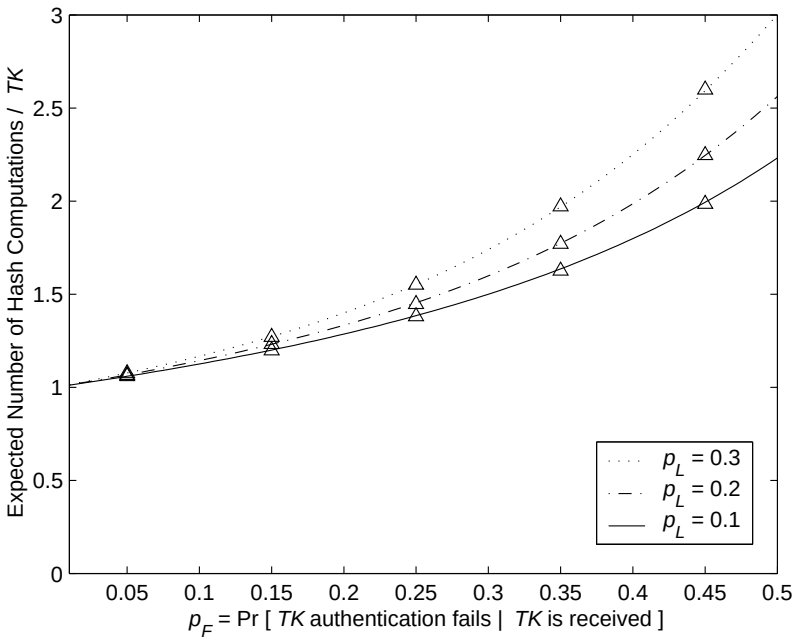


Figure 2.12: The expected number of hash computations per TK-disclosure at the client node vs. p_F: p_L = 0.1 $\sim$ 0.3, $t = 10$

simulation results closely match those obtained from the above equation, verifying the accuracy of the equation.

Figures 2.11 and 2.12 show that p_F has greater influence on the expected hash computations than p_L. For instance, if $p_F = 0$, each node incurs one hash computation, while if $p_F = 0.5$, it computes 2.5 $\sim$ 3 hash functions, because all incoming TKs must be authenticated via hash computations. Each client computes less than three hash functions per TK-disclosure even in the worst case, i.e., when a half of TK broadcasts get corrupted by the attacker ($p_F = 0.5$).

2.5.2 Communication Overhead

We evaluate the overhead of communication between the KS and a client, and show that the key-buffer length determines the communication overhead. The expected communication cost, C_{comm}, normalized by the communication cost of `UpdateKey` transmission

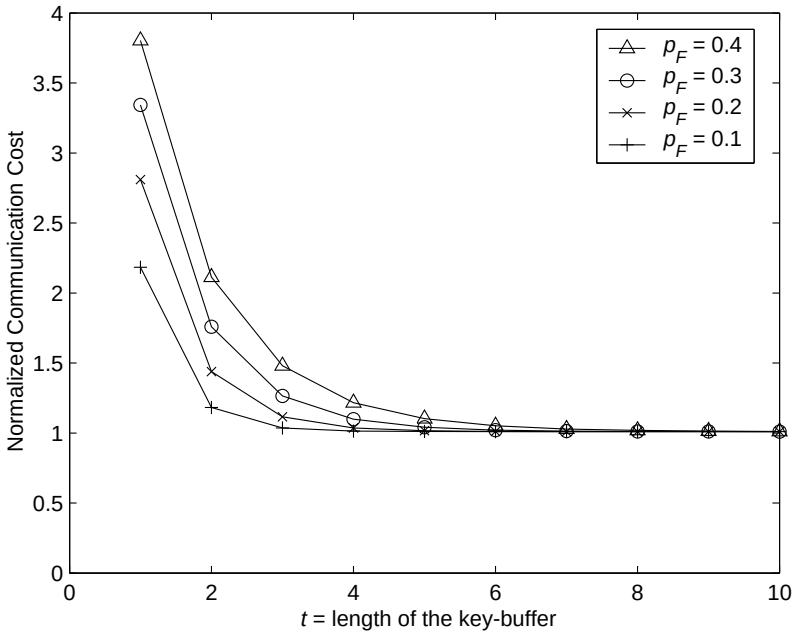


Figure 2.13: The normalized communication cost at the client node vs. t: p_F = 0.1 – 0.4, $\alpha = 10$, $p_L = 0.05$, $n = 1,000$

is:

$$C_{comm} = \alpha \left[\frac{1}{n} + (p_L + p_F)^t p(0,0) \right] + \sum_{k=0}^{t-1} (p_L + p_F)^k p(0,0). \tag{2.16}$$

where α is the ratio of the communication cost of `InitKey` to that of `UpdateKey`. The value of C_{comm} close to 1 means that most TK-disclosures are made through `UpdateKey`, and hence, GKMP is efficient in terms of communication overhead. By contrast, GKMP gets less efficient as C_{comm} approaches α.

Figure 2.13 plots C_{comm} as a function of t when $\alpha = 10$, $p_L = 0.05$ and $n = 1,000$. This choice of α implies that the cost of `InitKey` transmission is 10 times higher than that of `UpdateKey` broadcasting, due to the larger packet size of `InitKey` and requirements for reliability and authentication services. The results show that, regardless of p_F, a smaller t incurs a higher communication overhead. Thus, it is better for the KS to configure GKMP with a large t, as far as clients can allocate the required key-buffer space. A desirable value of t is 5 or 6, with which the normalized communication cost approaches

Table 2.2: Transmission costs

	Num. of packets	Total cost in bytes
GKMP	$0.7N$	$12.6N$
Unicast	$4N$	$58N$

1, even under serious attacks.

2.5.3 Efficiency of TK Management

We would like to evaluate the performance of GKMP, and show its effectiveness in reducing energy consumption. Since a significant portion of energy is spent on packet transmission, we measure transmission costs (both the number of control packets and the total cost in number of bytes) per TK-disclosure. We consider a group consisting of a KS and N group members. It is reasonable to assume that the spanning tree for the group has been established, and hence, each node in the group knows its child nodes.

We compare the proposed TK distribution with the scheme based on unicasts plus explicit message authentication. Other link-layer broadcasting schemes [105, 108] are excluded because they do not ensure 100% reliability. In GKMP, the KS broadcasts `UpdateKey` and, upon receiving the packet, intermediate nodes rebroadcast it if they have at least one child node. Therefore, each TK distribution incurs strictly less than N broadcasts. On the other hand, in the unicast scheme, each node in the group must receive a packet from its parent node via handshake (RTS/CTS/Data/ACK), requiring exactly N unicasts. Also, the unicast packet must include a message authentication code in it.

For the purpose of evaluation, we chose parameter values based on settings in [127], i.e., the sizes of the link-layer header, CRC, RTS, CTS and ACK are 6, 2, 8, 8 and 8 bytes, respectively. Each TK is set to be 10 bytes long and the message authentication code

(MD4) is 16 bytes long. Then, the size of broadcast (`UpdateKey`) and unicast packets are 18 and 34 bytes, respectively. To determine the number of broadcast packets generated in GKMP, we conducted simulation, in which locations of N nodes were randomly generated, a spanning tree among them was built, and the total number of broadcast packets were counted. The result is that a group of N nodes (excluding the KS), on average, generated $0.7N$ broadcast packets per TK-disclosure. From these results, we can calculate the transmission costs of GKMP and the unicast scheme, as given in Table 2.2.[10] Clearly, GKMP outperforms the unicast scheme in that the transmission cost of GKMP is only 18% (in number of packets) and 22% (in number of bytes) of the unicast case.

2.5.4 Security Analysis

We now discuss how GKMP defends against various attacks. As described in Section 1.3, an adversary either passively eavesdrops ongoing packet transmissions or actively inject packets into the network to disrupt network functions. In particular, the adversary will likely mount active attacks against TK management, i.e., modifying/injecting false TKs, jamming the channel to disrupt TK reception, inducing collisions on packets conveying TK, forcing nodes to repeat TK retransmission, etc. These attacks may, in turn, result in DoS, replay and man-in-the-middle attacks.

GKMP is effective in defeating attacks on TK management as follows. First, any modification to the TK will be rejected by the authentication test at the receiver. Similarly, any dropped TK due to collision will be recovered before its activation. Second, the expected computational overhead is bounded: (i) as shown in Section 2.5.1, each node incurs, on average, less than 3 hash computations, even when about 65% of TKs are manipulated/compromised by the attacker; (ii) the KS can easily detect the presence of attacks and disable the associated nodes, if more than a half of TKs fail authentication tests.

[10] In the unicast scheme, we ignored possible collisions on RTS packets.

Therefore, GKMP is robust to DoS attacks (attempting to interfere with TK distribution via high-power jamming and forced collisions or retransmissions) in that each client is expected to compute up to 3 hash computations per TK-disclosure and lost/corrupted TKs can be recovered without retransmissions. Third, replay attacks will not succeed; since the TK manager expects a unique TK in the given refreshment interval that cannot be inferred from the past TKs, replayed TKs will not pass authentication tests. Fourth, GKMP defeats main-in-the-middle attacks, in which the attacker fools the group as if s/he were the KS, because each client is capable of rejecting a false KS in the mutual authentication stage with the KS. Finally, from the fact that TK is transmitted every $T_{refresh}$, the adversary can predict when to launch an attack. We can prevent this by introducing randomization: the KS adds Δ, chosen from the interval $[-T_{refresh}/8,\ T_{refresh}/8]$, to the scheduled broadcast time. However, this scheme will reduce the timing margin against clock skews to $T_{refresh}/4$.

The attacker may subvert a node and acquire all key information. In such a case, s/he can eavesdrop communications and immediately inject bogus messages within the group. However, GKMP preserves the security of the whole network as follows. First, this attack will be valid only until an IDS, running on KS, detects/disables the node. Note that it is crucial to have a good IDS, which can uncover any compromise on keys or the node itself with minimal latency. Second, the scope of this attack will be limited to a single group.

GKMP also prevents attacks on data packets. It does not allow the attacker to mount keystream reuse-based attacks by periodically renewing TK, and mixing $nodeID$ and per-packet IV in the generation of keystreams, as explained in Section 2.3.4. Moreover, s/he can neither decipher nor inject/modify data packets, without the knowledge of TK.

A sybil attack is not possible in our two-tier architecture because each KS can act as the authority (or an IDS) within each group that monitors the ongoing traffic to verify the

1-to-1 correspondence between IDs and locations of the group members. Although the KS can be a single point of failure of the group, it is much better protected from attacks as it is more capable than usual sensors, and moreover, a powerful IDS is running on the KSN to uncover/remove compromised KSs, as described in Section 2.3.2. Finally, wormhole attacks are ineffective because a wormhole link between two different groups doesn't increase the attack strength thanks to the independent management of GKs.

2.6 Conclusion

In this chapter, we proposed a Group-based Key Management Protocol (GKMP) that makes security & energy-efficiency tradeoffs via efficient refreshment of keys. In GKMP, a key-server independently maintains the security of a group using two main components: intrusion detection and TK management. By employing the cryptographic one-way function and TK double-buffering, the TK management offers (i) efficient TK broadcasting without relying on retransmissions/ACKs; (ii) authentication and TK recovery without incurring additional overhead; (iii) seamless TK re-keying without disrupting ongoing data traffic. Moreover, it withstands very loose time-synchronization and does not require reliability support at the link layer.

We evaluated the performance of GKMP using its overhead measurements and security analysis. The measurement results have shown that (1) each node computes, on average, less than 3 hash functions per TK-disclosure, even in the presence of severe attacks on TKs, and (2) with the storage of $\geq$ 8 TKs ($t \geq 6$), GKMP distributes most TKs via broadcasting, and hence, it is very efficient compared to other reliability mechanisms. Our security analysis has demonstrated GKMP's effectiveness in defeating various security attacks. GKMP's strength lies in meeting conflicting goals of providing high-level security and maximizing energy-efficiency.

CHAPTER III

SECURE ROUTING BASED ON DISTRIBUTED KEY SHARING

3.1 Introduction

The critical role of sensor networks in their intended applications requires high-level security throughout their lifetime. TinySec [107] realizes a link-layer security mechanism for message encryption and authentication based on symmetric-key ciphers. In this scheme, keys can be established and renewed with conventional public-key algorithms [4, 20], such as the well-known Diffie-Hellman (DH) protocol. However, they are not suitable for sensor networks, as they usually require complicated processing, extensive usage of memory, and large key length, causing faster depletion of battery if they were used in sensor devices. Attempts [68, 106] have been made to realize public-key algorithms on a well-known Motes platform [33], to show the feasibility of public-key algorithms on sensor devices. Their implementations are, however, still too "heavy" to be employed in sensor devices: each public-key operation consumed 1.19 $\sim$ 12.64 [J], allowing just 51,731 $\sim$ 4,870 operations even when a sensor's total energy budget of 61,560 [J] was devoted solely to this task. Obviously, this result is not acceptable as sensor nodes must also perform other tasks and are required to be operational for at least several months without replacing/recharging their batteries.

We, therefore, need a *lightweight* security protocol whose primary design objective is energy-efficiency. To this end, we have taken an approach to building a secure network layer via 'cooperation' among sensor nodes themselves, instead of relying on a trusted central server. This approach is motivated by the fact that a sensor network inherently relies on collective assurance among multiple low-cost sensor nodes to execute high-precision applications/missions, and hence, it is important to use mutual cooperation in developing energy-efficient security protocols that achieve high-level security without using resource-demanding public-key ciphers, thus extending the network lifetime significantly.

It is essential to share keys among sensors for their proper operations in the link-layer security mechanism. Clearly, the degree of key sharing is inversely proportional to the level of security the sensor network can achieve. The highest level of security would be achieved if every pair of sensors share their own keys independently of others, in that individual subversions do not compromise the rest of the network. But, this scheme is not realistic due mainly to the large ($\mathcal{O}(N^2)$ per sensor) storage requirement where N is the total number of sensors. Sharing an identical key among all sensors offers the least security because a single compromised sensor completely reveals the secret of the entire network. Localized key sharing—either cluster-based [7, 19] or pairwise [19, 22, 37]—schemes may mitigate the risk of sensor subversions, but cannot effectively meet the requirements of both security and performance, especially in a large-scale network where it is not uncommon for a sensor node to communicate with a remote peer. Hence, the degree (and the way) of key sharing must be chosen (designed) so as to make a tradeoff between security and performance.

In this chapter, we propose a novel *distributed* way of sharing keys that optimizes the tradeoff between (1) *security*, in terms of reducing the effects of a compromised sensor node on the rest of the network; and (2) *performance*, in terms of achieving high

energy-efficiency and low-degree key sharing. The heart of the proposed key sharing is the concept of *distributed key servers* (DKSs): a sensor node serves as DKSs for a small subset of other sensor nodes (chosen according to their geographic distance and routing direction throughout the entire network) by sharing unique keys. Using DKSs, we present the following secure routing protocols:

- **secure geographic forwarding** that reinforces the conventional geographic forwarding protocol [52, 55] with a *secure distributed lookup service* that delivers packets based on limited (and distributed) knowledge of shared keys; and
- **key establishment** between any pair of sensor nodes, in which two sensors *equally* contribute to the value of the key while achieving *forward secrecy* to others, without relying on the resource-demanding DH protocol.

To our best knowledge, this is the first 'distributed' approach to (i) systematically building a secure network-layer; (ii) realizing a purely symmetric-cipher-based key setup protocol that is flexible enough to trade security for residual energy; and (iii) gracefully tolerating device compromises in that the network security is gracefully degraded with the number of undetected compromised sensors. We finally show, via analysis and simulation, that the proposed protocols are indeed energy-efficient, scalable, flexible, and robust to subversion of individual sensors.

The rest of the chapter is organized as follows. Section 3.2 describes the proposed protocols. Section 3.3 presents the analytical results. Section 3.4 analyzes the security of the proposed protocols while Section 3.5 evaluates their performance via simulation. Finally, the chapter concludes with Section 3.6.

3.2 The Proposed Secure Routing

We first give an overview of the proposed approach, and then describe details of its components.

3.2.1 Overview

We propose lightweight, secure routing protocols for a network of resource-constrained sensor devices. The proposed protocols:

- are tailored to secure communications between *distant* sensor nodes;
- are *flexible* enough to make a tradeoff between security and energy consumption;
- augment the existing localized key sharing with a *global* distributed key sharing infrastructure;
- preserve *compatibility* with existing link-layer security mechanisms; and
- support
    1. *confidentiality* that protects data from unauthorized disclosures,
    2. *data integrity* that does not allow unauthorized creation or modification of data,
    3. *authenticity* that correctly associates the sensors' IDs with data/services/keys, and
    4. *availability* that prevents the disruption of network services.

These salient features will enable the proposed protocols to play a crucial role in securing many critical applications/services, such as data storage based on GHT, data dissemination, location management, and inter-cluster communications for cluster-based networks.

We use the following widely-accepted assumptions.

A1. Each sensor is uniquely identified by its location estimate obtained from the localization service executed during bootstrapping (Section 3.2.2).

A2. Used as an underlying routing protocol is a well-known geographic forwarding protocol (GFP), in which a source or an intermediate sensor sends each packet to one of its neighbors closest to the packet's destination.

A3. For the proper operation of GFP, each sensor keeps a list of its neighbors' locations based on BEACON packets exchanged.

For compatibility, each sensor maintains its own key, as well as the cluster key (shared within its cluster) and/or pairwise keys (shared with its neighbors). The cluster and pairwise keys are created during the initial bootstrapping of the network, via the cluster-based key management and key pre-deployment schemes, respectively. The challenge in this environment is that each sensor doesn't have keying relationships with most of other sensors located outside the local cluster and/or its neighborhood. To address this challenge, we present a distributed key sharing scheme in which a chosen sensor elects, from the entire network, a small number of other sensors to serve as distributed key servers (DKSs) by creating/sharing unique keys. The proposed scheme builds an efficient, global key-sharing framework that covers the entire network throughout its lifetime.

Built on top of this framework, we propose two protocols for secure routing: a *secure geographic forwarding protocol* (SGFP) and a *temporal-key establishment protocol* (TKEP). SGFP provides a secure distributed lookup service that executes recursive DKS queries, each secured with its shared key, until a neighbor of the destination node is found. This can be viewed as a secure extension of the Distributed Hash Table (DHT) routing [62, 89, 93, 103] for peer-to-peer and ad hoc networks. TKEP then relies on SGFP to realize the purely symmetric-cipher-based key setup. As shown in Figure 3.1, these se-

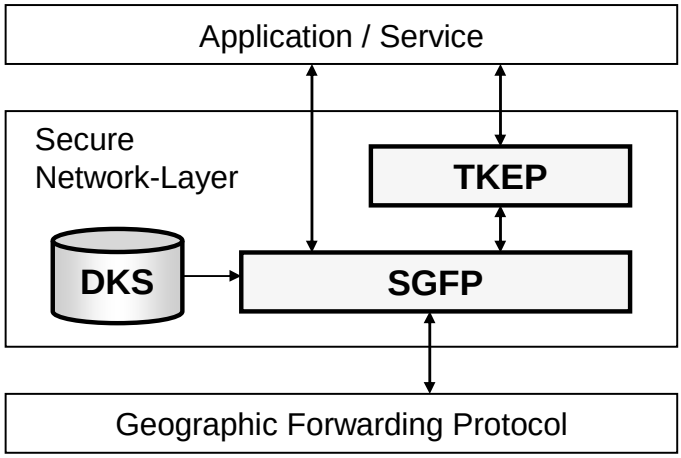


Figure 3.1: The proposed secure network-layer

curity building blocks interact with each other to form a secure network-layer between applications/services and GFP. Accordingly, applications/services invoke either SGFP (for packet-by-packet protection) or TKEP (for secure session establishment) which will then ask GFP to deliver the security-added packets. Note that we use GFP simply as a mechanism to forward packets created/processed securely by SGFP and TKEP, and that the security depends on how the packet's payload is handled, but not on how the packet gets to its destination.

We define and use three types of unique symmetric keys: (1) a *sensor key* (SK) individually generated by each sensor, (2) a *mission key* (MK) shared by a sensor and its DKS (or each of its direct neighbors), and (3) a *temporal key* (TK) for encrypting/authenticating a session of data traffic. We use notation, SK_s, $MK_{i,j}$ and $TK_{i,j}$, to refer to the SK of sensor s, MK and TK shared between sensors i and j, respectively. During initialization, sensors i and j agree on a unique $MK_{i,j}$. $TK_{i,j}$ will be established, whenever needed, between i and j using TKEP.

3.2.2 Distributed Key Sharing

The distributed key sharing is proposed to address the challenges in dealing with a large number of battery-powered sensors. It differs from the dedicated key-server solution in that (chosen) sensors act both as *key-servers* (i.e., DKSs) storing a small number of MKs shared with other sensors chosen in the network coverage area, and as *key-clients* querying DKSs for secure routing. It is essentially a distributed database that is cooperatively maintained and accessed.

DKS Architecture

To control the overhead of initially setting up DKSs, we enforce that a sensor, called a *DKS-sensor*, builds the DKS map only if it has no DKS-sensor within its neighborhood (just like Bluetooth's piconet). Otherwise, a sensor establishes a pairwise key shared with one of its neighbors that acts as a DKS-sensor, thereby relying on that sensor for secure routing. That is, if a sensor has not heard from the DKS-sensor (e.g., via a BEACON packet), it declares itself as a DKS-sensor by broadcasting this decision to all its neighbors. This way, only a small portion ($\leq 10\%$) of sensors will execute the DKS setup process.

Figure 3.2 shows how a DKS-sensor constructs its map of DKSs associated with its geographic location. It partitions the network into squares of various levels and elects DKS(s) in each square. A sensor elects its DKSs based on the facts that (1) distribution of its DKSs should be denser in its proximity but sparser farther away from it, and (2) DKSs are direction-aware, i.e., one DKS for each of 8 directions at the same level.[1] Here we allow up to level-K squares and DKSs.

The DKS map of a DKS-sensor s is built as follows. First, a level-0 square, $L_0(s)$, with a pre-defined area λ^2, is formed around s. s then establishes pairwise keys shared

[1] The 8 directions—NW, N, NE, E, SE, S, SW and W—are assign numbers $1, \ldots, 8$, respectively.

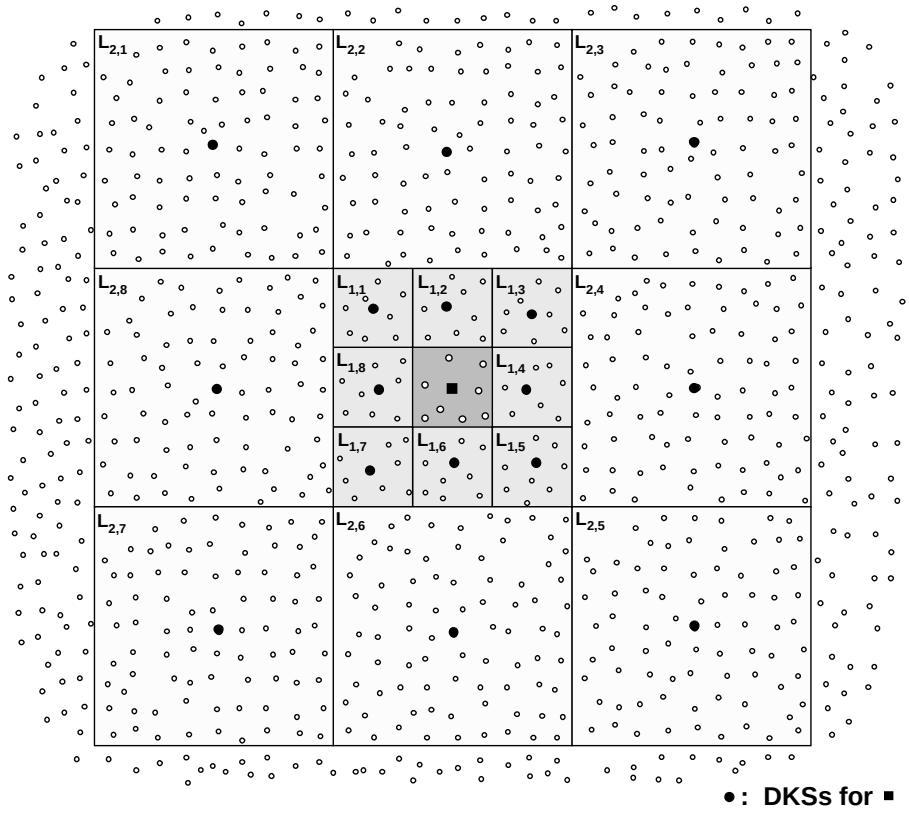


Figure 3.2: The map of DKSs for a DKS-sensor (located at the center) when K = 2.

with each of the sensors in $L_0(s)$, as well as the shared cluster key. Second, 8 level-1 squares, $L_{1,m}(s), m = 1, \ldots, 8$, each of the same size as $L_0(s)$, is located around $L_0(s)$. Then, the DKS-sensor closest to the center of each $L_{1,m}(s)$ is selected as a level-1 DKS, denoted as $\mathcal{DKS}_{1,m}(s)$. Third, 8 level-2 squares, $L_{2,m}(s)$'s, are formed to surround the cumulative area of level-0 and all level-1 squares, each of area $9\lambda^2$, i.e., 9 times the area of $L_0(s)$. Again, the DKS-sensors closest to the center of each level-2 square are elected as level-2 DKSs, i.e., $\mathcal{DKS}_{2,m}(s), m = 1, \cdots, 8$. Likewise, higher-level squares (up to level-K) are constructed and DKSs elected. All DKS-sensors in the network construct their own map of DKSs by using the above procedure.

Each DKS-sensor elects no more than $8K$ DKSs when the network has been configured to have up to level-K squares. For instance, when $K = 2$, it elects up to 16 DKSs regardless of the total number of sensors in the network. The average value (over the entire network) is slightly less than $8K$, since sensors near the border of the network area would elect less than 8 DKSs for outer-levels.

Initial Bootstrapping

Each and every sensor, right after its deployment, executes the conventional bootstrapping process that consists of the following sequential steps:

B1. determine its location estimate by running the (attack-tolerant) localization algorithm with other sensors;

B2. generate its own SK;

B3. set up pairwise keys (and a cluster key) with its neighbors (and a cluster) according to the existing key pre-deployment schemes such as [37]; and

B4. elect a DKS-sensor (either itself or a neighbor) with all its neighbors.

In step B1, we may apply the existing localization protocols that can defeat and/or resist localization-targeted attacks. One of such protocols is VeIL (Chapter IV), under which sensors cooperatively safeguard the localization services by exploiting the high spatio-temporal correlation between adjacent nodes, hence requiring no cryptographic bindings among sensors at this stage.

After step B3, any sensor s establishes cryptographic bindings between the pairwise keys and the neighbors' IDs leading to the construction of secure links with each of its neighbors, e.g., it agrees on a unique $MK_{s,g}$ with its neighbor g. The key pre-deployment method, albeit communication-intensive, has been widely used in, and is known to be a feasible solution for, resource-constrained sensor devices. The communication overhead wouldn't be an issue as long as the bootstrapping is executed only once per sensor.

DKS Setup

After bootstrapping, a sensor s, if elected as a DKS-sensor, executes the DKS setup procedure to build key sharing relationships with other DKS-sensors. This is to construct, for each of the DKS-sensors, a cryptographic binding between MK and the IDs of the two remote DKS-sensors based on already-established pairwise keys.

During the DKS setup, s sequentially contacts each of its candidate DKSs and establishes a shared MK. That is, **for** $1 \leq k \leq K$ and $1 \leq m \leq 8$, sensor s:

1. identifies the location of the center of $L_{k,m}(s)$, based on its own location and DKS map;
2. discovers $f_{k,m} = \mathcal{DKS}_{k,m}(s)$ that is closest to this desired location; and
3. sets up a shared $MK_{s,f_{k,m}}$ with $f_{k,m}$.

Here, $f_{k,m}$ can be found easily as follows. First, an $f_{k,m}$-discovery packet is relayed us-

ing GFP until it arrives at the first sensor that has the center of $L_{k,m}(s)$ within its transmission range. Note that GFP is guaranteed to find such a sensor, if exists, even in the presence of hole(s) along the path. If the sensor has a neighbor closer to the center of $L_{k,m}(s)$, it forwards the packet to that neighbor; otherwise, the sensor determines itself or its DKS-sensor as $f_{k,m}$ of s. If the sensor fails to find $f_{k,m}$ (possibly due to a hole near the desired location), it may flood the received packet within its proximity (neighborhood) to find a DKS-sensor eligible to be $f_{k,m}$. Also note that the use of insecure GFP in the $f_{k,m}$-discovery causes no security vulnerabilities because GFP is just an underlying routing protocol used to deliver the security-added packets that will be processed by external mechanisms like the DH protocol, as described next.

There are several ways to set up $MK_{s,f_{k,m}}$. First, the DH protocol can be applied to establish a unique MK between s and $f_{k,m}$. The use of DH protocol at this stage is acceptable because it is executed only during the DKS setup while future transactions will be secured by our proposed protocols. Second, s and $f_{k,m}$ can use the key pre-deployment scheme (over a multi-hop path) to find if they have a common preloaded key. If so, they can come up with their own $MK_{s,f_{k,m}}$ using this common key. Third, we may simplify the MK establishment under the assumption that each sensor is safe against physical attacks for a certain period of time after its initial deployment, during which it can complete the DKS setup.[2] Then, relaying sensors would not harm the DKS setup process, and hence, s and $f_{k,m}$ exchange their SKs and random numbers, n_1 and n_2, and then compute $MK_{s,f_{k,m}}$ using these values. The delivery of $\{SK_s, n_1\}$ from s to $f_{k,m}$ is protected by pairwise keys of intermediate sensors: s uses its pairwise key to get to one of its neighbors, which will then forward it to the next sensor closer to $f_{k,m}$ after re-encrypting it with its own pairwise key; the subsequent forwarding is processed in the same way until it arrives at

[2]The rationale behind this assumption is that it would take time for an adversary to locate/capture the victims.

$f_{k,m}$. Likewise, $\{SK_{f_{k,m}}, n_2\}$ will be delivered from $f_{k,m}$ to s via hop-by-hop transcoding. Finally, s and $f_{k,m}$ computes $MK_{s,f_{k,m}} = F(SK_s, SK_{f_{k,m}}, n_1, n_2)$ where F is a fixed hash function. Again, the hop-by-hop transcoding is used only once during this stage while SGFP will be applied to protect the future communications from attacks and compromised sensors.

A complete DKS setup protocol is summarized as follows. For each $k \leq K$ and $m \leq 8$,

D1. s generates a packet containing the location of $L_{k,m}(s)$'s center and the security context (necessary to set up MK), and geographically forwards the packet towards the center of $L_{k,m}(s)$;

D2. a sensor receiving the packet:

D2.1. **if** it (or its neighboring DKS-sensor) is closest to the location marked in the packet, declares itself (or its neighbor) as $f_{k,m}$ who will then reply back to s with its own location and security context;

D2.2. **else**, relays the received packet to the next hop towards the center of $L_{k,m}(s)$;

D3. both s and $f_{k,m}$ compute (or agree on) a unique $MK_{s,f_{k,m}}$ and store it in their routing table.

Figure 3.3 shows the structure of the routing table of DKS-sensor s built according to the above procedure. It consists of three fields—the DKS level, the location and a shared MK—for all $8K$ DKS-sensors chosen during the DKS setup. Hence, each DKS-sensor stores at most $8K$ additional MKs (e.g., 16 MKs when $K = 2$), which is significantly fewer than the requirement of N^2 MKs (where N is the total number of sensors in the network) needed for maximum security. This indicates that our distributed key sharing

Level	Location	Key
1	$x_{f_{1,1}}, y_{f_{1,1}}$	$MK_{s,f_{1,1}}$
	...	...
	$x_{f_{1,8}}, y_{f_{1,8}}$	$MK_{s,f_{1,8}}$
...	...	...
K	$x_{f_{K,1}}, y_{f_{K,1}}$	$MK_{s,f_{K,1}}$
	...	...
	$x_{f_{K,8}}, y_{f_{K,8}}$	$MK_{s,f_{K,8}}$

Figure 3.3: The routing table of s, having $8K$ DKS entries

is very efficient in terms of the key storage requirement, incurs a very low degree of key sharing, and scales well with the network size.

When $K = 2$, the DKS setup incurs 8 medium-distance (level-1) and 8 long-distance (level-2) handshaking processes to less than 10% of sensors in the network. This overhead of initially setting up DKSs shouldn't be an issue, as it takes place only *once* in the beginning (thus incremental reconfiguration of DKSs afterwards as described in Section 3.2.5) and only those chosen sensors participate in computing/sharing MKs. Moreover, this overhead is not significant at all, compared to the overhead of localization (that must be executed for other applications/services). Both SGFP and TKEP, regardless of this overhead, will extend the lifetime of the network significantly by consuming much less energy than existing schemes, especially for the long-distance traffic. Finally, the time duration of DKS setup is small because it can *simultaneously* set up individual DKSs.

3.2.3 Secure Geographic Forwarding

The SGFP is a multi-hop routing protocol that establishes a secure, unidirectional path between two arbitrary sensors based on the limited and distributed knowledge of DKS-sensors' locations/MKs. SGFP achieves a high level of tolerance/robustness to sensor

compromises by minimizing the number of transcodings per route discovery.[3]

We use the term "link" to refer to GFP between two sensors in the DKS relationship. Each $f_i \rightarrow f_j$ is said to be a level-k link if $f_j = \mathcal{DKS}_{k,m}(f_i)$. The security is preserved over each link by using the shared MK, e.g., MK_{f_i,f_j} for the $f_i \rightarrow f_j$ link. Since each link uses a unique MK, the edge sensor relaying the packet from one link to another should transcode the packet, e.g., from MK_{s,f_i} to MK_{f_i,f_j}.

The heart of SGFP is the DKS selection rule that determines a DKS to establish a link to: each non-DKS-sensor chooses a neighbor DKS-sensor; else, chooses a DKS that considerably reduces the distance to the destination. Using this simple DKS selection rule, SGFP constructs the path as a concatenation of multiple GFP links, each of which is secured by a distinct MK. Below is a description of SGFP:

S1. if a sensor s is a DKS-sensor,

- **S1.1.** it forwards the packet to an intermediate sensor $f_{k,m}$ ($k \leq K$)—instead of the destination d—that is closest to d among DKSs listed in its routing table;
- **S1.2.** $f_{k,m}$, upon receiving the packet, forwards the packet to one of its own DKSs closer to d;
- **S1.3.** the subsequent forwarding is handled in the same way until the packet reaches $f_{1,m'}$ for which $d \in L_0(f_{1,m'})$; and finally,
- **S1.4.** $f_{1,m'}$ uses a pairwise/cluster key to deliver the packet to d;

S2. else, it asks its nearby DKS-sensor to deliver the packet to d (through steps S1.1 $\sim$ S1.4).

[3]The transcoding of a packet consists of (1) decrypting and verifying the authenticity of the packet using the previous MK, then (2) re-encrypting and re-computing the message authentication code with the next MK.

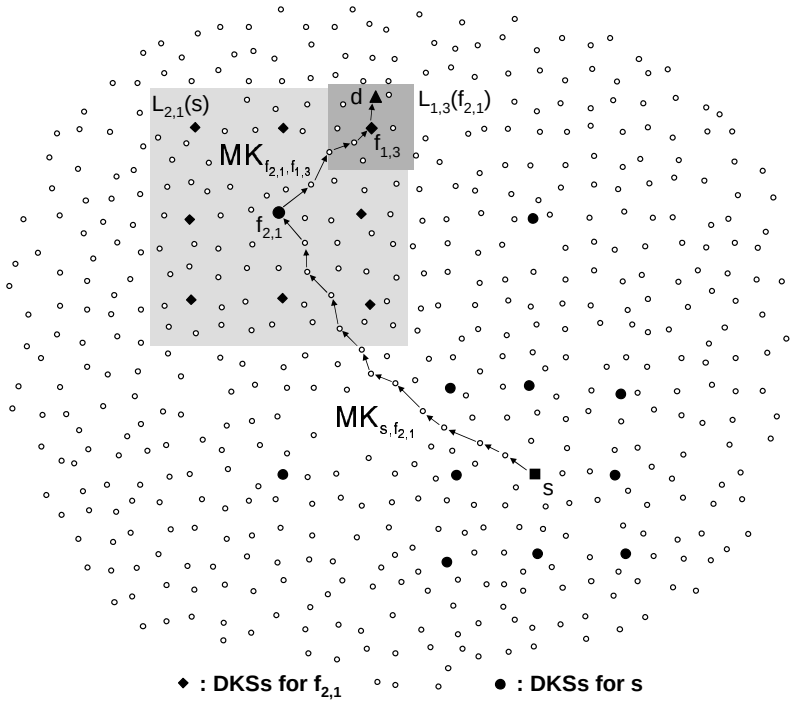


Figure 3.4: SGFP from s to d

SGFP takes a "divide-and-conquer" approach: when d belongs to a level-k square of s where $1 \leq k \leq K$, SGFP typically forms at most k links, $s \rightarrow f_{k,*} \rightarrow \cdots \rightarrow f_{1,*}$, until a DKS-sensor belonging to $L_0(d)$ is found. Figure 3.4 illustrates how SGFP forwards the packet from s to d when d resides in $L_{2,1}(s)$. s (being a DKS-sensor) selects $f_{2,1} = \mathcal{DKS}_{2,1}(s)$ according to the DKS selection rule, encodes the packet with $MK_{s,f_{2,1}}$, then geographically forwards it to $f_{2,1}$. Since $f_{2,1} \neq d$, $f_{2,1}$ repeats the same procedure. d now belongs to $L_{1,3}(f_{2,1})$, and hence, $f_{2,1}$ gets $f_{1,3} = \mathcal{DKS}_{1,3}(f_{2,1})$, transcodes the packet with $MK_{f_{2,1},f_{1,3}}$, and geographically forwards it to $f_{1,3}$. Note that $f_{2,1}$ suffices to search up to level-1 DKSs. Finally, $f_{1,3}$ finds that d is its neighbor, thus forwarding the received packet to d using the pairwise (cluster) key.

When d is outside of all level-K squares, s chooses and forwards the packet to its level-K DKS, $f'_{K,*}$ closest to d. If d belongs to one of $f'_{K,*}$'s squares, the packet is routed via $f'_{K,*} \rightarrow f_{K,*} \rightarrow \cdots \rightarrow f_{1,*}$; otherwise, $f'_{K,*}$ again forwards the packet to one of its

level-K DKS. Therefore, it incurs one or more level-K links in the beginning.

It is possible that d is not directly reachable from $f_{1,*}$ due mainly to the errors in the location estimates, irregular deployment of sensors, absence of the shared key, and so on. In such a case, $f_{1,*}$ selects, and forwards the packet to, its neighbor g which is closer to d. Then, g will likely have a pairwise key, $MK_{g,d}$, shared with d, thus successfully delivering the packet. Otherwise, g will repeat the same procedure. So, SGFP incurs additional (hop-by-hop) transcoding to reach d.

3.2.4 Temporal-Key Establishment

When two sensors need to maintain a persistent session for a certain period of time, it is preferable to establish a shared TK dedicated to that session if they do not yet have a shared key. To address this need, we present TKEP that enables any two sensors to agree on a common TK, meeting the following two requirements:

- *contributory establishment*—none of the two sensors can dictate the value of TK, and
- *forward secrecy*—the rest of the network should not be able to duplicate the established TK.

TKEP is a purely symmetric-cipher-based key setup protocol, and hence, serves as a lightweight alternative to the resource-demanding DH protocol in a large-scale network of sensors.

Basic TKEP

Built on top of SGFP, TKEP realizes the concept of spatial diversity by exploiting the novel DKS infrastructure. Suppose s initiates TKEP between itself and d where both s and d are DKS-sensors (for ease of description). If forward and backward SGFP paths (s-to-d

and d-to-s, respectively) were run via different sets of DKSs, both sensors could contribute to TK via each of the two paths, while no other sensors could duplicate the complete TK. In practice, they exchange random seeds, R_s and R_d (generated by s and d, respectively) using SGFP, and then, individually compute $TK_{s,d} = F(R_s, R_d)$ where F is a fixed hash function. TKEP consists of the following steps:

T1. In the forward path, s:

T1.1. randomly generates R_s;

T1.2. transmits R_s to d using SGFP.

T2. In the backward path, d:

T2.1. randomly generates R_d;

T2.2. transmits R_d to s using SGFP.

T3. s and d individually compute $TK_{s,d} = F(R_s, R_d)$.

Figure 3.5 illustrates how TKEP works between s and d. Let $f_{k,*}$'s and $r_{k,*}$'s refer to DKSs on the forward and backward paths, respectively. Then, the forward and backward SGFP paths are routed via $\{f_{2,1}, f_{1,3}\}$ and $\{r_{2,5}, r_{1,7}\}$, respectively. As shown in this example, TKEP satisfies the contributory establishment condition in that both s and d equally contributes to the value of TK. The forward secrecy is preserved if no sensors other than s and d can get the plaintexts of both R_s and R_d. This condition is automatically met by using SGFP, as justified next.

Thanks to the way DKSs are constructed, all $f_{k,*}$'s and $r_{k,*}$'s must be distinct sensors. For example, in Figure 3.5, $f_{2,1}$ and $f_{1,3}$ reside in $L_{1,7}(d)$ (one of level-1 squares of d) and $L_0(d)$ (level-0 square of d), respectively, while $r_{2,5}$ and $r_{1,7}$ must belong to the level-2 square of d ($= L_{2,5}(d)$), and trivially, DKSs in each of the two paths must be distinct.

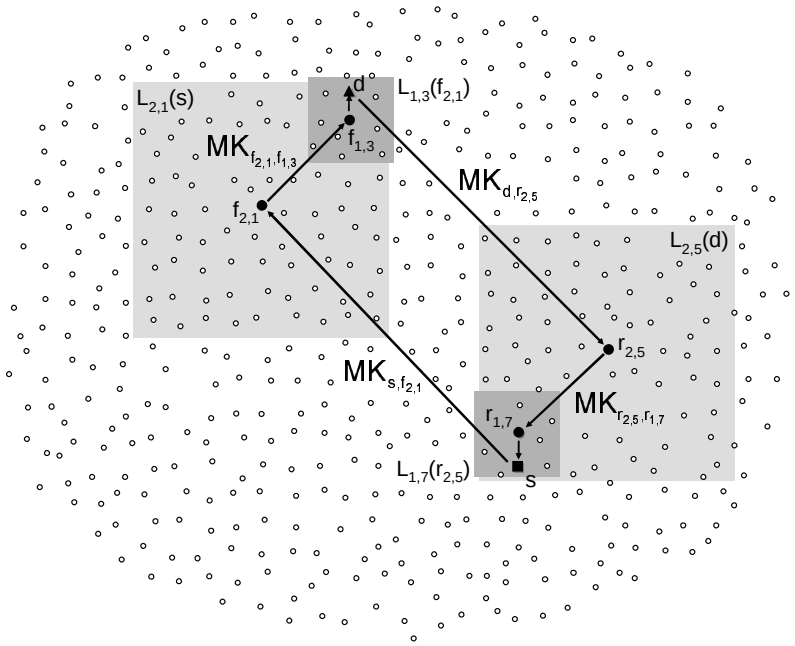


Figure 3.5: TKEP between s and d

Accordingly, no sensor can serve as DKS for both forward and backward paths. (It is clear that these properties hold for general K.) Moreover, all MKs involved in TKEP are unique. Each link is thus secured using MK known only to the edge sensors, implying that no intermediate sensors on that link can decipher it. As a result, sensors other than s and d may decrypt at most one of R_s and R_d, but cannot reproduce both of them. For example, even if the link $d \rightarrow r_{2,5}$ uses $f_{1,3}$ as a relaying node (in Figure 3.5), $f_{1,3}$ does not know $MK_{d,r_{2,5}}$, and hence, it can neither decrypt R_d nor construct $TK_{s,d}$.

TKEP with Randomization

To withstand attacks from compromised DKSs,[4] we may apply "randomization" in which s (and d) randomly picks the next DKS $f_{k,*}$ (and $r_{k,*}$) to form an SGFP path to d (and s). Because intermediate DKSs are randomly picked, the odds that two compromised nodes (sharing information) are on each of the two paths are very small. Thus, it is very

[4]See Section 3.4.3 for details of the attack scenario.

difficult for the adversary to invent a *general* policy to choose victims to be compromised. For example, he has to compromise 14 sensors to figure out TK of a single (s,d) pair. But, this would be too much of an effort for the attacker to make because there are no hot spots or dedicated devices (like cluster-heads) in our distributed environment,[5] and hence, he has no other way but to compromise as many sensors as possible to take control of the network.

μ-Split TKEP

We may achieve the highest-level forward secrecy in the presence of compromised nodes by splitting each random seed into μ pieces and then forwarding each of them over a randomly-chosen SGFP path. Note that there may be as many as 8^K distinct routes because it may choose one of 8 DKSs per level. As a result, both s and d collect all the pieces and compute $TK_{s,d} = F(R_{s,1}, \cdots, R_{s,\mu}, R_{d,1}, \cdots, R_{d,\mu})$. This scheme is capable of trading security for computation (and energy consumption) via the choice of μ.

3.2.5 Steady-State Operations

We now describe DKS reconfiguration that adds/removes DKS-sensors to/from the DKS infrastructure and renews shared MKs.

DKS Reconfiguration

The DKS reconfiguration refers to the on-demand operation that sets up a new DKS-sensor to replace the old one within the proximity. It is triggered by (1) the detection of compromised and/or malfunctioning sensors and (2) the power-saving mode operations [25, 72], allowing sensors to sleep to conserve energy.

[5] As described in Section 1.2.2, many emerging applications and services rely more on the peer-to-peer model in which each sensor communicates directly with the other sensors without relying on dedicated devices.

We use a network intrusion detection system (NIDS) [73, 130] that probes and monitors network activities to uncover compromised sensors, and initiates reconfiguration of DKSs if a sensor is found suspicious of having been compromised. It also does bookkeeping of the thus-removed/blacklisted sensors to block them from re-entering the network. Possible NIDS deployment/usage scenarios are discussed in Chapters II and V.

To balance the energy consumption among sensors, nearby sensors may take turns to serve as a DKS-sensor by executing DKS reconfiguration (while keeping the rest in a low power state). This makes our proposed protocols compatible with existing energy-saving solutions like [25]. The frequency of re-electing DKSs is a configurable network parameter that makes a tradeoff between security and energy consumption. It is either pre-configured upon deployment or adaptively chosen based on local traffic patterns and the characteristics of applications.

The DKS reconfiguration that replaces c with s (a neighbor of c) consists of the following two operations: (1) s discovers $8K$ DKS candidates according to the DKS setup procedure; and (2) those who have been sharing MK with c, replace c with s. In the DKS architecture, these operations can be done efficiently via the following steps. For each $k \leq K$ and $m \leq 8$:

1. s contacts $g_{k,m}$ that is closest to the center of $L_{k,m}(s)$ by executing the discovery protocol of DKS setup;

2. $g_{k,m}$ broadcasts locally to find $f_{k,m}$ that has a shared MK with c; then

3. $f_{k,m}$ deletes (or deactivates) the entry corresponding to c from its routing table, elects a new DKS s, and establishes a shared $MK_{f_{k,m},s}$.

This protocol ensures that each and every DKS-sensor maintains at most one DKS per

square, i.e., the number of DKSs per DKS-sensor is bounded by $8K$, regardless of addition/removal of DKS-sensors.

Renewal of MKs

It is required to renew keys in stream ciphers such as that of TinySec due to the limitation in the maximum number of packets that can be transmitted using the same key. In our proposed key sharing, two DKS-sensors can set up a new MK from two random seeds, each (1) generated independently and individually, (2) encrypted with the current shared MK, and then (3) exchanged via GFP links. The MK can be renewed periodically to simplify the implementation on sensor devices.

3.3 Performance Analysis

We derive the probability that the adversary eavesdrops SGFP and TKEP when a portion of the network had been compromised, and the expected number of transcodings for both SGFP and TKEP.

3.3.1 Preliminaries

We make the following assumptions: (i) sensors are uniformly distributed in the entire network; (ii) each sensor has up to level-K DKSs; (iii) a level-0 square contains, on average, N_0 sensors in a $\lambda \times \lambda$ square area; and (iv) the network covers a square area of $(3^K\lambda) \times (3^K\lambda)$. Then, the expected total number of sensors in the network, N_{net}, is $9^K N_0$. The adversary randomly selects and manipulates sensors to acquire all MKs stored in the compromised sensors.

Let s and d denote source and destination sensors establishing an SGFP path. The number of compromised sensors is denoted by N_c. We define a set of all compromised sensors as $\mathcal{C} = \{c_i, i = 1, \cdots, N_c\}$. The set of all uncompromised sensors is then $\mathcal{U} =$

$\mathcal{C}^c$. The probability that a randomly-selected sensor f has already been compromised is defined as

$$p_c = Pr\{f \in \mathcal{C}\} = \frac{N_c}{9^K N_0}. \tag{3.1}$$

We define the level-k cumulative area of s as:

$$A_k(s) = \begin{cases} L_0(s) & k = 0 \\ \sum_{m=1}^{8} L_{k,m}(s) & 1 \leq k \leq K \end{cases} \tag{3.2}$$

Then, the probability that d lies within $A_k(s)$ is given by:

$$p_{A_k} = Pr\{d \in A_k(s)\} = \begin{cases} \frac{1}{9^K} & k = 0 \\ \frac{8 \cdot 9^{k-1}}{9^K} & 1 \leq k \leq K \end{cases} \tag{3.3}$$

We also define the conditional probability associated with $A_k(s)$ as:

$$p_{L_{k,m}|A_k} = Pr\{\, d \in L_{k,m}(s) \,|\, d \in A_k(s) \,\} = \frac{1}{8} \tag{3.4}$$

where $1 \leq k \leq K$ and $1 \leq m \leq 8$.

3.3.2 Eavesdropping Probabilities

The sample space is a set $\Omega = \{\, (s,d),\ s \neq d,\ d \in \sum_{k=0}^{K} A_k(s)\}$, and the event space $\mathcal{E}$ consists of (s,d) pairs for which the adversary successfully decrypts an SGFP packet. Let P_{SGFP} $(= Pr\{\,\mathcal{E}\})$ and P_{TKEP_μ} denote the probabilities of eavesdropping SGFP and μ-split TKEP, respectively. We then define the following conditional probabilities associated with $\mathcal{E}$:

$$P_k = Pr\{\, \mathcal{E} \,|\, s \in \mathcal{U},\ d \in A_k(s) \,\},\ 0 \leq k \leq K, \tag{3.5}$$

and

$$Q_{k,m} = Pr\{\, \mathcal{E} \,|\, s \in \mathcal{U},\ d \in L_{k,m}(s) \,\} \tag{3.6}$$

where $1 \leq k \leq K$, $1 \leq m \leq 8$.

We derive P_{SGFP} and P_{TKEP_μ} by considering the following two cases. First, when $s \in \mathcal{U}$ and d is located inside $A_0(s)$, the adversary can decrypt the packet if d and/or intermediate sensors have been compromised. Hence, $P_0 = \alpha \cdot p_c$ where α is the average number of hops taken inside the level-0 square. Second, when $s \in \mathcal{U}$ and d lies within $A_k(s)$, $k \geq 1$, P_k is derived by considering $d \in L_{k,m}(s)$, $m = 1, \ldots, 8$, for each of which the adversary decrypts the SGFP packet with the success probability $Q_{k,m}$. Therefore, the following relationship holds:

$$P_k = \sum_{m=1}^{8} p_{L_{k,m}|A_k} \cdot Q_{k,m} \tag{3.7}$$

If $d \in L_{k,m}(s)$, s asks $f_{k,m} = \mathcal{DKS}_{k,m}(s)$ to search, on behalf of itself, a reduced area $A_{k-1}(f_{k,m})$ for d. In this case, the adversary succeeds in eavesdropping if $f_{k,m} \in \mathcal{C}$ or $f_{k,m} \in \mathcal{U}$, but the subsequent forwarding inside $A_{k-1}(f_k)$ is routed via compromised DKSs. Therefore,

$$Q_{k,m} = p_c \cdot 1 + (1 - p_c) \cdot P_{k-1} \tag{3.8}$$

Consequently,

$$P_k = \begin{cases} \alpha p_c & k = 0 \\ p_c + (1 - p_c) P_{k-1} & 1 \leq k \leq K \end{cases} \tag{3.9}$$

From Eqs. (3.3) and (3.9), P_{SGFP} is derived as:

$$P_{SGFP} = \sum_{k=0}^{K} p_{A_k} P_k \tag{3.10}$$

P_{TKEP_1} of the basic TKEP (with no randomization) is then derived from the event that the attacker eavesdrops both R_s and R_d as:

$$P_{TKEP_1} = P_{SGFP}^2. \tag{3.11}$$

Finally, each SGFP path of the μ-split TKEP is bounded by $[\, p_c + (1 - p_c) P_K \,]^\mu$, and hence,

$$P_{TKEP_\mu} \leq [\, p_c + (1 - p_c) P_K \,]^{2\mu} \tag{3.12}$$

3.3.3 Expected Transcoding Attempts

Let T_{SGFP} and T_{TKEP_μ} denote random variables that count the transcoding attempts of SGFP and TKEP, respectively, and p_n the ratio of the number of non-DKS sensors to the total number of sensors. We first derive the conditional expectation, $E_{A_k}[T_{SGFP}] = E[T_{SGFP}|d \in A_k(s)]$, as:

$$E_{A_k}[T_{SGFP}] = k + \alpha + p_n, \quad 0 \leq k \leq K \tag{3.13}$$

$E[T_{SGFP}]$ is then derived as:

$$E[T_{SGFP}] = \sum_{k=0}^{K} p_{A_k} E_{A_k}[T_{SGFP}] \tag{3.14}$$

$$= K + \alpha + p_n - \frac{1}{8}\left(1 - \frac{1}{9^K}\right) \tag{3.15}$$

Finally, $E[T_{TKEP_\mu}]$ is approximated as:

$$E[T_{TKEP_\mu}] \simeq \begin{cases} 2(K + \alpha + p_n - \frac{1}{8}) & \mu = 1 \\ 2\mu(K + \alpha + p_n + \frac{7}{8}) & \mu \geq 1 \end{cases} \tag{3.16}$$

3.4 Security Analysis

This section discusses how our proposed protocols defend against possible attacks.

3.4.1 Prevention of Sybil Attacks

It is possible that a compromised sensor joins the network and creates/uses many different IDs/locations to mount a sybil attack. However, we detect/prevent sybil attacks by providing countermeasures in all our protocols as described below.

First, the DKS setup and reconfiguration mechanisms can defeat sybil attacks as follows. It is impossible for a DKS-sensor to claim multiple locations because the eligibility for DKS, as well as the underlying packet delivery, depend on "locations." That is, a

malicious DKS-sensor cannot make its (multiple) fake locations inserted into the others' routing tables during the DKS setup. Moreover, the sybil attack is not an issue for non-DKS-sensors because they should ask, for secure routing, their own DKS-sensors, each of which may act as a central entity for checking 1-to-1 correspondence between IDs and locations of sensors in its neighborhood.

Second, a malicious sensor node cannot claim arbitrary locations due mainly to the correlation among locations of neighboring sensors. That is, if the malicious device announces a new location without changing the ID, its neighbors, if more than 2/3 of them are well-behaving, would easily detect this discrepancy via cooperative location validation among themselves to blacklist/block the misbehaving sensor from the network. So, the malicious node must risk being detected if the false location is too far away from its true location because its unusual distances to its neighbors make it conspicuous in their neighbors' routing tables. Otherwise, the bogus locations wouldn't impose more threat than a compromised node which does not lie about its location. This is one of the characteristics that NIDS can exploit to tell misbehaving nodes.

Third, the malicious node may claim a new sensor node by creating the binding of ID and falsified location. In this case, however, it must go through the bootstrapping and DKS setup phases to be qualified as a legitimate sensor as well as establishing shared keys. If no new sensors are allowed after network-wide bootstrapping, the neighbors can simply ignore the new ID from the network; otherwise, we may easily capture the misbehaving device by applying a strong access control mechanism, such as PIV (in Chapter V), during initial setup.

In summary, the sybil attack takes place in a decentralized virtual network, the ID space of which is *completely* decoupled from physical network connectivity [34]. However, this is clearly not the case in our proposed protocols and environments thanks to the spatial

correlation. A systematic, distributed way of detecting invalid locations, i.e., far from the majority of others' locations in the neighborhood, will be investigated in Chapter IV for the development of attack-tolerant localization service (and an online guard mechanism against sybil attacks).

3.4.2 Attacks on DKS Setup

The key management protocols would become susceptible to masquerading and man-in-the-middle attacks if they don't properly address key authentication that establishes a cryptographic binding between the key and the communicants' IDs. However, this security risk doesn't exist in our distributed scheme because we first establish the "local" cryptographic bindings using a well-known method and then build our proposed "remote" bindings using the thus-established local bindings. This means that if the former resists the above-mentioned attacks, so does our distributed scheme. In addition, the DKS setup is safe against man-in-the-middle attacks if using the DH protocol in establishing MKs.

A malicious device can appoint itself as a DKS-sensor to intercept messages to be transcoded by itself. However, our proposed protocols successfully tolerate this threat via DKS reconfiguration (Section 3.2.5), under which all sensors in the neighborhood take turns to serve as a DKS-sensor. This means the malicious device cannot always eavesdrop messages. Moreover, as described in Section 3.2.4, the attacker owning multiple compromised slaves can only decipher a small portion of network traffic thanks to our distributed environment that uses neither hot spots nor dedicated devices. To further thwart attacks from compromised nodes disguised as DKS sensors, we may apply a soft tamper-proofing protocol (Chapter V) that does a deep inspection of the program code of the node chosen as a DKS-sensor to make sure it is genuine.

Finally, one may argue a malicious device can establish MKs with a large number of

other DKSs by replying, to the packet destined for some remote location, as though it were the correct destination. But, this belongs to the category of sybil attacks, and hence, a cooperative defense mechanism like VeIL (Chapter IV) can be applied to defeat this attack.

3.4.3 Attacks on TKEP/SGFP

To establish TK between s and d, s sends R_s through the first SGFP path, and d replies with R_d through the second SGFP path. With a collusion attack, if the two malicious nodes, m_1 and m_2 are on each of the two paths, they may share information to reconstruct TK. However, this collusion attack is quite opportunistic in that the secrecy of TK is broken only if the attacker happens to own DKSs on both paths. As analyzed in Section 3.3, this probability is very small unless he compromises a large number of sensors throughout the entire network. Moreover, the application of randomization and μ-split schemes in selecting DKSs, as well as DKS reconfiguration, make TK eavesdropping very unlikely. Therefore, TKEP "tolerates" collusion attacks by degrading its security gracefully as the number of undetected compromised sensors increases.

We do not consider DoS attacks on SGFP (precisely speaking, on GFP) assuming an external countermeasure to the DoS attacks.

3.4.4 Tolerance to Physical Attacks

Both SGFP and TKEP tolerate physical attacks very well since they are robust to compromises of individual sensors, thanks to the distributed key sharing that allows each DKS-sensor to share only a small number of MKs. By compromising/owning a sensor c, an attacker can only take over the data traffic passing through and transcoded by c. Therefore, the only way to take control of a significant portion of network traffic is to capture/compromise as many sensors as possible (without getting caught by the NIDS).

Table 3.1: P_{SGFP} and P_{TKEP_μ} vs. p_c

p_c	P_{SGFP}	P_{TKEP_1}	P_{TKEP_3}
0.001	0.0041	0.000017	1.95×10^{-14}
0.005	0.0202	0.000409	2.90×10^{-10}
0.010	0.0401	0.0016	1.75×10^{-8}
0.020	0.0791	0.0062	0.99×10^{-6}
0.050	0.1887	0.0356	1.66×10^{-4}

Moreover, there is no difference between random and 'planned' selection of victims. For instance, the adversary can capture a 'cut' through the network to monitor all traffic over the cut. But, he can decode only a small portion of the traffic that happens to have been encoded with the key he knows of. Consequently, the security will be degraded gracefully as the number of undetected compromised sensors increases. This is important as it constrains the adversary's attempts to subvert the entire network.

To quantify the tolerance/robustness of SGFP and TKEP to sensor compromises, we use the following probabilities: (1) P_{SGFP} of eavesdropping SGFP packets, and (2) P_{TKEP_μ} of breaking μ-split TKEP. Both probabilities provide a useful basis for evaluating robustness to sensor compromises in very large-scale sensor networks. Table 3.1 shows the numbers obtained from Eqs. (3.10)$\sim$(3.12) while varying the ratio of the number of compromised sensors to the total number of sensors (fully captured in p_c) when $K = 3$ and $\alpha = 1.2$.[6] P_{SGFP} is shown to be proportional to p_c, i.e., approximately 4 times p_c. This means that the required number of sensors to be compromised will be very large, demonstrating the robustness of SGFP. P_{TKEP_μ} is shown to be almost negligible when there are

[6]Please see Section 3.3 for the definition of p_c, K and α.

a small number of compromised sensors, and to increase gradually with p_c. Moreover, when $\mu = 3$, it is very unlikely for an adversary to be able to eavesdrop even after compromising 5% (e.g., 500 out of 10,000) of sensors, indicating TKEP's robustness to sensor compromises.

3.5 Performance Evaluation

Using simulation, we evaluate the performance of proposed protocols in terms of the overhead and energy consumption. We first quantify the initial DKS setup overhead, then compare the energy consumption of TKEP and the DH key setup protocol, and finally evaluate the security/energy tradeoffs.

3.5.1 Simulation Environment

Although *ns-2* is widely used to simulate network protocols, it cannot be used to evaluate our proposed protocols for the following reasons. First, the ns-2 simulation is limited to network sizes in the order of a couple of hundreds of sensors [90], and hence, it is very difficult, albeit not impossible, to simulate very large-scale networks. Since SGFP and TKEP are tailored to very large-scale networks of thousands to millions of sensors, *ns-2* is not suitable for the evaluation of these protocols. Second, we do not need a detailed simulation of link-layer behavior, packet losses, sensor dynamics, and the effects of energy depletion, because we are only interested in network-layer behaviors. We, therefore, developed a customized simulator with a simple radio transmission model: at any time, each sensor can directly communicate with all the sensors within its transmission range, and the packet delivery to neighbors is instantaneous and error-free. It is reasonable to use this simplified model as sensors are stable and stationary, and hence, the neighbors of each sensor do not vary with time [90].

Our simulation environment is based on a network of 10,000 sensors, placed in a square area of 200×200 [m^2] and electing up to level-3 DKSs ($K = 3$). Each sensor has a radio transmission range of radius $5 \sim 6$ [m].[7] The location (estimate) of each sensor is generated randomly within the network coverage area.[8] That is, we do not simulate the localization service because it is not our intended contribution in this chapter. The GFP is implemented/simulated as follows: either the source or the relaying sensor determines its next-hop sensor as the one closest in the direction towards the destination. The distance to a neighbor is not considered, as the selection of next-hop based on specific distance-based policies (e.g., either minimum- or maximum-distance policies) has its own merits and demerits. We, therefore, only use the direction-based policy, assuming that each sensor can adjust the transmission power according to the distance to the next-hop sensor so as to minimize interferences.

3.5.2 Overhead of DKS Setup

We measured the total number of packets generated and sent/relayed during the DKS setup (counting each hop as a distinct packet) while varying the size of level-0 square and the transmission range. We also counted the number of DKS-sensors chosen during this setup; it elected 6.8$\sim$9.7% of sensors as DKS-sensors depending on the transmission range (i.e., the larger the transmission range, the smaller the number of DKS-sensors). We then divided these values by the total number of sensors to compute the average number of packets relayed per sensor. This average behavior is important because all sensors take turns to serve as DKS-sensors throughout their lifetime. Figure 3.6 plots the results: each sensor received/relayed less than 20 packets during the DKS setup. Moreover, when K is

[7] Note that this choice is just for the purpose of simulation. The realistic values for transmission range and network coverage area would depend on other factors, such as the accuracy of localization service.

[8] We assign sensors' locations according to the 2-D uniform distribution used in Section 3.3. The uniform distribution may not represent all possible deployment scenarios, but still captures irregularities caused by (local) holes or clusters of sensors.

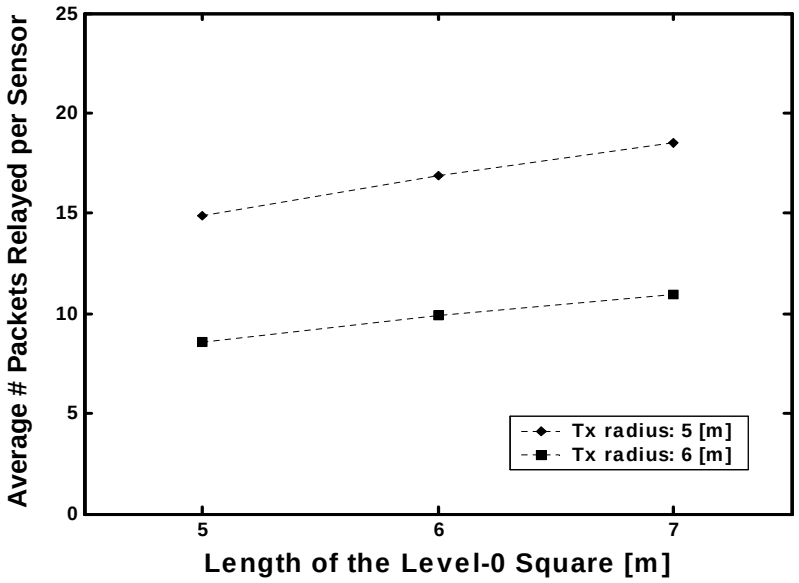


Figure 3.6: The average number of packets relayed per sensor

set to 2, this overhead gets even smaller. This initial DKS setup overhead is reasonable (and low) considering the localization overhead that takes place prior to the DKS setup.

3.5.3 Energy Consumption

Experimental results [19, 68] have shown public-key algorithms to consume a significant amount of energy. To compare the energy consumption of TKEP and DH protocols,[9] we use the measurement results of [68] for the energy costs of relevant ciphers (summarized in Table 3.2).

As derived in Section 3.3, the average number of transcoding attempts per TK setup for both basic and μ-split TKEP are $2(K + \alpha + p_n - \frac{1}{8})$ and $2\mu(K + \alpha + p_n + \frac{7}{8})$, respectively. Using the fact that each transcoding requires two TinySec encryptions and MAC computations, we can compute the energy consumption of TKEP given the protocol parameters such as μ, K, α and p_n. When $K = 3$, $\alpha = 1.25$, and $p_n = 0.9$, Table 3.3

[9] We compare TKEP against the DH protocol because, to our best knowledge, it is the only secure key-setup protocol currently available on sensor devices.

Table 3.2: The energy costs of TinySec and DH protocols

	TinySec		DH
	encryption	MAC	protocol
Energy [mJ]	0.04796	0.06677	1185

Table 3.3: Comparison of energy costs for TKEP and DH

	Energy [mJ]		TKEP/DH
μ	TKEP	DH	[%]
1	2.2603	1185	0.19
2	5.4382		0.46
3	8.1573		0.69

presents, as a function of μ, the energy costs for TKEP and the DH protocol, and the amount of energy savings by TKEP. The result shows that TKEP consumes energy far less than 1% of the DH protocol, confirming its high energy-efficiency.

3.5.4 Security/Energy Tradeoffs

Tables 3.1 and 3.3 demonstrate how TKEP can make a tradeoff between security and energy consumption of cryptographic operations. Figure 3.7 plots the probability, P_{TKEP_μ}, of eavesdropping TKEP as a function of energy consumption in [mJ], while varying the percentage of compromised sensors. The result confirms P_{TKEP_μ} to be inversely propor-

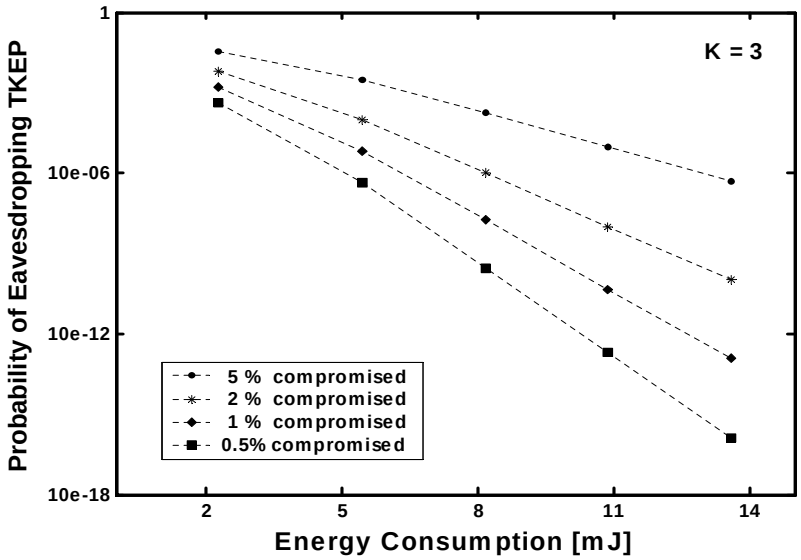


Figure 3.7: P_{TKEP_μ} vs. energy consumption

tional to the energy consumption. Thus, TKEP is very "flexible" in that any sensor, either as source or destination, can reconfigure TKEP according to its residual energy.

We also measured the number of transcodings for both SGFP and GFP to evaluate SGFP's capability to withstand compromised sensors. GFP incurred 24.7$\sim$27.8 transcodings (hops) per path, while SGFP required about 5.5 transcodings per path. That is, the risk of compromised sensors to SGFP is just about one fifth of the hop-by-hop transcoding scheme.

3.6 Conclusion

In this chapter, we proposed two protocols for secure routing—a secure geographic forwarding protocol (SGFP) and a temporal-key establishment protocol (TKEP)—as cost-effective security solutions for large-scale sensor networks. The distributed key sharing played a crucial role in our proposed protocols: by having a sensor share keys only with a small number of other sensors chosen based on their geographic location, we successfully (1) constructed a lightweight, secure network layer and (2) replaced the resource-expensive

Diffie-Hellman key-setup protocol with a purely symmetric-cipher-based (hence energy-efficient) alternative.

Our security analysis and performance evaluation have shown that the distributed key sharing is practically useful and effective in defeating and/or tolerating many critical attacks, such as sybil, physical, man-in-the-middle and collusion attacks, while incurring (consuming) only a small amount of overhead (energy) in the packet forwarding and key setup. This, in turn, enabled the realization of lightweight, secure routing protocols at the expense of initial setup overhead in constructing DKS relationships.

CHAPTER IV

ATTACK-TOLERANT LOCALIZATION

4.1 Introduction

Various localization schemes [16, 29, 43, 49, 53, 76, 83, 88, 100] have been developed for sensors to determine, with reasonable accuracy, their relative locations within the network coverage area. All these schemes employ location-information-equipped *anchors* that provide reference locations, and sensor nodes determine their relative locations with respect to the anchors' reference locations. Accordingly, they can successfully accomplish their application/mission only if all participants are benign and strictly follow the localization protocol.

However, sensor networks are usually deployed in a hostile, unattended, and untrusted environment, and hence, they face various critical security attacks from (malicious) compromised nodes. Specifically, an adversary may attempt to fail the localization service by advertising false locations, causing errors in distance measurements, or introducing bogus anchors. Despite its importance, the problem of determining sensor nodes' locations in the presence of attacks has not yet been addressed effectively. Existing solutions either rely on traditional authentication mechanisms [49] or simply use anchor-provided information [61, 64, 65], but they fail to completely safeguard the location service due mainly to the non-cryptographic nature of attacks, the requirement of unrealistically-powerful anchors,

and/or ignorance of the relationship/correlation among sensors' locations, as described in Section 1.4.3.

In this chapter, we address the problem of "attack-tolerance" in the design of a localization protocol. We consider a *large-scale* sensor network equipped with only a very small number of less-capable anchors than assumed in the existing schemes, which makes the problem more realistic but challenging. We take an approach to building an attack-tolerant localization protocol, called *Verification for Iterative Localization* (VeIL). This approach is motivated by the fact that a sensor network inherently relies on collective assurance among multiple low-cost sensors to execute high-precision missions, where attack-tolerance is one of such missions.

The heart of VeIL is the use of *spatio-temporal correlation* among adjacent nodes in the development of anomaly-based attack detection. In essence, VeIL is a cooperative intrusion detection system tailored to localization, and consists of

- a profile manager that captures and adaptively tracks the profile of normal localization behavior; and
- an attack detector that detects and locates attacks by iteratively verifying location announcements via their comparison against the profiled normal profile.

These two building blocks together achieve a high-level tolerance to attacks by rejecting any information that exhibits a noticeable deviation from the normal profile, thereby forcing the attacker to weaken the attack strength so as not to be caught, which, in turn, makes it very unlikely for the attacker to fail the localization service. Moreover, our security analysis and simulation results demonstrate the effectiveness and robustness of VeIL that defeats many critical attacks while incurring the processing overhead amenable to resource-poor sensor nodes, and no additional communication overhead.

The rest of the chapter is organized as follows. Section 4.2 describes the proposed protocol. Section 4.3 analyzes the security of the proposed protocol while Section 4.4 evaluates its performance via simulation. Finally, the chapter concludes with Section 4.5.

4.2 The Proposed Protocol

We propose an attack-tolerant localization protocol, called *Verification for Iterative Localization* (VeIL), that

1. tolerates attacks (and faults) by malicious (misbehaving) devices,
2. incurs a small processing overhead without any communication overhead,
3. preserves compatibility with other services like the authentication framework, and
4. achieves high localization accuracy and efficiency.

In what follows, we describe the network and threat models, the proposed countermeasures, the underlying localization algorithm, and the details of VeIL.

4.2.1 The Network Model

The sensor network under consideration consists of *sparsely-deployed*, less-capable, static anchor nodes and a large number of sensor devices. This is a realistic deployment scenario in that the sensor network is inherently an infrastructure-less network in which sensors autonomously organize themselves into a connected structure, and hence, it is desirable to minimize the dependency of localization on infrastructure nodes, such as anchors. In this environment, it is not uncommon that a sensor cannot directly hear from any of the anchors, necessitating collective assurances among sensors to reach network-wide consistent location assignments.

We choose the MDS-based iterative localization algorithm [29, 53] as our underlying localization scheme, in which each and every sensor keeps refining its location estimates based on location announcements from, and distance measurements with, its direct neighbors. To this end, two nodes within each other's transmission range establish a *mutual* neighborhood relationship. Any of the ranging techniques (RSS, TOA, TDOA, and AOA) described earlier can be used to estimate distances between *direct* neighbors. Sensors may optionally use location verification protocols [96, 117] to ensure their neighbors are really within their communication range.

4.2.2 The Threat Model

Anchors are assumed to be *trusted* entities, i.e., the reference locations provided by them are trustworthy and cannot be spoofed by the adversary.[1] Accordingly, each sensor can authenticate the reference locations to confirm that they are indeed from genuine anchors. By contrast, sensors can be physically compromised or tampered with by the adversary at any time. Therefore, the localization takes place in the presence of malicious/compromised devices (the number of which is less than one-third of that of entire network nodes) disguised as normal participants who will do their best to disrupt the localization service by mounting the attacks described in Section 1.3.

4.2.3 The Proposed Approach

To maximize both attack-tolerance and localization-accuracy, we exploit the *spatio-temporal correlation* among neighboring nodes' locations to determine if a malicious node claims/announces false locations. Basically, the adversary must be *aggressive* enough to disrupt the localization service. However, if a malicious device advertises arbitrary locations that deviate significantly from what the protocol expects, its neighbors, if a majority[2]

[1] The effects of malicious anchors will be discussed in Section 4.3.

[2] simple or two-thirds majority in case of Byzantine faults

of them are well-behaving, would easily detect the discrepancy via cooperative location validation to blacklist/block the misbehaving sensor from participating in the localization service. So, the malicious sensor must risk getting caught if its falsification deviates too much from its normal location because its unusual distances to its neighbors make it conspicuous during the localization process. On the other hand, the false locations with small perturbations wouldn't do any harm, since the effects of small perturbations would easily be canceled out.

We take an *anomaly detection* approach [73, 130] to realize this idea. That is, each sensor maintains, and adaptively updates, a baseline profile of the normal localization behavior based on past announcements of all its neighbors. Then, upon reception of new announcements, it compares them with this normal profile, and if a noticeable deviation is found, decides on the presence of a possibly adversarial behavior and takes an action to locate the malicious or misbehaving sensor. Consequently, VeIL consists of the following two building blocks that closely interact with each other:

- **a profile manager** (described in Section 4.2.5) that constructs and maintains the compact profile of normal localization behavior; and
- **an attack detector** (described in Section 4.2.6) that detects, locates, and rejects false location announcements, as well as updates the normal profile.

VeIL is essentially a *cooperative* intrusion detection mechanism tailored to localization, in which each and every sensor checks if the location/distance announcement from each of its neighbors is "abnormal," and, if so, removes the sensor from the rest of the localization process. Unlike the other schemes that rely solely on anchors, VeIL uses peer sensors as active information sources that provide distances information and incremental location updates, thereby maximizing the attack-detection capability.

4.2.4 The Underlying Localization Algorithm

Let the index s refer to the sensor performing localization, and n_s denote the number of s's direct (one-hop) neighbors. The (local) indices, $i = 1, \cdots, n_s$, are assigned to s's neighbors, each of which may or may not be an anchor node. There exists 1-to-1 correspondence between the index and ID of a sensor. The distance estimate between s and i is denoted by $\delta_{s,i}$. Also, let $w_{s,i}$ denote a weight assigned between s and i, the value of which is either binary (1 if $\delta_{s,i}$ is known; 0 otherwise) [53] or adaptively chosen [28]. We define the *individual cost* between s and i in the k^{th} iteration as

$$c_{s,i}(k) = w_{s,i} \left[\delta_{s,i} - \|\mathbf{x}_s(k) - \mathbf{x}_i(k)\| \right]^2 \tag{4.1}$$

where $\mathbf{x}_s(k)$ and $\mathbf{x}_i(k)$ are $p \times 1$ coordinate vectors (p = 2 or 3 for two- or three-dimensional coordinates, respectively) representing estimated locations of s and i at iteration k (≥ 0), respectively, and $\|\mathbf{x}\|$ denotes the Euclidean norm of the vector $\mathbf{x}$. Then, the *local cost* of sensor s, $c_{s,0}(k)$, at iteration k is computed by summing up all individual costs:

$$c_{s,0}(k) = \sum_{i=1}^{n_s} c_{s,i}(k). \tag{4.2}$$

The localization algorithm searches s's true location by iteratively minimizing $c_{s,0}(k)$. The entire localization process works as follows.

- Initially, all sensors in the network randomly choose their initial location estimates while the anchors use their own fixed reference locations, i.e., $\mathbf{x}_s(0)$ and $\mathbf{x}_i(0)$ are the initial locations of s and i.

- At iteration k (≥ 0), s refines its location estimate, $\mathbf{x}_s(k+1)$, by processing $\{\delta_{s,i}, \mathbf{x}_i(k)\}_{i=1}^{n_s}$ with the update formula in [16, 28, 53], then exchanges the new locations with all its neighbors.

- s terminates the algorithm if the location estimate gets stabilized (i.e., $c_{s,0}(k) - c_{s,0}(k+1) < \epsilon$); otherwise, repeat the process at the next iteration.

The communication overhead incurred by exchanging location estimates is small because each sensor may simply broadcast this information as part of the beaconing process (that periodically exchanges BEACON packets to refresh sensors' neighbor-lists). Also, note that beaconing is one of the basic operations a sensor must execute throughout its lifetime. Therefore, s may keep on fine-tuning its location estimate after terminating the above algorithm based on the BEACON packets exchanged. Our proposed protocol can be used to verify BEACON packets, thus serving as an online guard mechanism against attacks targeting at sensors' locations, such as sybil attacks.

As mentioned in Section 4.2.1, we use existing iterative localization algorithms as the underlying localization layer. Our contribution/goal is to reinforce this localization layer with attack-tolerance by providing two mechanisms: (1) construction/management of the normal profile based on intermediate results of the iterative localization, and (2) detection/identification of attacks. As long as the underlying localization layer ensures convergence in an attack-free environment, the entire localization process will converge in the presence of attacks because VeIL will detect and reject any falsified information as early as possible, thus feeding the localization layer with attack-free information only.

4.2.5 Construction of Normal Profiles

We want to construct a profile of s for the normal localization behavior, based solely on the information collected by s during the localization. That is, in the k^{th} iteration, s has been processing $\{ \mathbf{x}_i(t) \}_{t=1}^{k}$ using Eq. (4.1) to compute $k \times 1$ vectors,

$$\mathbf{c}_{s,i}(k) = [\, c_{s,i}(k), \cdots, c_{s,i}(1) \,]^T, \;\; 1 \leq i \leq n_s. \tag{4.3}$$

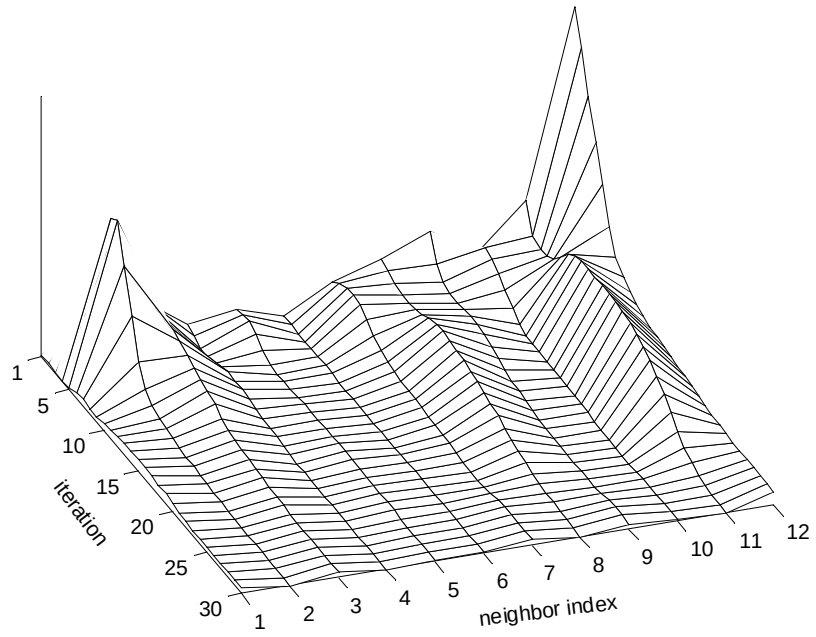


Figure 4.1: Example plot of $\mathbf{c}_{s,i}(k)$ for $1 \leq i \leq 12$ and $k = 30$

Then, a $k \times 1$ local cost vector of s, $\mathbf{c}_{s,0}(k)$, is given by

$$\mathbf{c}_{s,0}(k) = [\, c_{s,0}(k), \cdots, c_{s,0}(1) \,]^T = \sum_{i=1}^{n_s} \mathbf{c}_{s,i}(k) \tag{4.4}$$

Figure 4.1 plots typical $\mathbf{c}_{s,i}(k)$ values when $i = 1, \cdots, 12$ and $k = 30$. From this figure, we observe the following two facts:

- each $\mathbf{c}_{s,i}(k)$ exhibits strong temporal correlation; and
- $\mathbf{c}_{s,0}(k)$ is strictly decreasing as the iteration progresses, due to the spatial correlation among neighbors.

We can, therefore, derive as compact a description of the normal profile as possible by removing this redundancy. Below, we describe how we exploit this property *optimally* (in the sense of achieving the highest attack-resolution, i.e., if any other scheme can resolve an attack, so can VeIL).

Problem Formulation

Our problem is cast into the design of an *adaptive* transversal filter bank that consists of n_s filters, each with M taps. We first formulate a *least squares prediction* problem as follows:

$$\begin{aligned} \hat{c}_{s,1}(t) &= \mathbf{h}_{s,1}^T(k)\,\mathbf{c}_{s,1}(t-1;M) \\ &\dots \qquad\qquad , \; M < t \le k \\ \hat{c}_{s,n_s}(t) &= \mathbf{h}_{s,n_s}^T(k)\,\mathbf{c}_{s,n_s}(t-1;M) \end{aligned} \tag{4.5}$$

where $k \ge M+1$ and $\mathbf{h}_{s,i}(k)$ is the $M \times 1$ filter-weight vector for neighbor i at iteration k defined by

$$\mathbf{h}_{s,i}(k) = [\, h_{s,i1}(k), \cdots, h_{s,iM}(k) \,]^T \tag{4.6}$$

and $\mathbf{c}_{s,i}(t-1;M)$ is the $M \times 1$ past individual cost vector for i given by

$$\mathbf{c}_{s,i}(t-1;M) = [\, c_{s,i}(t-1), \cdots, c_{s,i}(t-M) \,]^T. \tag{4.7}$$

Our objective is to find estimators $\{\, \hat{\mathbf{h}}_{s,i}(k) \}_{i=1}^{n_s}$, each of which minimizes the sum of squared errors (SSE):

$$SSE_{s,i}(k) = \sum_{t=M+1}^{k} \lambda^{k-t} |\, c_{s,i}(t) - \mathbf{h}_{s,i}^T(k)\mathbf{c}_{s,i}(t-1;M) \,|^2 \tag{4.8}$$

where λ (≤ 1) is an exponential forgetting factor.

Recursive Least Squares Algorithm

We apply the method of recursive least squares (RLS) [42] to develop a recursive algorithm that updates the filter-weight vectors $\{\, \hat{\mathbf{h}}_{s,i}(k) \}_{i=1}^{n_s}$ upon reception of $\{\, \mathbf{x}_i(k) \}_{i=1}^{n_s}$ (translated into $\{\, c_{s,i}(k) \}_{i=1}^{n_s}$), given $\{\, \hat{\mathbf{h}}_{s,i}(k-1) \}_{i=1}^{n_s}$, where $k \ge M+1$. The RLS algorithm first calculates, for each i, *a priori* prediction error based on old filter-weight estimates at iteration k, as follows:

$$\alpha_{s,i}(k) = c_{s,i}(k) - \hat{\mathbf{h}}_{s,i}^T(k-1)\mathbf{c}_{s,i}(k-1;M). \tag{4.9}$$

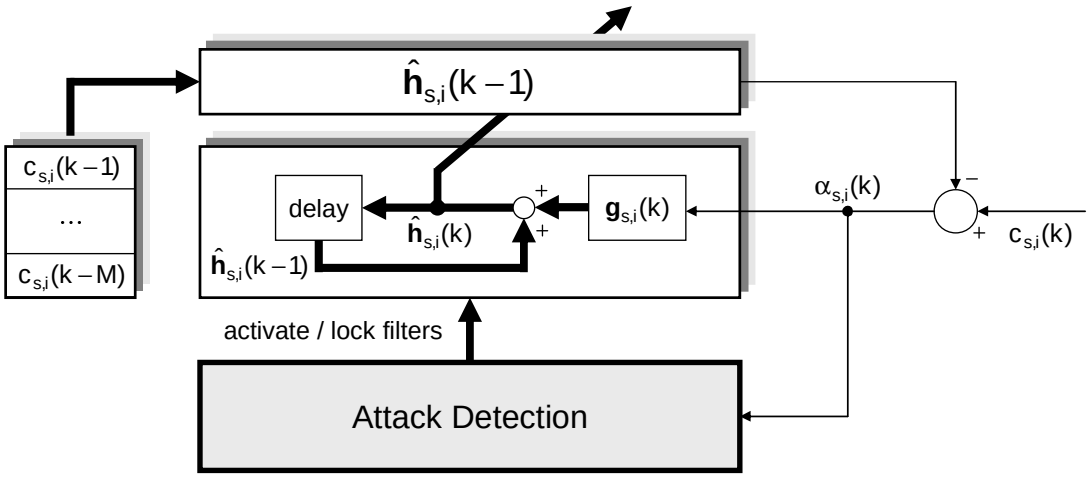


Figure 4.2: The VeIL architecture

The filter-weight vector is then updated as

$$\hat{\mathbf{h}}_{s,i}(k) = \hat{\mathbf{h}}_{s,i}(k-1) + \alpha_{s,i}(k)\,\mathbf{g}_{s,i}(k) \tag{4.10}$$

where $\hat{\mathbf{h}}_{s,i}(M) = \mathbf{0}$ and an $M \times 1$ gain vector $\mathbf{g}_{s,i}(k)$ is computed by

$$\mathbf{g}_{s,i}(k) = \frac{\mathbf{P}_{s,i}(k-1)\mathbf{c}_{s,i}(k-1;M)}{\lambda + \mathbf{c}^T_{s,i}(k-1;M)\mathbf{P}_{s,i}(k-1)\mathbf{c}_{s,i}(k-1;M)} \tag{4.11}$$

$\mathbf{P}_{s,i}(k)$ is an $M \times M$ inverse correlation matrix, initialized to

$$\mathbf{P}_{s,i}(M) = \rho^{-1}\mathbf{I} \tag{4.12}$$

with a small positive ρ, and recursively updated by

$$\mathbf{P}_{s,i}(k) \;= \lambda^{-1}\mathbf{P}_{s,i}(k-1) - \lambda^{-1}\mathbf{g}_{s,i}(k)\mathbf{c}^T_{s,i}(k-1;M)\,\mathbf{P}_{s,i}(k-1). \tag{4.13}$$

For details of the RLS algorithm, see [42].

Discussion

With the above RLS algorithm, the *entire* localization history of s, up to the k^{th} iteration, is fully captured in $n_s \times M$ filter-weights, $\{\, h_{s,ij}(k) \,|\, 1 \le i \le n_s, 1 \le j \le M \,\}$,

that constitute a normal profile at iteration k. This normal profile must be built only from *attack-free* data, and hence, $\{\hat{\mathbf{h}}_{s,i}(k)\}_{i=1}^{n_s}$ will be updated after making sure $\{\mathbf{x}_i(k)\}_{i=1}^{n_s}$ not to be adversarial according to the procedure described in Section 4.2.6.

As shown in Figure 4.2, the profile manager of s includes n_s RLS filters, each storing M past individual costs, $\mathbf{c}_{s,i}(k-1;M)$, and M filter-weights, $\hat{\mathbf{h}}_{s,i}(k)$, that are adaptively and recursively updated using Eqs. (4.9) – (4.13) starting at $k = M+1$. Possible attacks during the first M iterations are not an issue, as the localization starts with high individual costs even in an attack-free environment due to the randomly-assigned initial locations, effectively paying no attention to identification of attacks during the initial period. That is, the malicious neighbors will soon get caught if they keep misbehaving after this initial period;[3] they wouldn't otherwise be able to confuse/disrupt the localization process.

The computational requirement at each iteration is $\mathcal{O}(n_s \cdot M^2)$. This overhead is reasonable even for resource-constrained sensors, since it suffices to choose a small (like 3 – 5) value of M, thanks to the high temporal correlation. To determine an optimal value of M, one may use either the information-theoretic criterion (AIC) or the minimum description length (MDL) criterion [42].

Finally, it is possible to use the Kalman filter instead of RLS, in the sense that the above RLS algorithm (which is a tailored solution to our prediction problem) is indeed a special case (i.e., sharing the same mathematical structure) of the Kalman filter. The application of Kalman filtering to our prediction problem given by Eq. (4.5) will yield Eqs. (4.9) – (4.13) to be replaced with the Kalman filter recursions. The computational complexity in using this general-purpose filter is acceptable for small M values.

[3] This is because a majority of neighbors are adjusting their locations towards their true locations, thus steadily reducing their individual costs, while the individual cost of a malicious neighbor remains high due mainly to the discrepancy of locations between itself and others.

4.2.6 Detection of Attacks

A malicious sensor i may attempt to have a falsified $\mathbf{x}_i(k)$ accepted by s so that $c_{s,i}(k)$ can be boosted to a large value. This will cause $\mathbf{x}_s(k+1)$ to have a large deviation from its desired value, making it impossible, or at least take a very long time, for s to determine its true location. We describe below how VeIL defends a network against this threat, and then qualitatively analyze its attack-detection capability.

Proposed Detection Scheme

Every sensor must verify the trustworthiness of incremental location updates, $\{\, \mathbf{x}_i(k)\}_{i=1}^{n_s}$, from its neighbors by comparing them with the normal profile built in the $(k-1)^{\text{st}}$ iteration. VeIL achieves this easily by evaluating Eq. (4.9) for $i = 1, \cdots, n_s$. Clearly, $\alpha_{s,i}(k)$, $1 \leq i \leq n_s$, quantifies the difference of i's new announcement from its value predicted from the most-recent profile, and hence, s should suspect i to be malicious if $\alpha_{s,i}(k)$ exceeds a certain threshold. Accordingly, for each i, s decides $\mathbf{x}_i(k)$ to be disruptive/harmful to the location service if

$$|\alpha_{s,i}(k)| \geq \eta_t \cdot \max\{\, c_{s,i}(k),\, c_{\min}\,\} \tag{4.14}$$

where η_t (≤ 1) is a pre-configured network-wide threshold for detecting anomalies, and $c_{\min}$ is the minimum individual cost, below which the prediction error becomes negligible because $\|\mathbf{x}_s(k) - \mathbf{x}_i(k)\|$ falls well within $\delta_{s,i}$. Moreover, $c_{s,0}(k)$ must decrease with the iteration count. Hence, every sensor should monitor if this condition is violated, i.e., in the k^{th} iteration, s checks if

$$c_{s,0}(k) \geq \eta_0(k) \cdot c_{s,0}(k-1) \tag{4.15}$$

where $\eta_0(k)$ (≤ 1) is a threshold to verify the acceptability of local cost whose value is determined by the choice of the underlying localization algorithm. Eq. (4.15) will be met if $c_{s,0}(k)$ is dominated by a few anomalous individual costs (possibly from malicious nodes).

Using Eqs. (4.14) and (4.15), s can detect, for each iteration, if there exist anomalies in its neighbors' location announcements, and if so, can identify which of the neighbors caused the anomalies. s then blacklists a neighbor if it has been caught more than N_B times out of B iterations. The ratio N_B/B (≤ 1) is a design parameter; the choice of N_B/B close to 0 implies an overly-conservative mode of operation, while the opposite ($N_B/B \rightarrow 1$) means lenience against attacks. For the purpose of blacklisting, s keeps track of the number of anomalous location announcements from i in blacklist_counter(i). Neighbor sensors may also exchange their detection results to speed up blacklisting i.

Figure 4.2 shows how the VeIL's attack detector interacts with its profile manager. First, both the profile manager and the attack detector are triggered by $\alpha_{s,i}(k)$'s computed at iteration k. The attack detector then processes $\alpha_{s,i}(k)$'s using Eqs. (4.14) and (4.15) to determine if there exist anomalies in the location announcements, and if so, identifies which of the neighbors caused the anomalies. Finally, this information is fed to the profile manager to reject (announcements from) those neighbors.

Attack-Detection Capability

VeIL must be able to *amplify* the prediction errors caused by false location announcements for M consecutive iterations, thus detecting anomalies with high accuracy (each of which presents multiple chances to be caught). In other words, sensor i's multiple false announcements within M consecutive iterations will cause VeIL to continuously produce high $\alpha_{s,i}(\cdot)$ values, making it highly likely to detect the misbehaving sensor i. On the other hand, sporadic attacks will be detected by the N_B/B scheme mentioned earlier. Weakening the attack frequency further than this imposes no threat to the localization service.

Let us consider the case where a malicious sensor i mounted an attack to force s to compute $c_{s,i}(k)$ that differs from the expected cost $c^*_{s,i}(k)$ by Δ, i.e., $c_{s,i}(k) = c^*_{s,i}(k) + \Delta$.

This will increase $\alpha_{s,i}(k)$ by Δ, because $\alpha_{s,i}(k) = \alpha^*_{s,i}(k) + \Delta$ from Eq. (4.9). Hence, the deviation of the prediction error at iteration k is proportional to the attack strength Δ. By contrast, the prediction error gets amplified rapidly at iteration $k+1$ for the following reason. From Eq. (4.10), the filter-weights are updated as $\hat{\mathbf{h}}_{s,i}(k) = \hat{\mathbf{h}}^*_{s,i}(k) + \Delta \cdot \mathbf{g}_{s,i}(k)$. Then, $\alpha_{s,i}(k+1)$ can be rewritten as a function of Δ:

$$\alpha_{s,i}(k+1) = \alpha^*_{s,i}(k+1) - \Delta^2 \cdot g_{s,i1}(k) - \Delta \cdot \left[\hat{h}_{s,i1}(k) + \mathbf{g}^T_{s,i}(k)\mathbf{c}_{s,i}(k;M)\right] \quad (4.16)$$

where $g_{s,i1}(k)$ is the first element of $\mathbf{g}_{s,i}(k)$. Therefore, the perturbation Δ introduced at iteration k results in a very large amount of prediction error in the next iteration. Moreover, this magnification of prediction errors will persist for the period of M consecutive iterations, during which the false cost $c_{s,i}(k)$ remains cached inside the profile manager, thus making it very difficult for the attacker to evade VeIL. This qualitatively illustrates the highest level of VeIL's attack-detection capability.

4.2.7 Protocol Description

Figure 4.3 provides the pseudocode of our proposed localization protocol executed at sensor s. (For simplicity, B is set to be unbounded in the pseudocode.) It integrates the operations for the attack detector, the profile manager, and the underlying localization algorithm. The localization at sensor s starts by initializing the profile manager and randomly choosing $\mathbf{x}_s(0)$ followed by announcement of the initial location to its neighbors. Then, s executes up to M iterations with VeIL disabled, activates VeIL at iteration $M+1$, and finally, performs the rest of localization until its convergence.

There are two response mechanisms to false location announcements. First, the false locations (at iteration k) must be excluded from (i) the computation of s's next location estimate $\mathbf{x}_s(k+1)$, and (ii) the recursion for the profile manager by letting $c_{s,i}(k) = \hat{\mathbf{h}}^T_{s,i}(k-1)\mathbf{c}_{s,i}(k-1;M)$. Second, malicious neighbor i must be removed from the future

Initialization: $\hat{\mathbf{h}}_{s,i}(M) = \mathbf{0}$;
$\mathbf{P}_{s,i}(M) = \rho^{-1}\mathbf{I}$;
randomly choose and announce $\mathbf{x}_s(0)$;

Iteration:

// *Execute localization until its convergence*
for ($k = 0$; $c_{s,0}(k-1) - c_{s,0}(k) < \epsilon$; k++)

receive $\mathbf{x}_i(k)$ from all i;
compute $c_{s,i}(k)$, $\forall i$, and $c_{s,0}(k)$ using Eqs. (4.1) and (4.2);

// *Activate VeIL at iteration* $M+1$
if $k \geq M+1$,
for $i = 1$ **to** n_s,
compute $\alpha_{s,i}(k)$ using Eq. (4.9);
if $c_{s,0}(k) \geq \eta_0(k) \cdot c_{s,0}(k-1)$ **and**
$|\alpha_{s,i}(k)| \geq \eta_t \cdot \max\{c_{s,i}(k), c_{\min}\}$,
$c_{s,i}(k) = \hat{\mathbf{h}}_{s,i}^T(k-1)\,\mathbf{c}_{s,i}(k-1;M)$;
if ++ blacklist_counter(i) $\geq N_B$,
blacklist i;
if i blacklisted,
deactivate $\hat{\mathbf{h}}_{s,i}(k)$;
else,
update $\hat{\mathbf{h}}_{s,i}(k)$, $\mathbf{g}_{s,i}(k)$ and $\mathbf{P}_{s,i}(k)$;

// *Execute the location-update algorithm*
update and announce $\mathbf{x}_s(k+1)$;

Figure 4.3: Pseudocode for VeIL at sensor s

localization process (e.g., by deactivating $\hat{\mathbf{h}}_{s,i}(k)$) if it had been caught more than N_B times. To do this, the blacklist_counter(i) is incremented whenever i's announcement is found to be "false."

4.3 Security Analysis

We classified the localization-specific attacks in Section 1.3 into three types: (1) location-targeted attacks, (2) distance-targeted attacks, and (3) anchor-targeted attacks. We now describe how VeIL counters each of these attacks.

4.3.1 Defense against Location-Targeted Attacks

The attacker may influence, and cause a significant bias in, the localization process by providing false location information. This threat can be caused by (i) physical attacks that compromise sensors and then deploy them, and (ii) wormhole attacks that create hidden links between malicious devices, both of which will then be used to replay/modify/create messages that carry location information. Traditional authentication-based countermeasures will likely fail since it is difficult to keep cryptographic keys secret under these attacks. Likewise, any protocol (including the one based on hop-counts) that uses sensors as relays, is vulnerable to these attacks.

By contrast, VeIL is very robust to these location-targeted attacks as it hinges on highly-correlated localization behaviors of sensors that update location estimates toward their true locations. Since every sensor keeps refining its location estimate such that the aggregate differences in location information received from its neighbors fit better with measured distances, the next location estimate of a sensor can be predicted, for the most part, from its past localization history. Actually, VeIL makes "optimal" prediction using the least squares formulation and the RLS method. Under this prediction framework, any location announcements that deviate significantly from the corresponding prediction are

highly likely from attackers.

An attacker may attempt to break VeIL in several ways. First, he may judiciously adjust the strength of perturbation to make the cost just below the detection threshold η_t. Since the location differences converge to the distance measurements during the localization, such an attempt will place great stress on the attacker into steadily lowering the attack strength not to be caught by VeIL. However, by doing so, the attacker cannot succeed in his mission to fail the localization process. Second, the attacker may try to inject false location information from the beginning, to "train" the profile manager to always predict high individual costs. However, such an act will soon be caught because the individual cost from the malicious node will become conspicuous among the costs of all the neighbors. Moreover, it is extremely difficult for the attacker to fool all the neighbors without physically compromising them. Third, the attacker may mount sybil attacks. But sybil attacks are ineffective under VeIL, because fictitious locations can only take values agreeing with the corresponding distance measurements to evade VeIL, forcing all the falsified locations to be close to one another. As a result, the only effective way to evade VeIL "locally" is to compromise at least one-third of the sensors within the local region. This proves the highest attack-tolerance of VeIL on location-targeted attacks.

4.3.2 Defense against Distance-Targeted Attacks

The distance measurements/estimates can be altered by distance enlargement/reduction attacks (e.g., via jamming, physical obstacles and the transmission power control), wormhole attacks on distance or hop-count information, etc. In general, any attempt by a malicious sensor i to modify the distance to its neighbor s makes only a one n_s-th contribution to s's location estimate, since s determines its location with respect to the locations/distances of all n_s neighbors. s's decision is then fed back to the others, canceling

the errors from i. So, the impact of the distance-targeted attacks of a single malicious node on sensors in its proximity is small, and smaller on farther-away sensors. Thus, in spite of its possible undesirable local effects, VeIL will not distort the network-wide connectivity, making it robust to distance-targeted attacks.

Specifically, jamming a local area (or placing obstacles) blocks sensors inside (or nearby) the area from participating in the localization. But, the unaffected sensors can still maintain consistent network connectivity among themselves, although their location estimates may differ from real locations (i.e., the closer to the jammed area, the larger the deviation). Moreover, as soon as the jamming is over, the sensors will start adjusting their locations in the course of neighborhood management via BEACON packets. Note that VeIL also plays a role of verifying BEACON packets. Besides, a malicious sensor i may amplify its transmission power in order to proliferate bogus information as well as to cause smaller-distance measurements, but i is more likely to be caught by VeIL because in such a case there will be more neighbors watching on it.

4.3.3 Defense against Anchor-Targeted Attacks

This type of attacks can be considered as a special case of location-targeted attacks in that malicious/bogus anchors will provide false information on location references. Typically, the effects of location-targeted attacks on anchors become much more serious than those on non-anchor nodes because they will eventually corrupt the entire network. For this reason, anchors are assumed to be trusted entities. However, in reality, anchors can be compromised (by determined attackers) no matter how well protected they are, and, if that happens, it is impossible for sensors to determine their true locations. Also, compromising all (or two-thirds of) the neighbors of an anchor would have the same effect as compromising the anchor. VeIL is capable of gracefully resisting this type of attacks as explained

below.

There exist two attack scenarios from a malicious anchor: (i) persisting the same falsified reference location, or (ii) spoofing as many false locations as possible (sybil attacks). The former applies to the network with static anchors only, while the latter applies to the network of mobile anchors. Under the first attack scenario, VeIL still maintains network-wide connectivity since the malicious anchor itself is just one of the neighbors. Therefore, VeIL can achieve 'coarse-grained' localization even when several (static) anchors are compromised. Moreover, VeIL inherently doesn't require any mobile anchor, and hence, disallows the use of mobile devices for security reasons, violation of which can be easily checked as follows. Each VeIL-enabled sensor builds a static neighbor list before starting localization (as in Section 4.2.1), and then binds the reference location with an anchor ID, if it is a neighbor, via the anchor authentication (described in Section 4.2.2). Hence, any reference location that differs from the initial, authenticated value will be rejected. As a result, the second attack scenario cannot happen in VeIL. By contrast, those protocols relying on mobile anchors become defenseless once the anchor has been compromised or successfully spoofed. Therefore, mobile anchors, if used, must have higher-level protection than immobile sensors; this may be acceptable since mobile devices (e.g., iPaqs or laptops) are usually equipped with more and better resources (e.g., faster CPUs and more powerful antennae) than static sensors (e.g., Motes). VeIL can also be configured to allow mobile anchors although the benefit of using them is marginal in VeIL.

4.4 Performance Evaluation

We evaluate the performance of VeIL using simulation. We will first describe our simulation environment and the metrics used. Then, we present our simulation results that consist of two parts: (1) quantification of the prediction error of profile management, and

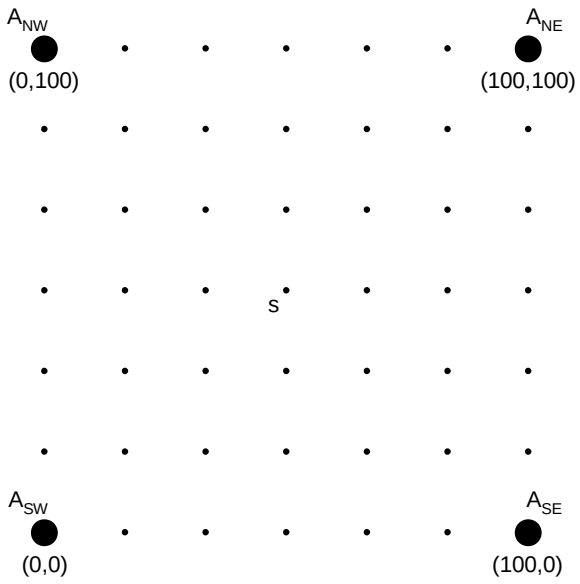


Figure 4.4: The simulation environment consisting of 4 anchors and 45 sensors

(2) evaluation of the attack/anomaly detection capability to mask the effects of malicious neighbors.

4.4.1 The Simulation Environment

As shown in Figure 4.4, our simulation environment consists of a network of 49 sensor nodes deployed on a uniform, 7×7 grid covering an area of 100×100 [m^2]. The four corner nodes are location-information-equipped anchors, transmitting their (fixed) geographic coordinates. The rest of the network nodes are normal sensors whose locations are unknown, and hence, must be determined. Sensors guess their initial locations completely randomly. Let s be the sensor at the grid center for convenience.

We use the RSS-based ranging technique because it has been widely used by real sensor platforms like Motes [33]. The radio model implemented for our simulation is the one in [84] based on real measurements that models the RSS as a log-normal-distributed random variable with its mean power decaying according to the pass loss model. This model captures the characteristics of RSS-based ranging that incurs high estimation errors

if two communicating parties are farther away from each other. To deal with these errors, we average 20 RSS measurements before deriving a distance estimate.

The maximum communication range is set to 40 [m] for both anchors and sensors, i.e., s accepts i as a neighbor if the distance estimate derived from RSS is smaller than 40 [m]. This effectively limits the number of neighbors for each anchor to be around 7 sensors, and hence, about a half of sensors cannot directly hear from any of the anchors, while the rest can directly hear from at least one anchor. Sensors determine their own list of neighbors based on RSS measurements before starting the localization process.

During the localization process, each sensor executes VeIL (Figure 4.3) that verifies, computes, and exchanges incremental location updates with its direct neighbors. Throughout the simulation, VeIL is configured with $\lambda = 0.95$ and $\rho = 0.1$. The iterative method of [28] is used to update the location estimates at each iteration, but other iterative schemes can be used as well. The choice of location-update method determines how fast it converges and how computationally-efficient it is, without affecting the detection capability of VeIL.

4.4.2 Metrics for Evaluation

We define and use the normalized prediction error (NPE) of s at iteration k as

$$NPE_s(k) = \frac{\sum_{i=1}^{n_s} |\alpha_{s,i}(k)|}{c_{s,0}(k)}, \quad k \geq M+1. \tag{4.17}$$

That is, $NPE_s(k)$ is the sum of absolute values of $\alpha_{s,i}(k)$'s, normalized to $c_{s,0}(k)$. It follows from Eq. (4.9) that $\alpha_{s,i}(M+1) = c_{s,i}(M+1)$, $\forall i$, because $\hat{\mathbf{h}}_{s,i}(M) = \mathbf{0}$, and hence, $NPE_s(M+1)$ always equals 1.

We also introduce an individual prediction error (IPE) between s and i at iteration k to quantify the attack-detection capability. Based on Eq. (4.14), we define

$$ipe_{s,i}(k) = \frac{|\alpha_{s,i}(k)|}{\max\{c_{s,i}(k), c_{\min}\}}, \quad k \geq M+1. \tag{4.18}$$

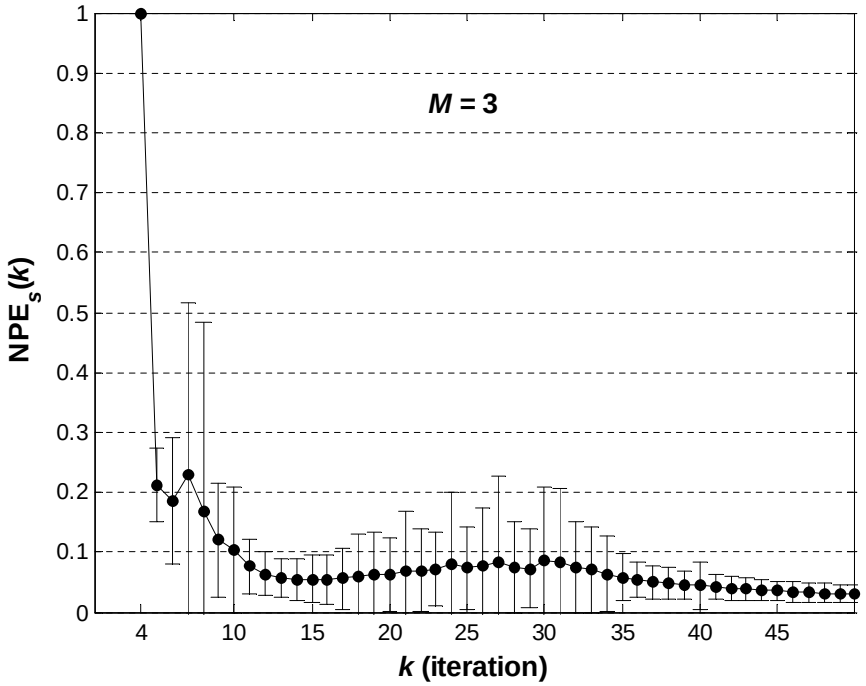


Figure 4.5: Attack-free normalized prediction error

4.4.3 Performance of the Profile Manager

We evaluate the performance of the profile manager in terms of its prediction accuracy by executing VeIL in an attack-free environment. To quantify this, we collected NPE values of s from 200 independent simulation runs of the localization process. Figure 4.5 plots the average $NPE_s(k)$, as well as the $(-\sigma_k, +\sigma_k)$ interval, as a function of k, where M = 3 and σ_k is the standard deviation of NPE measurements at iteration k. The results for M = 4 and 5 were similar to this. From this figure, we make the following observations. First, NPEs were mostly less than 0.1, demonstrating high accuracy of the profile manager. Second, NPE was around 0.2 at the $(M+2)^{\text{nd}}$ iteration, meaning that it constructed a ready-to-use profile pretty quickly, i.e., right after the filter gets activated. Finally, the profile manager incurred small processing and storage overheads thanks to the small order (=3) of the filters. As mentioned in Section 4.2.5, the processing overhead of s for updating

the profile is $\mathcal{O}(M^2)$ per neighbor, which is acceptable even for resource-limited sensors when $M = 3$.

In summary, our proposed profile manager based on adaptive filtering captures the localization behavior in as compact a form as possible, thus achieving both accuracy and computational efficiency.

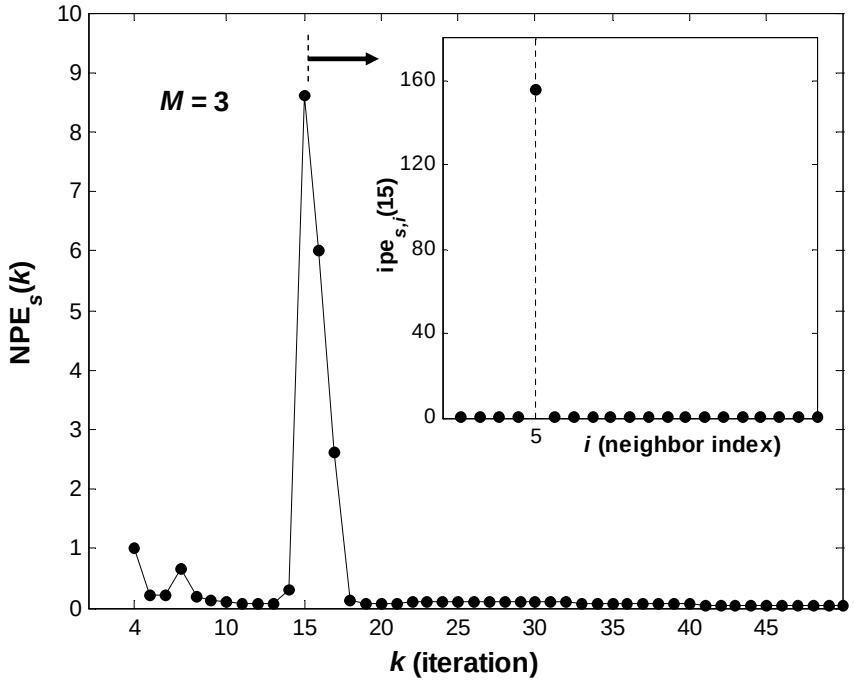
4.4.4 Performance of the Attack Detector

To evaluate the attack detector's performance, we simulate VeIL under the attack scenarios of (i) a single attack source and (ii) multiple simultaneous attack sources. Described below are the simulation results and their analyses for each of the two scenarios.

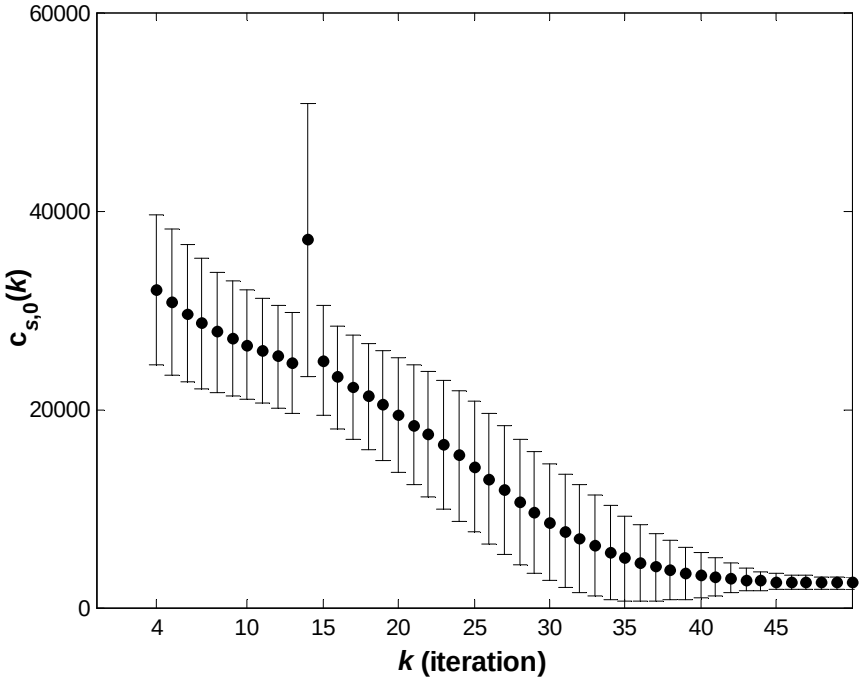
Defense against a Single Attack Source

In this simulation, a malicious neighbor (say, sensor 5) of s injects at iteration 14 false location information that deviates from the expected location by 40 [m], where the direction of deviation is determined randomly. This attack scenario is suitable for evaluating the VeIL's ability to handle false location announcements. We carried out 200 simulation runs under this attack scenario, and measured/computed $NPE_s(k)$, $ipe_{s,i}(k)$ and $c_{s,0}(k)$.

Figure 4.6(a) plots both $NPE_s(k)$ and $ipe_{s,i}(k)$. The former quantifies the attack strength, while the latter identifies the source of the false information. We observe that the attack occurred at iteration 14 increased $\alpha_{s,5}(14)$ in proportion to the attack strength, and boosted $\alpha_{s,5}(15)$ by orders of magnitude. Moreover, the next two iterations also exhibited unusually large prediction errors. This implies that the adversarial cost disrupted the prediction mechanism while it resided in the profile manager. Clearly, these results agree with our analysis in Section 4.2.6. In Figure 4.6(b), we also plot $c_{s,0}(k)$ as a function of k. The figure shows the local cost created a spike at iteration 14, thus causing the test of Eq. (4.15) to fail. This is an evidence that the increase in $\alpha_{s,5}(14)$ is due to an attack.



(a) Normalized and individual prediction errors



(b) Local cost

Figure 4.6: Attack-detection capability: a single false location announcement at iteration 14

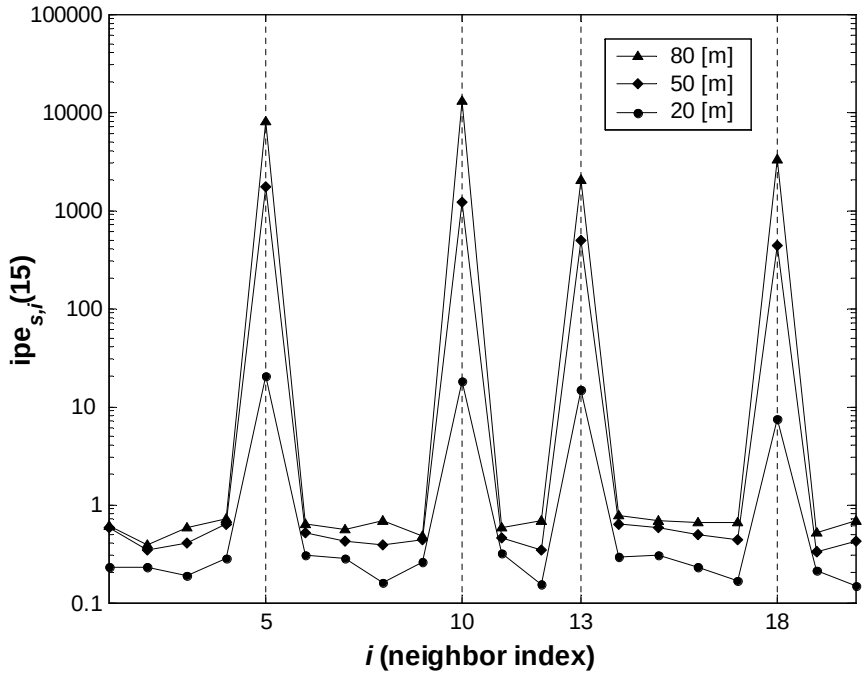


Figure 4.7: Individual prediction errors with varying attack strengths

Based on these results, we constructed multiple layers of defense mechanisms against attacks as follows. First, the tests based on Eqs. (4.14) and (4.15) serve as the first line of defense that diagnoses and combats attacks as early as possible. Second, VeIL checks and monitors the strength of $\alpha_{s,5}(\cdot)$ for the next M iterations to uncover the attacks that somehow evaded the first line of defense.

Defense against Multiple Attack Sources

We now evaluate the VeIL's capability to detect multiple malicious nodes that have simultaneously announced false locations. Figure 4.7 presents the IPE of s at iteration 15 when 4 malicious neighbors (whose indices are 5, 10, 13 and 18) simultaneously mounted attacks of varying strengths that deviate from the desired values by 20, 50 and 80 [m]. We make the following observations from the figure. First, $ipe_{s,i}(15)$ increased very rapidly with the attack strength, demonstrating VeIL's effectiveness in countering location-

targeted attacks. Second, VeIL successfully detected attacks incurring small perturbations, e.g., a half of the communication range. This demonstrates the VeIL's very high resolution in detecting attacks. Finally, VeIL preserved its attack-detection capability regardless of the number of attack sources as far as they are less than one-third of the total number of neighbors, which is obvious from the fact that VeIL separately maintains an adaptive filter for each neighbor.

4.5 Conclusion

In this chapter, we proposed a novel attack-tolerant localization scheme, called VeIL, for a large-scale sensor network deployed with only a small number of less-capable anchors. The use of spatio-temporal correlation among adjacent nodes played a key role in developing VeIL as a cooperative intrusion/anomaly detection system tailored to localization that consists of (1) adaptive management of the profile for normal localization behavior, and (2) distributed detection of false locations via comparison with the thus-managed profile. Our security analysis and performance evaluation demonstrated the high-level attack-tolerance and the feasibility of VeIL on resource-limited sensors, in that VeIL successfully defeats many critical attacks while incurring only small overheads.

CHAPTER V

PROGRAM INTEGRITY VERIFICATION

5.1 Introduction

Sensor networks are vulnerable to various security attacks, especially because they are deployed in a hostile and/or harsh environment. In such an environment, a captured sensor may be reverse-engineered, modified and abused by the adversary. That is, the adversary can (i) acquire (via analysis of the sensor memory) detailed knowledge of what the sensor's program is supposed to do and what the master secret is; (ii) modify the program with a malicious code; and (iii) produce and deploy multiple copies of the manipulated sensor device in the network. This is a serious problem, as sensor devices, once compromised, can subvert the entire network, e.g., blocking nodes within its communication range from receiving and/or sending/relaying any information. Consequently, it is essential to make a sensor device *tamper-proofing*.

Traditionally, the tamper-proofing of programs or master secret relies on tamper-resistant hardware [3, 19]. However, this hardware-based protection will likely fail to provide acceptable security and efficiency, because (i) strong tamper-resistance is too expensive to be implemented in resource-limited sensor devices, and (ii) the tamper-resistant hardware itself is not always absolutely safe due to various tampering techniques [3, 2, 11] such as reverse-engineering on chips, micro-probing, glitch and power analysis, and cipher

instruction search attacks. Existing approaches to generating tamper-resistant programs without hardware support can be classified as:

code obfuscation [26, 113, 114, 123] that transforms the executable code to make analysis/modification difficult;

result checking [10, 36, 116] that examines the validity of intermediate results produced by the program;

self-decrypting programs [5, 27] that store the encrypted executables and decrypt them before execution; and

self-checking [5, 23, 48] that embeds, in programs, codes for hash computation as well as correct hash values to be invoked to verify the integrity of the program under execution.

However, for the following reasons these approaches are unsuitable for sensor networks where a program runs on a slow, less-capable CPU in each sensor device. First, in the case of code obfuscation, it becomes easier to tamper with the program code as the code size in low-cost sensor devices shrinks, let alone the theoretical difficulty of obfuscation [6]. Moreover, just making it difficult to tamper with program code is not sufficient, as it cannot protect against "determined" attackers. Second, techniques based on result-checking or self-decryption are too expensive to be employed in resource-limited sensor devices because it repetitively incurs the overhead of verification or decryption, shortening the sensor's battery lifetime and degrading the network throughput. Third, the security of self-decrypting programs can be easily broken unless the decryption routines are protected from reverse-engineering, for example, by means of hardware. Likewise, self-checking techniques become defenseless once the hash computation code and/or the hash values have been identified/analyzed by the adversary.

In spite of these threats, little has been done on tamper-proofing tailored to resource-limited sensor devices. To defend the sensor network against the above-mentioned attacks, the following security conditions should be met: (i) the program residing in a sensor is not modified (*integrity*), and optionally, (ii) the sensor identifier (ID) is unique in a network (*uniqueness*). The second condition is needed only if certain services rely on unique IDs for their proper operation as the adversary may deploy the cloned sensors to sabotage the services. However, these conditions are difficult to meet due mainly to the usually hostile operational environment, as well as the very large size of sensor networks, under which it is easy for an adversary to capture and compromise sensors. We, therefore, need an approach that creates a network of mutually trusted sensors, i.e., each sensor can trust that the rest of the network has not been tampered with. To achieve this, we require each sensor to register itself with a dedicated server after verification of its program.

In this chapter, we propose a protocol, called *Program-Integrity Verification* (PIV), that verifies the integrity of the program residing in each sensor device, when it (1) joins the network or (2) has experienced a long service blockage. The latter is based on the fact that an adversary may have to disrupt the sensor's normal function for an extended period in order to capture/reverse-engineer/reprogram a sensor device and deploy the manipulated sensor in the network. Examining and verifying the program itself is easy to do for small, low-cost devices: the verification of a small program is fast and necessary only infrequently. The PIV protocol is very attractive because it

- prevents manipulation/reverse-engineering/reprogramming of sensors;
- does not degrade normal sensor functions since PIV is triggered infrequently and relies on neither self-decryption nor result checking;
- is purely software-based (and thus, can be used with/without tamper-resistant hard-

ware); and

- is tailored to the sensor devices with severe resource limitation (e.g., Motes with an 8-bit CPU and 4 KB RAM each [33]).

Moreover, the verification of each program incurs a very small overhead, as it only defines/uses cryptographic hash functions, which are orders-of-magnitude cheaper and faster than non-trivial cryptography like public-key algorithms.

A naive way of ensuring program integrity is to use digital signatures [15, 39, 70, 71, 87] as follows. During the pre-deployment stage, the digest of the original program is computed using an agreed-on hash function and then a signature is derived from the digest. The verifier (i.e., a server in charge of verification) processes the signature with a trapdoor one-way function (OWF) and compares the result with the digest for the current program. However, this digital signature-based scheme will likely fail regardless of the cryptographic strength of the OWF, since (part of) the verification procedure should be executed on a remote, untrusted sensor. For instance, the malicious sensor can deceive the verifier either by tampering with the digest or by faking/replaying messages (conveying the digest and/or verification results) to the verifier. One cannot avoid this type of attacks due mainly to (i) fixed and agreed-on algorithms for hashing and signature verification, and (ii) short lengths of the digest/signature. Applying public-key algorithms on the entire program (similarly to that in [58]) may solve these attacks, but it is too costly to employ public-key algorithms in severely resource-constrained sensor devices.

We, therefore, need an efficient way of protecting the verification process from being replayed/forged, in which the verifier randomly generates the hash computation algorithm for each verification. Keyed hash functions (e.g., [59, 92]) with randomly-chosen keys [98] could compute random hash values. Unfortunately, they are not feasible for

sensor networks because they stress both the sensor (into computing 32-bit operations) and the verifier (into storing/processing the entire programs). The scheme in [98] also randomly "traverses" program contents in order to slow down the hash calculation by a malicious device. However, this scheme guarantees detection of malicious programs only probabilistically, thus requiring a large number of memory accesses to achieve a high detection probability. Thus, we need a random hash computation algorithm that meets the following requirements:

- the hash computation optimized for embedded CPUs (e.g., 8- or 16-bit CPUs);
- examination of every location in both volatile and non-volatile memory; and
- incurring low processing/storage overheads to the verifier.

To meet these requirements, we propose the concept of a *randomized hash function* (RHF) which provides (i) random encoding of the hashing algorithm over a finite field $GF(2^n)$ where n is typically equal to 8, and (ii) two ways of computing the hash value, i.e., from the program (for sensors) and the digest (for the verifier). We also enforce PIV to process both code (in the non-volatile memory) and data (in RAM or EEPROM) initialized to uncompressible values, ensuring no room left for the attacker to copy the malicious code in. Based on RHFs, we realize PIV by constructing the security framework, the sensor pre-deployment scheme, and the verification protocol. We finally analyze the security and performance of the PIV protocol, and evaluate the RHF on Motes.

The remainder of the chapter is organized as follows. Section 5.2 describes the proposed protocol. Section 5.3 evaluates the performance for the PIV protocol. Finally, the chapter concludes with Section 5.4.

5.2 Program-Integrity Verification

We propose a protocol for program-integrity verification (PIV) in sensor networks, which prevents the compromised sensors from joining the network, without relying on tamper-resistance of hardware. The PIV protocol aims to form a closed network among those sensor devices that have correct (uncompromised) program. To achieve this, we require each sensor device to prove integrity (authenticity) of its program via a verification server, before gaining access to the network resources. In other words, each sensor must register itself with a verification server by having its program checked by the server. Otherwise, it cannot acquire any meaningful information from the network. This approach is particularly suitable for use in sensor networks for both security and performance reasons. For security, the network becomes more robust to physical-level attacks, in that it attempts to proactively prevent attacks rather than just detecting them afterwards. Accordingly, existing services are free from the fault-tolerance (Byzantine generals) problem in the presence of faulty/misbehaving devices. For performance, the latency to examine the entire program will be reasonably low, because the program in a sensor is relatively small as compared to the software for PCs/workstations. Moreover, it does not degrade normal sensor functions since PIV is triggered only infrequently and the program will remain unencrypted.

In this section, we present an attack model and the PIV goal, the rationale behind the PIV protocol and the randomized hash function, and describe the components for PIV and security and performance analyses.

5.2.1 The Attack Model and the PIV Objective

The Attack Model

Evaluating the degree of tamper-proofing is an important problem. Abraham *et al.* [1] discussed this issue in the design of tamper-resistant hardware, and classified attackers as clever outsiders, knowledgeable insiders, and funded organizations. However, the degree of tamper-proofing in the networked sensor devices should be defined differently, i.e., in terms of preserving the availability of the network. We claim that the strength of a given tamper-proofing solution be evaluated by the cost (e.g., time and effort) that the adversary should pay to acquire an adequate number of compromised sensors necessary to subvert the network. The degree of tamper-proofing is, therefore, categorized according to the complexity of (re)producing malicious sensors (in the order of increasing strength) as follows.

Level 1: the attacker may convert a sensor to a malicious slave by simply reprogramming the sensor without modifying its hardware. After securing the first slave, the attacker can subvert others very easily, for example, by cloning the compromised sensor.

Level 2: the attacker should pay a similar amount of time and effort each time (but without augmenting the sensor hardware) for an individual compromise, i.e., the attacker does not exploit the knowledge gained from previous subversions.

Level 3: the attacker subverts a sensor by modifying the sensor hardware, for example, attaching more memory, a more powerful CPU, and/or another device via a secondary RF interface.

Clearly, if the captured sensor is modified with more memory that can store both the original and malicious code (an attack of Level 3), it can deceive any defense mechanism, e.g.,

by feeding the original program to the verifier. No software-only schemes can defeat such an attack, because they cannot tell if the sensor hardware is modified or not.

The PIV Objective

We would like to support the tamper-proofing of sensors as strong as Level 2, making it extremely difficult for the adversary to modify the program without changing the sensor hardware. Here we do not consider Level-3 attacks for the following reasons. First, it is too costly to manipulate an adequate number of sensors for the intended attack due mainly to the large network size. Second, the PIV serves as a first line of defense even in the presence of Level-3 attacks, because it stresses the adversary into either manually modifying individual sensors or designing/manufacturing sensors of increased storage capacity. To completely protect the network against the above-mentioned attacks, one may use the PIV protocol together with network intrusion detection systems [9, 51, 60, 129, 130] that uncover suspicious sensors by monitoring network activities.

5.2.2 How to Secure PIV?

The proposed protocol uses *PIV Servers* (PIVSs), distributed over the entire network, so as to examine each sensor's program and check if it is the same as the original one. PIVSs are equipped with more computation and storage capacities than sensors. We also employ a special-purpose mobile agent, called a *PIV Code* (PIVC), which is generated by a PIVS and executed on a sensor being verified to read/process the program. We need the following two types of security on each verification:

- *sensor security* that protects the sensor from a malicious server/code disguised as a PIVS/PIVC, and
- *code security* that protects the PIVC from a malicious sensor.

The sensor security is achieved by using a conventional authentication server (AS) that acts as a trusted third party, by which the sensor can make sure that the PIVS is authentic, and hence, it is safe to execute the PIVC. Ensuring code security is more complicated than sensor security, mainly because the PIVC is almost defenseless when it is running on a remote sensor. Hence, we will develop a protocol that does not require the guarantee of code security.

Conventionally, data integrity is ensured by using digital signatures. Digital signatures can be applied to verify program integrity as follows. Each sensor has been programmed with a program $\mathbf{x}$ and a signature $\mathbf{s}_p$, where $\mathbf{s}_p$ has been computed from $\mathbf{x}$ by compressing $\mathbf{x}$ into a digest $\mathbf{d}_p$ with an agreed-on hash function, and then processing $\mathbf{d}_p$ with a signature function. Then, the PIVS restores $\mathbf{d}_p$ by applying a trapdoor one-way function to $\mathbf{s}_p$, computes another digest $\mathbf{d}_v$ for the program to be verified, and checks if the two digests match. However, this digital signature-based scheme will likely fail, as the computation of $\mathbf{d}_v$ and the transmission of $\mathbf{s}_p$ and $\mathbf{d}_v$ to the PIVS should be done on a remote, untrusted sensor device that has not yet been verified. In particular, a malicious sensor can (1) reverse-engineer and modify the code for $\mathbf{d}_v$ computation (PIVC); (2) read/change data of $\mathbf{d}_v$ computation; and (3) fake messages (containing $\mathbf{s}_p$ and $\mathbf{d}_v$) from the PIVC to the PIVS. We should, therefore, assume that adversaries can arbitrarily modify $\mathbf{x}$, $\mathbf{s}_p$, and $\mathbf{d}_v$. In particular, the adversary who attempts to re-program the sensor with a malicious program $\tilde{\mathbf{x}}$ (a program with malicious codes appended to, or inserted in, the original program) may mount the following attacks.

A1. Tampering with the digest computation into calculating $\mathbf{d}_v$, instead of $\tilde{\mathbf{d}}_v$, on $\tilde{\mathbf{x}}$: since $\tilde{\mathbf{d}}_v$ is computed via a well-known cryptographic hash function and the length of $\tilde{\mathbf{d}}_v$ is very short (e.g., 16 bytes in case of MD5), the adversary can easily deceive the PIVS without knowledge of the underlying signature function.

A2. Intercepting the message exchanged between PIVC and PIVS to replace $\tilde{\mathbf{d}}_v$ with $\mathbf{d}_v$: the adversary can experiment with the PIVS and an unaltered sensor to get the value of $\mathbf{d}_v$; once $\mathbf{d}_v$ has been identified, it can be repeatedly replayed.

Clearly, these attacks are difficult to defend against when the algorithm for hash computation is fixed and the length of the digest is short. Creating a secure channel between PIVC and PIVS does not help because key materials and encryption/decryption routines can also be reverse-engineered. Making it just difficult to reverse-engineer them (e.g., via conventional code obfuscation techniques) is not enough, because, once they are compromised, the same method can be applied for subsequent break-ins (Level-1 attack). Applying public-key algorithms directly on the program, instead of the digest, may solve these attacks. However, it is very costly for severely resource-constrained sensor devices to process the entire program with the public-key algorithm.

We, therefore, propose an efficient way of protecting the verification process from being replayed or tampered with. To meet this need, we enforce that the PIVS randomly generate a hash calculation algorithm for each PIVC creation. Keyed hash functions with randomly-chosen keys could produce random hash values. However, they are not suitable for low-cost embedded devices like sensors, because (1) they are based on 32-bit operations, thus performing poorly in 8-bit CPUs which are currently the most commonly-used CPUs in low-cost sensor devices, and (2) the PIVS is required to store/process the programs, instead of digests, incurring high processing and memory overheads; although PIVSs are more capable than sensors, it still severely limits scalability. What is needed is a special class of cryptographic hash functions, called *randomized hash functions* (RHFs) which, in addition to random hash computation, provide two ways of computing $\mathbf{d}_v$, i.e., (i) from $\mathbf{x}$ and (ii) from $\mathbf{d}_p$. By using RHFs, the PIVS can randomly encode the hash computation algorithm for each PIVC it creates. That is, while keeping $\mathbf{d}_p$ internally, the PIVS

randomly chooses an RHF to generate the PIVC and allows the PIVC executed on the sensor to compute $\mathbf{d}_v$. Then, it is possible for the PIVS to check if $\mathbf{d}_v$ agrees with $\mathbf{d}_p$ via the same RHF. Using this idea, we can successfully defend against the above-mentioned attacks, thus achieving highly secure tamper-proofing on sensor-resident programs, i.e., sensors with modified programs cannot pass the PIV test.

Although we propose RHF as a hash function tailored to the resource-limited sensor devices, the entire PIV framework (in Section 5.2.4) doesn't depend on the choice of a specific hash function. That is, the PIVS can use, and dynamically switch to, any of the existing hash functions (such as RHF and HMAC) with no additional cost, because the hash computation algorithm itself is transmitted via PIVC upon *each* verification. For example, if the currently-active hash function is found to be insecure, the PIVS will use another hash function for new verifications.

Most of the successful attacks/cryptanalyses on cryptographic algorithms reduce the effective key lengths, which, in turn, facilitates exhaustive-search attacks. However, it still takes a significantly longer time than a single execution of the algorithm to compute a plaintext for a given ciphertext. To exploit this fact, PIV strictly enforces a time-limit on each verification (as described in Section 5.2.4), thereby tolerating attacks on hash algorithms as well as allowing the use of less secure algorithms.

5.2.3 The Randomized Hash Function

To design RHFs, we apply *multivariate quadratic* (MQ) polynomials over $\mathbb{F} = GF(2^n)$, where n is typically 8 to allow for byte-oriented processing. The use of small finite fields does not degrade the level of security and, if designed properly, it can achieve both strong security and fast processing. In fact, public-key signature schemes [30, 31, 32, 75] that belong to the category of multivariate cryptography, rely on small finite fields (e.g., $GF(2^7)$

or $GF(2^8)$) for their faster and shorter signatures. MQ polynomials have been used successfully to realize trapdoor one-way functions in the above-mentioned multivariate signature schemes, and hence, it is reasonable to characterize them as a one-way hash function.

We partition the program into multiple program blocks. Let Λ denote the size of the entire program in bytes and η the length (in bytes) of an element in $\mathbb{F}$, i.e., $\eta = \lceil \frac{n}{8} \rceil$. We build, from the original program $\mathbf{x}$, B program blocks, $\mathbf{x}_1, \ldots, \mathbf{x}_B$, where $\mathbf{x}_l = [x_{l,1} \ \ldots \ x_{l,m}]^T$ is an $m \times 1$ vector and $x_{l,i} \in \mathbb{F}$.[1] Likewise, the program $\tilde{\mathbf{x}}$ to be verified consists of $\tilde{\mathbf{x}}_1, \ldots, \tilde{\mathbf{x}}_B$, where $\tilde{\mathbf{x}}_l = [\tilde{x}_{l,1} \ \ldots \ \tilde{x}_{l,m}]^T$ and $\tilde{x}_{l,i} \in \mathbb{F}$. We define a digest for $\mathbf{x}_l$ as an $m \times m$ matrix X_l, which consists of all quadratic terms, $x_{l,i}\, x_{l,j}$. That is, $X_l = \mathbf{x}_l \mathbf{x}_l^T = (\, x_{l,i}\, x_{l,j}\,)$. The PIVS will process and store X_l's in its database. The size of this database will be much smaller than that of storing all sensor programs, since there exist program blocks common to all, or at least a group, of sensors (for the homogeneity of their missions) and multiple digests can be combined into one. That is, the more common program blocks or combined digests they have, the smaller the database gets.

The RHF computes the same hash value from both (i) the program block $\mathbf{x}_l$ (for hash computation in PIVC) and (ii) the digest X_l (for hash verification in PIVS), and possesses the following algebraic structure. The RHF is specified over spaces of program blocks $\mathcal{P} = \mathbb{F}^m$, digests $\mathcal{D} = \mathbb{F}^{m \times m}$, random keys $\mathcal{G} = \mathbb{F}^B$ and $\mathcal{H} = \mathbb{F}^{k \times m}$, and hash values $\mathcal{Y} = \mathbb{F}^{k \times k}$ ($k^2 \ll m$), and consists of

- a hash computation algorithm, $\mathsf{Hash} : \mathcal{G} \times \mathcal{H} \times \mathcal{P}^B \rightarrow \mathcal{Y}$, and
- a verification algorithm, $\mathsf{Vrfy} : \mathcal{G} \times \mathcal{H} \times \mathcal{D}^B \times \mathcal{Y} \rightarrow \{\, \text{pass}, \text{fail} \,\}$,

such that $\mathsf{Vrfy}(G, H, \{X_l\}, \mathsf{Hash}(G, H, \{\tilde{\mathbf{x}}_l\})) = \text{pass}$, if $\mathbf{x}_l = \tilde{\mathbf{x}}_l$ for all $l = 1, \ldots, B$. Note that k is a parameter determining the complexity of Hash and the size of Hash output

[1] $\mathbf{x}^T$ (A^T) is the transpose of a vector $\mathbf{x}$ (a matrix A).

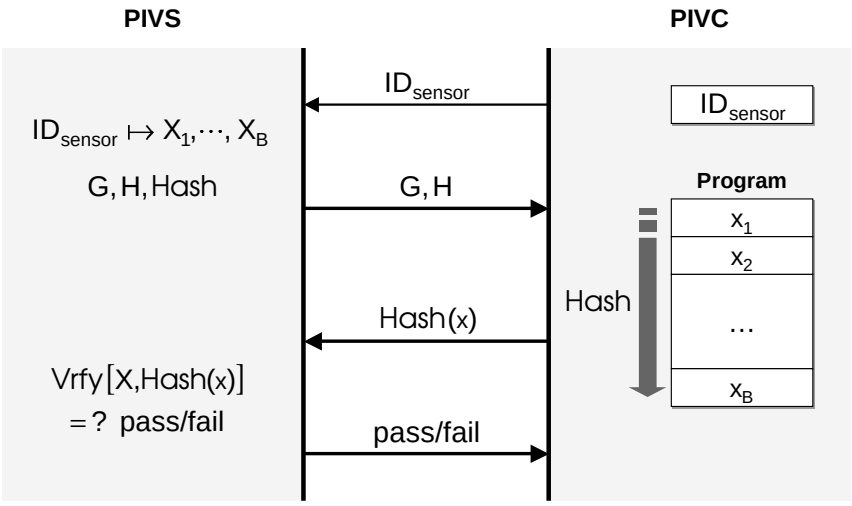


Figure 5.1: Hash and Vrfy algorithms

accordingly. Let $G = (g_l) \in \mathcal{G}$ and $H = (h_{ij}) \in \mathcal{H}$, where $g_l, h_{ij} \in \mathbb{F}$, denote randomly-chosen keys for each verification. The two ways to compute a hash value $Y = (y_{ij}) \in \mathcal{Y}$, $y_{ij} \in \mathbb{F}$ are as follows. First, the algorithm Hash computes Y from $\mathbf{x}_1, \ldots, \mathbf{x}_B$ as:

$$Y = \sum_{l=1}^{B} g_l \left(H\,\mathbf{x}_l\right) \left(H\,\mathbf{x}_l\right)^T. \tag{5.1}$$

Second, the algorithm Vrfy hashes $X_l, \ldots, X_B$ into Y as:

$$Y = H \left[\sum_{l=1}^{B} g_l X_l\right] H^T. \tag{5.2}$$

Clearly, Y can be represented as a set of MQ polynomials. Rewriting Eqs. (5.1) and (5.2) yields

$$y_{ij} = \sum_{l=1}^{B} \sum_{i'=1}^{m} \sum_{j'=1}^{m} g_l h_{ii'} h_{jj'} x_{l,i'} x_{l,j'}, \tag{5.3}$$

where $1 \leq l \leq B$ and $1 \leq i, j \leq k$. So, the RHF evaluates k^2 MQ equations in $m \times B$ variables.

Figure 5.1 shows how PIVS and the sensor interact with each other to cooperatively execute Hash and Vrfy. The PIVS and the sensor exchange the following messages.

M1. Sensor → PIVS: ID_{sensor}

M2. PIVS → Sensor: G, H

M3. Sensor → PIVS: $\mathsf{Hash}(G, H, \{\tilde{\mathbf{x}}_l\})$

M4. PIVS → Sensor: pass or fail

Accordingly, PIVC and PIVS proceed as follows.

PIVC initializes $\tilde{Y}$ to 0 then computes $\tilde{Y}$ from $\tilde{\mathbf{x}}_1, \cdots, \tilde{\mathbf{x}}_B$, i.e., for each $1 \leq l \leq B$, it calculates

1. $\tilde{\mathbf{z}}_l = H\,\tilde{\mathbf{x}}_l$,
2. $\tilde{Y}_l = \tilde{\mathbf{z}}_l\,\tilde{\mathbf{z}}_l^T$,
3. $\tilde{Y} = \tilde{Y} + g_l\tilde{Y}_l$.

PIVS retrieves $X_l, \ldots, X_B$ corresponding to the target sensor, generates a PIVC with G, H and the Hash algorithm, lets the PIVC to be executed on the sensor, and receives $\tilde{Y}$. It then executes Vrfy as follows:

1. $X = \sum_{l=1}^{B} g_l X_l$ for $1 \leq l \leq B$,
2. $Y = HXH^T$.

Finally, it checks if $Y = \tilde{Y}$.

As will be described in Section 5.2.4, X_l's can be combined into a few digest values, reducing the PIVS's processing and memory requirements.

5.2.4 Realization of PIV

In what follows, we describe how to realize the PIV protocol based on RHFs. We discuss all aspects of the protocol including the security framework, the PIV architecture, the pre-deployment phase of sensors, the state transition diagram for sensors, the verification protocol, and the realization of PIVS and PIVC.

The Security Framework

Figure 5.2 shows how to construct, based on PIV, the security framework of a sensor network. The three core building blocks of this framework are detailed below.

- **PIV** consists of PIVSs that interact with PIV-compliant sensors to verify programs in the sensors. PIV is triggered only (i) when a new sensor joins the network, or (ii) when an existing sensor is removed from the network, and optionally, (iii) if a sensor is suspected to have been compromised. Upon verifying the sensor, the PIVS either activates or locks the sensor.

- **Key management** (Chapter II) typically hinges on a cluster-based architecture,[2] in which a cluster-head distributes/renews a cluster-specific key periodically or whenever a sensor within its cluster is found (via PIV) to have been compromised.

- **Intrusion detection**, running on each cluster-head, continuously monitors/probes network activities (e.g., BEACON packets between neighbors) to detect malfunctioning devices (activities of which deviate significantly from those of agreed-on services/protocols) and, upon finding a suspicious device, requests its re-verification.

It is crucial to deny network access from those sensors blacklisted or unverified. To achieve this (as well as checking uniqueness of the ID in the sensor being verified),

[2]The entire network is divided into multiple clusters, each of which is controlled by a better-equipped cluster-head. Each sensor is associated with the cluster-head closest to itself.

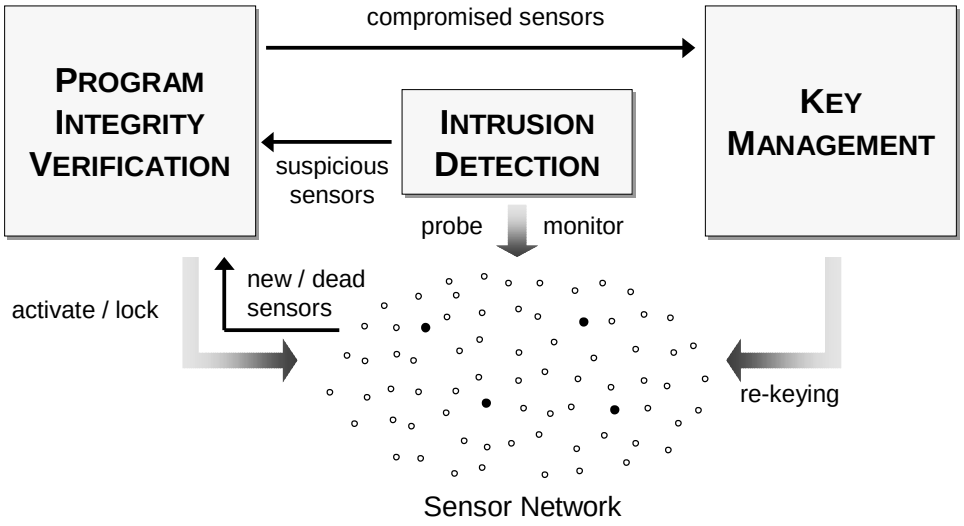


Figure 5.2: The security framework for sensor networks based on PIV

PIV maintains a database, called PIV_DB, of all successfully-verified IDs; it inserts into (deletes from) the PIV_DB, the ID upon activation (removal) of the sensor. Moreover, each ID in the PIV_DB is associated with certain attributes like the sensor's location. This is to make it impossible for a malicious sensor to spoof the IDs of the verified sensors, as those IDs will be easily traced back to inconsistent attributes. Hence, the only feasible way to gain access to the network is to execute and pass the PIV test. We also offer ways of actually *locking* a sensor, say **f**, that failed to register itself in the PIV_DB: (1) the PIVS asks all neighbors of **f** not to relay packets from **f**; (2) the key manager of a cluster for **f** refreshes the cluster key, thus disallowing **f** to access/eavesdrop packets; and (3) other services like routing may look up PIV_DB (via PIV) to ensure that the sensors are indeed genuine. The overheads of these operations are fairly small because they incur local traffic only.

The program within a sensor should be inspected as infrequently as possible (to reduce the overhead), inasmuch as it safeguards network resources (to maintain the required level of security). We meet this requirement by having each registered sensor monitor others

in its neighborhood to detect if they ceased normal operation (e.g., sending out BEACON packets) for an extended period of time and, if they did, request the PIV to delete their IDs from PIV_DB. The PIV does so if sensors in the proximity of the dead sensor had reported the same information. As a result, any non-member sensor must register with the PIV by verifying its program with the PIV protocol. Note that it is impossible for an attacker to remove a valid sensor from the network (by falsely reporting its death) unless he compromises most of its neighbors. This cooperative monitoring among sensors is important to prevention of attacks because the adversary may turn off a sensor for a certain period of time, during which it captures, reverse-engineers and reprograms the victim.

The PIV Architecture

The sensor network contains two types of dedicated servers — PIVSs and ASs. The roles of these servers are as follows.

- The PIVS performs the PIV protocol on a sensor, and cooperates with other PIVSs in the network to update/manage PIV_DB. For scalability, we let cluster-heads in a cluster-based hierarchical architecture serve as PIVSs. This allows each PIVS to maintain a local PIV_DB that stores IDs of the sensors belonging to its own cluster. Clearly, the more PIVSs (cluster-heads) a network has, the smaller the distance between PIVS and the sensor, and the more compact the local PIV_DB. PIVSs are deployed as uniformly as possible to balance the workloads among themselves.

- The AS acts as a trusted third party for the sensor in testing the PIVS. It, therefore, maintains a list of all legitimate PIVSs in the network and updates the list whenever a PIVS is added or removed. It is undesirable to equip only one AS in a network, as the AS becomes a single point of failure and the performance bottleneck, and in such a case, we must use multiple ASs deployed over the entire network. Each AS

authenticates a PIVS using either public-key cryptography or a secret authentication key shared with each sensor.

We assume that there exists a mechanism for a sensor to learn how to discover, and reach, a PIVS/AS. One possible realization of such a mechanism is as follows. The PIVS/AS periodically floods its whereabouts (within a limited scope), and hence, those sensors that have already been verified, can update how to reach the closest (and active) PIVS/AS. The newly-deployed sensor will then ask nearby sensors for the location of PIVS/AS to contact. This mechanism can easily tolerate occasional failures of PIVS/AS. When a sensor did not receive any packet from its chosen PIVS/AS for a certain period of time, it switches to an alternative PIVS/AS as follows: if it had recently heard from other PIVSs/ASs, it chooses the closest one among them; else floods its PIVS/AS search request, waits for responses from PIVSs/ASs, and then selects an alternative. Besides, the above mechanism works seamlessly with *mobile* PIVSs/ASs/sensors by simply increasing the frequency of periodic broadcasting, which allows PIV to be applicable to mobile environments.

Figure 5.3 shows the interactions among AS, PIVS and the sensor during PIV. It consists of the following three tasks: (1) authentication of PIVS via AS; (2) transmission and execution of PIVC; and (3) program verification by PIVS/PIVC. That is, the sensor first asks one of the ASs for authentication of a PIVS (probably the one closest to itself) and, if authentication succeeds, requests the PIVS to verify its program. Then, the PIVS sends the PIVC to the sensor, receives a hash value for the current program (computed by the PIVC with the algorithm Hash), runs Vrfy, and finally, determines whether the program is compromised or not. If the sensor passes the verification test, then the PIVS registers it in the PIV_DB.

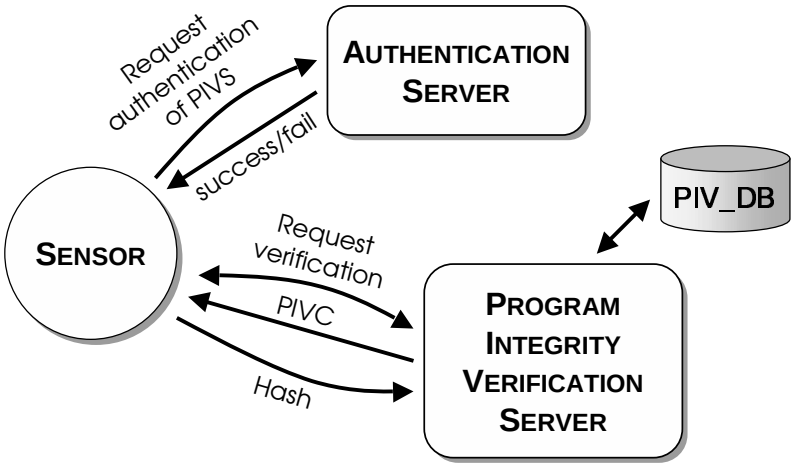


Figure 5.3: Interactions among AS, PIVS and the sensor during PIV

Pre-deployment of Sensors

A sensor device contains a unique master secret and ID. Each sensor also has two distinct programs: a *boot code* (executed for bootstrapping and initiation of the verification) and a *main code* (executed after the sensor has been successfully verified). Then, it is possible to take a snapshot of sensors' data space (excluding the area where the PIVC will be copied to) just before the execution of PIVC. The data space must be initialized to random values that can neither be predicted (e.g., all 0s or all 1s) nor compressed into a more compact form by an adversary. This is to prevent an attack where a tampered sensor abuses the free data space obtained by prediction/compression, for example, to keep a copy of the original program or the PIVC. An alternative (and more secure) way is to let the PIVC initialize the data space upon its execution, thus erasing hidden data, if any. We will henceforth use the terms "program" and "malicious program," as defined below.

DEFINITION 1. A (pre-deployed) program, $\mathbf{x}$, is a collection of boot and main codes, the master secret and ID, and the snapshot of the data space.

DEFINITION 2. A malicious program, $\tilde{\mathbf{x}}$, is the program containing one or more malicious code blocks that have been inserted in, or appended to, the pre-deployed program.

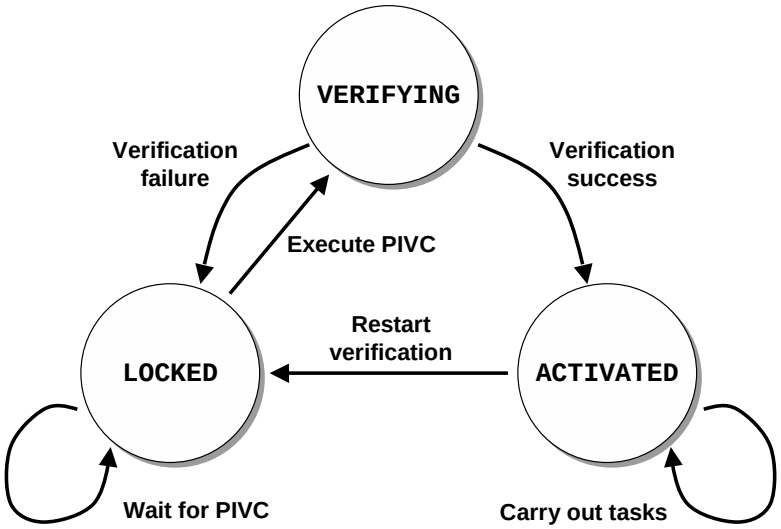


Figure 5.4: State-transition diagram for sensors

Pre-deployment of a sensor device consists of four off-line steps.

P1. Generation of a program $\mathbf{x}$, i.e., compilation of boot and main codes, selection of the master secret and ID, and construction of a data snapshot.

P2. Population of the sensor memory with $\mathbf{x}$.

P3. Computation of per-block digests X_l's from $\mathbf{x}$.

P4. Insertion of X_l's into PIV_DB.

State-Transition Diagram for Sensors

Figure 5.4 shows the state-transition diagram of each sensor. Each sensor device is associated with one of three states, namely "LOCKED", "VERIFYING" and "ACTIVATED" states, throughout its lifetime. When a sensor is executing the boot code, it is said to be in LOCKED state. Similarly, executions of the PIVC and the main code are bound to VERIFYING and ACTIVATED states, respectively.

Upon deploying a sensor device, it is started with the boot code and will remain in `LOCKED` state until it receives the PIVC from the PIVS. Since it is not yet a member of the network, it can perform no other tasks but wait for the PIVC. After receiving the PIVC, it makes a transition to `VERIFYING` state by executing the PIVC. The PIVC then verifies the program cooperatively with the PIVS and, based on the verification result, executes either the boot code or the main code: if the verification fails, it returns to `LOCKED` state, causing the network to deny this sensor's access to the network. Otherwise, it transitions to `ACTIVATED` state, in which the main code performs normal sensor functions. Finally, the main code responds to an explicit request for re-verification from the PIVS. If this is the case, it will restart the boot code and make a transition to `LOCKED` state. As PIVSs bookkeep successfully-verified sensors, directly executing the main code or ignoring a request (possibly by the adversary) will result in denial of the sensor's access to the network resources (see below for details).

The Verification Protocol

Figure 5.5 describes the verification protocol between the PIVS and the sensor. The verification protocol is initiated by either the boot code of the sensor device that wants to join the network or the PIVS that wants to re-verify the sensor device. The PIVS located closest to the sensor will be in charge of the verification. The verification procedure will proceed as follows.

V1. `Initialize`: this step starts the verification protocol between the PIVS and the sensor by exchanging their IDs. The sensor, after receiving the ID of PIVS, asks an AS for authentication of the PIVS, and if the authentication fails, terminates the protocol.

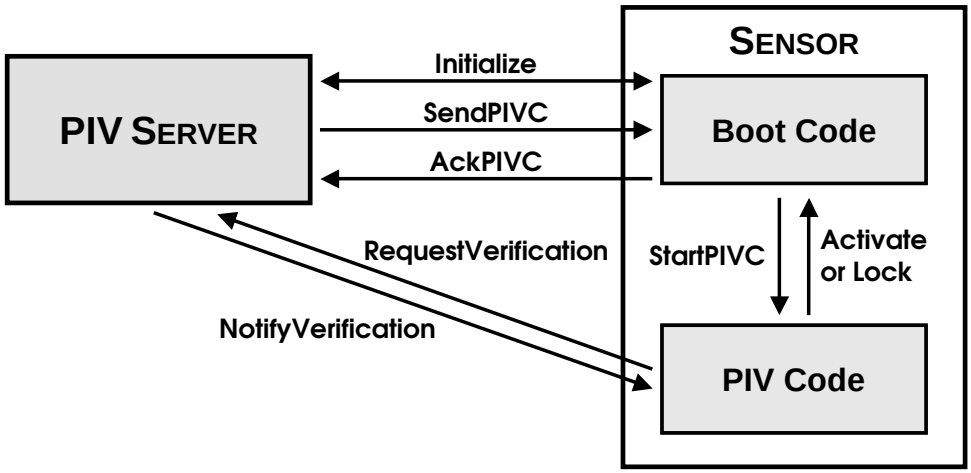


Figure 5.5: The verification protocol between the PIVS and the sensor

V2. `SendPIVC`: the PIVS generates a PIVC and then sends it to the sensor. It also records the time when PIV starts.

V3. `AckPIVC`: the sensor sends an acknowledgement back to the PIVS.

V4. `StartPIVC`: the sensor executes the received PIVC.

V5. `RequestVerification`: the PIVC computes a hash value on the program by executing Hash, and sends it back to the PIVS.

V6. `NotifyVerification`: the PIVS, if it received the hash result within a certain timeout period, examines the received hash value to check if the program has not been tampered with. If it passes the test, the PIVS registers the sensor in the PIV_DB. Then, the PIVS notifies the PIVC of the verification result.

V7. *Activate/lock sensor*: the PIVC, based on the verification result, either activates or locks the sensor. The sensor state will be changed to either `ACTIVATED` or `LOCKED`, accordingly.

The PIVS checks the latency between steps V2 and V5 and, if it exceeds a certain threshold, terminate the protocol. This time-limitation will place a great stress on the adversary's attempt to deceive the PIVS, e.g., emulating the PIVC's memory access or relaying the PIVC to an external machine that holds the original program. It is possible that an uncompromised sensor fails to verify itself due to transmission errors. Therefore, each sensor is allowed to retry the verification up to N times.

The first step, V1 ensures sensor security, i.e., a malicious device can neither pass the authentication procedure nor have its own code executed on the sensor, as far as the AS's authentication key is kept secret from the attacker. Thus, the attacker cannot abuse PIV to lock the other sensors. Finally, activating the sensor even when the PIVS indicates a verification failure (by the adversary), will result in denial of access to the network resources, as described in Section 5.2.4.

Realization of PIVS and PIVC

Figure 5.6 shows how to realize PIVS/PIVC based on Hash and Vrfy algorithms. In the pre-deployment stage, each sensor is programmed with a program $\{\mathbf{x}_l\}$. For all the sensors that have been successfully programmed, the PIVS computes and stores in PIV_DB the digests for $\{\mathbf{x}_l\}$. Thanks to the property that sensors share a portion of programs, the total number of distinct digests to be stored in the PIV_DB can be greatly reduced as discussed below. Each program block (digest) is classified as being (i) common to all sensors in the network; (ii) common to a group of sensors with the same missions; or (iii) unique to a specific sensor. We, therefore, reduce the size of PIV_DB by combining all digests belonging to the same class with a fixed combining factors, i.e., compute $X_{c,i} = \sum_{i^{th}common} g_l \mathbf{x}_l \mathbf{x}_l^T$, $i = 1, \cdots, N_c$, and $X_{\mathrm{u}} = \sum_{unique} g_l \mathbf{x}_l \mathbf{x}_l^T$, where N_c ($\ll B$) is the number of common digests. This pre-processing also relieves the Vrfy

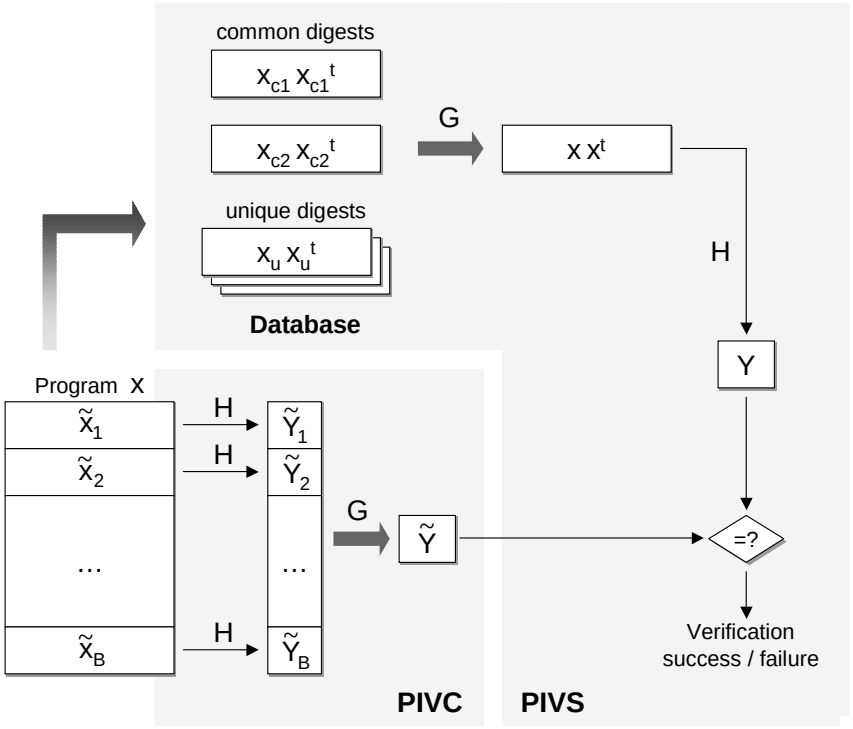


Figure 5.6: Realization of PIVS and PIVC

algorithm from the processing load, since the combining function G simply computes $X = \sum_i g_{c,i} X_{c,i} + g_u X_u$.

In the verification stage, the PIVS and PIVC cooperatively check the integrity of the program $\{\tilde{\mathbf{x}}_l\}$, of the sensor under verification, according to the protocol shown in Figures 5.1 and 5.5. Each message in Figure 5.1 triggers the following operations.

M1. The PIVS, using ID_{sensor}, retrieves, from PIV_DB, $X_{c,i}$'s and X_u that correspond to the program blocks of the sensor under verification. It also creates a PIVC with the Hash algorithm and random G and H.

M2. The PIVC computes $\tilde{Y} = \mathsf{Hash}(G, H, \{\tilde{\mathbf{x}}_l\})$ by executing the Hash algorithm.

M3. The PIVS executes the Vrfy algorithm to compute Y from $X_{c,i}$'s and X_u and check if $Y = \tilde{Y}$.

M4. The PIVC either activates or locks the sensor.

5.2.5 Security Analysis

We would like to show that (1) the proposed RHF can successfully defend itself against possible attacks, and (2) the only plausible attack requires modification of individual sensor hardware.

Replay attacks on messages M1 – M4 (i.e., intercepting a message and replacing it with an old message) cannot succeed as the proposed hash computation and verification are keyed operations, and random keys are mixed with the program blocks. Specifically, attacks on individual messages are defeated as follows. First, reporting a different ID_{sensor} (in M1) will be caught by the PIVS when its uniqueness is checked, and, moreover, the malicious sensor cannot pass the rest of the PIV test unless it has the matching program which must be free of malicious codes. Second, modifying G, H or the Hash algorithm will cause inconsistency between two hash outputs, and hence, the verification will fail. Third, replaying M3 does not work, because each verification will produce a distinct hash value even for the uncompromised sensor, and hence, old parameters (G and H) cannot be reused. Finally, intercepting M4 to always report "pass" may execute the main code. However, the subsequent requests to access the network resources will be denied, as explained in Section 5.2.4.

We then show that it is impossible for the adversary to forge the hash value without the knowledge of the original program. Consider the situation where the adversary re-programs the sensor with a malicious program $\{\mathbf{x}_l + \delta_l\}$, and attempts to fake the verification process by nullifying the effect of $\{\delta_l\}$ from the output of the Hash algorithm. This is impossible because the Hash algorithm is inherently a nonlinear function of program

blocks. By Eq. (5.1), the hash output Y yields

$$Y = H \left[\sum_{l=1}^{B} g_l \left\{ \mathbf{x}_l \, \mathbf{x}_l^T + 2\mathbf{x}_l \, \delta_l^T + \delta_l \, \delta_l^T \right\} \right] H^T, \tag{5.4}$$

which means

$$\mathsf{Hash}(G, H, \{\mathbf{x}_l + \delta_l\}) \neq \mathsf{Hash}(G, H, \{\mathbf{x}_l\}) + \mathsf{Hash}(G, H, \{\delta_l\}).$$

Therefore, to forge the hash output, the adversary must compute $\mathbf{x}_l \, \delta_l^T$ for all non-zero δ_l's as well as $\mathsf{Hash}(G, H, \{\delta_l\})$.

The only feasible attack is to store and feed either $\mathbf{x}_l$'s or X_l's to the PIVC. However, this type of attacks requires an excessively large amount of memory space, as opposed to that of conventional hashing schemes. First, the malicious sensor may disable the execution of PIVC and, instead, evaluate Eq. (5.2) using the original X_l's. But, since it cannot predict values of both G and H in advance, it must keep $X_{\mathrm{c},i}$'s and X_{u} to mimic the behavior of PIVC. The extra memory for storing them amounts to $(N_c + 1)m^2\eta$ bytes, e.g., 36 KB if $N_c = 3$, $m = 96$ and $\eta = 1$. Second, the malicious sensor may keep track of $\mathbf{x}_l$'s that differ from $\tilde{\mathbf{x}}_l$'s. If the malicious code is small and contiguous, it may suffice to save only a few program blocks. However, this attack can be defeated by applying "interleaving" on the program to construct program blocks, e.g., $B_i m\eta$-byte program space is interleaved into B_i (e.g., $B_i = \frac{B}{N_c+1}$) program blocks. A desirable property of interleaving is that the injection of a small malicious code affects no less than B_i blocks. Hence, the minimum requirement for the extra memory is $B_i m\eta$ bytes, e.g., 36 KB if $B_i = 384$, $m = 96$ and $\eta = 1$.

The replay attack on $\mathbf{x}_l$'s can be mounted if the malicious sensor has enough memory to maintain the original program blocks. However, as defined in Section 5.2.4, a program includes both code and data spaces, and a snapshot of data area (taken after initialization) is also inspected. Therefore, there is no room left in the sensor for the adversary to save

the original $\mathbf{x}_l$'s. The adversary may attach more memory to each sensor, but it will incur a considerable amount of hardware modification for each subversion. As mentioned in Section 5.2.1, we do not consider this kind of hardware-modifying attack, as it is unrealistic to mount such an attack in a large-scale network: the adversary must compromise multiple sensors (chosen from the entire network) with hardware modification to take control of the PIV-enabled network, but it is too costly to do so. Note that it does not increase the attack strength for the attacker to create one sensor with additional hardware (along with many reprogrammed slaves), then use it as a gateway/leader for the rest.

5.2.6 Performance Analysis

We analyze the performance of the proposed protocol by deriving the communication overhead between the PIVS and a sensor, and the computation and memory overheads of PIVC and PIVS. As defined earlier, k is the parameter that determines the length of the hash value. Λ, m and η refer to the size of the program, the size of the input block, and the size of a single word, all in bytes, respectively. Then, the number of input blocks, B, is derived as

$$B = \left\lceil \frac{\Lambda}{m\eta} \right\rceil \simeq \frac{\Lambda}{m\eta}. \tag{5.5}$$

Communication Overhead

We define the communication overhead as the total amount of the information exchanged between the PIVS and the sensor (normalized with respect to a per-hop value). The messages M2 and M3 dominate the communication overhead, and their lengths depend on the choice of protocol parameters. We, therefore, consider only these two messages. The sizes (in bytes) of G, H, the Hash code, and Y are $(N_c+1)\eta$, $km\eta$, L_{Hash}, and

$k^2\eta$, respectively. Hence, the communication overhead, C, is given by

$$\mathrm{C} = (km + k^2)\eta + L_{\mathsf{Hash}} + (N_c + 1)\eta. \tag{5.6}$$

The communication overhead depends on both m and k. Since $m \gg k$, we can control the communication overhead by the choice of m.

Processing Overhead of PIVC

The Hash algorithm relies on $GF(2^n)$ arithmetic. In finite fields, addition and subtraction are essentially "bitwise modulo 2", i.e., exclusive-OR of the corresponding bits of two operands, and hence very fast. In contrast, multiplication and division operations require lookup of two tables, each with 2^n elements. Obviously, multiplication and division are much more computationally expensive than addition and subtraction. We thus define the processing overhead of the PIVC as the average number of multiplications in $GF(2^n)$ per (η-byte) input word. The Hash algorithm iteratively evaluates (i) $\tilde{\mathbf{z}}_l = H\,\tilde{\mathbf{x}}_l$, (ii) $\tilde{Y}_l = \tilde{\mathbf{z}}_l\,\tilde{\mathbf{z}}_l^t$, and (iii) $\tilde{Y}$ += $g_l\tilde{Y}_l$, for $1 \le l \le B$. Each step incurs km, k^2 and k^2 multiplications, respectively. Hence, the algorithm computes $k(m + 2k)B$ multiplications over $GF(2^n)$ for processing the entire program. As a result, the processing overhead $\mathrm{P}_{\mathrm{PIVC}}$ is

$$\mathrm{P}_{\mathrm{PIVC}} = \frac{\eta}{\Lambda}k(m + 2k)B \simeq k + \frac{2k^2}{m}. \tag{5.7}$$

For its proper operation, Hash stores $\tilde{\mathbf{z}}_l$, $\tilde{Y}_l$, and $\tilde{Y}$, the sizes of which are $k\eta$, $k^2\eta$, and $k^2\eta$ bytes, respectively. Therefore, the PIVC allocates a buffer space of $\mathrm{M}_{\mathrm{PIVC}} = k(1 + 2k)\eta$ bytes.

Processing Overhead of PIVS

The Vrfy algorithm also performs addition and multiplication over $GF(2^n)$. Vrfy first computes X from $X_{\mathrm{c},i}$'s and X_{u}, then determines Y. Since each step incurs $(N_c + 1)m^2$

and $km^2 + k^2m$ multiplications, respectively, the processing overhead $\mathrm{P}_{\mathrm{PIVS}}$ (per input word) is

$$\mathrm{P}_{\mathrm{PIVS}} = \frac{1}{B}\left[(N_c+1)m + k(m+k)\right]. \tag{5.8}$$

Note that $\mathrm{P}_{\mathrm{PIVS}}$ is much smaller than $\mathrm{P}_{\mathrm{PIVC}}$ because $m \ll B$. For scalability, it is desirable to have a smaller $\mathrm{P}_{\mathrm{PIVS}}$ so that the server can handle as many concurrent verifications as possible. Vrfy reserves $k^2\eta$ bytes for storing Y. In addition, the PIVS maintains (i) N_c common digests, and (ii) digests (or program blocks) unique to individual sensors. If there are N sensors, the total amount of memory required by the PIVS is $\mathrm{M}_{\mathrm{PIVC}} = N_c m^2\eta + Nm\eta + k^2\eta \simeq Nm\eta$, because $N \gg N_c$.

5.3 Implementation and Evaluation

To evaluate the performance of our proposed approach, we first quantify the per-hop communication overhead between a sensor and the PIVS,[3] and the processing overhead that a sensor pays for each verification. Then, we demonstrate the strength of the proposed approach for typical choices of the parameters.

5.3.1 Overview of Implementation

We implemented Hash and Vrfy algorithms, with a sensor network of Motes and a laptop (acting as the PIVS). Because the Hash algorithm is downloaded to each sensor via the air medium and its execution is subject to severe resource constraints, it is important to make the algorithm as small as possible. In addition, for faster (byte-oriented) processing, we fix $\mathbb{F} = GF(2^8)$ (and $\eta = 1$, accordingly). The Hash algorithm is comprised of two parts: arithmetic operations over $GF(2^8)$ and the evaluation of Eq. (5.1). The code sizes and static data areas for these two modules are given in Table 5.1. The $GF(2^8)$ arithmetic

[3]Since we built PIV on top of the cluster-based architecture that uses a cluster-head as PIVS in each cluster, sensors can usually reach their PIVS in a small number of hops regardless of the network size. Hence, it suffices to consider the communication overhead normalized with respect to a per-hop value.

Table 5.1: Sizes of Hash components

	Code	Data
$GF(2^8)$ arithmetic	234	512
Hash computation	483	$km + N_c + 1$

uses two 256-byte tables for multiplication and division, while the module for hashing includes tables for G ($N_c + 1$ bytes) and H (km bytes).

To reduce the size of PIVC without loss of security, we can put the $GF(2^8)$-related routine in the boot code, and construct the PIVC using the hash computation routine only. Then, L_{Hash} becomes 483 bytes.

5.3.2 Communication Overhead

Using Eq. (5.6), Figure 5.7 plots the communication overhead C as a function of m, while varying k from 3 to 5. The figure shows that C is very small (e.g., $\mathrm{C} = 886$ bytes when $m = 96$ and $k = 4$) and depends on the choice of m and k. A considerable portion of C is due to the transmission of PIVC. If the effective data transmission rate (i.e., excluding headers, CRCs, error correction codes and control packets) of each Mote is 7 Kbps (out of the 40 Kbps raw data rate), the latency to transmit a PIVC is about 1 second per hop (when $m = 96$ and $k = 4$).

5.3.3 Processing Overhead

Figure 5.8 plots, using Eq. (5.7), $\mathrm{P}_{\mathrm{PIVC}}$ (the number of multiplications over $GF(2^8)$ per byte) of the Hash algorithm as a function of m, while varying k from 3 to 5. $\mathrm{P}_{\mathrm{PIVC}}$ is insensitive to the variations of m, while directly affected by the choice of k, e.g., $\mathrm{P}_{\mathrm{PIVC}}$ is around 4.33 multiplications per input byte when k = 4.

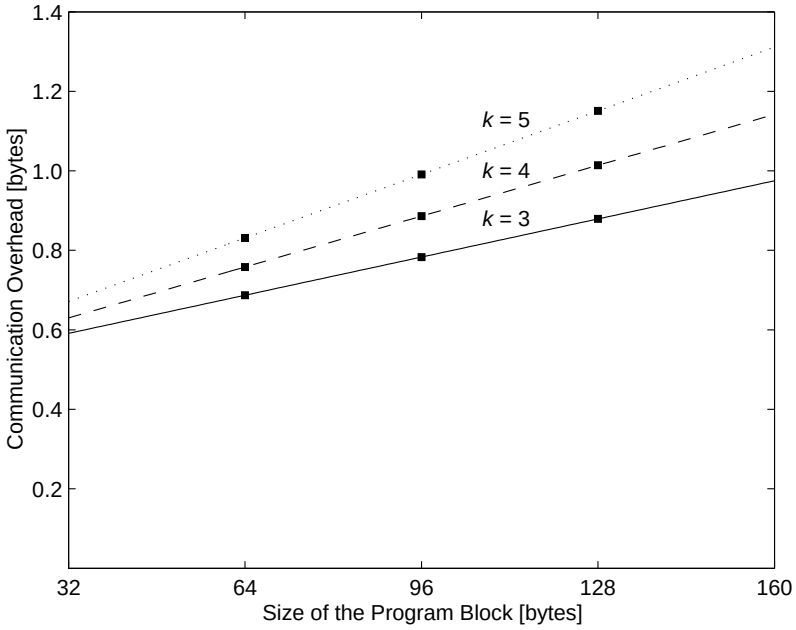


Figure 5.7: The communication overhead vs. m

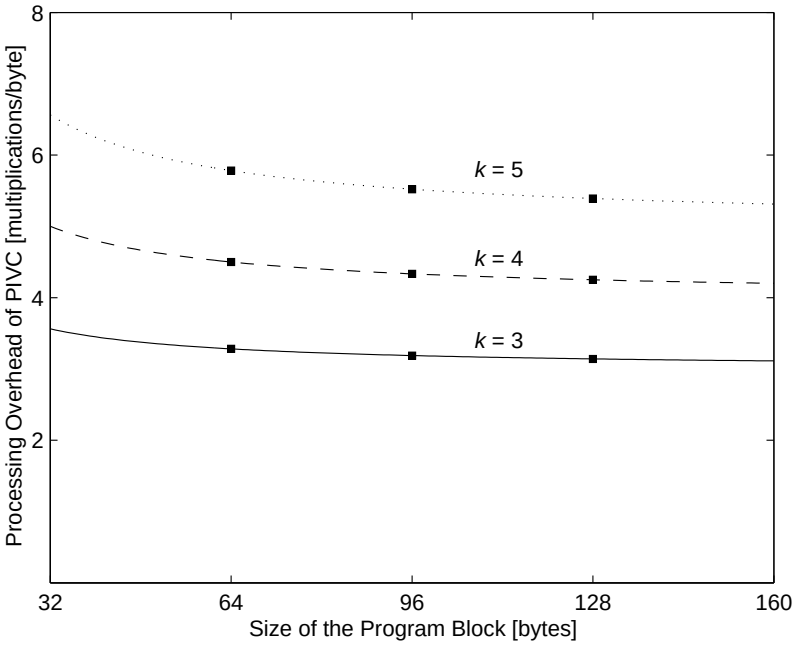


Figure 5.8: The processing overhead of PIVC vs. m

Table 5.2: The latency of Hash computation per 128 KB

		m	
		64	128
k	5	9.48	9.10
	4	7.51	7.29
	3	5.61	5.52

Table 5.2 shows the time (in seconds) for various values of m and k that the Hash algorithm spends to process the 128 KB program memory. Clearly, this time is proportional to the processing overhead. Moreover, since the Hash computation is executed very infrequently, the time in the order of tens of seconds at each sensor device is insignificant.

We also compare the processing overhead of the Hash algorithm against that of HMAC-MD5 by counting the number of CPU cycles per input-byte on an 8-bit CPU, called ATMega128L, of Motes. The HMAC-MD5 algorithm [59] performs two MD5 operations [91] on the input data (excluding key-related operations). Since MD5 consumes at least 55 cycles per input-byte, HMAC-MD5 incurs more than 110 cycles per input-byte. By contrast, the Hash algorithm uses about 38 cycles per input-byte when k = 4, making its processing overhead less than 35 % of that of HMAC-MD5 (when k = 4). Therefore, our proposed RHF is more computationally-efficient than HMAC-MD5. This computational efficiency is important because our PIV framework combats attacks on hash algorithms via time-limited verification, and hence, it is desirable to reduce the execution time as well as energy consumption.

Table 5.3: The PIV Parameters

Key length		$(km + N_c + 1)$	392 B
Block length		m	96 B
Hash length		k^2	16 B
Replay Protection	on digests	$(N_c + 1)m^2\eta$	72 KB
	prog. blocks	$\Lambda/(N_c + 1)$	81 KB
PIVC Latency	transmission	$\sim$ 1 second (per hop)	
	processing	$\sim$ 40 seconds	

5.3.4 Tradeoffs

The communication and processing overheads of PIV depends on the choice of k and m. The k value should be so chosen as to make the following tradeoff: a larger k yields higher security, but incurs more computation and processing delay. Once k is selected, m can be determined to reduce the communication overhead while maintaining an acceptable level of protection from replay attacks. That is, a smaller m yields less communication overhead, but the amount of data for the adversary to replay becomes smaller accordingly.

Table 5.3 lists typical values of the various parameters for verifying the program of length 648 KB (the total memory size of a Mote), when k, m and N_c are 4, 96 and 7, respectively. This PIV setup meets the requirement of both strong security and high performance as follows. First, it is secure in the sense that the adversary must modify sensor hardware (i.e., adding memory $>$ 72 KB) to evade PIV. Second, it takes less than 1 minute for the PIVS to verify each Mote. Moreover, thanks to its small processing overhead, the PIVS can verify multiple Motes in parallel, instead of sequentially, and hence, the initial network setup can be done very quickly.

5.4 Conclusion

In this chapter, we have proposed a soft tamper-proofing scheme based on Program-Integrity Verification (PIV), which offers (1) protection from manipulation, reverse-engineering, and re-programming of sensors; (2) purely software-based protection with/without tamper-resistant hardware; and (3) infrequent triggering of the verification. The PIVS plays a key role in our proposed scheme, i.e., verifying the integrity of the program of each sensor device and maintaining a database of digests for the original programs and sensor registry. For verification, it remotely calculates, via PIVC, a random hash value for the program being verified, computes another hash value from the digest for the original program, and checks if the two hash values match.

Our security analysis has shown that PIV effectively protects the network from possible attacks like replay attacks and the only plausible attack requires modification of sensor hardware. Our performance analysis/evaluation has demonstrated that the communication and processing overheads are very small (less than 1 KB and 4.5 multiplications over $GF(2^8)$ per byte, respectively), and the hash computation algorithm has a small time overhead ($5 \sim 9$ seconds per 128 KB) in 8-bit CPUs thanks to its byte-aligned operations.

CHAPTER VI

CONCLUSIONS AND FUTURE DIRECTIONS

6.1 Conclusions

This thesis aims to advance the secure networking technology for resource-limited embedded sensors by addressing the requirements of both high-level security and energy-efficiency. Its primary contribution lies in the development of unified, energy-efficient security framework, called *LiSP*, that enables low-cost, low-power sensors to provide high-level security at a very low cost. To this end, LiSP avoided using the traditional cryptography-based approaches intended for environments equipped with sufficient computation power & energy, and instead, focused on building security protocols via collaboration/cooperation among sensor nodes as well as among the protocols themselves. Our contributions are rehashed as follows.

We proposed two key mangement/sharing schemes that are complementary to each other. The first scheme, called GKMP, is designed specifically for group- or cluster-based network architectures that rely heavily on local transactions inside the group/cluster. By using the cryptographic one-way function and double-buffering of keys, it provides highly efficient re-keying without reliability support at the link layer, tolerates very loose synchronization of clocks on sensor devices, and offers the capability of trading security for residual energy. The results of performance evaluation and security analysis demonstrated

GKMP's effectiveness and efficiency in defeating various security attacks while incurring very small overheads.

Contrary to GKMP, distributed key sharing is tailored to securing communications between remote sensor nodes in existing/emerging applications for large-scale, distributed sensor networks. This scheme played a crucial role in the construction of attack-tolerant routing service that consists of two routing protocols, SGFP and TKEP. These protocols gracefully tolerated attacks from compromised sensors and successfully replaced the resource-expensive Diffie-Hellman key-setup protocol with a purely symmetric-cipher-based alternative. Our security analysis and performance evaluation showed that they are practically useful and effective in defeating/tolerating many critical attacks while consuming a moderate amount of energy.

We then investigated the problem of designing attack-tolerant protocols in the context of localization service. The proposed localization protocol, VeIL, exploited spatio-temporal correlation among adjacent nodes to characterize the normal localization behavior, detect/resist attacks, and accurately locate the attackers. This, in turn, led to the development of anomaly-based intrusion detection tailored to the localization service. The highest-level attack-detection capability and the feasibility of VeIL on resource-limited sensors were demonstrated via security analysis and performance evaluation.

Finally, we presented a soft tamper-proofing technique based on PIV (Program-Integrity Verification). While existing techniques were not suitable for low-cost (thus hardware resource-limited) sensor devices, the PIV protocol augmented such sensor devices to be usable for applications that require high-level security. In particular, it achieved purely software-based prevention of manipulation/reverse-engineering/re-programming/cloning of sensors based on a randomized hash function and mobile agent technology. As demonstrated with our security analysis, the proposed technique effectively defeated such attacks

as replay, impersonation, and masquerading, and hence, the only plausible attack required modification of sensor hardware. The performance analysis/evaluation showed verification to incur a very small overhead.

6.2 Future Directions

As described above, LiSP augments the emerging sensor networks to play key roles in safety-critical security applications like surveillance of the physical infrastructure that includes buildings, transportation systems, water supply systems, etc. However, to broaden the applicability of LiSP, there remains several open research problems associated with the attack-tolerant protocol design and the generalization of tamper-proofing technology, as described next.

Attack-tolerant protocol design: it is almost impossible to completely thwart all possible attacks in any form of systems including networked sensor systems, and hence, it is critical to make a system "attack-tolerant." To meet this need, we proposed a novel statistical approach to achieving attack-tolerance via the use of adaptive filtering to accurately track the normal behavior of the given service. In this thesis, this approach has been applied to the localization service, but it can also be extended to the other services. Hence, it is important to develop a generalized attack-tolerance framework that protects most of the core services from attacks. Also, it will be interesting to develop fault-tolerance and/or validation mechanisms for determining/maintaining the locations (IDs) of sensor nodes.

Generalized tamper-proofing: to completely safeguard the sensor network from physical attacks, the tamper-proofing will have to rely on a combination of both software- and hardware-based schemes. Towards this goal, we devised a software-based PIV

solution. Based on our analysis of, and experience with, PIV, we find it preferable to develop a complete tamper-proofing solution that extends PIV with low-cost hardware tamper-resistance techniques. Another possible research direction will be to apply our proposed tamper-proofing solution to the other embedded systems like cellular phones, for example, to diagnose if they've been infected with viruses/worms, and if so, to quarantine any malicious code.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] D. G. Abraham, G. M. Dolan, G. P. Double, and J. V. Stevens. Transaction Security System. *IBM Systems Journal*, 30(2):206–229, 1991.

[2] R. Anderson. Why Cryptosystems Fail. *Communications of ACM*, 37(11), November 1994.

[3] R. Anderson and M. Kuhn. Tamper Resistance — A Cautionary Note. In *Proceedings of 2nd Usenix Workshop Electronic Commerce*, Oakland, CA, November 1996.

[4] N. Asokan and P. Ginzboorg. Key Agreement in Ad Hoc Networks. *Computer Communications*, pages 1627–1637, 2000.

[5] D. Aucsmith. Tamper Resistant Software: An Implementation. *Information Hiding, LNCS 1174*, pages 317–333, 1996.

[6] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (Im)possibility of Obfuscating Programs. *CRYPTO'01, LNCS 2139*, 2001.

[7] S. Basagni, K. Herrin, D. Bruschi, and E. Rosti. Secure Pebblenets. In *Proceedings of ACM MobiHoc '01*, Long Beach, CA, October 2001.

[8] W. Basalaj. Proximity Visualization of Abstract Data, January 2001.
Available: http://www.pavis.org/essay/pavis.pdf.

[9] T. Bass. Intrusion Detection Systems and Multisensor Data Fusion. *Communications of the ACM*, April 2000.

[10] M. Blum and S. Kannan. Designing Programs that Check Their Work. *Journal of the ACM*, 42(1), 1995.

[11] S. Blythe, B. Fraboni, S. Lall, H. Ahmed, and U. Riu. Layout Reconstruction of Complex Silicon Chips. *IEEE J. Solid-State Circuits*, 28(2), February 1993.

[12] P. Bonnet, J. Gehrke, and P. Seshadri. Towards Sensor Database Systems. In *Mobile Data Management (MDM '01)*, Hong Kong, China, January 2001.

[13] N. Borisov, I. Goldberg, and D. Wagner. Intercepting Mobile Communications: The Insecurity of 802.11. In *Proceedings of ACM/IEEE MobiCom '01*, Rome, Italy, July 2001.

[14] G. Borriello, A. Liu, T. Offer, C. Palistrant, and R. Sharp. Wireless Acoustic Location with Room-Level Resolution using Ultrasound. In *Proceedings of ACM MobiSys '05*, Seattly, WA, June 2005.

[15] M. Brown, D. Cheung, D. Hankerson, J. L. Hernandez, M Kirkup, and A. Menezes. PGP in Constrained Wireless Devices. In *Proceedings of USENIX Security Symposium*, August 2000.

[16] N. Bulusu, J. Heidemann, and D. Estrin. GPS-less Low Cost Outdoor Localization for Very Small Devices. *IEEE Personal Communications Magazine*, 7(5), October 2000.

[17] M. Burnside, D. Clarke, T. Mills, S. Devadas, and R. Rivest. Proxy-based Security Protocols in Networked Mobile Devices. In *Proceedings of SAC '02*, March 2002.

[18] S. Capkun and J.-P. Hubaux. Secure Positioning in Sensor Networks, May 2004. Technical Report EPFL/IC/200444,
Available: http://www.terminodes.org/micsPublications.php?action=tr.

[19] D. W. Carman, P. S. Kruus, and B. J. Matt. Constraints and Approaches for Distributed Sensor Network Security, September 2000. NAI Labs Technical Report #00-010.

[20] D. W. Carman, B. J. Matt, and G. H. Cirincione. Energy-efficient and Low-latency Key Management for Sensor Networks. In *Proceedings of 23rd Army Science Conference*, December 2002.

[21] A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao. Habitat Monitoring: Application Driver for Wireless Communications Technology. In *Proceedings of ACM Workshop on Data Communications in Latin America and Caribbean*, April 2001.

[22] H. Chan, A. Perrig, and D. Song. Random Key Predistribution Schemes for Sensor Networks. In *Proceedings of IEEE Symposium on Security and Privacy '03*, May 2003.

[23] H. Chang and M. J. Atallah. Protecting Software Code by Guards. *DRM'01, LNCS 2320*, pages 160–175, 2002.

[24] I. Chang, R. Engel, D. Kandlur, D. Pendarakis, and D. Saha. Key Management for Secure Internet Multicast Using Boolean Function Minimization Techniques. In *Proceedings of IEEE INFOCOM '99*, March 1999.

[25] B. Chen, K. Jamieson, H. Balakrishnan, and R. Morris. SPAN: An Energy-Efficient Coordination Algorithm for Topology Maintenance in Ad Hoc Wireless Networks. In *Proceedings of ACM/IEEE MobiCom '01*, Rome, Italy, July 2001.

[26] C. Collberg, C. Thomborson, and D. Low. Breaking Abstractions and Unstructuring Data Structures. In *Proceedings of IEEE ICCL '98*, May 1998.

[27] C. S. Collberg and C. Thomborson. Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection. *IEEE Transactions on Software Engineering*, 28(8), August 2002.

[28] J. Costa, N. Patwari, and A. O. Hero III. Distributed Multidimensional Scaling with Adaptive Weighting for Node Localization in Sensor Networks. *submitted to ACM Transactions on Sensor Networks*.

[29] J. Costa, N. Patwari, and A. O. Hero III. Achieving High-Accuracy Distributed Localization in Sensor Networks. In *Proceedings of IEEE ICASSP '05*, Philadelphia, PA, March 2005.

[30] N. Courtois, L. Goubin, and J. Patarin. SFLASH, a fast asymmetric signature scheme for low-cost smartcards. Available: http://www.minrank.org/sflash-b.pdf.

[31] N. Courtois, L. Goubin, and J. Patarin. Flash, A Fast Multivariate Signature Algorithm. In *Cryptographers' Track RSA'01*, 2001.

[32] N. Courtois, L. Goubin, and J. Patarin. Quartz, 128-bit Long Digital Signatures. In *Cryptographers' Track RSA'01*, 2001.

[33] Crossbow. MICA, MICA2 Motes & Sensors. Available: http://www.xbow.com/.

[34] J. Douceur. The Sybil Attack. In *Proceedings of 1st International Workshop on Peer-to-Peer Systems*, 2002.

[35] G. L. Duckworth, D. C. Gilbert, and J. E. Barger. Acoustic Counter-Sniper System. In *International Symposium on Enabling Technologies for Law Enforcement and Security*, Boston, MA, November 1996. SPIE.

[36] F. Ergun, S. Kannan, S. R. Kumar, R. Rubinfeld, and M. Vishwanathan. Spot-Checkers. In *Proceedings of ACM Symp. Theory of Computing (STOC'98)*, May 1998.

[37] L. Eschenauer and V. D. Gligor. A Key-Management Scheme for Distributed Sensor Networks. In *Proceedings of ACM CCS '02*, Washington, DC, November 2002.

[38] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next Century Challenges: Scalable Coordination in Sensor Networks. In *Proceedings of ACM/IEEE MobiCom '99*, Seattle, Washington, August 1999.

[39] S. Even, O. Goldreich, and S. Micali. On-line/Off-line Digital Signatures. *Advances in Cryptology – Crypto'89, LNCS 435*, 1989.

[40] N. Haller. The S/KEY One-Time Password System. In *RFC 1760*. IETF, February 1995.

[41] H. Harney and C. Muchenhirn. Group Key Management Protocol (GKMP) Architecture. In *RFC 2094*. IETF, 1997.

[42] S. Haykin. *Adaptive Filter Theory*. Prentice-Hall, 2 edition, 1991.

[43] T. He, C. Huang, B. M. Blum, J. A. Stankovic, and T. Abdelzaher. Range-Free Localization Schemes for Large Scale Sensor Networks. In *Proceedings of ACM/IEEE MobiCom '03*, September 2003.

[44] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building Efficient Wireless Sensor Networks with Low-Level Naming. In *Proceedings of ACM SOSP '01*, October 2001.

[45] J. P. Hespanha, H. J. Kim, and S. Sastry. Multiple-Agent Probabilistic Pursuit-Evasion Games. In *Proceedings of the 38th Conf. on Decision and Control*, Phoenix, AZ, December 1999. IEEE.

[46] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture Directions for Networked Sensors. In *Proceedings of ACM ASPLOS '00*, Cambridge, MA, November 2000.

[47] F. Hohl. Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts. *Mobile Agents and Security, LNCS 1419*, 1998.

[48] B. Horne, L. Matheson, C. Sheehan, and R. E. Tarjan. Dynamic Self-Checking Techniques for Improved Tamper Resistance. *DRM'01, LNCS 2320*, pages 141–159, 2002.

[49] L. Hu and D. Evans. Localization for Mobile Sensor Networks. In *Proceedings of ACM/IEEE MobiCom '04*, October 2004.

[50] IEEE Std 802.11-1997. *Part 11: wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. IEEE, 1997.

[51] K. Ilgun, R. A. Kemmerer, and P. A. Porras. State Transition Analysis: A Rule-based Intrusion Detection Approach. *IEEE Transactions on Software Engineering*, 21(3), 1995.

[52] R. Jain, A. Puri, and R. Sengupta. Geographical Routing using Partial Information for Wireless Ad Hoc Networks. *IEEE Personal Communications*, February 2001.

[53] X. Ji and H. Zha. Sensor Positioning in Wireless Ad-hoc Sensor Networks Using Multidimensional Scaling. In *Proceedings of IEEE INFOCOM '04*, Hong Kong, March 2004.

[54] C. Karlof and D. Wagner. Secure Routing in Wireless Sensor Networks: Attacks and Countermeasures. *Ad Hoc Networks*, 2003.

[55] B. Karp and H. T. Kung. GPSR: Greedy Perimeter Stateless Routing for Wireless Networks. In *Proceedings of ACM/IEEE MobiCom '00*, Boston, MA, August 2000.

[56] R. Kennell and L. H. Jamieson. Establishing the Genuinity of Remote Computer Systems. In *Proceedings of USENIX Security Symposium*, August 2003.

[57] J. Kong, P. Zerfos, H. Luo, S. Lu, and L. Zhang. Providing Robust and Ubiquitous Security Support for Mobile Ad-Hoc Networks. In *Proceedings of ICNP '01*, Riverside, CA, October 2001.

[58] P. Kotzanikolaou, M. Burmester, and V. Chrissikopoulos. Secure Transactions with Mobile Agents in Hostile Environments. *Proceedings of the ACISP'00, LNCS 1841*, 2000.

[59] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. In *RFC 2104*. IETF, February 1997.

[60] S. Kumar and E. H. Spafford. A Software Architecture to Support Misuse Intrusion Detection. In *Proceedings of the 18th National Information Security Conference*, October 1995.

[61] L. Lazos and R. Poovendran. SeRLoc: Secure Range-Independent Localization for Wireless Sensor Networks. In *Proceedings of ACM WiSe '04*, October 2004.

[62] J. Li, J. Jannotti, D. S. J. De Couto, D. R. Karger, and R. Morris. A Scalable Location Service for Geographic Ad Hoc Routing. In *Proceedings of ACM/IEEE MobiCom '00*, Boston, MA, August 2000.

[63] X. S. Li, Y. R. Yang, M. G. Gouda, and S. S. Lam. Batch Rekeying for Secure Group Communications. In *Proceedings of 10th International World Wide Web Conference*, May 2001.

[64] Z. Li, W. Trappe, Y. Zhang, and B. Nath. Robust Statistical Methods for Securing Wireless Localization in Sensor Networks. In *Proceedings of IPSN '05*, April 2005.

[65] D. Liu, P. Ning, and W. Du. Attack-Resistant Location Estimation in Sensor Networks. In *Proceedings of IPSN '05*, April 2005.

[66] S. Madden, R. Szewczyk, M. J. Franklin, and D. Culler. Supporting Aggregate Queries over Ad-Hoc Wireless Sensor Networks. In *Proceedings of IEEE WMCSA '02*, New York, June 2002.

[67] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless Sensor Networks for Habitat Monitoring. In *Proceedings of ACM WSNA '02*, Atlanta, GA, September 2002.

[68] D. Malan. Crypto for Tiny Objects, 2004. Technical Reprot TR-04-04, Harvard University.

[69] D. A. McGrew and S. R. Fluhrer. The Stream Cipher Encapsulating Security Payload, July 2000. IETF, draft-mcgrew-ipsec-scesp-01.txt.

[70] R. C. Merkle. A Digital Signature Based on Conventional Encryption Function. *Advances in Cryptology – Crypto'87, LNCS 293*, 1987.

[71] R. C. Merkle. A Certified Digital Signature. *Advances in Cryptology – Crypto'89, LNCS 435*, 1989.

[72] M. J. Miller and N. H. Vaidya. A MAC Protocol to Reduce Sensor Network Energy Consumption Using a Wakeup Radio. *IEEE Transactions on Mobile Computing*, 4(3), May/June 2005.

[73] A. Mishra, K. Nadkarni, and A. Patcha. Intrusion Detection in Wireless Ad Hoc Networks. *IEEE Wireless Communications*, February 2004.

[74] S. Mittra. Iolus: A Framework for Scalable Secure Multicasting. In *Proceedings of ACM SIGCOMM '97*, Cannes, France, September 1997.

[75] T. Moh. A Public Key System with Signature and Master Key Functions. *Communications in Algebra*, 27(5), 1999.

[76] D. Nicolescu and B. Nath. Ad-Hoc Positioning Systems (APS). In *Proceedings of IEEE GLOBECOM '01*, November 2001.

[77] E. Pagani and G. P. Rossi. Reliable Broadcast in Mobile Multihop Packet Networks. In *Proceedings of ACM/IEEE MobiCom '97*, Budapest, Hungary, September 1997.

[78] T. Park and K. G. Shin. LiSP: A Lightweight Security Protocol for Wireless Sensor Networks. *ACM Transactions on Embedded Computing Systems*, 3(3), August 2004.

[79] T. Park and K. G. Shin. Attack-Tolerant Localization via Iterative Verification of Locations in Sensor Networks. *submitted for publication*, 2005.

[80] T. Park and K. G. Shin. Optimal Tradeoffs for Location-Based Routing in Large-scale Ad Hoc Networks. *IEEE/ACM Transactions on Networking*, 13(2), April 2005.

[81] T. Park and K. G. Shin. Secure Routing Based on Distributed Key Sharing in Large-scale Sensor Networks. *submitted for publication*, 2005.

[82] T. Park and K. G. Shin. Soft Tamper-Proofing via Program Integrity Verification in Wireless Sensor Networks. *IEEE Transactions on Mobile Computing*, 4(3), May/June 2005.

[83] P. N. Pathirana, N. Bulusu, A. V. Savkin, and S. Jha. Node Localization Using Mobile Robots in Delay-Tolerant Sensor Networks. *IEEE Transactions on Mobile Computing*, 4(3), May/June 2005.

[84] N. Patwari, A. O. Hero III, M. Perkins, N. S. Correal, and R. J. O'Dea. Relative Location Estimation in Wireless Sensor Networks. *IEEE Transactions on Signal Processing*, 51(8), August 2003.

[85] A. Perrig, R. Canetti, D. Song, and J. D. Tygar. Efficient and Secure Source Authentication for Multicast. In *Proceedings of NDSS '01*, San Diego, CA, February 2001.

[86] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and J. D. Tygar. SPINS: Security Protocol for Sensor Networks. In *Proceedings of ACM/IEEE MobiCom '01*, Rome, Italy, July 2001.

[87] G. Poupard and J. Stern. On the Fly Signatures Based on Factoring. In *Proceedings of ACM CCS '99*, November 1999.

[88] N. B. Priyantha, H. Balakrishnan, E. D. Demaine, and S. Teller. Mobile-Assisted Localization in Wireless Sensor Networks. In *Proceedings of IEEE INFOCOM '05*, Miami, FL, March 2005.

[89] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of ACM SIGCOMM '01*, San Diego, CA, August 2001.

[90] S. Ratnasamy, B. Karp, S. Shenker, D. Estrin, R. Govindan, L. Yin, and F. Yu. Data-Centric Storage in Sensornets with GHT, a Geographic Hash Table. *Mobile Networks and Applications (MONET) Special Issue on Algorithmic Solutions for Wireless, Mobile, Ad Hoc and Sensor Networks*, 2003.

[91] R. Rivest. The MD5 Message-Digest Algorithm. In *RFC 1321*. IETF, April 1992.

[92] P. Rogaway. OCB Mode: Parallelizable Authenticated Encryption. Available: http://csrc.nist.gov/encryp-tion/aes/.

[93] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *IFIP/ACM International Conference on Distributed Systems Platforms*, November 2001.

[94] S. Setia and S. Zhu and S. Jajodia. A Comparative Performance Analysis of Reliable Group Rekey Transport Protocols for Secure Multicast. In *Proceedings of Performance '02*, September 2002.

[95] T. Sander and C. Tschudin. Towards Mobile Cryptography. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 1998.

[96] N. Sastry, U. Shankar, and D. Wagner. Secure Verification of Location Claims. In *Proceedings of ACM WiSe '03*, September 2003.

[97] C. Savarese, J. Rabay, and K. Langendoen. Robust Positioning Algorithms for Distributed Ad-Hoc Wireless Sensor Networks. In *USENIX Technical Annual Conference*, Monterey, CA, June 2002.

[98] A. Seshadri, A. Perrig, L. Doorn, and P. Khosla. SWATT: SoftWare-based ATTestation for Embedded Devices. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2004.

[99] S. Setia, S. Koussih, S. Jajodia, and E. Harder. Kronos: A Scalable Group Rekeying Approach for Secure Multicast. In *Proceedings of IEEE Symposium on Security and Privacy '00*, May 2000.

[100] Y. Shang, W. Ruml, Y. Zhang, and M. P. J. Fromherz. Localization for Mere Connectivity. In *Proceedings of ACM MobiHoc '03*, June 2003.

[101] S. Singh and C. S. Raghavendra. PAMAS: Power Aware Multi-Access Protocol with Signalling for Ad Hoc Networks. *ACM Computer Communication Review*, 28(3), July 1998.

[102] M. Steiner, G. Tsudik, and M. Waidner. CLIQUES: A New Approach to Group Key Agreement. In *Proceedings of IEEE ICDCS '98*, May 1998.

[103] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM '01*, San Diego, CA, August 2001.

[104] M. Sun, L. Huang, A. Arora, and T. H. Lai. Reliable MAC Layer Multicast in IEEE 802.11 Wireless Networks. In *Proceedings of IEEE ICPP '02*, August 2002.

[105] K. Tang and M. Gerla. MAC Layer Broadcast Support in 802.11 Wireless Networks. In *Proceedings of MILCOM '00*, Los Angeles, CA, October 2000.

[106] BBN Technologies. TinyPK. Available: http://www.is.bbn.com/projects/lws-nest/.

[107] TinySec. Link Layer Security for Tiny Devices.
Available: http://www.cs.berkeley.edu/ nks/tinysec/.

[108] J. Tourrilhes. Robust Broadcast: Improving the Reliability of Broadcast Transmissions on CSMA/CA. In *Proceedings of IEEE PIMRC '98*, Boston, MA, September 1998.

[109] R. Vidal, O. Shakernia, H. J. Kim, H. Shim, and S. Sastry. Probabilistic Pursuit-Evasion Games: Theory, Implementation and Experimental Evaluation. *IEEE Transactions on Robotics and Automation*, 18(5), October 2002.

[110] G. Vigna. Cryptographic Traces for Mobile Agents. *Mobile Agents and Security, LNCS 1419*, 1998.

[111] J. R. Walker. Unsafe at Any Key Size; an Analysis of the WEP Encapsulation. IETF, October 2000.

[112] D. M. Wallner, E. G. Harder, and R. C. Agee. Key Management for Multicast: Issues and Architecture. In *RFC 2627*. IETF, June 1999.

[113] C. Wang, J. Hill, J. Knight, and J. Davidson. Software Tamper Resistance: Obstructing Static Analysis of Programs, 2000. Technical Report, Dept. of Computer Science, Univ. of Virginia.

[114] C. Wang, J. Hill, J. Knight, and J. Davidson. Protection of Software-based Survivability Mechanisms. In *Proceedings of DSN '01*, July 2001.

[115] H. Wang, D. Estrin, and L. Girod. Preprocessing in a Tiered Sensor Network for Habitat Monitoring. *EURASIP JASP Special Issue of Sensor Networks*, 2003.

[116] H. Wasserman and M. Blum. Software Reliability via Run-Time Result-Checking. *Journal of the ACM*, 44(6), 1997.

[117] B. R. Waters and E. W. Felten. Secure, Private Proofs of Location, January 2003. Technical Report TR-667-03, Princeton University,
Available: http://www.cs.princeton.edu/research/techreps/TR-667-03.

[118] K. Whitehouse and D. Culler. Calibration as Parameter Estimation in Sensor Networks. In *Proceedings of ACM WSNA '02*, Atlanta, GA, September 2002.

[119] U. G. Wilhelm, S. Staamann, and L. Buttyan. Introducing Trusted Third Parties to the Mobile Agent Paradigm. *Secure Internet Programming: Security Issues for Mobile and Distributed Objects, LNCS 1603*, 1999.

[120] C. K. Wong, M. G. Gouda, and S. S. Lam. Secure Group Communications Using Key Graphs. In *Proceedings of ACM SIGCOMM '98*, September 1998.

[121] A. Woo and D. Culler. A Transmission Control Scheme for Media Access in Sensor Networks. In *Proceedings of ACM/IEEE MobiCom '01*, Rome, Italy, July 2001.

[122] A. D. Wood and J. A. Stankovic. Denial of Service in Sensor Networks. *IEEE Computer*, 35(10), October 2002.

[123] G. Wroblewski. General Method of Program Code Obfuscation. In *Proceedings of SERP '02*, June 2002.

[124] Y. Xue, B. Li, and K. Nahrstedt. A Scalable Location Management Scheme in Mobile Ad Hoc Networks. In *Proceedings of IEEE Conf. on Local Computer Networks (LCN)*, 2001.

[125] Y. R. Yang, X. S. Li, X. B. Zhang, and S. S. Lam. Reliable Group Rekeying: Design and Performance Analysis. In *Proceedings of ACM SIGCOMM '01*, San Diego, CA, August 2001.

[126] F. Ye, H. Luo, J. Cheng, S. Lu, and L. Zhang. A Two-Tier Data Dissemination Model for Large-scale Wireless Sensor Networks. In *Proceedings of ACM/IEEE MobiCom '02*, Atlanta, GA, September 2002.

[127] W. Ye, J. Heidemann, and D. Estrin. An Energy-Efficient MAC Protocol for Wireless Sensor Networks. In *Proceedings of IEEE INFOCOM '02*, June 2002.

[128] M. Youssef and A. Agrawala. The Horus WLAN Location Determination System. In *Proceedings of ACM MobiSys '05*, Seattly, WA, June 2005.

[129] R. Zhang, D. Qian, C. Ba, W. Wu, and X. Guo. Multi-Agent Based Intrusion Detection Architecture. In *Proceedings of Int. Conf. on Computer Networks and Mobile Computing*, Beijing, China, October 2001. IEEE.

[130] Y. Zhang and W. Lee. Intrusion Detection in Wireless Ad Hoc Networks. In *Proceedings of ACM/IEEE MobiCom '00*, Boston, MA, August 2000.

[131] L. Zhou and Z. J. Haas. Securing Ad Hoc Networks. *IEEE Network Magazine*, 13(6), November 1998.

[132] L. Zhou, F. B. Schneider, and R. van Renesse. COCA: A Secure Distributed On-line Certification Authority. *ACM Transactions on Computer Systems*, 20(4), November 2002.

[133] S. Zhu, S. Setia, and S. Jajodia. LEAP: Efficient Security Mechanisms for Large-Scale Distributed Sensor Networks. In *Proceedings of ACM CCS '03*, October 2003.

ABSTRACT

LiSP: Lightweight Security Protocols for Wireless Sensor Networks

by

Taejoon Park

Chair: Kang G. Shin

Sensor networks, usually built with a large number of small, low-cost sensor devices, are characterized by their large-scale and unattended deployment that invites many critical attacks, thereby necessitating high-level security support for their intended applications and services. However, making sensor networks secure is challenging due mainly to the fact that sensors are battery-powered and it is often very difficult to change or recharge their batteries. To address this challenge, we design, develop and evaluate ***Li****ghtweight* ***S****ecurity* ***P****rotocols* (LiSP) that cooperatively build a unified, energy-efficient security framework for sensor networks.

We present two (group-based and distributed) key management/sharing schemes that are tailored to local and remote transactions, respectively. While the group-based scheme achieves efficient and robust re-keying via key broadcasting/authentication/recovery, distributed key sharing enables the development of attack-tolerant routing protocols capable of gracefully resisting device compromises as well as replacing resource-expensive, public-key-cipher-based protocols with a purely symmetric-cipher-based alternative.

The problem of attack-tolerance is further investigated for the development of a secure localization protocol. The proposed protocol uses mutual collaboration among sensors to achieve high-level attack-tolerance in terms of detecting/identifying/rejecting sources of attacks, if present. Accordingly, it plays the role of an anomaly-based intrusion detection system tailored to localization that safeguards the network from localization-targeted attacks.

As a countermeasure against physically tampering with sensors, we develop a novel soft tamper-proofing technique that verifies integrity of the program residing in each sensor device whenever it joins the network, or is suspected to have been compromised. Unlike other techniques unsuitable for low-cost, resource-limited sensors, our technique augments such sensors to be usable for applications that require high-level security.

Finally, the benefits of our protocols are demonstrated via analysis and evaluation of their capability to defeat known security attacks, and their performance in terms of processing, communication and memory overheads.