Soft Tamper-Proofing via Program Integrity Verification in Wireless Sensor Networks

Taejoon Park, *Student Member*, *IEEE*, and Kang G. Shin, *Fellow*, *IEEE*

Abstract—Small low-cost sensor devices, each equipped with limited resources, are networked and used for various critical applications, especially those related to homeland security. Making such a sensor network secure is challenging mainly because it usually has to operate in a harsh, sometimes hostile, and unattended environment, where it is subject to capture, reverse-engineering, and manipulation. To address this challenge, we present a *Program-Integrity Verification* (PIV) protocol that verifies the integrity of the program residing in each sensor device whenever the device joins the network or has experienced a long service blockage. The heart of PIV is 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, the PIV protocol 1) prevents manipulation/reverse-engineering/reprogramming of sensors unless the attacker modifies the sensor hardware (e.g., attaching more memory), 2) provides purely software-based protection, and 3) triggers the verification infrequently, thus incurring minimal intrusiveness into normal sensor functions. Our performance evaluation shows that the PIV protocol is computationally efficient and incurs only a small communication overhead, hence making it ideal for use in low-cost sensor networks.

Index Terms—Tamper-proofing, program-integrity verification, a randomized hash function, sensor networks.

1 INTRODUCTION

A sensor network is usually built with a large number of small devices, each of which has limited battery energy, memory, computation, and communication capacities. Such sensor networks can be used for various critical applications such as the 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. Despite the critical role in their intended applications, 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 1) 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, 2) modify the program with a malicious code, and 3) 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-proof*.

Traditionally, the tamper-proofing of programs or a master secret relies on tamper-resistant hardware [1], [2]. However, this hardware-based protection will likely fail to provide acceptable security and efficiency because 1) strong tamper-resistance is too expensive to be implemented in resource-limited sensor devices and 2) the tamper-resistant hardware itself is not always absolutely safe due to various tampering techniques [1], [3], [4] such as reverse-engineering on chips, microprobing, 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* [5], [6], [7], [8] that transforms the executable code to make analysis/modification difficult,
- *result checking* [9], [10], [11] that examines the validity of intermediate results produced by the program,
- *self-decrypting programs* [12], [13] that store the encrypted executables and decrypt them before execution, and
- *self-checking* [12], [14], [15] 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 [16]. 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 they continuously incurs the overhead of verification or decryption, shortening the sensor's battery lifetime

- *The authors are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109. E-mail: {taejoonp, kgshin}@eecs.umich.edu.*

Manuscript received 30 July 2004; revised 2 Dec. 2004; accepted 9 Dec. 2004; published online 29 Mar. 2005.
For information on obtaining reprints of this article, please send e-mail to: tmc@computer.org, and reference IEEECS Log Number TMC-0237-0704.

1536-1233/05/$20.00 © 2005 IEEE Published by the IEEE CS, CASS, ComSoc, IES, & SPS

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: 1) The program residing in a sensor is not modified (*integrity*) and, optionally, 2) 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 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 paper, 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 occurs 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 hardware), and
- is tailored to the sensor devices with severe resource limitation (e.g., Motes with an 8-bit CPU and 4 KB RAM each [17]).

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 nontrivial cryptography like public-key algorithms.

A naive way of ensuring program integrity is to use digital signatures [18], [19], [20], [21], [22] as follows: During the predeployment 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 1) fixed and agreed-on algorithms for hashing and signature verification and 2) short lengths of the digest/signature. Applying public-key algorithms on the entire program (similarly to that in [45]) 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., [23], [24]) with randomly chosen keys [47] 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 [47] 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 nonvolatile 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 1) random encoding of the hashing algorithm over a finite field $GF(2^n)$, where n is typically equal to 8, and 2) 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 nonvolatile 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 paper is organized as follows: Section 2 gives an overview of sensor networks and possible security attacks. Section 3 describes the proposed protocol. Section 4 evaluates the performance for the PIV protocol. Section 5 describes the related work. Finally, the paper concludes with Section 6.

2 Overview of Sensor Networks

Wireless sensor networks are deployed to collect information for various applications ranging from physical infrastructure [25], [26], [27] to habitat monitoring [28]. The success of their intended applications hinges on their own security, protecting individual sensors from compromise/reverse-engineering, detecting intrusions, and securing communications. In what follows, we briefly present a system architecture for sensor networks and possible security attacks.

2.1 System Architecture

For cost and size reasons, sensors are designed to minimize resource requirements, e.g., Motes [17] are built with an 8-bit CPU running at 4 MHz, 128 KB of program memory, 4 KB of RAM, 512 KB of serial flash memory, and two AA batteries. That is, sensors are usually built with limited battery energy, computation, memory, and communication capabilities.

The sensor network under consideration consists of a number of sensors and several better-equipped devices. The sensor network typically covers a wide area, requiring thousands or even millions of sensors, each of which is capable of, for example, reading temperature or detecting (part of) an object moving nearby. Moreover, the sensor network is usually deployed in a hostile and/or harsh environment and removal (due to device failures or depletion of battery energy) and addition of sensor nodes are not uncommon. Sensors coordinate with one another to achieve a higher-level sensing task, e.g., reporting with accuracy the characteristics of a moving object such as the speed and direction of movement. The sensor network consists of 1) sensors, 2) *data-collection* nodes, which process and make the sensed information available to data sinks, and 3) *control* nodes, which coordinate (multihop) data routing among sensors and broadcast commands to sensors. Each group or cluster of sensors—which is formed dynamically around moving objects to aggregate and/or route sensor data—elects a data-collection node as in [29], [30]. Clusters in a two-tier hierarchical system rely on control nodes, called *cluster-heads*, for managing the cluster topology, routing information, etc.

2.2 Security Attacks

Adversaries can be classified as *passive* or *active*. Passive attackers only eavesdrop on conversations in the network, while active attackers own keys of compromised sensors and inject packets into the network in addition to eavesdropping. Attacks on the sensor network can be classified as:

1. *physical attacks* on sensor devices, e.g., destroying, analyzing, and/or reprogramming sensors,
2. *service disruption attacks* on routing, localization, and time synchronization,
3. *data attacks*, e.g., traffic capture, replaying, and spoofing, and
4. *resource-consumption and denial-of-service (DoS) attacks*.

One of the serious attacks to the sensor networks deployed in an unattended environment is physical tampering with sensors. An adversary can easily 1) capture one or more sensors, 2) reverse-engineer/alter the program and/or master-secret in the sensor, and 3) create/deploy (multiple clones of) manipulated sensors. The compromised sensors will then be exploited by the adversary to mount actual attacks, e.g., initiating DoS attacks or sabotaging certain services of the sensor network, which will, in turn, facilitate the subversion of the entire network.

3 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 the tamper-resistance of hardware. The PIV protocol aims to form a closed network among those sensor devices that have a correct (uncompromised) program. To achieve this, we require each sensor device to prove the 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 afterward. 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.

3.1 The Attack Model and the PIV Objective

3.1.1 The Attack Model

Evaluating the degree of tamper-proofing is an important problem. Abraham et al. [31] 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 the 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.

3.1.2 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 [32], [33], [34], [35], [36] that uncover suspicious sensors by monitoring network activities.

3.2 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 reprogram the sensor with a malicious program $\tilde{\mathbf{x}}$ (a program with malicious codes appended to, or inserted into, 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 the 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., 1) from $\mathbf{x}$ and 2) 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.

3.3 The Randomized Hash Function

To design RHFs, we apply *multivariate quadratic* (MQ) polynomials over $\mathcal{F} = GF(2^n)$, 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 [37], [38], [39], [40] 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 $\mathcal{F}$, i.e., $\eta = \lceil \frac{n}{8} \rceil$. We build, from the original program $\mathbf{x}$, B program blocks, $\mathbf{x}_1, \dots, \mathbf{x}_B$, where $\mathbf{x}_l = [x_{l,1} \ \dots \ x_{l,m}]^T$ is an $m \times 1$ vector and $x_{l,i} \in \mathcal{F}$.[1] Likewise, the program $\tilde{\mathbf{x}}$ to be verified consists of $\tilde{\mathbf{x}}_1, \dots, \tilde{\mathbf{x}}_B$, where $\tilde{\mathbf{x}}_l = [\tilde{x}_{l,1} \ \dots \ \tilde{x}_{l,m}]^T$ and $\tilde{x}_{l,i} \in \mathcal{F}$. 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_ls 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 1) the program block $\mathbf{x}_l$ (for hash computation in PIVC) and 2) the digest X_l (for hash verification in PIVS) and possesses the following algebraic structure. The RHF is specified over spaces of program blocks $\mathcal{P} = \mathcal{F}^m$, digests $\mathcal{D} = \mathcal{F}^{m \times m}$, random keys $\mathcal{G} = \mathcal{F}^B$ and $\mathcal{H} = \mathcal{F}^{k \times m}$, and hash values $\mathcal{Y} = \mathcal{F}^{k \times k}$ ($k^2 \ll m$), and consists of

- a hash computation algorithm, `Hash` : $\mathcal{G} \times \mathcal{H} \times \mathcal{P}^B \rightarrow \mathcal{Y}$ and
- a verification algorithm,

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

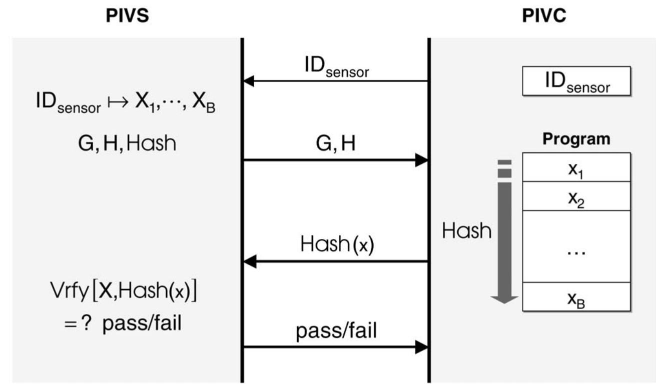

Fig. 1. `Hash` and `Vrfy` algorithms.

$$\texttt{Vrfy} : \mathcal{G} \times \mathcal{H} \times \mathcal{D}^B \times \mathcal{Y} \rightarrow \{\texttt{pass}, \ \texttt{fail}\},$$

such that $\texttt{Vrfy}(G, H, \{X_l\}, \texttt{Hash}(G, H, \{\tilde{\mathbf{x}}_l\})) = \texttt{pass}$, if $\mathbf{x}_l = \tilde{\mathbf{x}}_l$ for all $l = 1, \dots, B$. Note that k is a parameter determining the complexity of `Hash` and the size of `Hash` output accordingly. Let $G = (g_l) \in \mathcal{G}$ and $H = (h_{ij}) \in \mathcal{H}$, where $g_l, h_{ij} \in \mathcal{F}$, denote randomly chosen keys for each verification. The two ways to compute a hash value $Y = (y_{ij}) \in \mathcal{Y}$, $y_{ij} \in \mathcal{F}$ are as follows: First, the algorithm `Hash` computes Y from $\mathbf{x}_1, \dots, \mathbf{x}_B$ as:

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

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

$$Y = H \left[\sum_{l=1}^{B} g_l X_l \right] H^T. \tag{2}$$

Clearly, Y can be represented as a set of MQ polynomials. Rewriting (1) and (2) yields

$$y_{ij} = \sum_{l=1}^{B} \sum_{i'=1}^{m} \sum_{j'=1}^{m} g_l h_{ii'} h_{jj'} x_{l,i'} x_{l,j'}, \tag{3}$$

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.

Fig. 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: $\texttt{Hash}(G, H, \{\tilde{\mathbf{x}}_l\})$.
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. $\tilde{Y}_l = \tilde{\mathbf{z}}_l \ \tilde{\mathbf{z}}_l^T$, and
3. $\tilde{Y} = \tilde{Y} + g_l \tilde{Y}_l$.

PIVS retrieves $X_l, \dots, X_B$ corresponding to the target sensor, generates a PIVC with G, H, and the `Hash`

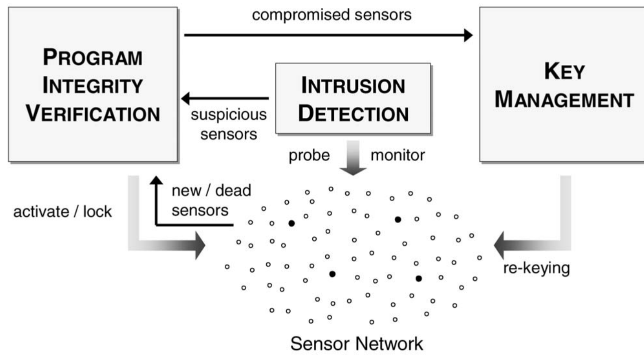

Fig. 2. The security framework for sensor networks based on PIV.

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 $1 \leq l \leq B$ and
2. $Y = HXH^T$.

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

As will be described in Section 3.4.6, X_ls can be combined into a few digest values, reducing the PIVS's processing and memory requirements.

3.4 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 predeployment phase of sensors, the state transition diagram for sensors, the verification protocol, and the realization of PIVS and PIVC.

3.4.1 The Security Framework

Fig. 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 1) when a new sensor joins the network or 2) when an existing sensor is removed from the network and, optionally, 3) if a sensor is suspected to have been compromised. Upon verifying the sensor, the PIVS either activates or locks the sensor.
- **Key management** [48] 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 reverification.

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.

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

3.4.2 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

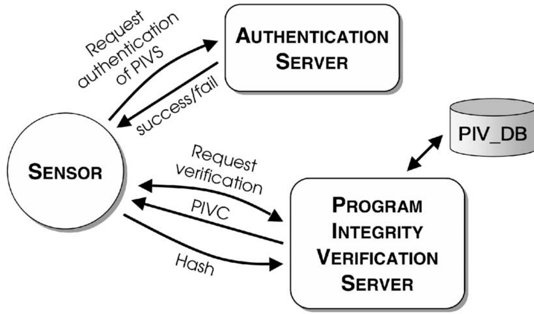

Fig. 3. Interactions among AS, PIVS, and the sensor during PIV.

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 it 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.

Fig. 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`.

3.4.3 Predeployment 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.

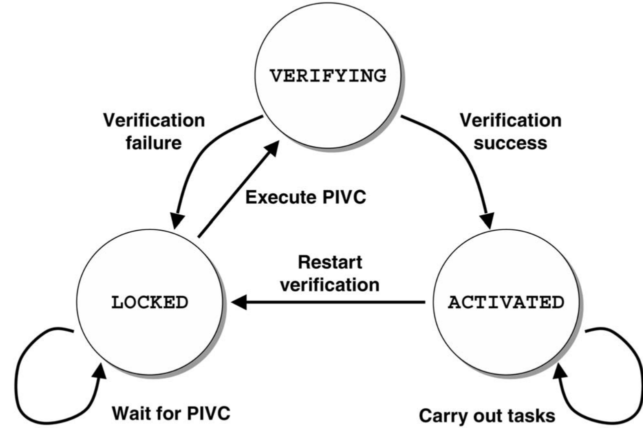

Fig. 4. State-transition diagram for sensors.

Definition 1. *A (predeployed) 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 into, or appended to, the predeployed program.*

Predeployment of a sensor device consists of four offline 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_ls from $\mathbf{x}$.
P4. Insertion of X_ls into `PIV_DB`.

3.4.4 State-Transition Diagram for Sensors

Fig. 4 shows the state-transition diagram of each sensor. Each sensor device is associated with one of three states, namely, the "`LOCKED`," "`VERIFYING`," and "`ACTIVATED`" states, throughout its lifetime. When a sensor is executing the boot code, it is said to be in the `LOCKED` state. Similarly, executions of the PIVC and the main code are bound to the `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 the

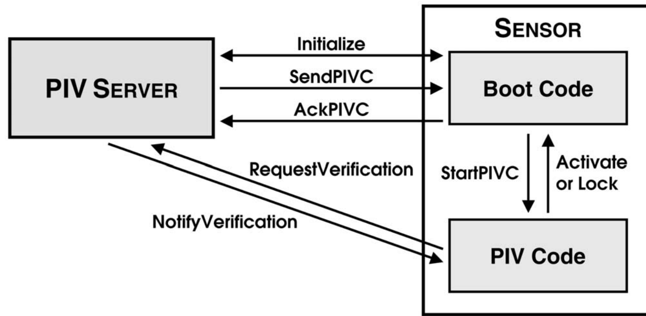

Fig. 5. The verification protocol between the PIVS and the sensor.

`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 the `LOCKED` state, causing the network to deny this sensor's access to the network. Otherwise, it transitions to the `ACTIVATED` state, in which the main code performs normal sensor functions. Finally, the main code responds to an explicit request for reverification from the PIVS. If this is the case, it will restart the boot code and make a transition to the `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).

3.4.5 The Verification Protocol

Fig. 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 reverify 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.
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 acknowledgment 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 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.

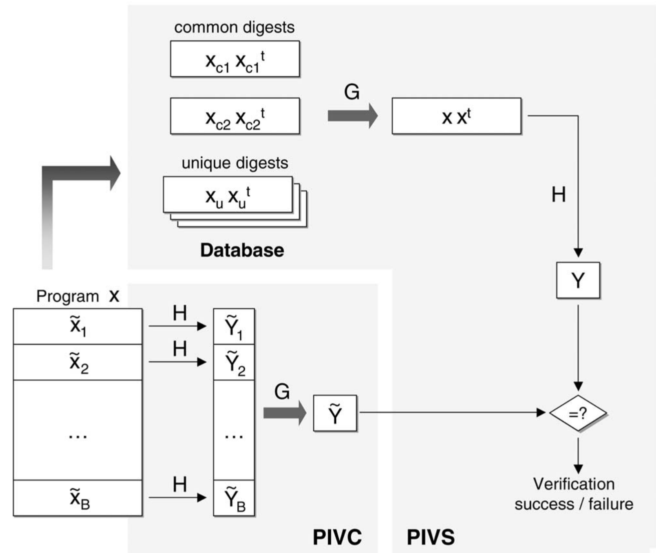

Fig. 6. Realization of PIVS and PIVC.

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 2 and 5 and, if it exceeds a certain threshold, terminates the protocol. This time-limitation will place 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.

Step 1 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 3.4.1.

3.4.6 Realization of PIVS and PIVC

Fig. 6 shows how to realize PIVS/PIVC based on the `Hash` and `Vrfy` algorithms. In the predeployment 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 1) common to all sensors in the network, 2) common to a group of sensors with the same missions, or 3) 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 $X_u = \sum_{unique} g_l \mathbf{x}_l \mathbf{x}_l^T$, where N_c ($\ll B$) is the number of common digests. This preprocessing also relieves the `Vrfy`

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 Figs. 1 and 5. Each message in Fig. 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 $\tilde{Y} = \texttt{Hash}(G, H, \{\tilde{\mathbf{x}}_l\})$ 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.

3.5 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 above (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 above) 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 3.4.1.

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 reprograms 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 (1), the hash output Y yields

$$Y = H \left[\sum_{l=1}^{B} g_l \{ \mathbf{x}_l\, \mathbf{x}_l^T + 2\mathbf{x}_l\, \delta_l^T + \delta_l\, \delta_l^T \} \right] H^T, \tag{4}$$

which means

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

Therefore, to forge the hash output, the adversary must compute $\mathbf{x}_l\, \delta_l^T$ for all nonzero δ_ls as well as $\texttt{Hash}(G, H, \{\delta_l\})$.

The only feasible attack is to store and feed either $\mathbf{x}_l$s or X_ls to the PIVC. However, this type of attack 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 (2) using the original X_ls. But, since it cannot predict values of both G and H in advance, it must keep $X_{c,i}$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 3.4.3, 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 3.1, 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.

3.6 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

$$B = \left\lceil \frac{\Lambda}{m\eta} \right\rceil \simeq \frac{\Lambda}{m\eta}. \tag{5}$$

3.6.1 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). Messages M2 and M3 dominate the communication overhead and their lengths depend on the choice of protocol parameters. We, therefore, consider only these

TABLE 1
Sizes of Hash Components

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

two messages. The sizes (in bytes) of G, H, the Hash code, and Y are $(N_c+1)\eta$, $km\eta$, $L_{\rm Hash}$, and $k^2\eta$, respectively. Hence, the communication overhead, C, is given by

$$\mathrm{C} = (km+k^2)\eta + L_{\rm Hash} + (N_c+1)\eta. \quad (6)$$

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

3.6.2 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 1) $\tilde{\mathbf{z}}_l = H\,\tilde{\mathbf{x}}_l$, 2) $\tilde{Y}_l = \tilde{\mathbf{z}}_l\,\tilde{\mathbf{z}}_l^t$, and 3) $\tilde{Y} += g_l\tilde{Y}_l$, 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 $\mathrm{P_{PIVC}}$ is

$$\mathrm{P_{PIVC}} = \frac{\eta}{\Lambda}k(m+2k)B \simeq k + \frac{2k^2}{m}. \quad (7)$$

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 $\mathrm{M_{PIVC}} = k(1+2k)\eta$ bytes.

3.6.3 Processing Overhead of PIVS

The Vrfy algorithm also performs addition and multiplication over $GF(2^n)$.

Vrfy first computes X from $X_{c,i}$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 $\mathrm{P_{PIVS}}$ (per input word) is

$$\mathrm{P_{PIVS}} = \frac{1}{B}\,[(N_c+1)m + k(m+k)]. \quad (8)$$

Note that $\mathrm{P_{PIVS}}$ is much smaller than $\mathrm{P_{PIVC}}$ because $m \ll B$. For scalability, it is desirable to have a smaller $\mathrm{P_{PIVS}}$ 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 1) N_c common digests and 2) digests (or program blocks) unique to individual sensors. If there are N sensors, the total amount of memory required by the PIVS is $\mathrm{M_{PIVC}} = N_c m^2\eta + Nm\eta + k^2\eta \simeq Nm\eta$ because $N \gg N_c$.

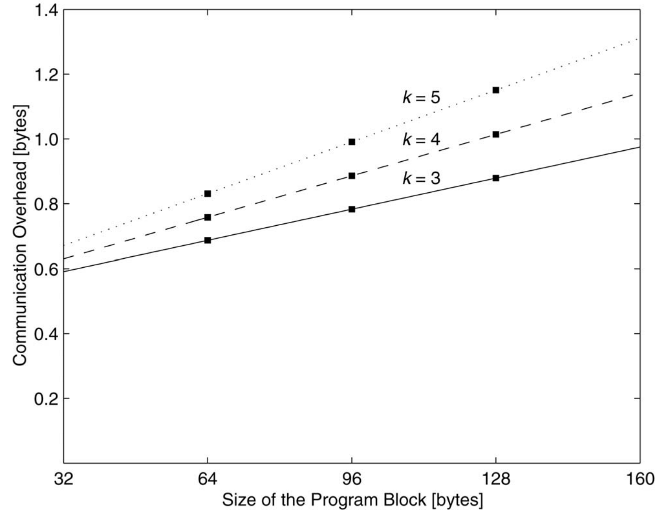

Fig. 7. The communication overhead versus m.

4 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.

4.1 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 $\mathcal{F} = GF(2^8)$ (and $\eta = 1$, accordingly). The Hash algorithm is comprised of two parts: arithmetic operations over $GF(2^8)$ and the evaluation of (1). The code sizes and static data areas for these two modules are given in Table 1. The $GF(2^8)$ arithmetic 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, $L_{\rm Hash}$ becomes 483 bytes.

4.2 Communication Overhead

Using (6), Fig. 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

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.

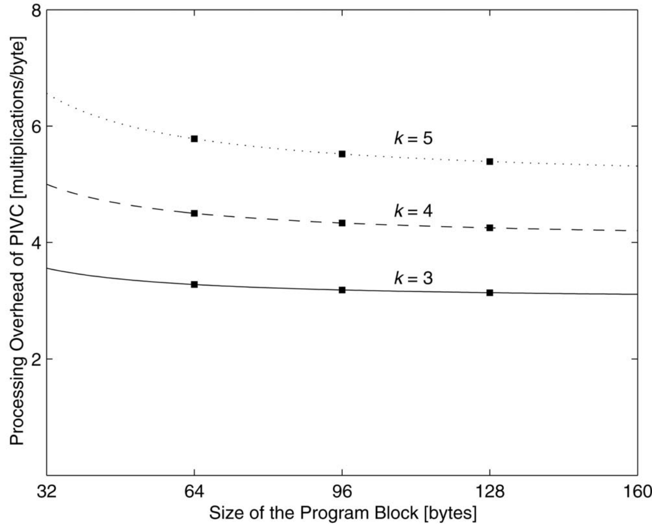

Fig. 8. The processing overhead of PIVC versus m.

TABLE 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

$k = 4$) 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 $k = 4$).

4.3 Processing Overhead

Fig. 8 plots, using (7), P_{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. P_{PIVC} is insensitive to the variations of m, while directly affected by the choice of k, e.g., P_{PIVC} is around 4.33 multiplications per input byte when $k = 4$.

Table 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.

4.4 Trade-Offs

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 trade-off: 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 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	

Table 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 Related Work

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 [5], [6], [7], [8] converts the executable code into an unintelligible form that makes analysis/modification difficult. However, the level of difficulty to tamper with gets substantially lowered as the program becomes smaller and, hence, it cannot protect against determined attackers. Furthermore, as Barak et al. [16] showed, obfuscating programs while preserving its functionality is theoretically impossible. Result checking [9], [10], [11] 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 [12] proposed storing the encrypted executable and decrypting it before execution. However, this scheme suffers from very high overhead of decryption/reencryption 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 [12], [14], [15] aim at detecting changes in the program and taking appropriate actions against those changes, as the program is running. To this end, they use embedded codes (e.g., testers [14] or guards [15]) 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 have 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 dealt with the tamper-proofing of mobile software agents. Wilhelm et al. [41] 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 [42] attempts to detect unauthorized modifications of the 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 to be retained. Blackbox security [43] scrambles the code in such a way that no one can gain a 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 [44] proposed the concept of computing with encrypted functions by which mobile agents can safely compute cryptographic primitives in untrusted computing environments. However, they failed to offer a general scheme for creating mobile agents that encode arbitrary functions. Kotzanikolaou et al. [45] realized Sander's idea 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, PIV 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 [46] presented a software-based scheme to verify authenticity 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 tell its authenticity. 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. [47] 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) into checking if the current memory access is made to a modified location, causing a detectable increase in the hash calculation time. However, this scheme is not efficient as it incurs many more memory accesses than sequential scanning of the program, without guaranteeing 100 percent detection of memory modifications. Moreover, the (random) communication latency in a networked environment may significantly reduce the detectability of this scheme. The PIV is different from [47] in that it accesses each memory location exactly once and allows for byte-oriented processing of program contents, resulting in much faster and more accurate verification.

6 Conclusion

In this paper, we have proposed a soft tamper-proofing scheme based on Program-Integrity Verification (PIV), which offers 1) prevention of manipulation, reverse-engineering, and reprogramming 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 defeats 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.

Acknowledgments

The work reported in this paper is supported in part by the US Office of Naval Research and the US Naval Research Laboratory under Grant No. N00014-04-10726, by the US National Science Foundation under Grant CCR-0329629, by the US Defense Advanced Research Projects Agency under contract F33615-02-C-4031 administered by the Air Force Research Laboratory, and by Cisco Corporation.

References

[1] R. Anderson and M. Kuhn, "Tamper Resistance—A Cautionary Note," *Proc. Second USENIX Workshop Electronic Commerce,* 1996.
[2] D.W. Carman, P.S. Kruus, and B.J. Matt, "Constraints and Approaches for Distributed Sensor Network Security," NAI Labs Technical Report #00-010, Sept. 2000.
[3] R. Anderson, "Why Cryptosystems Fail," *Comm. ACM,* vol. 37, no. 11, Nov. 1994.
[4] S. Blythe, B. Fraboni, S. Lall, H. Ahmed, and U. Riu, "Layout Reconstruction of Complex Silicon Chips," *IEEE J. Solid-State Circuits,* vol. 28, no. 2, Feb. 1993.
[5] C. Collberg, C. Thomborson, and D. Low, "Breaking Abstractions and Unstructuring Data Structures," *Proc. IEEE Int'l Conf. Computer Languages (ICCL '98),* May 1998.
[6] C. Wang, J. Hill, J. Knight, and J. Davidson, "Software Tamper Resistance: Obstructing Static Analysis of Programs," technical report, Dept. of Computer Science, Univ. of Virginia, 2000.
[7] C. Wang, J. Hill, J. Knight, and J. Davidson, "Protection of Software-Based Survivability Mechanisms," *Proc. Int'l Conf. Dependable Systems and Networks,* July 2001.
[8] G. Wroblewski, "General Method of Program Code Obfuscation," *Proc. Int'l Conf. Software Eng. Research and Practice (SERP),* June 2002.
[9] M. Blum and S. Kannan, "Designing Programs that Check Their Work," *J. ACM,* vol. 42, no. 1, 1995.
[10] H. Wasserman and M. Blum, "Software Reliability via Run-Time Result-Checking," *J. ACM,* vol. 44, no. 6, 1997.
[11] F. Ergun, S. Kannan, S.R. Kumar, R. Rubinfeld, and M. Vishwanathan, "Spot-Checkers," *Proc. ACM Symp. Theory of Computing (STOC '98),* May 1998.
[12] D. Aucsmith, "Tamper Resistant Software: An Implementation," *Information Hiding,* pp. 317-333, Springer-Verlag, 1996.
[13] C.S. Collberg and C. Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation—Tools for Software Protection," *IEEE Trans. Software Eng.,* vol. 28, no. 8, Aug. 2002.
[14] B. Horne, L. Matheson, C. Sheehan, and R.E. Tarjan, "Dynamic Self-Checking Techniques for Improved Tamper Resistance," *Proc. First ACM Workshop Digital Rights Management (DRM),* pp. 141-159, 2002.

[15] H. Chang and M.J. Atallah, "Protecting Software Code by Guards," *Proc. Second ACM Workshop Digital Rights Management (DRM),* pp. 160-175, 2002.

[16] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (Im)possibility of Obfuscating Programs," *Proc. 21st Ann. IACR Crypto Conf.,* 2001.

[17] Crossbow Co., "MICA, MICA2 Motes & Sensors," http://www.xbow.com/, 2005.

[18] R.C. Merkle, "A Digital Signature Based on Conventional Encryption Function," *Advances in Cryptology—Proc. Crypto '87,* 1987.

[19] R.C. Merkle, "A Certified Digital Signature," *Advances in Cryptology—Proc. Crypto '89,* 1989.

[20] S. Even, O. Goldreich, and S. Micali, "On-Line/Off-Line Digital Signatures," *Advances in Cryptology—Proc. Crypto '89,* 1989.

[21] G. Poupard and J. Stern, "On the Fly Signatures Based on Factoring," *Proc. ACM Conf. Computer and Comm. Security (CCS 1999),* Nov. 1999.

[22] M. Brown, D. Cheung, D. Hankerson, J.L. Hernandez, M Kirkup, and A. Menezes, "PGP in Constrained Wireless Devices," *Proc. Ninth USENIX Security Symp.,* Aug. 2000.

[23] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC2104, Feb. 1997.

[24] P. Rogaway, "OCB Mode: Parallelizable Authenticated Encryption," http://csrc.nist.gov/encryption/aes/, 2005.

[25] J.P. Hespanha, H.J. Kim, and S. Sastry, "Multiple-Agent Probabilistic Pursuit-Evasion Games," *Proc. 38th Conf. Decision and Control,* Dec. 1999.

[26] R. Vidal, O. Shakernia, H.J. Kim, H. Shim, and S. Sastry, "Probabilistic Pursuit-Evasion Games: Theory, Implementation and Experimental Evaluation," *IEEE Trans. Robotics and Automation,* 2002.

[27] G.L. Duckworth, D.C. Gilbert, and J.E. Barger, "Acoustic Counter-Sniper System," *Proc. SPIE Int'l Symp. Enabling Technologies for Law Enforcement and Security,* 1996.

[28] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson, "Wireless Sensor Networks for Habitat Monitoring," *Proc. ACM Int'l Workshop Wireless Sensor Networks and Applications (WSNA),* Sept. 2002.

[29] S. Madden, R. Szewczyk, M.J. Franklin, and D. Culler, "Supporting Aggregate Queries over Ad-Hoc Wireless Sensor Networks," *Proc. Fourth IEEE Workshop Mobile Computing Systems and Applications (WMCSA),* 2002.

[30] F. Ye, H. Luo, J. Cheng, S. Lu, and L. Zhang, "A Two-Tier Data Dissemination Model for Large-Scale Wireless Sensor Networks," *Proc. IEEE/ACM MobiCom 2002,* 2002.

[31] D.G. Abraham, G.M. Dolan, G.P. Double, and J.V. Stevens, "Transaction Security System," *IBM Systems J.,* vol. 30, no. 2, pp. 206-229, 1991.

[32] S. Kumar and E.H. Spafford, "A Software Architecture to Support Misuse Intrusion Detection," *Proc. 18th Nat'l Information Security Conf.,* 1995.

[33] K. Ilgun, R.A. Kemmerer, and P.A. Porras, "State Transition Analysis: A Rule-Based Intrusion Detection Approach," *IEEE Trans. Software Eng.,* vol. 21, no. 3, pp. 181-199, 1995.

[34] T. Bass, "Intrusion Detection Systems and Multisensor Data Fusion," *Comm. ACM,* Apr. 2000.

[35] Y. Zhang and W. Lee, "Intrusion Detection in Wireless Ad Hoc Networks," *Proc. IEEE/ACM MobiCom 2000,* 2000.

[36] R. Zhang, D. Qian, C. Ba, W. Wu, and X. Guo, "Multi-Agent Based Intrusion Detection Architecture," *Proc. Int'l Conf. Computer Networks and Mobile Computing,* 2001.

[37] N. Courtois, L. Goubin, and J. Patarin, "Quartz, 128-bit Long Digital Signatures," *Proc. Cryptographers' Track of the RSA '2001,* 2001.

[38] N. Courtois, L. Goubin, and J. Patarin, "Flash, A Fast Multivariate Signature Algorithm," *Proc. Cryptographers' Track of the RSA 2001,* 2001.

[39] N. Courtois, L. Goubin, and J. Patarin, "SFLASH, A Fast Asymmetric Signature Scheme for Low-Cost Smartcards," http://www.minrank.org/sflash-b.pdf, 2004.

[40] T. Moh, "A Public Key System with Signature and Master Key Functions," *Comm. Algebra,* vol. 27, no. 5, 1999.

[41] 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, Springer-Verlag, 1999.

[42] G. Vigna, "Cryptographic Traces for Mobile Agents," *Mobile Agents and Security,* 1998.

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

[44] T. Sander and C. Tschudin, "Towards Mobile Cryptography," *Proc. IEEE Symp. Research in Security and Privacy,* 1998.

[45] P. Kotzanikolaou, M. Burmester, and V. Chrissikopoulos, "Secure Transactions with Mobile Agents in Hostile Environments," *Proc. Australasian Conf. Information Security and Privacy,* 2000.

[46] R. Kennell and L.H. Jamieson, "Establishing the Genuinity of Remote Computer Systems," *Proc. 12th USENIX Security Symp.,* Aug. 2003.

[47] A. Seshadri, A. Perrig, L. Doorn, and P. Khosla, "SWATT: SoftWare-Based ATTestation for Embedded Devices," *Proc. IEEE Symp. Security and Privacy,* May 2004.

[48] T. Park and K.G. Shin, "LiSP: A Lightweight Security Protocol for Wireless Sensor Networks," *ACM Trans. Embedded Computing Systems,* vol. 3, no. 3, Aug. 2004.

Taejoon Park (S'04) received the BS (summa cum laude) degree from Hong-Ik University, Seoul, Korea, in 1992 and the MS degree from the Korea Advanced Institute of Science and Technology, Taejon, Korea, in 1994. From 1994 to 2000, he worked at LG Electronics, Seoul, Korea, as a research engineer. Since the Fall of 2000, he has been with the University of Michigan, Ann Arbor, where he is currently a research assistant working toward the PhD degree in electrical engineering and computer science. His research interests include security and routing in sensor/ad hoc/peer-to-peer networks. He is a student member of the IEEE.

Kang G. Shin (S'75-M'78-SM'83-F'92) received the BS degree in electronics engineering from Seoul National University, Seoul, Korea, in 1970 and both the MS and PhD degrees in electrical engineering from Cornell University, Ithaca, New York, in 1976 and 1978, respectively. He is the Kevin and Nancy O'Connor Professor of Computer Science and founding director of the Real-Time Computing Laboratory in the Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor. His current research focuses on QoS-sensitive networking and computing as well as on embedded real-time OS, middleware, and applications, all with emphasis on timeliness and dependability. He has supervised the completion of 51 PhD theses and authored/coauthored around 600 technical papers and numerous book chapters in the areas of distributed real-time computing and control, computer networking, fault-tolerant computing, and intelligent manufacturing. He has coauthored (jointly with C.M. Krishna) *Real-Time Systems* (McGraw Hill, 1997). He has received a number of best paper awards and he has also coauthored papers with his students which have received best student paper awards. From 1978 to 1982, he was on the faculty of Rensselaer Polytechnic Institute, Troy, New York. He has held visiting positions at several universities and in industry. He chaired the Computer Science and Engineering Division, EECS Department, The University of Michigan for three years beginning in January 1991. He is a fellow of the IEEE and the ACM and a member of the Korean Academy of Engineering. He is serving or has served as general chair or program chair for several conferences and symposiums. He has been the guest editor of the 1987 August special issue of the *IEEE Transactions on Computers* on real-time systems, an associate editor of the *IEEE Transactions on Parallel and Distributed Computing*, and an area editor of the *International Journal of Time-Critical Computing Systems*, *Computer Networks*, and the *ACM Transactions on Embedded Systems*. He was a distinguished visitor of the Computer Society of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.