Autonomous and Distributed Node Recovery in Wireless Sensor Networks

Mario Strasser
Computer Engineering and Networks Laboratory
ETH Zurich, Switzerland
strasser@tik.ee.ethz.ch

Harald Vogt
Institute for Pervasive Computing
ETH Zurich, Switzerland
vogt@inf.ethz.ch

ABSTRACT

Intrusion or misbehaviour detection systems are an important and widely accepted security tool in computer and wireless sensor networks. Their aim is to detect misbehaving or faulty nodes in order to take appropriate countermeasures, thus limiting the damage caused by adversaries as well as by hard or software faults. So far, however, once detected, misbehaving nodes have just been isolated from the rest of the sensor network and hence are no longer usable by running applications. In the presence of an adversary or software faults, this proceeding will inevitably lead to an early and complete loss of the whole network.

For this reason, we propose to no longer expel misbehaving nodes, but to recover them into normal operation. In this paper, we address this problem and present a formal specification of what is considered a secure and correct node recovery algorithm together with a distributed algorithm that meets these properties. We discuss its requirements on the soft- and hardware of a node and show how they can be fulfilled with current and upcoming technologies. The algorithm is evaluated analytically as well as by means of extensive simulations, and the findings are compared to the outcome of a real implementation for the BTnode sensor platform. The results show that recovering sensor nodes is an expensive, though feasible and worthwhile task. Moreover, the proposed program code update algorithm is not only secure but also fair and robust.

Categories and Subject Descriptors

C.2.0 [**Computer-communication networks**]: General—*Security and protection*

General Terms

Algorithms, Reliability, Security

Keywords

Wireless Sensor Networks, Node Recovery, Intrusion Detection

SASN'06, October 30, 2006, Alexandria, Virginia, USA.

1. INTRODUCTION

Wireless sensor networks (WSNs) consist of many wireless communicating sensor nodes. Essentially, these are microcontrollers including a communication unit and a power supply, as well as several attached sensors to examine the environment. Sensor nodes typically have very limited computing and storage capacities and can only communicate with their direct neighbourhood. In addition, WSNs have to work unattended most of the time as their operation area cannot or must not be visited. Reasons can be that the area is inhospitable, unwieldy, or ecologically too sensitive for human visitation; or that manual maintenance would just be too expensive.

More and more, WSN applications are supposed to operate in hostile environments, where their communication might be overheard and nodes can be removed or manipulated. Regarding attacks on sensor networks, one differentiates between so called *outsider* and *insider attacks* [21]. In the former, a potential attacker tries to disclose or influence a confidential outcome without participating in its computation; for instance, by intercepting, modifying, or adding messages. In the latter, by contrast, an attacker appears as an adequate member of the WSN by either plausibly impersonating regular nodes or by capturing and compromising them.

Cryptographic methods, such as encrypting or signing messages, are an effective protection against attacks from outside the network, but are of only limited help against insider attacks. Once an adversary possesses one or several valid node identities (including the associated keys), it can actively participate in the operations of the WSN and influence the computed results.

Intrusion or misbehaviour detection systems (IDS), on the other hand, are an important and widely accepted security tool against insider attacks [18, 21]. They allow for the detection of malicious or failed nodes and the application of appropriate countermeasures. So far, however, once detected, misbehaving nodes have just been isolated from the rest of the network and hence are no longer usable by running applications. In the presence of an adversary or software faults, this proceding will inevitably result in an early and complete loss of the whole network. Therefore, not only the detection of misbehaving nodes is important, but also the selection and application of effective countermeasures. Their aim must not be to simply expel suspected nodes but to recover them into correct operation. In combination, the advantages of an IDS together with the appropriate recov-

ery measures are manifold. Not only do they help in case of program faults (e.g., deadlocks or crashes) but even if an attacker manages to capture a node and to abuse it for his own purposes, there is a chance that the aberrant behaviour of this node will be detected and the node be recovered, thus nullifying the attack. However, due to the size of sensor networks, both the IDS functionality as well as the recovery measures should be autonomously executed by the involved nodes in a distributed and cooperative manner and without the need for central instances with extended functionality.

Motivated by the above mentioned insights, this paper focuses on autonomous an distributed node recovery in wireless sensor networks and proposes three alternative countermeasures to node expelling; namely to switch a node off, to restart it, and to update its program code. We formally specify what we consider a secure and correct recovery algorithm, present a distributed algorithm which meets these properties, and reason why it can help to extend the overall lifetime of a sensor network. In addition, we discuss the limitations of the proposed countermeasures, show which hard- and software parts of a corrupted node must still work correctly to make them applicable, and explain how this can be achieved with current and upcoming technologies. More precisely, the contributions of this paper are as follows:

- We propose to no longer expel misbehaving nodes, but to either (i) switch them off, (ii) restart them, or (iii) update their program code.
- We give a formal specification of a secure and correct recovery algorithm.
- We present a provably secure and robust distributed node recovery algorithm.
- We discuss the requirements on the soft- and hardware of a node in order to make the countermeasures applicable and show how they can be fulfilled with current and upcoming technologies.

The algorithm is evaluated analytically as well as by means of extensive simulations and the findings are compared to the outcome of a real implementation for the BTnode sensor platform. The results show that recovering sensor nodes is an expensive, though feasible and worthwhile task. Moreover, the proposed program code update algorithm is not only provably secure but also fair and robust. It distributes the update load equally over all participating nodes and terminates as long as at least one of them remains correct.

The rest of this paper is organised as follows. Section 1.1 presents the related work in the area of intrusion detection and node recovery in wireless sensor networks. Section 2 states the required definitions and assumptions. Section 3 specifies the proposed recovery algorithm whose correctness is proven in section 4. The algorithm is evaluated in section 5 and section 6 concludes the paper.

1.1 Related Work

In this section, we present related work in the area of intrusion detection in wireless sensor networks. Additionally, related work regarding program code updates in sensor networks is also discussed, as we propose program code updates as a mean to recover nodes.

Intrusion Detection

In recent years, intrusion detection systems for wireless sensor networks have become a major research issue and several approaches have been proposed. However, to our best knowledge, the only countermeasure applied so far was to (logically) exclude malicious nodes.

Khalil, Bagchi, and Nina-Rotaru present a distributed IDS where nodes monitor the communication among their neighbours [14]. For each monitored node a malignity counter is maintained and incremented whenever the designated node misbehaves. Once a counter exceeds a predefined threshold, an according alert is sent to all neighbours and if enough alerts are received the accused node is revoked from the neighbourhood list. Hsin and Liu suggest a two-phase timeout system for neighbour monitoring which uses active probing to reduce the probability of false-positives [10].

A rule-based IDS, which proceeds in three phases, is proposed by da Silva et al. [5]. In the first phase, messages are overheard and the collected information is filtered and ordered. In the second phase, the detection rules are applied to the gathered data and each inconsistency counted as a failure. Finally, in the third phase, the number or failures is compared to the expected amount of occasional failures and if too high an intrusion alert is raised.

Inverardi, Mostarda and Navarra introduce a framework which enables the automatic translation of IDS specifications into program code [12]. The so generated code is then installed on the sensor nodes in order to locally detect violations of the node interaction policies. In the approach by Herbert et al. [6], predefined correctness properties (invariants) are associated with conditions of individual nodes or the whole network and program code to verify these invariants is automatically inserted during compilation.

A reputation-based IDS framework for WSNs where sensor nodes maintain a reputation for other nodes is presented by Ganeriwal and Srivastava [8]. It uses a Bayesian formulation for reputation representation, update, and integration.

Program Code Update

The main difference between the already available reprogramming algorithms and the proposed recovery measures are that the former focus on the propagation of new program releases among all nodes of the network, whereas the aim of the latter is the local and autonomous update of a single node. Furthermore, most reprogramming mechanisms do not care about security at all, or rely on expensive public key cryptography.

Kulkarni and Wang propose a multihop reprogramming service for wireless sensor networks which uses a sender selection algorithm to avoid collisions [16]. Impala, a middleware system for managing sensor systems is presented by Liu and Martonosi [17]. Its modular architecture supports updates to the running system. An application consists of several modules which are independently transferred; an update is complete if all its modules have been received. Jeong and Culler introduce an efficient incremental network programming mechanism [13]. Thanks to the usage of the Rsync algorithm, only incremental changes to the new program must be transferred.

A secure dissemination algorithm to distribute new program releases among nodes is presented by Dutta et al. [7]. Program binaries are propagated as a sequence of data blocks of which the first is authenticated with the private key of

the base station and the subsequent ones by means of a hash chain. In order to improve the fault tolerance of the sensor network, nodes use a grenade timer to reboot periodically. During the boot process neighboring nodes are asked whether a new program release is available and if so, its download is initiated.

2. DEFINITIONS

In this section, we define our assumptions regarding the observation of nodes and the network communication model. We specify the capabilities of a potential adversary, explain what we consider a correct recover algorithm, and discuss the requirements on the hard- and software of a sensor node.

2.1 Intrusion and Misbehaviour Detection

Troughout this paper, we assume that the network is divided into N_C so called *observation clusters* $C_i = (V_i, E_i)$, $0 \leq i < N_C$ of size n, $n = |V_i|$. Within a cluster each node is connected to each other ($\forall v_i, v_j \in V_k,\ v_i \neq v_j : \{v_i, v_j\} \in E_k$) and observes the behaviour of its cluster neighbours. For the actual monitoring of the neighbours an arbitrary IDS can be used, as long as each node ends up with an (individual) decision about whether a certain node behaves correct or malicious. The set of malicious nodes in a cluster is denoted by M_i and their number by t, $t = |M_i| \leq n$.

2.2 Network Model

In the following, p_s (p_r) denotes the probability that the sending (receiving) of a message fails. Thus, for $0 \leq p_s, p_l < 1$ the resulting probability for an unsuccessful transmission (packet loss ratio, PLR) is

$$p_l := 1 - (1 - p_s)(1 - p_r) = p_s + p_r + p_s p_r$$

Additionally, we assume that there exists a constant upper bound $\tau_p \in O(1)$ on the transmission time of a message.

2.3 Adversary Model

We consider an omnipresent but computationally bounded adversary who can perform both outsider as well as insider attacks. This means that a potential adversary is able to intercept and create arbitrary messages but unable to decrypt or authenticate messages for which he does not possess the required keys. We further assume that nodes can be either logically (i.e., by exploiting a software bug) or physically captured. However, the time to compromise a node physically is considered non-negligible (i.e., it takes some time to move from node to node and to perform the physical manipulations) and to not significantly decrease with the number of already captured nodes.

2.4 Hard- and Software Requirements

To all presented recovery measures applies that they are only applicable if at least the therefore needed systems of the corrupt node – in the following denoted as the *recovery system* – still work correctly. In order to achieve this, one has to make sure that the recovery system is logically and, if feasible, physically protected.

Logical Protection of the Recovery System

Logical protection means that it should not be possible for a running application to prevent the execution of the recovery procedures. That is, if the program code running on a node has crashed or been corrupted by an adversary (e.g., by exploiting a security hole), this should not affect the integrity and availability of the recovery system.

One mechanism to achieve this is to set up a hardware interrupt which cannot be suppressed or redirected by the application and by locating the dedicated interrupt routine in a write protected memory area. Consequently, on each interrupt request, control is handed over to the immutable interrupt routine an thus to the recovery system. A simple variant of this mechanism in which a grenade timer periodically reboots the system and the bootloader is located in read only memory (ROM) is used by Dutta et al. [7]. Another approach would be to misuse some additionally available MCUs [23], for example the ARM CPU on the ARM-based Bluetooth module on the BTnode. Some of these MCUs are powerful enough to take on additional tasks like monitoring the main MCUs activities or rewriting the application memory. In case of the BTnode that extra MCU is directly responsible for communication and thus it would be guaranteed that it has access to all received packets as well.

On more advanced systems, mechanisms as provided by Intel's protected mode (e.g., isolated memory areas, privilege levels, etc.) could be used to protect the recovery system more efficiently. Current technologies such as ARM's TrustZone [1] for embedded devices or Intel's LaGrande technology [11] go even further and enable a comprehensive protection of the CPU, memory, and peripherals from software attacks.

Physical Protection of the Recovery System

The physical protection of current sensor node platforms is very poor because of their focus on simple maintenance [9]. However, although it is generally agreed that entirely tamper-proof sensor nodes would be too expensive, current trends in the hardware development of embedded devices indicate that some level of physical protection will be available in the near future [20, 15]. Security mechanisms regarding the packaging of sensor nodes as, for instance, those proposed by FIPS 140-2 level 2 [19] could already significantly increase the cost for an adversary. For integrity and not confidentiality is the main concern with the recovery module, it has only to be protected against manipulations but not against unintended disclosure or side-channel attacks. In fact, it would be sufficient to have mechanisms which render a node useless if the case of the recovery system was opened; complete tamper resistance is not required.

2.5 Correct Node Recover Algorithms

A node recovery algorithm for a cluster $C_i = (V_i, E_i)$ is considered correct if the following liveness and safety properties hold:

L1 If all correct nodes ($V_i \setminus M_i$) accuse a node $m \in V_i$ to be faulty or malicious, its recovery process will finally be initiated with high probability.

L2 Once the recovery process for a node $m \in V_i$ has been initiated, it will eventually terminate as long as there remain at least $k \geq 1$ correct nodes $\subseteq V_i \setminus M_i$.

S1 If no more than $\frac{n-1}{3}$ correct nodes (i.e., a minority) accuse another correct node $v \in V_i \setminus M_i$ the recovery process will not be initiated.

S2 After the recovery process, a node $m \in V_i$ must either (i) be halted, (ii) contain the same program code as before, or (iii) contain the correct program code.

The two liveness properties L1 and L2 ensure that each malicious node is recovered if its aberrant behaviour is detected by enough neighbours. Safety property S1 is required to make sure that a node is only recovered if a majority of correct nodes accuses it and property S2 ensures that things are not worsened by applying the recovery process.

3. DISTRIBUTED NODE RECOVERY

In this section, we present a distributed node recovery algorithm which is autonomously executed within an observation cluster. The supported recovery measures are: node shutdown, node restart, and program code update. As long as the recovery module of an otherwise faulty or malicious node is still intact, it is tried to recover it by restarting it or updating its program code; or to at least eliminate its interfering influence by turning it off. If a node does not respond to any of these measures, it is still possible to logically expell it; preferably by means of a reliable majority decision [22] to avoid inconsistencies among the cluster members.

3.1 Description of the Recovery Procedure

The proposed recovery algorithm consists of two phases. In the first, so called *accusation phase*, nodes accuse all neighbours which are regarded as being malicious. If a node is accused by at least two third of its neighbours it initiates the second, so called *recovery phase*, during which the actual countermeasures are executed. To simplify the cooperative program code update, the program memory of a node is divided into F frames f_i, $0 \leq i < F$ of size fs. Additionally, for each frame f_i its corresponding hash value $h_i := h(f_i)$ is computed.

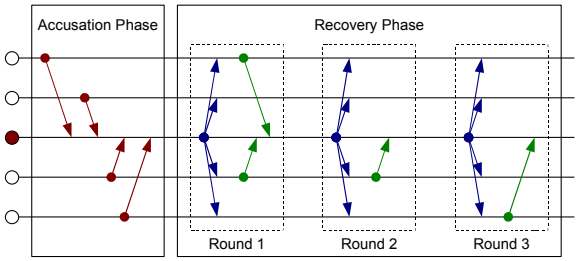

Figure 1: Schematic depiction of a recovery procedure which performs a program update as the countermeasure.

Accusation Phase

Nodes which conclude that one of their neighbours behaves maliciously, send it an authenticated accusation message[1]. The proposed countermeasure depends on the observed aberration and can be either of type *shutdown*, *reset*, or *update*, if the node should be halted, restarted, or its program code updated, respectively. Accusation messages have to be acknowledged and are resent up to r times otherwise.

[1]For simplicity, it is assumed that nodes can accuse their neighbours at any time. However, if the recovery module is only active from time to time, nodes could of course also actively ask for (pending) accusations.

In case that a program update is requested, the accusation messages also include a list of the sender's F frame hash values h_i. They represent the current state of its program memory and are required to deduce the correct program code. Therefore, for each frame f_i not only its hash value h_i but also a counter c_i, which is initialised with zero, is stored. Upon reception of a accusation message, each included hash value is compared to the already stored one and if they are equal, c_i is incremented by one. If they differ and $c_i > 0$ the counter is decremented by one; otherwise (i.e., they are not equal and $c_i = 0$) the stored hash value is replaced with the received value. This procedure ensures that, for $3t < n - 1$, every h_i will contain the hash value of the correct program code frame after $\geq \frac{2(n-1)}{3}$ accusations have been received (see Proof 4).

Recovery Phase

When a node m has received $\geq \frac{2(n-1)}{3}$ accusations of a certain recovery type, the corresponding measure is initiated. In the non trivial case of a distributed program code update, the correct program code has therefore to be downloaded from the neighboring nodes. Otherwise, the node is just rebooted or shutdown and no further communication or coordination is required.

The autonomous program code transfer is performed in rounds of which each starts with the broadcasting of an authenticated *update request* message by the accused node m. Essentially, the message contains a list of so called *frame descriptors* (u_i, Q_i), consisting of a node id u_i and a set of requested frame numbers $Q_i := \{r_0, r_1, \ldots, r_{|Q|-1}\}$. Upon reception of a valid request, a node v seeks for descriptors which contain its own id (i.e., $u_i = v$). If present, for each requested frame number $r_j \in Q_i$ the corresponding program code frame is sent back to m with an *update* message. All received program code fragments f_i, in turn, are verified by m using the stored hash values h_i. Valid code fragments are copied into the program memory[2] and the frame marked as updated. If for a duration of τ_{round} no update messages arrive although there are still some outstanding frames, a new update request message is broadcasted and the next round initiated. As soon as all frames have been received, the node is rebooted and thus the new program code activated.

In order to distribute the transfer load equally among all participating nodes and to ensure that the update procedure terminates if at least one correct node is available, the frame descriptors are determined as follows: First, the $n-1$ participating nodes are ordered such that $id(v_0) < id(v_1) < \ldots < id(v_{n-2})$. Next, the F memory frames are divided into $n-1$ sectors of length $l := \lceil \frac{F}{n-1} \rceil$. Finally, to each node one such fragment is assigned per update round in a round robin fashion. Thus, in round i node v_j, is responsible for the segment $s := j + i \bmod (n-1)$, that is, for the frames sl to $\min((s+1)l - 1, F - 1)$. In the first round, for example, the first node is responsible for the first l frames, the second node for the second l frames and so on. In the second round, however, the assignment is rotated by one and thus the outstanding frames of the first sector are now requested from the second node. This process has to be continued until all required frames have been received.

[2]On most sensor node platforms, new code is not directly written into program memory but into a therefore available Flash memory and installed during a subsequent reboot.

Extensions and Optimisations

Even though not all but only the subset of modified program code frames has to be requested, updating a node is still a time consuming and expensive task. Consequently, the amount of update load that a specific node can cause should be restricted, for instance by limiting the number of update messages that are sent to it. To further reduce the load for the participating nodes, the F hash values h_i in an accusation message can be replaced by the hash value $\overline{h} := h(h_0||h_1||\ldots||h_{F-1})$. Once the correct value $\overline{h}$ has been determinded using the corresponding counter $\overline{c}$ in analogy to the above mentioned algorithm, the actual hash values can be requested from the neighbours in a second step and verified with $\overline{h}$. In order to decrease the total number of required accusation messages, more than one recovery measure per message should be allowed. Alternatively, the measures could be hierarchically organised, having the type update also counting as a reboot or shutdown request.

3.2 Algorithms

Listing 1: Algorithm for an accusing node v.

```
var
  acc_retries[n-1] := {0,...,0}
  acc_failed[n-1]  := {FALSE,...,FALSE}
  num_updates[n-1] := {0,...,0}

upon misbehavior detection of node m
  choose an appropriate accusation-type a_m
  if a_m = ACC_UPDATE
    send ⟨accusation, v, m, τ, a_m, {h(f_0),...,h(f_{F-1})}⟩
    to m
  else
    send ⟨accusation, v, m, τ, a_m⟩ to m
  start timer A_m

upon reception of ⟨accusation_ack, m, τ, a⟩ from m
  stop timer A_m
  acc_retries[m] := 0

upon timeout of timer A_m
  if acc_retries[m] < MAX_ACC_RETRIES
    acc_retries[m] := acc_retries[m] + 1
    send ⟨accusation, v, m, τ, a_m, {h(f_0),...,h(f_{F-1})}⟩
    to m
    start timer A_m
  else
    acc_failed[m] := TRUE

upon reception of ⟨update_request, m, τ, R⟩ from m
  if num_updates[m] < max_updates and
    (u, {r_0,...,r_k}) ∈ R
    num_updates[m] := num_updates[m] + 1
    ∀r_i, 0 ≤ i ≤ k send ⟨update, v, m, r_i, f_{r_i}⟩ to m
```

Listing 2: Algorithm for the accused node m.

```
var
  updating          := FALSE
  num_acc_reset     := 0
  num_acc_update    := 0
  num_acc_shutdown  := 0
  start_node        := 0
  acc_reset_recvd[n-1]    := {0,...,0}
  acc_update_recvd[n-1]   := {0,...,0}
  acc_shutdow_recvd[n-1]  := {0,...,0}
  frame_updated[F-1]  := {FALSE,...,FALSE}
  frame_digest[F-1]   := {h(f_0),...,h(f_{F-1})}
  frame_count[F-1]    := {0,...,0}

upon reception of ⟨accusation, v, m, τ, ACC_RESET⟩
    from v
  send ⟨accusation_ack, m, τ, ACC_RESET⟩ to v
  if not updating and not acc_reset_recvd[v]
    acc_reset_recvd[v] := TRUE
    num_acc_reset := num_acc_reset + 1
  if not updating and num_acc_reset ≥ 2(n-1)/3
    reset node

upon reception of ⟨accusation, v, m, τ, ACC_SHUTDOWN
    ⟩ from v
  send ⟨accusation_ack, m, τ, ACC_SHUTDOWN⟩ to v
  if not updating and not acc_shutdown_recvd[v]
    acc_shutdown_recvd[v] := TRUE
    num_acc_shutdown := num_acc_shutdown + 1
  if not updating and num_acc_shutdown ≥ 2(n-1)/3
    shutdown node

function setup_update_request()
  k := 0
  R := {}
  for 0 ≤ i < n, i ≠ m
    w := (start_node + i) mod n
    Q := {}
    for 0 ≤ j < ⌈F/n⌉
      if not frame_updated[k]
        Q := Q ∪ {k}
      k := k + 1
    if Q ≠ {}
      R := R ∪ {(w, Q)}
  start_node := start_node + 1
  return R

upon reception of ⟨accusation, v, m, τ, ACC_UPDATE
    , {h_0,...,h_{F-1}}⟩ from v
  send ⟨accusation_ack, m, τ, ACC_UPDATE⟩ to v
  if not updating and not acc_update_recvd[v]
    acc_update_recvd[v] := TRUE
    num_acc_update := num_acc_update + 1
    for 0 ≤ i < F
      if frame_digest[i] = h_i
        frame_count[i] := frame_count[i] + 1
      else
        if frame_count[i] > 0
          frame_count[i] := frame_count[i] - 1
        else
          frame_digest[i] := h_i
  if not updating and num_acc_update ≥ 2(n-1)/3
    R := setup_update_request()
    broadcast ⟨update_request, m, τ, R⟩
    start timer U
    updating = TRUE

upon timeout of timer U
  R := setup_update_request()
  broadcast ⟨update_request, m, τ, R⟩
  start timer U

upon reception of ⟨update, v, m, i, f⟩ from v
  reset timer U
```

```
if h(f) = frame_digest[i] and not
  frame_updated[i]
  update memory frame i
  frame_updated[i] := TRUE
if ∀i, 0 ≤ i < F frame_updated[i]
  reset node
```

4. PROOF OF CORRECTNESS

In this section, we proof the correctness of the proposed algorithm with respect to the specifications of section 2.

THEOREM 1. *Given the network and adversary model specified in section 2, the proposed recovery algorithm is correct and fulfils the properties L1, L2, S1, and S2 if the recovery module of the accused node m is intact, if h() is a secure hashfunction, and if less than one third of the participating nodes are malicious (i.e., $3t < n - 1$).*

In order to prove Theorem 1 we have to show that the properties L1, L2, S1, and S2 hold. We therefore first prove some helper Lemmas.

LEMMA 1. *If all correct nodes accuse a node m, its recovery process will be initiated with high probability.*

PROOF. The probability that less than $\frac{2(n-1)}{3}$ accusations are received is equal to the probability that more than $\frac{n-1}{3}$ messages are either not sent or lost. Assuming that the t malicious nodes do not participate in the distributed update, at least $\frac{n-1}{3} - t + 1$ accusations must get lost. Given $0 \leq p_l < 1$, the probability for this is $\leq (p_l^r)^{\frac{n-1}{3}-t+1} + (p_l^r)^{\frac{n-1}{3}-t+2} + \ldots + (p_l^r)^{n-1} \leq \frac{2(n-1)+3t}{3} (p_l^r)^{\frac{n-1}{3}-t+1}$. It holds that $\forall c > 1 : \exists r \geq 1$ such that $\frac{2(n-1)+3t}{3} \left(p_l^{\frac{n-1}{3}-t+1} \right)^r < n^{-c}$. Thus, the node m gets $\geq \frac{2(n-1)}{3}$ accusations w.h.p. and the recovery process is initiated. □

LEMMA 2. *Once the recovery process for a node m has been initiated it will eventually terminate as long as there remain at least $k \geq 1$ correct nodes.*

PROOF. In order that a frame is updated in a specific round, the dedicated request as well as its actual transmission must succeed. The probability that this is the case is $(1 - p_l)^2$. With only one correct node ($k = 1$) the expected number of update rounds per frame a can be described as a Markov chain described by the expression $a = (a + 1)(1 - (1 - p_l)^2) + (1 - p_l)^2$ with the solution $a = \frac{1}{(1-p_l)^2}$. The overall expected number of rounds is thus $aF = \frac{F}{(1-p_l)^2} \in O(1)$. In each round at most one request and F updates are transmitted, leading to an upper bound for its duration of $(F + 1)\tau_p \in O(1)$. Altogether, the expected worst case duration is $\frac{F(F+1)}{(1-p_l)^2}\tau_p \in O(1)$. □

LEMMA 3. *If no more than $\frac{n-1}{3}$ correct nodes accuse another correct node $v \notin M$ the recovery process will not be initiated.*

PROOF. From each node only one accusation is accepted and thus the number of valid accusations is at most $\frac{n-1}{3} + t < \frac{2(n-1)}{3}$. □

LEMMA 4. *At the start of a program code update the target node m has stored the correct hash value h_i for all frames, given that all correct nodes have loaded the same program code.*

PROOF. Let's assume that there is a hash value h_i which is not correct when the program code update starts. As a stored hash value is only substituted if the dedicated counter c_i is zero, the node must have received at least as many wrong values as correct ones. From each node only one accusation is accepted, thus of the $a \geq \frac{2(n-1)}{3}$ received values at most $t < \frac{n-1}{3}$ are false. It follows that at least $a - t > \frac{2(n-1)}{3} - \frac{n-1}{3} = \frac{n-1}{3} > t$ values must be correct, which contradicts the assumption that at least as many false as correct hash values were received. □

The properties L1, L2, and S1 are proven by Lemma 1, 2, and 3, respectively. If the accused node is turned off or restarted, property S2 holds by definition. Otherwise, if the program code is updated, Lemma 4 and property L2 guarantee that only correct code frames are installed and that the procedure finally terminates.

5. EVALUATION

In this section we provide an analytical evaluation of the proposed algorithm and present the findings of extensive simulations as well as of a real implementation for the BTnodes. The evaluated metrics are: (a) number of update rounds, (b) update load for the accused nodes, (c) update load for the other participating nodes, and (d) update duration.

5.1 Analytical Evaluation

Number of Update Rounds

In order to update a frame it is required that the dedicated request as well as the actual frame itself are successfully transmitted. The expected fraction of erroneous updates is therefore $1 - (1 - p_l)^2$. If one further assumes that the t malicious nodes do not participate in the program update, the fraction increases to $1 - \frac{n-1-t}{n-1}(1 - p_l)^2$. Thus, the expected number of outstanding frames after the first and second update round are $E_f(1) = F(1 - \frac{n-1-t}{n-1}(1 - p_l)^2)$ and $E_f(2) = E_f(1)(1 - \frac{n-1-t}{n-1}(1-p_l)^2) = F(1 - \frac{n-1-t}{n-1}(1-p_l)^2)^2$, respectively. In general, the expected number of outstanding frames after $i > 0$ rounds is:

$$\begin{aligned} E_f(i) &= E_f(i-1)\left(1 - \frac{n-1-t}{n-1}(1-p_l)^2\right) \\ &= F\left(1 - \frac{n-1-t}{n-1}(1-p_l)^2\right)^i \end{aligned}$$

Consequently, the expected number of update rounds is

$$E_r \approx \frac{\log(0.5) - \log(F)}{\log(1 - \frac{n-1-t}{n-1}(1-p_l)^2)}$$

For reliable connections ($p_l = 0$) we get $E_r \approx \frac{\log(0.5)-\log(F)}{\log(1/3)} \in O(1)$. In a worst case scenario a continuous sequence of frames is assigned to the t malicious nodes and thus at least $t + 1$, that is $O(t) = O(n)$ rounds are required. Moreover, for a fixed p_l, the expected number of rounds is (almost) independent of the cluster size n as $\frac{2}{3} < \frac{n-1-t}{n-1} \leq 1$ for a fixed t, $0 \leq t < \frac{n-1}{3}$.

Update Load

The expected amount of data (in bytes) to transfer for the accused node is

$$E_{tm} = \sum_{i=0}^{\lceil E_r \rceil - 1} \left(C_{req} + C_{mac}(n-1) + C_{sel}(n-1)\frac{E_f(i)}{F} \right) \leq (C_{req} + C_{mac}(n-1) + C_{sel}(n-1))E_r$$

and

$$E_{tv} = \sum_{i=0}^{\lceil E_r \rceil - 1} \frac{E_f(i)}{n-1}(1-p_l)(C_{update} + fs) \leq \frac{F}{n-1}(1-p_l)(C_{update} + fs)E_r$$

for the other participating nodes. In the above expressions $(n-1)\frac{E_f(i)}{F}$ is the expected number of addressed nodes and $\frac{E_f(i)}{n-1}(1-p_l)$ expresses the expected number of successfully requested frames per node.

Update Duration

The total number of sent messages E_m is bound by $(F+1)E_r \in O(E_r)$ as there are only one request and no more that F update messages per round. Thus, the expected value of E_m is in $O(1)$ for reliable connections and in $O(n)$ in the worst case. More precisely, the expected number of messages is given by

$$E_m = \sum_{i=0}^{\lceil E_r \rceil - 1} \left(1 + \frac{E_f(i)}{n-1}(1-p_l)\right) = E_r + \frac{(n-1-t)E_{tv}}{C_{update} + fs}$$

Neglecting the delays caused by the involved software routines, a good approximation for the update duration can be achieved by considering the overall transfer time and the delays caused by the round timeouts. The expected time to transfer all messages is $\frac{E_{tm}+(n-1-t)E_{tv}}{B} + E_m\tau_{mac}$ whereas the overhead of the round timer is given by $(E_r-1)\tau_{round}$, resulting in a total update duration of

$$E_d \approx \frac{E_{tm} + (n-1-t)E_{tv}}{B} + E_m\tau_{mac} + (E_r-1)\tau_{round}$$

Parametrisation

For the comparison of the analytical results with the simulation and implementation of the algorithm, the following parameters were used:

Baudrate	B	19.2 kBit/s
B-MAC preamble	τ_{mac}	100 ms
Round timeout	τ_{round}	3 s
Request header size	C_{req}	12 Bytes
Update header size	C_{update}	11 Bytes
Frame selector site	C_{sel}	10 Bytes
MAC size	C_{mac}	20 Bytes
Frame size	fs	1024 Bytes

5.2 Simulation

The simulation of the algorithm was carried out with the Java based JiST/SWANS simulator [2]. In order to make the results comparable to the real BTnode implementation, the radio module was set up according to the characteristics of

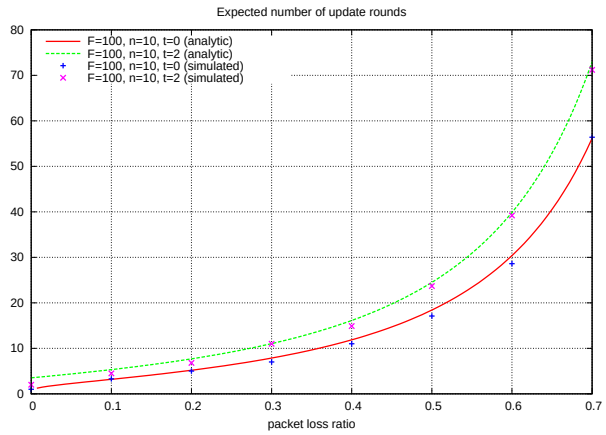

Figure 2: Expected number of rounds to update a node.

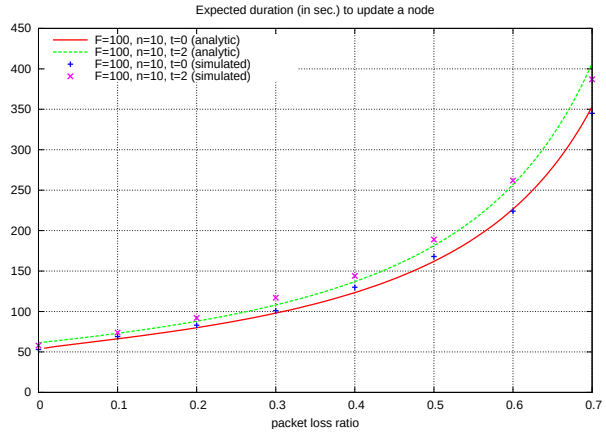

Figure 3: Expected duration to update a node.

the Chipcon CC1000 transceiver [4] and B-MAC was chosen as the data link layer protocol. The complete parametrisation of the simulation is given in the table below:

Transmission frequency	868 MHz
Transmission power	5 dBm
Receiver sensitivity	-100 dBm
Memory size	100 kByte
Number of nodes	10
Deployment area	20 x 20 m (u.r.d.)

5.3 Implementation

In addition to the above mentioned simulations, the algorithm was also implemented for the BTnodes, a wireless sensor platform running NutOS [3]. A detailed description of the created software is omitted due to space reasons but can be found in [22]. The implementation was evaluated by randomly distributing a cluster of 10 nodes in a field of 20 x 20 m, whereupon each node in turn initiated a complete program update. Altogether, over 100 recovery procedures where measured.

5.4 Results

The packet loss ratio has, as expected, a significant effect on all evaluated metrics and each of them increases exponentially if the ratio worsens. The number of nodes, in contrast,

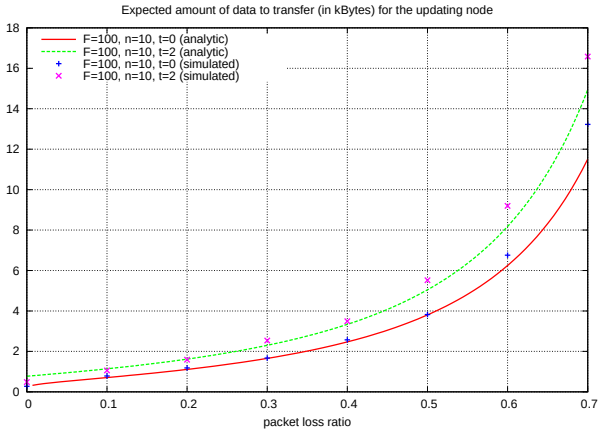

Figure 4: Expected update load for the accused node.

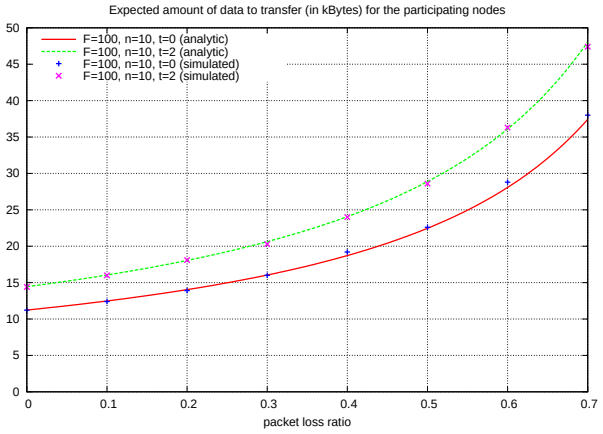

Figure 5: Expected update load for the participating nodes.

has for a fixed packet loss ratio almost no negative impact on the evaluated metrics, showing that the algorithm itself scales well. Furthermore, the results show that the update algorithm is fair and equally distributes the update load over all participating nodes.

Update Rounds and Update Duration

Whilst the expected number of update rounds (see Figure 2) is only of secondary importance, the update duration (see Figure 3) is of major interest for the feasibility of the algorithm. The faster a node recovery is completed, the sooner the network is operable again. Even though the update duration almost triples from 50 to 150 s if the packet loss ratio increases from 0 to 40 percent, it is still in a range which most WSN application should be able to cope with.

Update Load

In a cluster of 10 nodes the update load for the accused node (see Figure 4) is 0.5 to 3.5 kByte for $0 \leq p_l \leq 0.4$ and thus considerably smaller than for the other participating nodes (see Figure 5) with a load of 12 to 24 kByte. However, the latter is, as expected, inverse proportional to the cluster size (see Figure 7): the larger a cluster and the lower the number of malicious nodes, the smaller the expected update load per participating node.

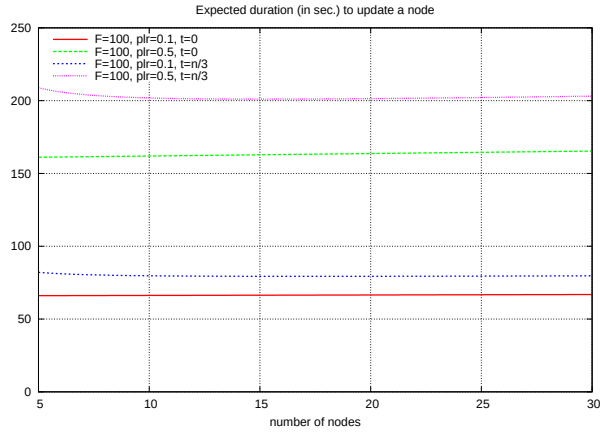

Figure 6: Influence of the cluster size on the update duration.

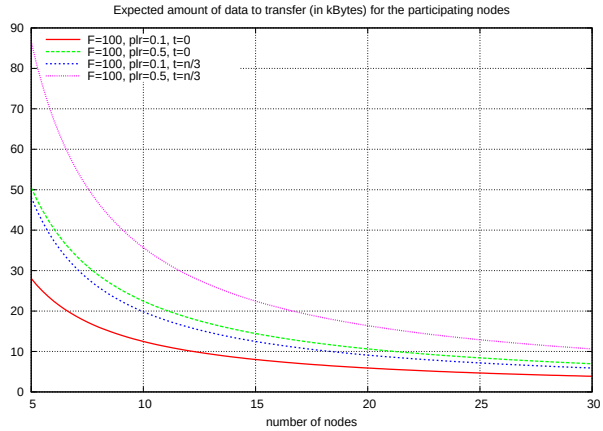

Figure 7: Influence of the cluster size on the expected update load for the participating nodes.

Implementation

In the experiments conducted with the BTnode implementation, the average number of update rounds was five, the update duration about 100 s ($\sigma \approx 10$ s), and the load for the participating nodes about 15 kByte ($\sigma \approx 1$ kByte). Applied to the analytical model this would mean that the gross packet loss ratio was roughly 20%.

6. SUMMARY AND CONCLUSIONS

In this paper, we presented an autonomous and distributed recovery algorithm for sensor networks. The algorithm allows for bringing malicious or failed nodes back into normal operation or, at least, for securely shutting them down. Particularly in remote or unwieldy areas, such as deserts, the bottom of the sea, mountains, or even on planets in outer space, where redeployment is expensive and sensor nodes cannot easily be exchanged or maintained, the application of a node recovery system is most likely to extend the lifetime of the whole network.

The results of the simulation and analytical analysis were confirmed by the real BTnode implementation. They show that recovering sensor nodes is – as any form of reprogramming – an expensive, though feasible task. Moreover, the proposed program code update algorithm is not only prov-

ably secure but also fair and robust. It distributes the update load equally over all participating nodes and terminates as long as at least one of the nodes remains correct.

To all presented recovery measures applies that they are only applicable if at least the therefore needed systems of the corrupt node still work correctly. However, although it is generally agreed that entirely tamper-proof sensor nodes are too expensive, current trends in the hardware development of embedded devices indicate that at least some logical and physical protection (e.g., CPUs which support isolated memory areas or automatic memory erason if a node is tempered with) will be available in the near future. We discussed how these upcoming technologies can be exploited to protect the recovery mechanisms of a sensor node and what is already feasible with existing systems.

7. REFERENCES

[1] T. Alves and D. Felton. *TrustZone: Integrated Hardware and Software Security*. ARM Ltd, July 2004.

[2] R. Barr, Z. J. Haas, and R. van Renesse. Jist: an efficient approach to simulation using virtual machines: Research articles. *Softw. Pract. Exper.*, 35(6):539–576, 2005.

[3] J. Beutel, O. Kasten, and M. Ringwald. Poster abstract: Btnodes – a distributed platform for sensor nodes. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, pages 292 – 293, Los Angeles, California, USA, Jan. 2003. ACM Press. http://www.btnode.ethz.ch/.

[4] Chipcon AS, Oslo, Norway. *Single Chip Very Low Power RF Transceiver, Rev. 2.1*, Apr. 2002. http://www.chipcon.com/.

[5] A. P. R. da Silva, M. H. T. Martins, B. P. S. Rocha, A. A. F. Loureiro, L. B. Ruiz, and H. C. Wong. Decentralized intrusion detection in wireless sensor networks. In *Q2SWinet '05: Proceedings of the 1st ACM international workshop on Quality of service & security in wireless and mobile networks*, pages 16–23, New York, NY, USA, 2005. ACM Press.

[6] S. B. Douglas Herbert, Yung-Hsiang Lu and Z. Li. Detection and repair of software errors in hierarchical sensor networks. *To appear in IEEE conference on Sensor Networks and Ubiquitous Trustworthy Computing (SUTC)*, June 2006.

[7] P. K. Dutta, J. W. Hui, D. C. Chu, and D. E. Culler. Towards secure network programming and recovery in wireless sensor networks. Technical Report UCB/EECS-2005-7, Electrical Engineering and Computer Sciences University of California at Berkeley, Oct. 2005.

[8] S. Ganeriwal and M. B. Srivastava. Reputation-based framework for high integrity sensor networks. In *SASN '04: Proceedings of the 2nd ACM workshop on Security of ad hoc and sensor networks*, pages 66–77, New York, NY, USA, 2004. ACM Press.

[9] C. Hartung, J. Balasalle, and R. Han. Node compromise in sensor networks: The need for secure systems. Technical Report CU-CS-990-05, Department of Computer Science, University of Colorado, Jan. 2005.

[10] C. Hsin and M. Liu. A distributed monitoring mechanism for wireless sensor networks. In *WiSE '02: Proceedings of the 3rd ACM workshop on Wireless security*, pages 57–66, New York, NY, USA, 2002. ACM Press.

[11] Intel Corporation. *LaGrande Technology Architectural Overview*, Sept. 2003.

[12] P. Inverardi, L. Mostarda, and A. Navarra. Distributed IDSs for enhancing security in mobile wireless sensor networks. *AINA*, 2:116–120, 2006.

[13] J. Jeong and D. Culler. Incremental network programming for wireless sensors. In *Proceedings of the First IEEE Communications Society Conference on Sensor and Ad-Hoc Communications and Networks (SECON)*, 2004.

[14] I. Khalil, S. Bagchi, and C. Nina-Rotaru. Dicas: Detection, diagnosis and isolation of control attacks in sensor networks. *securecomm*, 00:89–100, 2005.

[15] P. Kocher, R. Lee, G. McGraw, and A. Raghunathan. Security as a new dimension in embedded system design. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 753–760, New York, NY, USA, 2004. ACM Press. Moderator-Srivaths Ravi.

[16] S. S. Kulkarni and L. Wang. Mnp: Multihop network reprogramming service for sensor networks. *icdcs*, 00:7–16, 2005.

[17] T. Liu and M. Martonosi. Impala: a middleware system for managing autonomic, parallel sensor systems. *SIGPLAN Not.*, 38(10):107–118, 2003.

[18] S. Northcutt and J. Novak. *IDS: Intrusion Detection-Systeme*. mitp Verlag Bonn, 2001.

[19] N. B. of Standards. *Security Requirements for Cryptographic Modules*. National Bureau of Standards, Dec. 2002.

[20] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady. Security in embedded systems: Design challenges. *Trans. on Embedded Computing Sys.*, 3(3):461–491, 2004.

[21] E. Shi and A. Perrig. Designing secure sensor networks. *IEEE Wireless Communication Magazine*, 11(6):38–43, Dec. 2004.

[22] M. Strasser. Intrusion detection and failure recovery in sensor networks. Master's thesis, Department of Computer Science, ETH Zurich, 2005.

[23] H. Vogt, M. Ringwald, and M. Strasser. Intrusion detection and failure recovery in sensor nodes. In *Tagungsband INFORMATIK 2005, Workshop Proceedings*, LNCS, Heidelberg, Germany, Sept. 2005. Springer-Verlag.