Locating Compromised Sensor Nodes through Incremental Hashing Authentication

Youtao Zhang[1], Jun Yang[2], Lingling Jin[2], and Weijia Li[1]

[1] Computer Science Department, University of Pittsburgh, Pittsburgh, PA 15260
[2] Computer Science and Engineering Department, University of California at Riverside, Riverside, CA 92507

Abstract. While sensor networks have recently emerged as a promising computing model, they are vulnerable to various node compromising attacks. In this paper, we propose COOL, a COmpromised nOde Locating protocol for detecting and *locating* compromised nodes once they misbehave in the sensor network. We exploit a proven collision-resilient *incremental hashing* algorithm and design secure steps to confidently locate compromised nodes. The scheme can also be combined with existing en-route false report filtering schemes to achieve both early false report dropping and accurate compromised nodes isolation.

1 Introduction

The sensor network has recently emerged as a promising computing model for many applications e.g. patient status monitoring in a hospital, and target tracking in a battlefield. However, its unattended nature makes the network vulnerable to varying forms of security attacks such as a compromised node dropping true data reports [9] or injecting false reports [18, 22]. Without being detected, compromised nodes may prevent the sink from reaching a correct or optimal decision. In addition, routing false reports wastes the energy of relay nodes, which reduces the lifetime of the network.

The previous work proposed either to locate compromised nodes through en-network detection [12, 15] or to filter false reports early in routing [18, 22]. While they are effective in many cases, both approaches have limitations — the former suffers from low accuracy due to possible collusion attacks and the latter cannot exclude the compromised nodes. In this paper we propose COOL, a COmpromised nOde Locator for locating malicious nodes if they send out false data reports or drop real reports. Our design is based on an intuitive observation — for any well-behaved node in the sensor network, the set of outgoing messages should be equal to the set of incoming and locally generated or dropped messages [3]. We exploit a proven collision-resilient incremental hashing scheme — AdHASH [1] and show how to securely collect the AdHASH values and confidently locate compromised nodes. We incrementally extend the testing so as to capture an inconsistency when a bad link is included. A bad link is a hop between

[3] Some messages may be lost due to weak connection in the sensor network. It is also considered as one type of fault. We detect such links and let the sink decide if the involved nodes should be excluded.

two nodes in which at least one is compromised. For such links, we drop both nodes achieving an upper bound of $2m$ excluded nodes if there are m malicious ones.

The remainder of the paper is organized as follows. We describe the problem, and the network and attack models in Section 2. The COOL protocol is then presented in Section 3 with optimizations presented in Section 4. We evaluate proposed schemes and show the results in Section 5. Section 6 discusses the related work. Section 7 concludes the paper.

2 Problem Statement

2.1 The network model

We assume that the sink assign a unique ID and a unique secret key to each sensor before deployment. Sensors are left unattended after deployment and monitor events of interests. While some may be compromised, we assume that the majority of sensing nodes for any single event are trustworthy.

We adopt a cluster-based multi-hop routing scheme due to its energy efficiency [6, 19]. Sensing readings (including the timestamps [18]) are first sent to the cluster head (CH) at which the are aggregated to a data report. By taking the majority of the readings, the CH includes the selected sensing node IDs and their MACs (message authentication code, discussed next) in the content of the aggregated report. The CH also appends its own ID and MAC to the report. After generating the report, the CH forwards it along the routing path to the sink. Messages from the sink are first sent to CH and then broadcasted within the cluster.

At the cluster head level, the routing graph is built using directed diffusion protocol [2]. Paths are set up to monitor different *interests*. We assume reports are forwarded according to the routing path in one epoch. Each node checks the received report and drops it if not for a cached interest.

2.2 The attack model

After compromising a sensor node, the adversary can retrieve all security information including the secret key. (S)He can then inject false reports ([22, 18]), or drop some of its received reports ([9]). The adversary knows the COOL or other security enhancement algorithms, and may strive to send back data targeting at defeating the protection.

The injection attack or the dropping attack may occur at a sensing node; at a source CH; or at a relay CH. In this paper we address all these types except the dropping attack at a source CH node. Dropping at a source CH is more difficult to defend since a compromised CH may refuse to form a report even after receiving several sensing readings. On the other hand, a CH is usually granted the power to *legally* drop some readings when constructing the report (to shield random erroneous readings). If it is a concern, then each sensing reading could be sent to more than one source CH nodes resulting in increased routing overhead as we will discuss later.

A compromised node is located if and only if its node id is known to the sink who can then securely notify other sensors (using broadcast authentication [13]). Without being located, the compromised node can be elected as a CH node and continuously inject or drop reports. After being located, the network is free from its injection and

dropping attacks since others know it is excluded. Of course additional mechanism might be needed to prevent it from malicious signal collision or changing its id.

Reports may be lost due to weak connections. This is one type of faults that should also be identified. Identifying a weak connection is beneficial since it gives the accurate location where a problem occurs. Based on the frequency of a faulty link, the sink can always make the decision whether or not to exclude the involved nodes. Since it is straightforward to detect/eliminate such links, we will focus on the report loss due to security attacks in the rest of the paper.

2.3 The design objectives

For a sensor network with above settings and models, our design goal is to effectively identify those compromised nodes and then exclude them from the network. The proposed algorithm meets the following requirements.

- The sink has the ability to discriminate the false reports;
- The scheme can defend both true report dropping at the relay nodes and all types of injection attacks;
- The algorithm can locate compromised nodes;
- The algorithm is effective with small overhead introduced to existing clustering and routing algorithms.

3 The COOL Protocol

In this section we present the basic design of the COOL protocol. We first discuss the incremental hash function, and then describe the high-level idea of malicious node detection using a simple example. The details of the systematic protocol operations are then discussed, followed by security analyses of the protocol.

3.1 The incremental hash function

Fig. 1 illustrates the concept of the incremental hash [1]. It computes a cryptographic hash value for a finite set of elements. Each element is first concatenated with a unique *id* and then hashed by a standard cryptographic hash function e.g. MD5 or SHA [14]. Those intermediate hash values are then combined by a combining operator to get the incremental hash value.

In this paper, we use the AdHASH introduced in [1] (abbreviated as AH(...)):

$$AH^h_M(x_1, x_2, .., x_n) = \sum_{i=1}^{n} h(\langle i \rangle . x_i) \ mod \ M$$

where h is a standard cryptographic hash function and M is a very large integer value with k bits. The $\langle i \rangle$ is an *id* assigned to each message such that the concatenation of them is unique in the entire set. When we apply the AdHASH in our sensor network, each report is assigned with the sensor's ID and a local report sequence number. Therefore, each report received by the forwarding cluster head is unique. As we can see, the AH computed by the cluster head is independent of the order at which reports are received. This incremental hash function has the following properties that are useful in our design.

- Compression. It compresses inputs of larger size into k bits such that each incremental hash value can be stored using small number of bits in each node.

– Incrementality. The AH of a larger set can be computed incrementally from the AH of its subset. In particular, when a new item is inserted to the set, the new AH can be computed from the old value and the $h()$ value of the new item. i.e. $AH^h_M(x_1,...,x_{n+1}) = (AH^h_M(x_1,..,x_n) + h(\langle n+1\rangle.x_{n+1}))\ mod\ M$
– Effi ciency. The computation of an AH hash value just needs several additions and one modulation in addition to the standard hashing. Particularly, for the insertion of a new item, the computation overhead is one addition and one modulation only (the width of the h and AH is of the same order). This is important as most hash values are to be maintained by resource constrained relay nodes in our design.
– Proven collision-resilience. It is computationally infeasible to forge another set of items that can result in a same hash value [1]. This gives us a solid security ground for designing security enhancement schemes for sensor networks.

We selected h to be MD5 [14] and k to be 128 in the design. The selection of MD5 is independent and can be substituted if for example security is a concern [17, 16]. The security of AdHASH requires that the number of reports should be greater than k [1]. This is generally not a restriction — the protocol can start after the network has been warmed up.

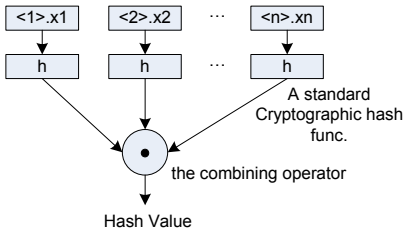

Fig. 1. An incremental hash function.

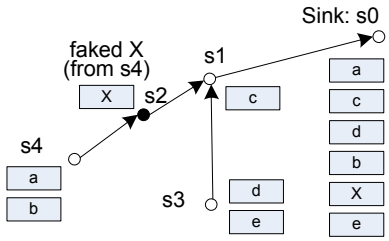

Fig. 2. Locate compromised nodes using incremental hashing.

3.2 The basic design — a simple example

We next show how an incremental hash function can be applied to authenticate messages and in particular how to locate malicious nodes in a sensor network.

The design is based on an intuitive observation, i.e. the set of outgoing (forwarded) messages of a well-behaved node equals the set of the incoming (received) and locally generated/dropped messages. Unfortunately, these message sets are maintained on different sensor nodes across the network making it impractical to pass them around and compare. Luckily with the incremental hash function, we only need to compare the hash values of different sets while keeping suffi cient confi dence to claim that *their hash values match indicates that the message sets also match*, i.e. Fig. 2(a).

No node being compromised
$\Leftrightarrow$
$\{msg_{out}\} = \{msg_{in}\} \cup \{msg_{local}\}$
$\Leftrightarrow$ (with sufficient confidence)
$AH(\{msg_{out}\}) = (AH(\{msg_{in}\}) + AH(\{msg_{local}\}))\ mod\ M$

To see how this principle is applied in our sensor network, let us look at a simple example (Fig. 2). Here we show four cluster head sensors (s1-s4) and one sink (s0). The messages are labeled in letter a, b etc. Suppose the compromised node s2 injects a false message X and pretends that X is sent by s4 (in this section, we will also discuss what if X is forged as if it is sent by s2). All messages are forwarded to the sink, but the AH values are calculated and kept locally. Specifically, s4/s3 calculates outgoing AHs for $(a,b)/(d,e)$; s1 calculates two incoming AHs for (d,e) and (a,b,X) respectively, and one outgoing AH for (a,b,c,d,e,X); s2, as a compromised node, can fake the incoming or outgoing AHs for either (a,b) or (a,b,X). Note that s2 will not produce another incoming AH for X as s2 tries to hide itself from being detected immediately (X is "originated" from s4). As we will elaborate later that the AHs for locally generated legitimate messages are computed at the sink.

The sink receives all messages including X. It can immediately identify that X is false since s2 does not have the secret key of s4 and cannot generate the consistent MAC for X. Next, the sink tries to locate the sender of X, i.e., the node that has been compromised. At this time, the sink collects all the AHs. We can assume that they all arrive correctly as simple endorsement using secret keys can assure this, and there are fixed number of them for a given routing so that no AHs are dropped. The sink then starts to check the "node consistency", i.e., if

$$AH(incoming \cup generated) = AH(outgoing \cup dropped) \tag{1}$$

holds for every node. Note that due to the additivity of the AH function, $AH(incoming \cup generated) = (AH(incoming) + AH(generated)) mod\ M$ (and the same for the right hand side). It is easy to see those conditions for s1, s3 and s4 satisfy. For s2, as we mentioned, there are two options —$AH(a,b)$ or $AH(a,b,X)$— for both the incoming and outgoing AH values, forming four possibilities. If s2 choses the different values for incoming and outgoing AHs, it would immediately be identified as compromised as equality (1) would not satisfy. Let us assume s2 is intelligent enough not to expose itself too easily, and thus chose a consistent incoming and outgoing AH pair. Thus, it will pass at least the "node consistency" check.

Next the sink starts a "link consistency" check in which the AH of the outgoing message on a link should equal the AH of the upstream incoming message. This can be easily checked for links without the node s2. Let us assume that the s2's incoming and outgoing hashes are both $AH(a,b,X)$. Then, as an outgoing AH, $AH(a,b,X)$ is consistent with one of the incoming AHs for s1. However, as an incoming AH, $AH(a,b,X)$ is inconsistent with the outgoing AH for s4 which is $AH(a,b)$. In other words, s2 chose that value to lie that s4 had given it a false incoming message. The sink now cannot distinguish who is the real compromised node, but can at least conclude that one of them is flawed. A new routing graph will be generated excluding both s4 and s2 after detecting the link inconsistency.

Notice that node s2 can destroy s1 in the same way by producing $AH(a,b)$ for link consistency checking, or even destroy all the adjacent nodes by forging an arbitrary AH making none of the links consistent. It would be unnecessarily conservative if all involved nodes in such a scenario are eliminated since the faults are due to only one evil axial node. Instead, our protocol removes one link at a time, removing the axial node in the first place and saving the other nodes being circumvented.

Let us discuss what if X is forged as if it is sent by s2. The sink can still identify X as a false report since each legal one should have multiple sensing node MACs — s2 cannot construct these sensing node MACs as it does not have their secret keys. Old readings cannot be replayed since a timestamp is included in the reading. Notice that the sink computes local AH values only from legitimate reports, that is, it excludes X from generating the local AH value for s2. Hence, no matter which incoming or outgoing AH values s2 chooses (either $AH(a,b)$ or $AH(a,b,X)$), there bounds to be a node or link inconsistency.

Before presenting the COOL protocol, we summarize the benefits from excluding compromised nodes from the network.

- Communication energy savings. Once the compromised nodes have been excluded, no false reports can be injected into the network. As a result, the energy drained by forwarding false reports can be saved. This is different from en-route filtering schemes in which false reports are still forwarded several hops ([18, 22]) before being detected and dropped.
- Computation energy savings. In the basic COOL scheme, we do not perform en-route packet authentication but rather only update incremental hash values. We can afford less frequent authentication by excluding compromised nodes and form a network with trustworthy nodes.

3.3 The COOL protocol

The goal of the COOL protocol is straightforward: we securely collect AH hash values from the network and send them to the sink; we drop the identified node if a node inconsistency is found, and drop both nodes if an inconsistent link is found. The detailed protocol contains the following phases.

(1) In the initialization phase, we assign a unique ID and a secret symmetric key to each sensor node. The sensor nodes are deployed thereafter.
(2) In the routing graph discovery phase, we broadcast *hello* messages along the downstream routing path and collect the current routing graph from replies.
(3) In the report forwarding phase, we endorse each report by the secret key of the sender and have it forward along the routing path. Each node maintains the AH hash values for each of its incoming and outgoing links.
(4) In the hash value collection phase, the sink sends out a request to collect hash values from the path where the false message belongs to.
(5) In the compromised node detection phase, the sink finds inconsistent nodes and links using the incremental hash function AH.
(6) In the routing graph fix phase, the sink replaces the excluded cluster heads with newly selected ones.

Next, we elaborate each of the phases with more details.

Phase 1: Initialization. Each sensor is assigned a unique integer ID and a secret symmetric key before being deployed into the field. Both the ID and the key are known to the sink. This is similar to [22] but we do not request any key sharing among sensor nodes. The node ID occupies two bytes (16 bits) which can distinguish 64K sensors in

a network. Larger networks can adaptively adjust this bit width to accommodate their needs. The key is used to generate a MAC by a sensing node for the sensing reading sent to the source CH node, and by a source CH node for the report sent to the sink. By checking the MACs using the secret keys, the sink can detect tampered reports or injected false reports from compromised source CH or relay CH nodes.

Phase 2: Routing graph discovery. The discovery starts at the beginning of an epoch, e.g. cluster heads are reselected and a new routing graph is constructed according to [2]. The sink forms an entire routing graph through collecting information from distributed cluster heads. It sends out a timestamped "hello" message to all its adjacent cluster heads who then forward this message downstream until it reaches the leaf nodes. All the nodes then respond with their node IDs as well as their adjacent node IDs. For those messages, a MAC using the local secret key is also attached to ensure their integrity.

Each node only collect replies from its downstream nodes. To prevent malicious report dropping, the reply is collected in order — each node first collects the replies from all its children nodes and then sends out its own reply. The sink finally assembles the complete routing graph from all replies.

To ensure "hello" messages are not abused, a broadcast authentication [13] is applied. In addition, a selected cluster head may try to find a different routing path if it does not receive a "hello" message within a certain time interval after its election to avoid being isolated from the network.

Phase 3: Report endorsement and forwarding. Fig. 3 illustrates the report generation, endorsement and forwarding in the network. We also list the related authentication actions and discuss why injected false reports and dropped legitimate reports can be detected.

To check equation 1, each node maintains its incoming AHs and outgoing AHs. The local AH (generated from locally generated reports) is computed at the sink rather than the individual node. This is because false reports should not be used to compute the local AH (otherwise there is no inconsistency) and the sink knows what those false reports are. The sink generates a local AH for each node using only the legitimate reports whose source IDs are that node, discarding all false reports discriminated. **For example**, a node m may forge a false report using its own ID as the source ID. This false report can be identified by the sink since it does not have enough legal sensing node MACs. After identifying X as a false report, the sink will exclude it from updating the local AH of m, which creates a node inconsistency if node m updated the false report into its outgoing AH, or a link inconsistency otherwise.

The drop AH (generated from locally dropped reports) is not used in the basic scheme i.e. we assume a non-compromised node does not drop reports intentionally. A report is dropped only by compromised nodes who always try to conceal themselves as much as possible. Consequently, the AHs for dropped reports are never created.

Either false report injection or legitimate report dropping creates AH inconsistency for some nodes or links. The difficulty is how to expose the inconsistency. The technique presented in phase 4 handles this problem.

	Sensing node	Cluster head (CH) node	
		Data aggregation	Report relay
Report generation and endorsement	An event is detected by at least M surrounding sensing nodes. Each of them e.g. m sends the sensing reading and the MAC (generated using m's secret key) to C CH nodes.	By taking the majority of received readings, the source CH constructs a report containing the sensing value and received MACs. It also appends its ID and a unique sequence number. A unique MAC is generated for the report using its own secret key. Both the report and the MAC are forwarded along a multi-hop route to the sink.	Since the keys to attached MACs are only known to the sink, relay nodes do not perform any en-route checks in the basic scheme. A relay node receives the report from one downstream link and forwards it along one upstream link.
Authentication	An AH value is maintained for the outgoing link of m. The value is updated with the sensing reading and the MAC each time when m sends them out.	A source CH maintains one outgoing AH and in the case it is also a relay node, it maintains several AH values with one for each of its incoming/outgoing links respectively. After sending the report generated by itself, it updates the outgoing AH value with the report and the MAC.	A relay CH node maintains a different AH value for each of its incoming and outgoing links respectively. For the forwarded report, it updates two AH values — the incoming AH value for the link from which the report is received and the outgoing AH value for the outgoing link.
Detecting injection attack	A source CH should receive readings from at least M sensing nodes. If some ($< M/2$) are compromised and send back false readings, these readings will not affect the report generation at the CH node.	A compromised source CH may forge false reports. It cannot accumulate $M/2$ legal MACs for the false reading. A report with such a value can be detected at the sink. Old readings and MACs cannot be replayed since the sensing reading has a timestamp indicating when the event happens [18]. Different sensing node MACs are expected even for the same sensing reading but at a different time.	If a relay node forges a report with the source node as itself, the false report can be detected as the data aggregation case. If a relay node forges a report with the source node id as one of its downstream nodes, it does not have the secret key of the faked sender to generate a matching MAC. The report will be detected by the sink. The compromised node is then located using our algorithm.
Detecting dropping attack	Dropping readings at some ($< M/2$) compromised sensing nodes does not affect the generation of the legitimate report at a source CH node.	[Discussion only: not the focus in the paper] We can elect more than one CH to perform data aggregation. If some but not majority of them refuse to generate reports from received readings, the sink can still receive the legitimate report and thus detect packet dropping in the network. In the rest of the paper we assume that for each event one CH node is elected for data aggregation.	If a relay node drops some reports, the sink can check the sequence number from a source CH and reveal the dropping from non-contiguous numbers. If the compromised relay node chooses to drop all following reports from a CH, the sink can periodically collects all AH values to detect the dropping attack.

Fig. 3. Actions taken by different nodes.

Phase 4: Hash value collection. Hash value collection is triggered by any of the following two conditions: (i) the sink has detected one or multiple false reports; (ii) a preset timer has elapsed. The former is to detect injected false report attack while the latter is to detect report dropping attack. In the first case, AH values are collected only from the path where the erroneous report belongs to. Such a path can be identified correctly as we explained earlier that a spoofed report can reach the sink only if it is generated from a downstream node. Collecting the hash values along a path greatly reduces the number of messages introduced to the network. In the second case, however, all the hash values in the network are queried as the sink has no clue of where reports could be dropped. Next we show how to collect AHs from the erroneous path. It is trivial to extend the scheme to all nodes.

To collect the hash values, the sink sends out an inquiry message onto the erroneous path. For example, in Fig. 2, to detect the compromised node on path s4-s2-s1-s0, we only collect hash values on this path but not from the link s1-s3 (but in phase 5 the sink still needs to compute the AH values for these reports injecting to the path from s3). We must be very careful in this collection process as the forwarded AHs may be altered by compromised nodes as well. Thus, we treat all the AH values as *normal data reports* and send them upstream starting from the leaf cluster nodes. The only constraint is: the outgoing AH on a link does not update the incoming AH on the same link since this would result in link inconsistency and make the hash collection process and later checking too convoluted. As a result, the node consistency checking would be adjusted to accommodate this exception. We will give formal derivation later.

Phase 5: Identify compromised nodes.

The sink performs two types of tests: the node consistency and the link consistency test. The first is to test the matching of incoming and outgoing AH values for each node on the erroneous paths. The AHs for the incoming links not on erroneous paths, e.g, the s1-s3 path in our example, and for locally generated reports are calculated by the sink directly; other AHs are from the returned AH reports. If there is a mismatch, the node is tagged as a compromised node. For example, in Fig. 2, the sink tests s1 using

$$(\underbrace{AH_{2\to 1}}_{collected} + \underbrace{AH(c) + AH(d,e)}_{calculated\ by\ the\ sink})\ mod\ M =? \underbrace{AH_{1\to 0}}_{collected}$$

The second type is to test if the outgoing and incoming AHs are consistent on all links. Each hash value should match with the one reported by the other end of the link. If any inconsistency is found, the sink tags both nodes as problematic as it is now hard to flag one node with 100% confidence. e.g. we test

$$(\underbrace{AH_{1\to 0}}_{collected} =? \underbrace{AH_{0\leftarrow 1}}_{computed\ at\ the\ sink}) \ and\ (\underbrace{AH_{1\to 2}}_{collected} =? \underbrace{AH_{2\leftarrow 1}}_{collected})$$

Phase 6: Excluding compromised nodes and routing graph fix. Once any nodes are tagged as suspicious, they should be excluded from the sensor network immediately. To do so, the sink broadcasts the IDs of the tagged nodes across the network, particularly to those nodes around the compromised ones. This can be done using broadcast authentication algorithms e.g. μTESLA [13]. This packet also initiates the selection of

new cluster heads to replace the excluded ones, and then incorporates new heads into the routing graph. The newly joined nodes send back their IDs to the sink to check if they are allowed to join the network. The sink acknowledges back with the most up-to-date AH values for the new cluster heads.

3.4 The security analyses

In this section we give the major security analysis results. Their proof details can be found in [20].

Theorem 1. *Any injection attacks can be detected by the COOL protocol.*

Corollary 1. *The report dropping attack at the relay nodes can also be detected by the COOL protocol.*

Corollary 2. *If there are m compromised nodes, our scheme removes at most* 2*m nodes including those m compromised nodes.*

Theorem 2. *The AH value collection process is secure: correct AH values can be retrieved from the received AH report; no AH value may be compromised or dropped without being detected.*

4 Optimizations

In this section, we discussion two optimizations to the basic design.

4.1 Drop-COOL: combining en-route filtering schemes

In the basic COOL protocol false reports are forwarded all the way to the sink. While the sink can detect these false reports, it is just too late since the energy has already been consumed along the routing paths. It may become even worse if a lot of false reports are injected before the COOL protocol is activated to collect AH values. We therefore propose Drop-COOL, a hybrid scheme that integrates an en-routing filtering scheme [18, 22].

The Drop-COOL scheme works as follows. The system initializes according to both the basic COOL and the SEF [18] protocols. In addition, each node is assigned an integer threshold which is the maximal number of false reports that the node can forward in one round. In the packet forwarding phase, detected false reports up to this threshold are forwarded while following ones are discarded. An additional AH hash value — *drop* hash value, is maintained on each node that drops false reports. It is updated incrementally each time when a false report is detected and dropped. To improve the effectiveness of Drop-COOL, a node may try to forward these false reports with different source CH IDs, which can trigger collecting and detecting more paths and thus expose more compromised nodes in one round.

The Drop-COOL protocol combines the advantages of both COOL and SEF protocols. It detects and excludes compromised nodes while saves the energy from routing less false reports. The energy spent to route a small number of false reports is small compared to the savings after excluding compromised nodes. It removes the worst case overhead that the basic COOL protocol has on routing false reports.

We next illustrate that the Drop-COOL does not affect the ability to locate compromised nodes although random false reports are dropped in the middle. Due to the introduction of the drop hash value, we have *for a well-behaved node in the routing graph, the set of forwarded and dropped messages should be the same as the set of received and locally generated messages*. By collecting and comparing the AH hash values of these message sets, we can adjust the node test to

$$(\text{AH}(\text{MESSAGE}_{forwarded} + \text{AH}(\text{MESSAGE}_{dropped}) \bmod \text{M} = (\text{AH}(\text{MESSAGE}_{received} + \text{AH}(\text{MESSAGE}_{local}) \text{ mode M}$$

The link test is unaffected and an inconsistent node or link test result exposes at least one compromised node.

4.2 Hi-COOL: a hierarchical authentication scheme

To further reduce the overhead, we propose Hi-COOL, a hierarchical approach which groups multiple adjacent nodes in the routing graph as a *super node*. We only collect hash values with respect to this super node, and refi ne the collection if a super node is found problematic.

The Hi-COOL scheme works as follows. First the sink picks up an integer number *l* and forwards this integer with the "hello" message (for collecting the routing graph). The integer value is decremented for each hop downstream along the routing path and reset after reaching zero. A node sets itself to be the head of a super node if it receives *l* and be the leaf of a super node if it receives zero. If a node receives two values from two upstream nodes, it picks up the smallest one. In Fig. 5 nodes s1 to s5 form a super node in which s2, s4, and s5 are leaf nodes while s1 is the head node. In the phase to collect the AdHASH values, only the incoming hash values to this super node and the outgoing hash values from this super node are collected. For example hash values AH_{23}, AH_{32} are omitted. The link test at the super node level is processed the same as the basic COOL — a failed link test removes two involved nodes. However an inconsistent (super) node test results in one additional round of hash value collection such that we can determine the exact location of the compromised node within the super node. In the second round, we only collect hash values from these inconsistent super nodes.

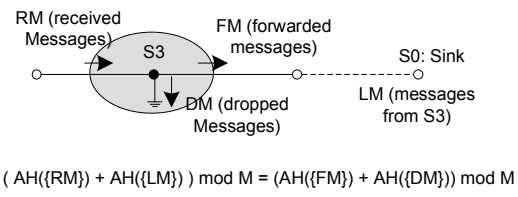

Fig. 4. Reducing routing energy through combined en-route filtering.

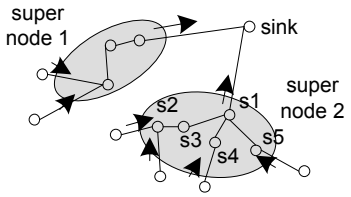

Fig. 5. Hierarchical authentication with super nodes.

To ensure high level security, in the routing graph collection phase, each node should reply with its received integer number. In addition, the head of each super node may need to collect all internal hash values. More details can be found in [20].

5 Limitations of the COOL protocols

The limitations of the COOL protocol are: (1) It is possible that a subregion is isolated from the sink. Without having the information about a cluster head in the routing graph collection phase, the sink cannot identify its status and decide if it is compromised or not. (2) Signal blocking or collision is another source of attack, if normal communication cannot be ensured between two nodes. Both of two involved nodes are excluded while the nodes themselves may not be hacked.

6 Performance Evaluation

6.1 Settings

To evaluate the effectiveness of the proposed COOL protocols, we simulate a sensor network with 450 cluster head nodes uniformly distributed in a field of 400x400m^2 area. Each sensor node is Mica2 running TinyOS [7] operating at 19.2Kbps data rate, with battery voltage 3V. It takes 16.25/12.5 μJ to transmit/receive a byte [18]. This will be referred as the baseline setting in the rest of the paper. The sink is located at (20,20) and the communication range of each node is 40m. These sensor nodes form a multihop routing network using the directed diffusion routing algorithm [2]. A normal packet is of 24 bytes long, a MAC is of 8 bytes (64 bits), and an incremental hash value is of 16 bytes (128 bits). The evaluation is based on false report injection attacks. All results are averaged from 100 different runs.

6.2 The overhead

The protocol overhead comes from four sources: (i) AH hash value computation overhead; (ii) hash value collection overhead; (iii) routing graph discovery overhead; and (iv) routing false reports overhead. Next we study them in more detail.

(i) Computation overhead. The computation overhead is for updating incremental hash values at each sensor node. As the incremental hash is maintained per link based, a received report updates two AH values on the relay node. The updates are done incrementally with the overhead mainly from computing the standard hashing MD5() on the input report. MD5() intermediate result is used to update both AH values. Our simulation results show that the incremental hash computation overhead is about twice of the overhead of one RC5 [14] computation (used in [18, 7]), that is, 30μJ per node. It is small and thus omitted in the rest of the discussion.

(ii) Hash value collection overhead. The main overhead of the COOL protocol comes from collecting hash values across the network. We first induce a theoretical formula about this overhead. Assume the routing graph is a b-nary balanced routing tree with height O($\log_b$(N)) where N is the total number of nodes in the graph. Since the hash value collection is per problematic routing path based, the number of node-to-node transmission T for one path is,

$$T = (1+2+3+...+H)\cdot(2\cdot k+M) = \frac{H\cdot(H+1)}{2}\cdot(2\cdot k+M) \quad (2)$$
$$H = log_b(N)$$

where H is the height of the b-nary tree, N is total number of sensor in the field, k is the length of the incremental hash value, and M is the length of the MAC value. From the

equation, T is in the range of $O((\log N)^2)$. Since we need to exclude all m compromised nodes, we may need to collect from m disjoint paths or detect in m rounds. Therefore the worst case hash value collection cost is $O(m \cdot (\log N)^2)$.

We next present the average number of rounds to exclude all compromised nodes in Fig. 6. In the experiment, false reports are randomly injected from the compromised nodes and we start a new detection round if 30 false reports are received. As expected it requires more rounds when there are a larger number of compromised nodes. On average we can detect and exclude more than 10 nodes in one round when more than 30 nodes are compromised.

Fig. 7 illustrates the total energy overhead for collecting hash values in multiple rounds. From equation 3, the AH collection cost is proportional to the number of compromised nodes and to the square of the tree height, these two factors are used as the x and y axes respectively. To change the tree height, we deploy in a different square field with the same node density, e.g. 1000 nodes are distributed in a field of 600x600 m^2. The tree height varies from 9.2 to 28.5.

The trend confirms what we observed from equation 3. For example, the energy overhead is about 0.87J with 20 compromised nodes and the tree height 13.6. It increases about 2 times to 1.56J if the number of compromised nodes increases 2 times to 40; It increases about 4 times to 3.89J if the tree height increases about 2 times to 28.4.

We also present the results using the Hi-COOL scheme. It effectively reduces the hash value collection cost. For example, when 20 nodes are compromised, the Hi-COOL overhead is 0.99J or 61% of the baseline setting (1.61J).

In collecting the results, we perform a simple optimization. For the two nodes of each link, they may report the same AH() value e.g. both nodes are healthy nodes. We therefore only need to transmit one AH hash value with two MACs to the sink.

	number of compromised nodes							
	10	20	30	40	50	60	70	80
Rounds	2.1	2.4	3.5	4.0	4.5	5.3	6.1	6.5

Fig. 6. The number of rounds to exclude all compromised nodes.

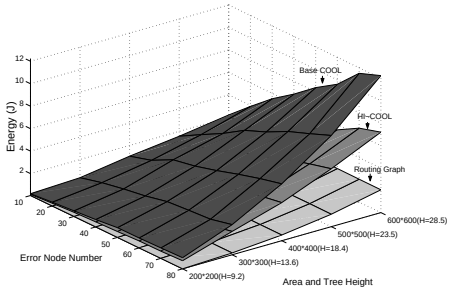

Fig. 7. Hash value collection overhead.

(iii) Routing graph overhead Fig. 7 also illustrates the energy overhead to collect the routing graph. Compared to the hash value collection overhead, it is usually small ranging from 0.5J when the tree height is 9.2 to 2.0J when the tree height is 28.5.

(iv) The overhead to route false reports. Fig. 8 illustrates the wasted routing energy in forwarding false reports. As we discussed, the sink has the option to start the hash value collection phase after accumulating a number of false reports. Clearly if we increase the threshold, the routing overhead increases as well. The benefit of accumulating a reasonable larger number of false reports is that we increase the chance that these false reports are from more problematic paths. They can then detect multiple paths and

reduce the total number of rounds. For example, if we detect after receiving one false report, we may need m rounds to exclude all m compromised nodes; on the other hand, we may need only m/2 round if we set the threshold to be 2 and these two false reports are always from 2 different compromised nodes on different routing paths. However, our experiments show that the difference is not signifi cant (with one or two rounds difference). In addition, if the threshold is set larger than 30, the number of rounds does not change much but the wasted energy increases drastically. Therefore we set the threshold to 30 in the paper.

6.3 The savings

We next study the benefi ts from applying COOL protocols and compare it with an en-route false report fi ltering scheme [18]. As discussed, the savings comes from two sources: the communication savings and the computation savings. The latter is omitted as it is usual very small.

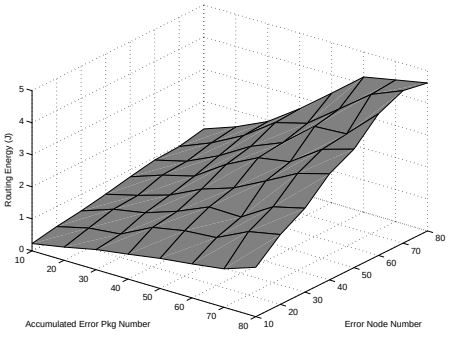

Fig. 8. The overhead to route false reports.

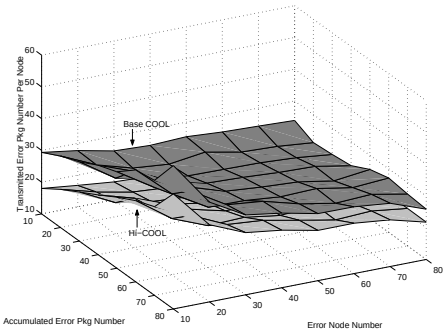

Fig. 9. Comparing the overhead to en-route filtering schemes.

Let us compare the overhead from different protocols. The overhead in the COOL protocol contains the energy to collect hash values, to discover routing graph, and to route false reports to the sink. The overhead in the SEF protocol is from routing false reports before detecting and dropping them. The exact number of hops varies with the key sharing scheme and the number of compromised nodes. For comparison purpose, we assume on average each false report is dropped after 5 hops in the paper.

In Figure 9, x and y axes are the detection trigger threshold and the number of compromised nodes in the network respectively. Each point in the fi gure is a number of false reports averaged to each compromised node. It is a break even point at which the overhead to route and drop this amount of false reports in SEF equals the overhead in the COOL protocol. For example, with 20 compromised nodes and trigger threshold set at 30, the number is 25 reports meanings that, if SEF routes and drops 500 reports (=25 reports/node$\times$20node) in 5 hops, the energy it wastes is the same as all of the overhead in the COOL protocol.

In addition, consider that the scheme needs to spend 3.3 rounds and a round is triggered at 30 false reports, there are another 100 injected false reports (=30 reports/round $\times$ 3.3 rounds). Therefore the COOL protocol outperforms the SEF if each compromised

node injects more than 30 reports (=25 reports/node + 100 reports/20 nodes). This is very small. For example, as suggested in [21], a compromised node may inject a faked report every 10 seconds. At this rate, we outperform SEF in 300 seconds or 5 minutes. With the Hi-COOL optimization, it is further reduced to 4 minutes, a 20% reduction. Of course, depending on the pattern that the false reports are injected, this number may vary but the results show that the COOL overhead is very modest.

7 Related Work

Extensive research has been done on sensor network security. Karlof *et al.* [9] identified several attacks for a multihop routing based sensor network.

Marti *et al.* proposed to monitor each node by a neighboring *watchdog* node. Wang *et al.* [15] improves the scheme through the collaborative decision of neighbors around a suspicious node. Both schemes have limitations [12] as the watchdog node may be compromised as well and multiple compromised nodes can collude to attack. The schemes to locate compromised nodes share some similarity with the approaches to detect faults in sensor networks [8, 11, 4]. However the significant difference lies in that a faulty node always returns a wrong report while a compromised node is smarter e.g. it may inject false reports but communicate normally with the sink.

En-route false data filtering schemes [18, 22] are proposed to actively detect and drop false reports early in the routing minimizing the impact of false report. In these schemes, each data report is attached with several MACs generated from different keys that are distributed probabilistically [18], set up before routing [22], or refreshed periodically [21]. However, those schemes also have limitations as compromised nodes are left undetected, which causes severe consequences in the long run.

Algorithms have been proposed to securely manage keys in sensor network. Eschenauer and Gligor [5] proposed a key pre-distribution scheme in which each sensor randomly selects a subset of all keys before deployment; Protocols were then developed for shared key discovery and path-key establishment. Improvements were later proposed for enhancing security [3] and achieving higher probability of key establishment [10]. Zhang and Cao [21] proposed to represent keys as group key polynomials whose shares are distributed around neighbors. New keys can be re-generated collaboratively by neighbors from these shares achieving better security and resilience.

8 Conclusion

In this paper we introduced the COOL protocol and its optimizations based on the provably secure incremental hash function AdHASH to detect and locate compromised nodes effectively. We first discussed how to securely maintain and collect AdHASH values on sensor nodes, and then use these values to perform node and link tests to expose the compromised nodes. Our experimental results showed that the COOL protocols are very effective and introduce very small overhead to the network.

Acknowledgment

This work is partially supported by the U.S. National Science Foundation under grants CCF CAREER 0447934 and CCF 0430021.

References

1. M. Bellare and D. Micciancio, "A New Paradigm for Collision-free Hashing: Incrementality at reduced cost," In *Eurocrypt'97*, LNCS 1233, 1997.
2. C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed Diffusion: a Scalable and Robust Communication in Wireless Sensor Networks," In *5th IEEE/ACM Mobicom*, pages 174-185, 1999.
3. H. Chan, A. Perrig, and D. Song, "Random Key Predistribution Schemes for Sensor Networks," In *IEEE Symposium on Security and Privacy"*, 2003.
4. P. Chew and K. Marzullo, "Masking Failures of Multidimensional Sensors," In *Proc. of the 10th Symposium on Reliable Distributed Systems*, pages 32-41, 1991.
5. L. Eschenauer and V. D. Gligor, "A Key-Management Scheme for Distributed Sensor Networks," In *Proc. of the 9th ACM Conference on Computer and Communication Security*, pages 41-47, November 2002.
6. W.R. Heinzelman, A. Chandrakasan, and H. Balakrishnan, "An Application-Specific Protocol Architecture for Wireless Microsensor Networks," *IEEE Transactions on Wireless Communications*, vol 1:4, pages 660-670, 2002.
7. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister, "System Architecture Directions for Networked Sensors," In ASPLOS IX, 2000.
8. C. Jaikaeo, C. Srisathapornphat, and C. Shen, "Diagnosis of Sensor Networks," In *IEEE international Conference on Communications*, June 2001.
9. C. Karlof and D. Wagner, "Secure Routing in Wireless Sensor Networks: Attacks and Countermeasures," In *IEEE international workshop on Sensor Network Protocols and Applications*, pages 113-127, 2003.
10. D. Liu and P. Ning, "Establishing Pairwise Keys in Distributed Sensor Networks," In *Proc. ACM CCS*, 2003.
11. K. Marzullo, "Tolerating Failures of Continuous-valued Sensors," In *ACM Transactions on Computer Systems*, November 1990.
12. S. Marti, T.J. Giuli, K. Lai, and M. Baker, "Mitigating Routing Misbehavior in Mobile Ad Hoc Networks,", In *MOBICOM*, 2000.
13. A. Perrig, R. Szewczyk, V. Wen, D.E. Culler, and J.D. Tygar, "SPINS: security protocols for sensor networks," In *Proc. of Seventh Annual International Conference on Mobile Computing and Networks*, 2001.
14. B. Schneier, "Applied Cryptography," 2nd Edition, John Wiley & Sons, 1996.
15. G. Wang, W. Zhang, G. Cao, and T.L. Porta, "On Supporting Distributed Collaboration in Sensor Networks," In *IEEE MILCOM*, 2003.
16. X. Wang, Y. Yin, H. Yu, "Finding Collisions in the Full SHA-1 Collision Search Attacks on SHA1," In *Crypto'05*, 2005.
17. X. Wang, D. Feng, X. Lai, H. Yu, "Collisions for Some Hash Functions MD4, MD5, HAVAL-128, RIPEMD," In *Crypto'04*, 2004.
18. F. Ye, H. Luo, S. Lu and L. Zhang, "Statistical En-route Detection and Filtering of Injected False Data in Sensor Networks," In *IEEE INFOCOM 2004*, 2004.
19. O. Younis, and S. Fahmy, "Distributed Clustering in Ad-hoc Sensor Networks: A Hybrid, Energy-Efficient Approach," In *INFOCOM*, 2004.
20. Y. Zhang, J. Yang, L. Jin, and W. Li, "Locating Compromised Sensor Nodes through Incremental Hashing Authentication," Technical Report, University of Pittsburgh, 2006.
21. W. Zhang and G. Cao, "Group Rekeying for Filtering False Data in Sensor Networks: A Predistribution and Local Collaboration-Based Approach," In *INFOCOM*, 2005.
22. S. Zhu, S. Setia, S. Jajodia, P. Ning, "An Interleaved Hop-by-Hop Authentication Scheme for Filtering of Injected False Data in Sensor Networks," In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, California, May 2004.