SIA: Secure Information Aggregation in Sensor Networks

Bartosz Przydatek, Dawn Song, and Adrian Perrig

(presented by Aleksandr Yampolskiy)
Outline

- Motivation
- The model
- Results
  - Median
  - Min/max
  - Counting distinct elements
- Conclusion
What is a sensor network?

- Thousands of sensor nodes
- Sensors collect data and relay information to users
- Many applications
  - battlefield surveillance
  - wildlife monitoring
  - seismic safety
  - tracking NFL players during SuperBowl XXXVIII
What is a sensor network? (cont.)

- Sensors are severely constrained: limited battery power, computation resources, bandwidth.

- **Example:** UC Berkeley mote
  - 4 MHz Atmel processor
  - 4kB RAM and 128 kB code space
  - 917 MHz RFM radio at 50 kb/s

- Conflict between **limited resources** and **security requirements**.
Why do we need aggregation?

**Problem:**
- Forwarding raw information is too expensive.
- Individual sensors readings are of limited use.
  - May want to know MAX of seismic readings and not the readings themselves.
  - We have a CPU on board. Let’s use it!
Why do we need aggregation? (cont.)

Solution:
- Dedicated sensor nodes, called **aggregators**
- Information is processed in the network and only results are forwarded to the user
- Common aggregation operators: COUNT, MIN/MAX, AVERAGE, SUM, ...
Why this paper?

- Most prior work [DNGS03, EGHK99, IEGH01, MFHH02] study aggregation under the assumption that every node is honest.
- But... what happens if the adversary takes possession of a sensor node or, even worse, an aggregator?
Outline

- Motivation
- The model
- Results
  - Median
  - Min/max
  - Counting distinct elements
- Conclusion
The model

- Sensors $S_1, \ldots, S_n$
- Single aggregator $A$
  - resource-enhanced
  - master key $K_A$
- Home server $B$
  - master key $K_B$
The model (cont.)

- Each sensor $S_i$ has a
  - unique ID$_i$
  - shared keys $\text{MAC}_{KA}(\text{ID}_i)$ and $\text{MAC}_{KB}(\text{ID}_i)$
  - sensor measurement $a_i \in \{1, 2, \ldots, m\}$

- Aggregator $A$ computes $y = f(a_1, \ldots, a_n)$
Attack model

- Aggregators or sensors may be compromised
- There are many kinds of attacks.
- **Example:** DDoS attacks, where nodes are too overwhelmed to respond to queries
The focus is on **stealthy attacks**.

They try to cheat the user into accepting false aggregation results which are significantly different from true results.
Suppose aggregator reports $y$ instead of the actual $y = f(a_1, \ldots, a_n)$.

We will look for $(\varepsilon, \delta)$-schemes so that if the server accepts $y$, we have:

$$\Pr[|y - y| \cdot \varepsilon y] \leq 1 - \delta$$
Some assumptions

- Byzantine fault model [LSP82]
- Polynomially bounded attacker
- Can corrupt a (small) fraction of sensors
- Uncorrupted sensors form a connected component containing an aggregator
- Home server and aggregator can broadcast to all sensors (e.g., using $\mu$TESLA)
Tradeoff

security
$a_1, \ldots, a_n$

efficiency
$y = f(a_1, \ldots, a_n)$

forward all inputs

forward only the final result
General approach

- Three phases: aggregate, commit, prove
  1. Aggregator collects data from sensors and locally computes aggregation: \( y = f(a_1, \ldots, a_n) \)
  2. Aggregator commits to the collected data: \( c = \text{commit}(a_1, \ldots, a_n) \).
  3. Aggregator sends \( y \) and \( c \) to the home server and engages in an IP:
     1. Home server checks that committed data is a good representation of true data values.
     2. Home server checks if the aggregator is cheating: Is \( c \) close to \( y \)?
Merkle hash tree [Merkle80] is used to commit to measurements $a_1, \ldots, a_n$.

Basic idea:

- Put data $a_1, \ldots, a_n$ at the leaves.
- Each node contains a hash of its children: $v_{i,j} = H(v_{i+1,2j} \| v_{i+1,2j+1})$.
- Root node $v_{0,0}$ is the commitment.
General approach (cont.)

\[ v_{0,0} = \text{commit}(a_1, \ldots, a_8) \]
Outline

Motivation
  ▪ The model

Results
  ➢ Median
  ➢ Min/max
  ➢ Counting distinct elements

Conclusion
Computing the median

- **Goal:** Securely compute the median of $a_1, \ldots, a_n$ (if $a_i$ are not distinct, use $(a_i, ID_i)$ pairs).

- Median is the middle of distribution. For example, the median of 3, 5, 11, 4, 7 is 5.

- $n'$ corrupted nodes can cause aggregated median to deviate by at most $n'$ positions from true value.
Computing the median (cont.)

- **Naïve approach 1**: send *all* measurements to the home server.
  - Too inefficient.
- **Naïve approach 2**: send sample of *l* measurements to the server.
  - Use sample’s median as an approximation to the true median.
Computing the median (cont.)

**Thm**: The median of a uniform sample of \( l \) out of \( n \) elements yields an element whose position in the sorted sequence \( a_1, \ldots, a_n \) is within \( \varepsilon n \) of \( n/2 \) with probability \( \geq 1 - (2/e^{2\varepsilon^2}) \).

**Proof idea:**
- Can bound \( \Pr[|X_l - n/2| > \varepsilon n] \) using Hoeffding bound.
- Sample size \( \Omega(1/\varepsilon^2) \) is needed [BKS01].
Digression: spot-checkers

- Program P computes function $f$ on input $x$ [EKK+99].
- Ascertain that program output $P(x)$ is reasonably correct in $o(|x| + |f(x)|)$ time.
- $\epsilon$-spot-checker for sorting: $\Omega(1/\epsilon)$ running time.

```
procedure Sort-Check-II(A, $\epsilon$):
  repeat $O(1/\epsilon)$ times
    choose $i_2 \in [1, n]$
    perform binary search as if to determine if $a_i$ is in A
    if not found
      return FAIL
  return PASS
```
Computing the median (cont.)

- **Better approach:**
  1. A commits to a sorted sequence \( \tilde{a} = sort(a_1, \ldots, a_n) \) using a Merkle tree.
  2. When B obtains an alleged \( a_{med} \), he first verifies that \( \tilde{a} \) is sorted using Sort-Check-II.
  3. Then B uses MedianCheck to check that \( a_{med} \) is close to the true median.
procedure MedianCheck(n, a_{med}, \varepsilon):
request a_{n/2}
if a_{n/2} \neq a_{med} then
    return REJECT
for i = 1\ldots(1/\varepsilon) do
    pick j \in 2 \times \{1...n\}\{n/2\}
    request a_j
    if j < n/2 and a_j > a_{med} then
        return REJECT
    if j > n/2 and a_j < a_{med} then
        return REJECT
return ACCEPT
Computing the median (cont.)

**Thm:** MedianCheck(n, a_{med}, \varepsilon)
requests 1/\varepsilon elements a_i, runs in time O(t\Theta/\varepsilon) and satisfies:

1. if a_{med} = a_{n/2}, then the result is “ACCEPT”
2. if a_{med} is not in the sequence or its position p satisfies |p – n/2| > \varepsilon n, then with probability > ½ the result is “REJECT”.
Computing the median (cont.)

Proof:
(1) Trivial.
(2) Notice that if $|p - n/2| > \varepsilon n$, then there are $\varepsilon n$ values of $j$, which yield reject. Hence, with probability $\cdot (1 - \varepsilon)^{1/\varepsilon} \cdot \frac{1}{e}$ for-loop completes without rejection. QED.
Computing min/max

- **Goal:** Securely compute $\min(a_1, ..., a_n)$, where $a_i \in [m]$.  
- Corrupted sensor can always claim his measurement to be 1 and disrupt the calculations.  
- Assume that sensors don’t lie about their values.
Computing min/max (cont.)

- First construct a spanning tree so that the root holds the minimum element.

<table>
<thead>
<tr>
<th>$p_i$</th>
<th>$V_i$</th>
<th>id$_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>$S_1$</td>
<td>3</td>
</tr>
<tr>
<td>$S_2$</td>
<td>$S_2$</td>
<td>5</td>
</tr>
<tr>
<td>$S_3$</td>
<td>$S_3$</td>
<td>4</td>
</tr>
<tr>
<td>$S_4$</td>
<td>$S_4$</td>
<td>1</td>
</tr>
</tbody>
</table>
Computing min/max (cont.)

- First construct a spanning tree so that the root holds the minimum element.

<table>
<thead>
<tr>
<th></th>
<th>$p_i$</th>
<th>$V_i$</th>
<th>id$_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>$S_1$</td>
<td>3</td>
<td>id$_1$</td>
</tr>
<tr>
<td>$S_2$</td>
<td>$S_1$</td>
<td>3</td>
<td>id$_1$</td>
</tr>
<tr>
<td>$S_3$</td>
<td>$S_4$</td>
<td>1</td>
<td>id$_4$</td>
</tr>
<tr>
<td>$S_4$</td>
<td>$S_4$</td>
<td>1</td>
<td>id$_4$</td>
</tr>
</tbody>
</table>
Computing min/max (cont.)

- First construct a spanning tree so that the root holds the minimum element.

<table>
<thead>
<tr>
<th></th>
<th>p_i</th>
<th>V_i</th>
<th>id_i</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_1</td>
<td>S_1</td>
<td>3</td>
<td>id_1</td>
</tr>
<tr>
<td>S_2</td>
<td>S_3</td>
<td>1</td>
<td>id_4</td>
</tr>
<tr>
<td>S_3</td>
<td>S_4</td>
<td>1</td>
<td>id_4</td>
</tr>
<tr>
<td>S_4</td>
<td>S_4</td>
<td>1</td>
<td>id_4</td>
</tr>
</tbody>
</table>
Computing min/max (cont.)

- First construct a spanning tree so that the root holds the minimum element.

<table>
<thead>
<tr>
<th>$p_i$</th>
<th>$V_i$</th>
<th>$id_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>$S_2$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$id_4$</td>
</tr>
<tr>
<td>$S_2$</td>
<td>$S_3$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$id_4$</td>
</tr>
<tr>
<td>$S_3$</td>
<td>$S_4$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$id_4$</td>
</tr>
<tr>
<td>$S_4$</td>
<td>$S_4$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$id_4$</td>
</tr>
</tbody>
</table>
Computing min/max (cont.)

**Algorithm:**
- First construct a spanning tree so that the root holds the minimum element.
- Aggregator $A$ commits the tree and reports the root of the tree to the server.
- Home server $B$ randomly picks a node in the list and traverses the path from the node to the root. If unsuccessful, $B$ rejects.

**Thm:** If no more than $\varepsilon$ fraction of sensors are corrupted, and diameter is $d$, home server requests $O(d/\varepsilon)$ and rejects invalid minimum with probability $\geq 1 - \varepsilon$. 
Counting distinct elements

- **Goal:** Given $a_1, \ldots, a_n$, how many distinct measurements (denoted $\mu$) are there?
- For example, $(1, 3, 5, 7, 3, 3)$ has 4 distinct measurements.
- $n'$ corrupted sensors can alter the result by at most $n'$
Counting distinct elements (cont.)

- **Basic idea:** use [FM83] algorithm for counting distinct elements in a data stream $a_1, \ldots, a_n$:
  - Pick a random hash function $h: [m] \rightarrow [0...1]$.
  - Keep the value $v = \min_{i=1}^n h(a_i)$.
  - Let $\mu' = 1/v$. Then $\mu/c \cdot \mu' \cdot c\mu$ for all $c > 2$ [AMS96].
Counting distinct elements (cont.)

- Can adapt the data-stream algorithm to compute the minimum:

```plaintext
procedure CountDistinct
1. Home station chooses $h \in H$ and through aggregator announces $h$ to each sensor.
2. Each sensor $S_i$ computes $h(a_i)$.
3. Run FindMin algorithm to compute $v = \min_{i=1}^{n} h(a_i)$.
4. return $1/v$.
```
Protocol to count distinct elements can be used to:

- Compute the size of the network.
  - Run it on the set of sensor identifiers \( \{\text{ID}_1, \ldots, \text{ID}_n\} \).

- Compute the average of \( a_1, \ldots, a_n \)
  [EKR99].
  - Run it on \( \Psi = \{(i, j) \mid 1 \leq i \leq n, 1 \leq j \leq a_i\} \). Then \( \text{avg} = |\Psi|/n = \left(\sum_{i=1}^{n} a_i\right)/n \).
Outline

- Motivation
- The model
- Results
  - Median
  - Min/max
  - Counting distinct elements
- Conclusion
Conclusion

Information aggregation when sensors and aggregator are malicious can be hard.

- Proposed aggregate-commit-prove framework.
- Gave concrete sublinear protocols for:
  - min/max
  - median
  - counting
  - average
My conclusion

- The article studies an interesting problem of SIA
- Aggregate-commit-framework is useful, yet gives no general cookbook recipe for constructing protocols
- Some assumptions are unrealistic:
  - single aggregator
  - knowledge of topology of the sensor network: diameter $d$, number of nodes $n$, connected components
- It would be interesting to try to lift some of these assumptions and consider classes of general aggregation operators.