Security and Reliability of the Internet Of Things (IoT): A Smart Meter Case Study

Karthik Pattabiraman
Farid Molazem Tabrizi, Maryam Raiyat, Abraham Chan, Ivan Beschastnikh

University of British Columbia (UBC)
My Research

• **Building fault-tolerant and secure software systems**

• **Application-level fault and attack tolerance**
  • Software resilience techniques [SC’16][DSN’16][DSN’15][DSN’14A][DSN14B]
  • Web applications’ reliability [ICSE’16][ICSE’15][ICSE’14A][ICSE’14B]
  • IoT Security [ACSAC’16][EDCC’15][HASE’14]

• **This talk**
  • IoT Security and Reliability: Smart Meter Case Study
IoT Systems are Everywhere
IoT Security and Reliability
IoT Security and Reliability: Challenges

• **IoT devices are resource constrained**
  • Low memory and computing capacity
  • Sometimes energy constrained

• **Large scale of deployment**
  • Worms can spread quickly in the network
  • Need scalable solutions with low false positives

• **Autonomous operation**
  • Need for human intervention should be minimal or none
  • Must be capable of operating continuously for a long time
IoT Example: Smart Meters
Smart Meter

- Energy
- Sensors
- Power line/Wireless
- Cellular
- Internet
- Utility Server
Global Status of Smart Meters

- 2009: 76 million
- 2010: 118 million
- 2012: 1 billion
Smart Meter Security

• **Smart meter Attacks**
  • No need for physical presence
  • Hard to detect by inspection or testing
  • Attacks can be large-scale
Smart Meter Security is a concern
Outline

• Motivation and Goals

• Host-based Intrusion Detection System (IDS) for smart meters [EDCC’15 – Distinguished Paper Award][HASE’14]

• Model checking to find design vulnerabilities in smart meters [ACSAC’16]

• Ongoing Work and Conclusions
IDS: Goal

- **Goal**: Make IoT embedded devices secure
  - Build a host-based intrusion detection system

- **Important constraints**
  - Small embedded devices => Low memory capacity
  - Large scale => No false positives
  - Low cost => Automated, no special hardware etc.
IDS Challenge: False Positives
IDS Challenge: Memory Constraints

```c
{a = receive();
if (a > 0)
    foo(a);
else
    bar(a);
}

void foo(int a) {
    if (a % 2 == 0)
        even(a);
    else
        odd(a);
}

void bar(int a) {
    if (a == -1)
        error1();
    else if (a == -2)
        error2();
}
```
IDS Existing Solutions

- Statistical Techniques [Moradi][Warrender]
- Program Analysis Techniques [Wagner][Giffin]

Our goal

False-Positives vs. Memory Consumption
IDS Threat model

- Adversary: Wants to change the execution of the software (in subtle ways) to avoid detection. Do not consider privacy or confidentiality.
IDS: Main Idea

• Quantify security to detect only the most critical attacks, subject to memory constraints
IDS Approach: Overview

Coverage function

Our work

Code

Software Design Documents (SDD)

Invariants

Monitoring Software trace

IDS
IDS Approach: Details

1. Study Software Design Document
2. Generating abstract Invariants
3. Static Analysis
4. Generating concrete invariants
5. Select optimized invariants

Software Design Documents (SDD)
• Storage/Retrieval integrity

Sensor data must eventually be stored on flash memory
\(\Box (getting\ sensorData \Rightarrow (\Diamond \text{store on flash}))\)
IDS Approach: Steps 3-4

Abstract invariants → Concrete invariants (contain system calls)

1- Study Software Design Document
2- Generating abstract invariants
3- Static Analysis
4- Generating concrete invariants
5- Select optimized invariants

Code

Coverage function
\(\square(\text{getting sensorData(data)} \Rightarrow (\Diamond \text{store on flash(data)}))\)

\[\square(\text{receive(d)} \Rightarrow (\Diamond \text{write(d)})]\)
IDS Approach: Step 5

1- Study Software Design Document
2- Generating abstract invariants
3- Static Analysis
4- Generating concrete invariants
5- Select optimized invariants

Software Design Documents (SDD)

Code

Coverage function
IDS Approach: Building the IDS

Formulate building the IDS as an optimization problem, where we maximize coverage subject to cost constraints
IDS Coverage: MaxMin Coverage

MaxMin Coverage IDS: Maximize minimum coverage i.e., distribute coverage among all properties
IDS Coverage: MaxProperty IDS

MaxProperty IDS:
Maximize security properties that are fully covered
IDS: Building the IDS

Select the invariants from the graph according to the coverage function

Automatically convert it to Buchi Automaton

Monitor the invariants at runtime
IDS Evaluation: Testbed

• Testbed: Smart Meter

• Meter:
  • Arduino board
    • ATMEGA 32x series microcontroller
    • Sensors
  • Gateway board
    • Broadcom BCM 3302 240MHz CPU
    • 16 MB RAM
    • 4 MB available for IDS
    • OpenWRT Linux

• IDS runs on the Gateway board
IDS Evaluation: Fault injection

• Flipping branches (surreptitiously)

```lisp
if (data_file =~ nil) then
  big_string = data_file:read("*all")
end

if (data_file == nil) then
  big_string = data_file:read("*all")
end
```
IDS Results (MaxMin IDS: 2 MB memory)

- How good is the coverage of the IDS (left)?
- How good the graph-based optimization is reflected at run-time (right)?

![Graphs showing detection and coverage precision for various security properties.]
IDS Results (MaxProperty IDS: 2 MB memory)

- How good is the coverage of the IDS (left)?
- How good the graph-based optimization is reflected at run-time (right)?
Outline

• Motivation and Goals

• Host-based Intrusion Detection System (IDS) for smart meters [EDCC’15 – Distinguished Paper Award][HASE’14]

• Model checking to find design vulnerabilities in smart meters [ACSAC’16]

• Ongoing Work and Conclusions
Model Checking: Problem

Enumerate all possible attacks

void foo() {
    
}
Model Checking: Challenge

• Formal analysis requires well-defined properties (e.g. TCP/IP)
  • Unclear in IoT devices

• The state space may be very large
  • Require the *right* level of abstraction
    • High-level enough to avoid state space explosion
    • Low-level enough to be translatable to device code
Model Checking: Our approach

- Key Idea: Each class of embedded devices performs similar operations
  - We can abstract the operations
  - Create an abstract model
    - Formalize the model (using Maude)
    - Formalize attacker actions
    - Define unsafe states
    - Run model checking to find attacker actions leading to unsafe states
Model Checking: Formal model

```
SENSOR-STATES
1. mod SENSOR-STATES is
2. op getSensorDataList : —> SensorDataList.
3. var dataList : SensorDataList.
4. var r n : Nat.
5. rl[r1]: getSensorDataList —> sensorDataElement(0,0).
6. crl[r2]: sensorDataElement(r,n) —> sensorDataElement(r,n)
   sensorDataElement(r+1, 0)  if r < maxSensorNumber.
7. crl[r3]: sensorDataElement(r,n) —> sensorDataElement(r,n+1)
   if n < maxSensorData.
8. endm
```

- Defines the operation of receiving data from sensors.
  SensorDataList is a list of tuples, each called sensorDataElement.
- Defines necessary variables for defining the operations.
- Base of recursion.
- Recursively defining the rule to extend one sensorDataElement, to up to maxSensorNumber elements. Each tuple is: [value, sensor channel number].

Model Checking: Threat model

Root access to a node in grid network [Mo et al. 2012]

• Actions
  • Drop messages
  • Replay messages
  • Reboot meter

Read/Write access to communication interfaces [McLaughlin et al. 2010]
Model Checking: Results

• For each attacker action: query for paths to unsafe states, e.g.,

  • \textit{search sensor}(N1, M1) \textit{sensor}(N2, M2) \textit{sensor}(N3, M3) \Rightarrow \textit{sensor}(N1, M1) \textit{sensor}(N2, M2)

• Checks if any data may be lost via dropping messages

• Found many attacks: Many map to the same execution path
Model Checking: Attacks example 1

```
Function confirm_time_is_OK()
    while time_is_ok == false do
        ...
        time_is_ok = check_time()
        if (time_is_ok == true) then
            set_time()
            break
        end
    end
end
```

Gets stuck in the loop:

- iptables -A INPUT -d ADDRESS -j DROP

Root access to a routing node

Add IPTables rule: drop messages to time server

Server

Meter
Model Checking: Attack example 2

Sensor board

Communication board

Data

Request

Normal behavior

Find serial communication configuration (a handful common configs, a couple of hundreds total configs)

One of the common configs worked in our case

Use USB to 6-pin serial connector from laptop to meter

Replay data request

Receive data on the laptop – data deleted from sensor board
Model Checking: Attack example 3
Model Checking: Performance

<table>
<thead>
<tr>
<th>Attacker action</th>
<th>Time (hrs)</th>
<th>Attacks found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dropping packets</td>
<td>0.002</td>
<td>12</td>
</tr>
<tr>
<td>Replay</td>
<td>0.005</td>
<td>845</td>
</tr>
<tr>
<td>System reboot</td>
<td>1.9</td>
<td>6452</td>
</tr>
</tbody>
</table>
Outline

• Motivation and Goals

• Host-based Intrusion Detection System (IDS) for smart meters [EDCC’15 – Distinguished Paper Award][HASE’14]

• Model checking to find design vulnerabilities in smart meters [ACSAC’16]

• Ongoing Work and Conclusions
Invariants: ARTINALI

- A Real-Time-specific Invariant Inference Algorithm
  - Mining independent properties

- Finding Temporal relationship of independent properties

- Incorporating time properties into data invariants
Invariants: ARTINALI VS. Previous work

- ARTINALI (D|E Miner)
- Dysy [ICSE’08]
- Daikon [IEEE’01]
- ARTINALI (D|T Miner)
- Quarry [ICSE’15]
- Gk-tail [ICSE’08]
- Perfume [ASE’14]
- ARTINALI (T|E Miner)
- Texada [ASE’15]
Invariants: Synchronization Tampering Attack

Detection: violation in time per event invariant:

\[ \text{send} (T_0 + K \times 15) \rightarrow \text{send} (T_0 + (K+1) \times 15) \]
Diversity: Motivation

- One compromised device will not lead to attacks on other similar devices
Diversity: Code Reuse Attacks

Code Injection Attack

Code Reuse Attack
Diversity: Functional Correctness vs Security?

- Compilable Variants
  - Variants Break Tests
    - Semantic Non-Preserving Variants but Passes Tests
      - Semantic Preserving Variants
Conclusions

• **IoT Security and Reliability are important**
  • Challenging due to memory and resource constraints
  • Physical access to the device is possible

• **Smart Meters: Important class of IoT device**
  • Host-Based IDS to detect intrusions
  • Model checking to find design defects

• **Ongoing Work**
  • Extracting invariants for runtime monitoring (ArtiNali)
  • Enhancing diversity among deployed variants (NVerD)
Security and Reliability of the Internet Of Things (IoT): A Smart Meter Case Study

Karthik Pattabiraman
Farid Molazem Tabrizi, Maryam Raiyat, Abraham Chan, Ivan Beschastnikh

University of British Columbia (UBC)