Model-based Intrusion Detection System (IDS) for Smart Meters

Karthik Pattabiraman and Farid Tabrizi
Dependable Systems Lab
University of British Columbia (UBC)
My Research

• Building fault-tolerant and secure software systems

• Application-level fault tolerance
  – Software resilience techniques [DSN’14][DSN’13][DSN’12]
  – Web applications’ reliability [ICSE’14][ICSE’14][ESEM’13]

• This talk
  – Smart meter security [HASE’14][WRAITS’12]
Smart Meter Security

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

Smart meter hacking tool released

Summary: Terminator, an open-source tool designed to assess the security of smart meters, has been released.

FBI: Smart Meter Hacks Likely to Spread

A series of hacks perpetrated against so-called “smart meter” installations over the past several years may have cost a single U.S. electric utility hundreds of millions of dollars annually, the FBI said in a cyber intelligence bulletin obtained by KrebsOnSecurity. The law enforcement agency said this is the first known report of criminals compromising the hi-tech meters, and that it expects this type of fraud to spread across the country as more utilities deploy smart grid technology.

Smart meters are intended to improve efficiency, reliability, and allow the electric utility to charge different rates for
Goal

• **Goal:** Make smart meters secure
  – Build a host-based intrusion detection system (IDS)
  – Detect attacks early and stop them

• **Why is this a new challenge?**
  – Smart meters have unique constraints that make them different from other computing devices
  – Existing techniques do not offer comprehensive protection
Outline

• Motivation and Goal

• Prior work and constraints

• Our approach

• Evaluation

• Formal modeling

• Conclusion
Prior Work on Smart Meter Security

- Network-based IDS [Barbosa-10][Berthier-11]

- Remote Attestation [LeMay-09][OMAP-11]
Why (bother with) Host-based IDS?

• Defense in depth
  – Complement network-based IDS: False negatives
  – Can detect both physical and network attacks

• Remote attestation techniques do not cover attacks that change dynamic execution of the meter at runtime, e.g., control-flow hijacking
Constraints of smart meters

• **Performance**
  – Low-cost embedded devices; memory constrained

• **No false positives**
  – False-positive rate of 1% => 10,000 FPs in 1 million meters

• **Software modification**
  – Software has real-time constraints; no modifications

• **Low cost**
  – Rules out special cryptographic hardware or other additions

• **Coverage of unknown attacks**
  – Attacks are rapidly being discovered; zero-day attacks
Prior Work on Host-based IDS

<table>
<thead>
<tr>
<th>System</th>
<th>Performance</th>
<th>No False Positives</th>
<th>No Software Modification</th>
<th>Low Cost</th>
<th>Unknown attacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dyck</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>NDPDA</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>HMM/NN/SVM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Statistical Techniques</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

No existing host-based IDS can satisfy all five constraints: Need for new IDS
Outline

• Motivation and Goal
• Prior work and constraints
• Our approach
• Evaluation
• Formal modeling
• Conclusion
Threat model

• Adversary: wants to change the execution path of the software (in subtle ways)
Approach

• Build a model of the meter software
  – Meters are designed to do specific tasks
Approach

Abstract Model

Concrete Model

University of British Columbia (UBC)
Abstract Model

• Build an abstract model based on standard specifications of smart meter functionality
Model

1. Initialization
2. Check for input commands
3. Process commands
4. Read data from sensors
5. Calculate consumption
6. Pass data to be sent to server
7. Receive consumption data from controller
8. Check for Availability of the server
9. Save data to the physical storage
10. Read data from physical storage
11. Submit all data to the server
12. Check for incoming commands from the server
13. Send commands to the controller

Communication Processes
Controller Processes

University of British Columbia (UBC)
Approach

Abstract Model

Concrete Model

Syscall1

Syscall2
Building the concrete model

• Use a tagging system

```c
// <network, serial, b2>
SerialHandler()
{
    ...
}
```

• Features
  – Ease of use
  – Flexibility
Concrete Model

1-Initialize

setup -> segMeterInitialize -> serialInitialize

2-Check input commands

serialHandler() -> parseCommand

3-Process commands

seg_commands.pars -> relayCommand

4-Read sensors

segMeterHandler

5-Calculate consumption

collectChannelTransduced

collectChannelRMS

6-Pass results to be Submitted to server

sendMessage

powerOutputHandler
Approach

Abstract Model

Concrete Model
IDS Generation: Attack Database

• Build the IDS based on system calls

7-Receive consumption data from controller

8-Check for Availability of the server

9-Save data to physical storage

10-Read data from physical storage

Attack Database
Example Attack

• Communication interface attack

6-Pass data to be sent to server

7-Receive consumption data from controller

Data spoofing

Pass data to be sent to the server

Save data in the buffer

sendMessage():

ser2net

Receiver consumption data from the controller

serial_handler():
System Call Selection: Algorithm

- Generate the set of all system calls of the meter
- Traverse the attack database
- Map the attacks to functionalities of the concrete model
- Map system calls to functionalities
- In the end: system calls associated with the attacks are mapped to the concrete model blocks
- Pick system calls that cover the most blocks until all blocks are covered
- Generate the state machine of the system calls based on the graph

(send, recv, connect)
Model-Based IDS: Implementation

• **Compile time:** Extract state machine of sys calls
  – Input: Annotated code
  – Output: state machine

• **Run time:** Check sys call sequences
  – Logger: attaches `strace` to the process being monitored and logs system call traces
  – Checker: Runs every $T$ second, parses the generated system calls, compares the logged trace with model
Outline

• Motivation and Goal

• Prior work and constraints

• Our approach

• Evaluation

• Formal modeling

• Conclusion
Experimental Setup

• SEGMeter
  – Arduino board
    • ATMEGA 32x series
    • Sensors

– Gateway board
  • Broadcom BCM 3302 240MHz
  • 16 MB RAM
  • OpenWRT Linux

– IDS runs on Gateway board
Results: Performance

Performance
- Time taken to check the syscall trace / time taken to execute the meter software - produce the trace

<table>
<thead>
<tr>
<th>Memory available</th>
<th>12 MB</th>
<th>9 MB</th>
<th>6 MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full-trace IDS</td>
<td>165.2%</td>
<td>214.6%</td>
<td>315.1%</td>
</tr>
<tr>
<td>Our Model-based IDS</td>
<td>4.0%</td>
<td>4.0%</td>
<td>4.0%</td>
</tr>
</tbody>
</table>

Full-trace IDS cannot keep up with the software, while our model-based IDS incurs low overheads
Results: Coverage (Known Attacks)

• Detection (Known attacks)
  – Implemented four different attacks [WRAITS’12]
    • Communication interface attack
    • Physical memory attack
    • Buffer filling attack
    • Data omission attack

  – Our Model-Based IDS detects all four attacks
    • If undetected, the attacks lead to severe consequences
Results: Coverage (Unknown Attacks)

- **Detection (Unknown attacks)**
  - **Code injection**
    - Select a procedure to inject in the smart meter
    - Mutate the procedure by copying and pasting 1-8 lines of code from some other part of it (harder to detect)

<table>
<thead>
<tr>
<th>Component</th>
<th>Random (%)</th>
<th>Popular system calls (%)</th>
<th>Full-trace (%)</th>
<th>Model-based</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Minimum</td>
<td>Average</td>
<td>Maximum</td>
<td></td>
</tr>
<tr>
<td>Server communication</td>
<td>32</td>
<td>36</td>
<td>92</td>
<td>59</td>
</tr>
<tr>
<td>Storage and retrieval</td>
<td>14</td>
<td>44</td>
<td>84</td>
<td>73</td>
</tr>
<tr>
<td>Serial communication</td>
<td>42</td>
<td>28</td>
<td>88</td>
<td>67</td>
</tr>
<tr>
<td>Average</td>
<td>29.3</td>
<td>36.0</td>
<td>88.0</td>
<td>67.4</td>
</tr>
</tbody>
</table>
Results: Monitoring Latency

- Monitoring latency
  - Smaller $T$: Faster detection, higher performance overhead
  - We pick $T = 10s$
    - Low performance overhead: 4%
    - Full trace can’t keep up even with $T=60s$
Outline

• Motivation and Goal
• Prior work and constraints
• Our approach
• Evaluation
• Formal modeling
Towards formal modeling

- Manual checking of IDS
  - Inaccuracy
  - Effort

- Formal Modeling
  - Formal definition of the flaws
  - Formal definition of the model

- Goals: Speed and accuracy
Formal Modeling: Approach

• **We model the operations of the smart meter**
  – Low level (code level)

• **What do we do with the model?**
  – Define invariants:
    • Is it possible to change the consumption data?
    • Is it possible that data not be stored?
    • Is it possible to skip consumption calculation?

• **Test the model against the invariants**
  – Find the flaws → provide potential solutions
Formal Modeling Approach - 1

- We model the operations of the smart meter
  - Low level (code level)

```lua
function process_seg_response(response)
    local win = true
    local command = nil
    ...
    if (response:sub(1, 7) == "(site= ") then
        ...
    if (response:sub(1, 6) == "(node ") then
        ...
    return win
end
```

```lua
module process_res(response, result)

    { input response: string; 
     output result: string; 
     if (...) 
         result = time + consumption; 
     .... 
    }
```

Our Input is the code

- Use the variables of the code as input
- Rewrite the statements
Formal Modeling Approach - 2

• What do we do with the model?
  – Define checks for different invariants

```plaintext
module process_resp(response, result)
{
    input response: string;
    output result: string;
    if (...)
        result = time + consumption;
    ....
    cond1: assert ~(result == nil)
    cond2: assert (response \rightarrow consumption > 0)
    ...
}
```

Will be checked against all possible inputs
• Test the IDS against the model and invariants
  – Find the flaws → provide potential solutions

Example:

response == "" → consumption = 0 (default value)

Attacker can make the string empty ("") even without knowing the encoding scheme

Solution

Add a check for empty string and raise an alarm for it
Outline

• Motivation and Goal
• Prior work and constraints
• Our approach
• Evaluation
• Formal modeling
• Conclusion
Conclusion

• Smart meters have special constraints that make existing host-based IDSes impractical

• Our model-based IDS: practical for smart meters
  – Low performance overhead
  – Good detection coverage
  – Low detection latency

• Formal modeling can help automate the analysis of the software: provide strong guarantees
Future Work and Discussion

• Extend to other SCADA systems (e.g., transportation systems, oil pipelines etc)

• Build a generic framework to reason about trading-off security for performance

• Automated inference of concrete model through static analysis without annotations