IOT: Formal Security Analysis of Smart Embedded Systems

Farid Molazem Tabrizi and Karthik Pattabiraman
University of British Columbia
Vancouver, BC, Canada
{faridm, karthikp}@ece.ubc.ca

ABSTRACT

Smart embedded systems are core components of Internet of Things (IoT). Many vulnerabilities and attacks have been discovered against different classes of IoT devices. Therefore, developing a systematic mechanism to analyze the security of smart embedded systems will help developers discover new attacks, and improve the design and implementation of the system. In this paper, we formally model the functionalitiy of smart meters, as an example of a widely used smart embedded device, using rewriting logic. We also define a formal set of actions for attackers. Our formal model enables us to automatically analyze the system, and using model-checking, find all the sequences of attacker actions that transition the system to any undesirable state. We evaluate the analysis results of our model on a real smart meter, and find that a sizeable set of the attacks found by the model can be applied to the smart meter, using only inexpensive, commodity off-the-shelf hardware.

Keywords

IoT, security analysis, formal model, Smart Meters

1. INTRODUCTION

The Internet of Things (IoT) is a collection of network-enabled physical objects that are embedded with sensors and software, and collect and exchange data [41]. Implanted medical devices, modern cars, and smart grids are examples of widely-used IoT systems. They are equipped with networked embedded devices that carry out critical tasks, and hence, are targets for malicious users.

Problem: There are many vulnerabilities and attacks that have been discovered against IoT devices such as smart meters, modern cars, and medical devices [11, 32, 20, 61, 30, 27, 25, 19]. However, most of these attacks were discovered in an ad-hoc or opportunistic manner, and may hence not be comprehensive. Therefore, developing a systematic mechanism to analyze the security of IoT devices will help developers discover the attacks, improve the design or implementation of the system, and find efficient ways to build security mechanisms to detect the attacks.

Existing solutions: Prior work for analysis of attacks against software systems falls into three classes: 1) attack trees [52, 36], 2) attack patterns [21, 22], and 3) attack graphs [54, 26, 12, 50]. These techniques may be used with known attacks and vulnerabilities. With these techniques, the security analyst builds a model of specific attacks, and analyzes the steps required to apply them. Thus, they require knowledge of the attacks that can be mounted. However, for modern embedded devices such as smart meters, there is no exhaustive database of attack vectors available, and creating such a database is difficult as new threats continue to be discovered against them. Also, as new IoT devices emerge, security analysis techniques that do not require knowledge of known attacks become necessary.

Our approach: To demonstrate our approach, we picked *smart meters* as a testbed. Smart meters are key components of the smart grid. They are installed at homes/businesses, calculate electricity consumption, and communicate with the utility server. It is estimated that by the end of 2016, the worldwide revenue of smart grids will exceed $12 billion [24]. The large scale deployment of smart meters and the criticality of their operations, make smart meters and their security an important concern [28, 48, 62, 37].

In this paper, we develop a three step approach for security of smart meters. In the first step, we build a formal model of smart meters, capturing its main functions. In the second step, we also formally define a primitive set of actions for attackers. This set may include the attacker dropping packets, replaying messages, etc. Note that these actions are not standalone attacks, but rather the capabilities of the attacker that are building blocks of attacks. In the third step, we perform automated search (using model checking) to find out whether it is possible for the attacker to *apply* a sequence of the primitive actions, and transition the system into an unsafe state. An unsafe state is any state for which a user-defined security invariant does not hold. For example, in a smart meter, a state where consumption data is lesser than zero is unsafe as it may result in incorrect billing.

The search of the formal model guarantees that, within the state space of the model, all possible scenarios or attacks that may cause the system to transition to an unsafe state are discovered. Therefore, in contrast to existing techniques, we do not need to have prior knowledge of attacks against the system to evaluate its security. *To the best of our knowledge, we are the first to systematically analyze the*

DOI: 10.1145/1235

security of a real-world IoT device using formal techniques, without requiring prior knowledge of the attacks.

Challenge: Existing papers on formal security analysis target systems with well-defined properties (such as communication protocols) [34, 16, 40], or specific implementations of an application [47]. These systems have standard properties (e.g., in the form of RFCs), which can be formally defined. Many IoT devices do not have such standard implementations that we can translate to a formal model. For these systems, we have to find an appropriate abstraction level that is applicable to different implementations. This is challenging as a very low-level model results in state space explosion [60], while a very high-level model cannot be easily mapped to a specific implementation. For example, in a smart meter, different sensor channels measure their own consumption data. This data, which is communicated between the components of the meter, may be formally modeled as a stream of bits. Any change to this data may be modeled as flipping one or more bits. This model can represent all the changes an attacker may potentially make. However, for a 32 bit data stream, the search space of the model will exceed 4 billion states.

To address the above challenge, we examined the design documents of smart meters, the type of *access* the adversaries have to the device, and attacks against smart meters presented in prior work [44]. We observed that these attacks are the results of specific *accesses* that the adversary has to the device. For example, changing the order of the meter's operations, or controlling the availability of network connection. These let the adversary exploit loopholes in the design-level architecture of the smart meter. We take advantage of this observation and identify the components that are targeted by the adversary and are generally present in various models of smart meters, and formally model them. For example, the communication interface between the sensors and gateway board in a smart meter is an important component. An adversary may drop or replay messages passing through this interface. Given this action, modeling the data as a sequence of tuples (indicating sensor channel and consumption data, as opposed to raw bit-streams) enables us to significantly reduce the search space *and* still model the attacker operations. This allows us to analyze the smart meter in a reasonable amount of time and find attacks against real smart meters (see sec.5).

Contributions of the paper: We make the following contributions:

- We build a formal model of a smart meter in Rewriting logic, using the abstract model of a smart meter presented in Molazem et al.[44], which represents the generic operations of a smart meter. We consider the use cases and specifications developed for smart meters by the corresponding utility providers [2, 17, 18]. Hence, our formal model does not depend on a specific implementation of the smart meter.
- We develop a formal model of capabilities of the attacker for a generic smart meter also in Rewriting logic. The attacker may use a sequence of these capabilities to mount sophisticated attacks on a smart meter.
- We use model-checking on the two models to automatically find sequences of actions that may take the system into an unsafe state. These sequences correspond to the attacks found against the smart meter.
- Using off-the-shelf, inexpensive equipment, we experimentally validate the attacks found on an open source smart meter: SEGMeter [55]. We find that the attacks found by the model checker represent both design-level and implementation-level bugs in the smart meter that cause it to lose data and get stuck in an infinite loop. The attacks were found by the model checker within a couple of hours on a regular desktop computer.

2. RELATED WORK

Below we discuss 1) technologies that may be used to provide security for smart meters and, 2) techniques for performing automated security analysis, and their limitations.

2.1 Techniques for building security mechanisms

Hardware-based techniques: Hardware based approaches provide security through special hardware modules, such as a Trusted Platform Module (TPM) [51]. For embedded systems, pushing the security down to the hardware level has three disadvantages. First, TPMs incur high cost. Using them in millions of embedded systems makes their use an expensive proposition [5]. Second, hardware based solutions are difficult to update. Finally, the memory and power limitations of embedded systems make the use of TMP-based techniques challenging [28].

Intrusion detection systems (IDS): Berthier et. al. [8] formulate a set of guidelines to build IDSes for AMI, and in follow-up work, propose an network-based IDS [9] satisfying the guidelines. Their IDS monitors the communication links and detects abnormality in the traffic according to a previously built model. Network-based IDSes however, cannot fully secure embedded systems, as they may have false negatives that allow attackers to bypass the security mechanism by exploiting software vulnerabilities. Mohan et. al. [43] propose a host-based IDS running on a Hypervisor, for embedded systems equipped with multicore processors. This IDS runs on a dedicated core, and monitors the controller of the system, which is running on the other cores. However, their approach may only be applied to devices that are equipped with multicore processor and a Timing Trace Module (TTM), a special hardware module for obtaining accurate timing information. In prior work [56], we have proposed a host-based IDS for smart meters subject to its memory limitations. However, we manually performed the security analysis for building the IDS, which is error prone.

Remote attestation: Software verification techniques such as virtualization-based remote attestation suggested by LeMay et. al. [31], Pioneer [53], and oblivious hashing [13] verify the integrity of software on a third party machine by executing an instance of the program on a remote server. These techniques require the embedded system to be connected to the network at all times and maintain a fast and reliable connection to the server. Also, the network interface must be constantly active to perform attestation, which consumes substantial power. These are limiting factors for embedded devices which may be mobile, in isolated places, or working on limited battery. Moreover, remote attestation does not ensure that the software running on the device is free of vulnerabilities that may be exploited by an attacker.

Summary: Existing security technologies do not address the limitations of IoT devices such as scalability and hardware constraints. Therefore, we need to develop techniques

for security analysis of IoT devices to be able to make them more secure.

2.2 Techniques for analyzing attacks

Attack patterns: Attack patterns capture the common methods for exploiting system vulnerabilities. Each attack pattern encapsulates information including attack prerequisites, targeted vulnerabilities, attacker goals, and resources required. Thonnar et. al. [58] study a large dataset of network attacks to find the common properties of some of the attacks. They develop a clustering tool and apply them on different feature vectors characterizing the attacks. Gegick et. al. [22] encode attacks in the attack database and use them in the design phase to identify potential vulnerabilities in the design components. Fernandez et. al. [21] study the steps taken to perform a set of attacks and abstract the steps into attack patterns. They study Denial of Service (DoS) attacks on VoIP networks and show that their patterns can improve the security of the system at design time, and help security investigators trace the attacks.

Although integrating attack patterns into the software development process improves the security of the software, it has two disadvantages. First, attack patterns are often at a high level of abstraction, and require significant manual effort to apply. Second, for new systems such as smart meters, there is no well-known attack vector from which we can develop attack patterns.

Attack trees: Attack trees are top-down hierarchical structures in which lower level activities combine to achieve the higher level goals. The final goal of the attacker is presented at the root. Byres et. al. [12] develop attack trees for power system control networks. They evaluate the vulnerability of the system and provide counter measures for improvements. McLaughlin et. al. [39] use attack trees for penetration testing of smart meters. Morais et. al. [45] use attack tree models to describe known attacks, and based on the trees develop fault injectors to test the attacks against the system. They test their analysis technique on a mobile security protocol.

Attack trees are mainly designed to analyze predefined attack goals. However, many security attacks are not targeted and are based on the vulnerabilities that the attackers opportunistically find in the system while testing it. In contrast, we are not bound to specific attack goals, and the user of our model may plug-in their own goals, which they would define as unsafe states of the system.

Attack graphs: Attack graphs have been mainly used to analyze attacks against networked systems. They take the vulnerability information of each host in a network of hosts, along with the network information, and generate the attack graph. Sheyner et. al. [54] and Jha et. al. [26] propose techniques for automatically generating and analyzing attack graphs for networks. They assume that the vulnerability information for each node is available. Based on this information, they analyze the chains of attacks and their effects in the network.

To use attack graphs, the programmer needs the complete set of known vulnerabilities on the host. If the hosts have unknown vulnerabilities, the analysis will be incomplete. In this sense, our work may complement this analysis - we provide security analysis for embedded devices at the node level which may be used as inputs for attack graphs.

Formal analysis: Formal techniques have been used to evaluate the security of computer systems [23]. For example, Matousek et. al. formally verify security constraints on networks with dynamic routing protocols [34]. Delaune et. al. analyze the security of PKCS#11, an API for cryptographic devices [16]. Miculan et. al. formally analyze the security of Signle-Sign-On (SSO) authentication protocols for Facebook [40]. However, these techniques target protocols that have a formal specification. Smart meters do not (yet) have a formal specification that we can convert to a model and formally analyze. Therefore, extending prior work for formally analyzing security of smart meters is challenging.

Summary: Existing techniques for analyzing attacks against embedded systems do not provide guarantees for finding all the attacks within a search space. Also, they require a precise model of the attacks and hence, do not consider unknown attacks. This is important for smart meters as they are relatively new, and do not have a comprehensive attack database. Further, given the long expected lifetime of the meters and the fact that updating them is harder than updating desktop systems, it is important to find vulnerabilities for which there are no attacks yet. Therefore, we need techniques that do not need a comprehensive and precise database of attacks for analyzing their security.

3. BACKGROUND

3.1 Smart Meter

A smart meter is a networked device that measures electricity and communicates with a server. Smart meters have three main components, as we explain below.

Control unit: Inside the meter, there is a Microcontroller that transfers data measured by the low-level meter engine to a flash memory. The Microcontroller can save logs of important events during the activity of the smart meter.

Communication unit: For the meters to be able to communicate with each other and the server, they are equipped with a Network Interface Card (NIC). Meters can be connected to in-home displays, programmable controllable thermostats, etc. to form a Home Area Network (HAN). In each area, smart meters will be connected to a collector through field area network (FAN). This collector gathers all data and communicates with the utility server through Wide Area Network (WAN). The communication interface differs from region to region.

Clock: For the meters to have the capability of providing time-of-use billing services, they are equipped with a real-time clock (RTC). This clock should be synchronized with the server clock on a regular basis to prevent any drift. This is done through synchronization messages.

3.2 Threat model

Access: In this paper, we consider both physical and network attacks against the meter. We assume that the adversary has both read and write access to the communication interfaces of the meter (e.g., WiFi, LAN, serial interface between the components of the meter). This is a realistic assumption as smart meters are installed in locations (e.g., homes, business entities) accessible to people other than the meter vendors. Due to financial benefits that can be gained by tampering with the meter, the owners of the meter installations may act as the adversary as well. For example, open source tools such as Termineter [57] allow communication via the serial interface and optical probe, and sending/re-

	Access	Actions
A1	Physical access to the device	- Tampering with the cover seal - Observing visible indicators - Disabling internal battery
A2	Physical access to the internal/external communication interfaces	- Send/Receive data via communication interfaces
A3	Access to a routing node in the grid network infrastructure	- Dropping data packets

Table 1: Accesses and capabilities of the adversary

playing messages. Accessing the serial interface between the control unit and communication unit of smart meters may need the attacker to remove the seal of the cover of the meter. However, it has been shown that it is relatively easy to do so, and the attacker can erase any traces that the cover has been removed [38]. We also assume that the attackers may intercept communication of the meter, for example, by obtaining root access to some node on the grid network. This is a realistic assumption as it has been shown that the *complexity* and the *scale* of smart grid infrastructure provides several entry points for attackers to obtain such access [42]. However, the attacker *does not* require *root* access to the smart meter itself.

Capabilities: We assume that the attacker is able to drop/replay the messages sent to the smart meter, drop/replay messages sent or received between the control unit of the meter and the network communication unit which is normally via a serial interface, and can restart the meter (e.g., by resetting the power) at a precise time. We do not require the adversary to be able to decrypt any encrypted messages, or to be able to spoof any cryptographic tokens. We do assume however that they are capable of reverse-engineering the meters' binary code or getting access to its source code. This is reasonable as security through obscurity (of the code) is known to be a weak defense strategy [35]. Further, there are many tools that will allow reverse engineering of the source code from a binary file, e.g., Ida Pro [49], BinNavi [10], and OllyDBG [46].

The capabilities of the attacker in our threat model are basic and do not require high level of expertise. We show that even considering these basic capabilities, attackers may mount severe attacks against smart meters (see Sec.5). Note that it is relatively straightforward to model additional capabilities of the attacker in our model as long as the capabilities can be described formally. Also, *increasing the capabilities of attackers makes it easier to find attacks against the system, in our model.*

We have presented the summary of accesses and capabilities of the adversary in Table 1. The attacks that we are targeting in this paper are the results of the vulnerabilities in the software running on the meter. These vulnerabilities may exist due to bugs in either the design or the implementation of the meter software. We are *not* considering network attacks on availability such as Denial of Service (DoS). We are also not considering attacks on privacy of smart meters, e.g., obtaining consumption data, in this paper.

3.3 Rewriting logic

In this paper, we use Rewriting logic [33] to formally model smart meters and attacker capabilities. Rewriting logic is a flexible framework for expressing proof systems. It allows us to define the transitional rules that transition the system from one state to another, as well as the functions defined in the system. Using rewriting logic, we can specify the behavior of the system as a series of states, functions, and rules of transitioning between the states. Also, we can query the transition paths of the system to verify the correctness of the system behavior.

Formally a rewrite theory is a 4-tuple $\Re = \{\Sigma, E, L, R\}$. Σ is the set of all functions (operations), and constants defined in $\Re$. E is the set of all the equations in $\Re$. Equations help define the operation of the system (for example how two variables are added together). R is a set of labeled rewriting rules and L indicates those labels. Rewriting rules model transitions in the system and express the way the system evolves. In a rewrite theory, any of the rewrite rules may be applied concurrently to represent the behavior of a system. This lets us build rules that capture the behavior of both the system *and* the attacker. Using model checking, we can check if any sequence of the attacker's rewrite rules together with the rewrite rules of the smart meter can transition the system to an unsafe state.

In this paper, we implement the formal model of the system in rewriting logic using Maude [15]. Maude is a tool that supports rewriting logic, and enables the users to both execute rewriting logic rules and formally verify them. This is useful as we can execute the model to gain confidence in it before formally verifying it.

Maude allows us to perform model checking of the system with regard to invariants. This means that we can check whether it is possible that from an initial state s in the model, we can reach a state x, in which an invariant I does not hold true. In Maude, this is done by using *search* command: $search(s) \Rightarrow x \; such \; that \; I(x) \neq true$. The invariant I (e.g., data being greater than or equal to 0), may be defined by the user of the system, and we define state x in which the invariant I does not hold as an *unsafe* state.

Maude executes the search by doing breadth-first-search, starting from initial state s, and applying one rewriting rule at a time to the previous state. This way, all possible transitions in the model are checked. This is known as explicit state model checking (ESMC) [6]. Note that other variants of search strategies are possible in Maude e.g., symbolic model checking [6], that aim to tackle the problem of state space explosion that ESMC incurs. We do not explore these in this paper as the state space of the smart meter was still amenable to be analyzed by ESMC within a reasonable amount of time, as shown in Sec. 5.

4. APPROACH

In this section, we propose a formal approach for analyzing security of smart meters. We define a formal model for smart meters, a set of basic actions for an attacker, and use model checking to automatically search all the possible scenarios of applying those actions on the model of smart meters. The search finds the scenarios that lead to potential attacks, and is guaranteed to find all the attacks within the state space of our model (both the smart meter and the attacker).

We follow a three step process for building the model as follows. In the rest of this section, we explain each of the steps in more detail.

- Step 1: We formally model the components of smart meters and their operations. Smart meters are computing devices and can be considered as small general purpose computers. However, unlike general purpose computers, smart meters have low memory, low computing-capacity, and are designed to carry out a specific set of operations. In prior work, we have proposed an abstract model for operations of smart meters in [44]. This abstract model represents an implementation-independent model of the components of the meter, their operations, and their execution order. In this paper, we express the abstract model formally in rewriting logic. Rewriting logic lets us model all the operations (functions) of the system, and the transitions between its states.

- Step 2: We define a set of capabilities for the attacker also in rewriting logic. Modeling both the smart meter and the attacker's capabilities in rewriting logic allows us to automatically and systematically search for all the possible scenarios in which the attacker's actions on the meter can take the system to an unsafe state. An example of an unsafe state is when consumption data calculated by the meter are lost, and not submitted to the server. The users of our model may define their own unsafe states as a first order logic formula over the states of the model.

- Step 3: We compose the model of the smart meter, concurrently with the model of attacker actions. Using model checking, our system searches through all the execution paths of the models that lead to unsafe states. The actions that take the smart meter into an unsafe state will be identified as a potential attack on the system. Because we use model-checking, we are guaranteed to find all the possible paths that may take the system into an unsafe state, within the scope of the model.

4.1 Formal Model

We use the abstract model of smart meters presented by Molazem et. al. [44] as our input to build the formal model of smart meters in rewriting logic. This abstract model presents an implementation-independent model of the components of the meter, their operations, and their execution order. Therefore, it is valid for different implementations of smart meters. Using the abstract model, we extract the execution paths of components of the meter, and formally describe them. Below, we briefly explain the major operations of a smart meter as per the abstract model.

Smart meter's operations: Upon starting, the meter initializes the sensors and communication interfaces. The microcontroller periodically collects data from all the sensor channels by polling them, and averages data samples to calculate consumption data for each channel. Then, the microcontroller listens to incoming data requests from the communication unit, via a serial interface. Upon receiving a data request, consumption data calculated so far are sent to the communication unit of the meter, which stores the data on physical storage. The meter verifies connection to the network and to the server by pinging the server periodically. At specific time intervals, the meter retrieves all the unsent consumption data from the physical storage and transmits them to the utility server via its network interface. The

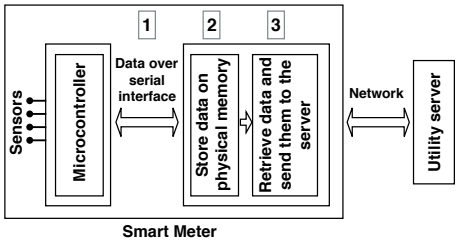

Figure 1: In this paper, we discuss and formalize the first two execution paths of the smart meter, shown in this figure.

communications unit of the meter also periodically checks for any input commands that may be sent from the utility server. The meter parses and verifies any incoming command from the server, and executes them.

We explain the formal model for two parts of the smart meter mentioned above (due to space limitations, we cannot describe the formal model of the entire meter). These parts include: 1) passing the consumption data from the microcontroller to the communication unit the meter, and 2) storing data on the flash memory before submitting it to the utility server. These paths are shown in Fig.1. For clarity and simplicity, we omit some details of the models.

4.1.1 *Passing consumption data to the storage component*

A smart meter has a number of sensor channels. A microcontroller periodically reads each of these channels in a loop, calculates the consumption data associated with them, and produces a stream of sensor data. Below we discuss the formal model for production of a stream of sensor data resulting from sensor channels in the meter. The illustration of sensor data is presented in Fig. 2. Sensors produce data tuples that indicate the index of the sensor and its value. A list of data is formed by putting these tuples together.

The formal model of sensor data is shown in Fig. 3. In line 2 of Fig. 3, we define *SensorElement*, *SensorList*, *SensorNumber*, and *SensorValue*. These are the data types that we use to formally define sensor data and the operations on it. In Maude[1], each of these types is called a *sort*. Each sensor element is a tuple $< s, v >$ (as shown in Fig. 2), which is the result of the operations of the microcontroller on sensor channels. s indicates the channel index, which is of type *SensorNumber*, and v indicates its value, which is of type *SensorValue*. This tuple is formally defined in line 3, by putting two natural numbers (indicated as 'Nat') together. A stream of these tuples forms *SensorList*, which is defined in line 4. *SensorList* is simply built by putting a series of *SensorElement*s together. In line 8 of Fig. 3, we define a common operation on the sensor data: *hasSensor*. This operation verifies whether a stream of *sensorElement*s (i.e., *sensorList*), contains data associated with a specific sensor index, and is defined recursively.

After defining the sensor data in a smart meter, we present the rules that define their production in Fig. 4. We define the production of sensor data using a recursive rule. At each step of the recursion, we either create a tuple of sensor data for a new sensor channel (line 6), or create a new value for

[1]We present a brief primer on Maude in Appendix B.

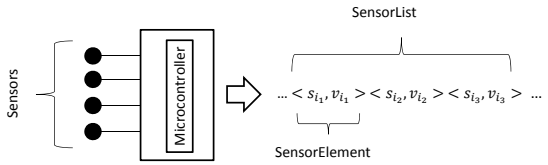

Figure 2: SensorList is a series of SensorElements and is the result of microcontroller operations.

```
SENSOR-DATA
1. fmod SENSOR-DATA is

2. sort SensorElement SensorList SensorNumber
        SensorValue.

3. op sensorElement : Nat Nat -> SensorElement.
4. op __  : SensorList SensorList -> SensorList.
5. op hasSensor : SensorList Nat-> Bool.

6. var r n t : Nat.
7. var dataList : SensorList.
8. eq hasSensor(sensorElement(r, n) dataList, t) =
      if r==t then true else hasSensor(dataList,t) fi.
9. endfm
```

Figure 3: Formal model of sensor data in Maude.

an existing sensor channel (line 7). Line 5 is simply the base case of recursion representing a tuple of sensor data for channel 0. The model lets us define a limit for the number of sensor channels, which is not shown here. The number of sensor channels depends on the specific model of the meter.

4.1.2 Receiving and storing sensor data

After a stream of sensor data is produced by the microcontroller, it is received by another process which is in charge of storing them. This communication is via a serial interface. The receiver process sends a data request message to the microcontroller, and waits for a response. If there is any sensor data available, the microcontroller sends the data, otherwise no data is sent and the request times out.

The state diagram of this procedure is based on the abstract model and is shown in Fig. 6. The formal model for receiving sensor data corresponding to the state diagram is shown in Fig. 5. First, the receiver process creates a socket to communicate with the microcontroller (line 6). This corresponds to states 1 and 2 of Fig. 6. Next, a request is sent (via *askForData* operation) to notify the microcontroller that it is expecting to receive sensor data (line 7). This step corresponds to state 4 of Fig. 6. Then the receiver

```
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
```

Figure 4: Formal model of states of sensor data in Maude.

```
RECEIVE-SENSOR-DATA
1. mod RECEIVE-SENSOR-DATA is
2. sort DataReques.

3. var s: Socket.
4. var request: DataRequest.
5. var dataList: SensorDataList.

6. rl [l1]: startReceivingSensorData -> createSocket.
7. rl [l2]: s -> askForData.
8. rl [l3]: request -> waitForSensorData.
9. rl [l4]: waitForSensorData -> dataList.
10.rl [l5]: waitForSensorData -> timeoutSensorData.
11.endm
```

Figure 5: Formal model of receiving sensor data from the microcontroller.

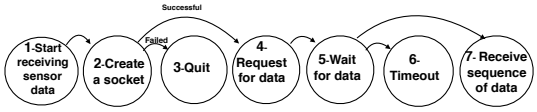

Figure 6: SensorList is a series of SensorElements and is the result of microcontroller operations.

process waits for data (line 8). This corresponds to state 5 of Fig. 6. Eventually, either a stream of sensor data is received (line 9), or the request times out (line 10). These steps correspond to states 6, and 7 of Fig. 6.

After sensor data is received, it will be stored in the flash memory. We model this process by changing the state of each tuple of sensor data (known as *sensorDataElement*), to a new tuple, namely *storedDataElement*. Similarly, a stream of *storedDataElement*s will form *storedDataList*. We present the formal model for representing data storage in Fig. 7. In this figure, we define the types *StoredDataElement* which is a tuple similar to *sensorData* and *StoredDataList*, which is a stream of *StoredDataElement*s (lines 2-5). In line 8 of Fig. 7, the transition of the state of data is defined. The way this transition works is that in the sequence of received sensor data, we change the state of each tuple of sensor data to a tuple of stored data. Eventually, the meter will have a sequence of stored data that it has received and not yet sent to the server (in the absence of attacks).

4.2 Attacker model

A classification of security attacks is presented in recom-

```
STORE-SENSOR-DATA
1.mod STORE-SENSOR-DATA is
2.sort StoredDataElement.
3.sort StoredDataList.

4.op storedDataElement : Nat Nat -> StoredDataElement.
5.op __ : StoredDataElement StoredDataList ->
          StoredDataList.

6.var r n : Nat.
7.var l : StoredDataList.
8.rl [r1] : sensorDataElement(r,n) l ->
            storedDataElement(r,n) l.
9.endm
```

Figure 7: Formal model of storing sensor data received from the microcontroller.

```
ATTACKER-ACTIONS
1. mod ATTACKER-ACTIONS is
2. op crash : -> state.

3. var num : NodeNumber.
4. var val : Nat.
5. var element : SensorDataElement.
6. var list : SensorDataList.
7. var s c p : State.

8. rl [DropMessage] : element list -> list.
9. rl [Reboot] : s -> reboot.
10.rl [Replay] : c -> p if before(c, p).
11.endm
```

Figure 8: Formal model of the attacker actions.

mendation X.800[2], and RFC 4949[3], which divide the attacks into passive attacks and active attacks. Passive attacks involve gaining information about the system, but do not affect the system resources. Active attacks modify system resources or its operations. We focus on active attacks.

We formally define actions for dropping messages, rebooting and restarting the system (to interrupt data flow and message processing), and replaying a message. These actions are simple and can be done by ordinary users of smart meters (see Sec.3.2). It is possible to extend the set of attacker actions to more sophisticated ones.

We present the formal rules for the attackers' action in Fig. 8. Dropping a message is defined in line 8 of Fig. 8 for dropping *SensorDataElement*s. The complete set of rules include other communication protocols of the meter. As a result of this rule, any element of sensor data, at random, may be dropped by an attacker.

Line 9 presents the general rule for rebooting the system. This action may correspond to simply rebooting the meter by unplugging it from power and plugging it back in. To define this action, we define an extra operation *reboot*. At any state s, we can transition to a *reboot* state from the current state s. For instance, while the system is generating a series of sensor data tuples, transitioning to the *reboot* state will interrupt the normal execution path as the rules for generating sensor data cannot be applied anymore. This action can hence lead to data loss.

Line 10 presents a rule that lets the system go from current state c to a previous state p. This transition is not part of the legitimate flow of the system. p is replaced by any state in the system that involves communication. By transitioning back to such a state, the model can re-execute the communication procedure. This rule models an attacker that replays messages sent between components of the meter via its interfaces, e.g, serial interface. Equation *before* in line 10, will return *true*, if state p is a prior state.

By adding these extra actions to the rules of the system, we are able to search through the execution steps and verify whether we can reach unsafe states. Examples of unsafe states are those in which produced sensor data are not stored on flash, and transitioning to a data submission state while the socket is closed. Note that not all the unsafe states produced necessarily represent a feasible attack on the real smart meter. We discuss this in more detail in Sec. 5.

[2]Security architecture for Open Systems Interconnection for CCITT applications

[3]Internet Security Glossary Version 2

Mapping the results of formal analysis to the code: We need to map the results of the formal model back to the meter's code to mount the attacks. To facilitate the process of mapping the results of the formal model to the code, we developed a semi-automated tool. The input to the tool is $L = (r_1, r_2, ..., r_n)$, a sequence of rewrite rules $r_i, 1 \leq i \leq n$ that lead to an unsafe state. The output of the tool is the execution paths of the code that may represent L. The process is semi-automated at present, as the the user of the tool needs to manually match the first and the last rewrite rules (r_1 and r_n) to two nodes of the control flow graph, v_1 and v_2. This can be done by providing the id of the rewrite rule and the corresponding function name in the code that implements the rule. The tool performs simple graph traversal and generates the paths between v_1 and v_2 in the control flow graph. These represent the viable paths corresponding to the input L, and are returned to the user. We used this semi-automated tool to translate the results of formal analysis to the meter's code.

5. EVALUATION

In this section we present the attacks found by the model checker, and the results of mounting the attacks on a real smart meter. We address the following research questions:

- **RQ1 (Practicality):** How applicable are the attacks discovered by the model checker on a real smart meter?
- **RQ2 (Performance):** How long does it take for the model checker to discover the attacks?

5.1 Testbed

Formal analysis testbed: Our test machine on which we run our Maude model checker [15] consists of a 3.4GHz Intel CPU and 16GB of RAM. It runs Ubuntu Linux.

Smart meter testbed: To evaluate the results of the formal analysis, we use SEGMeter, an open source smart meter from smart energy groups [55] (Fig. 9). SEGMeter is used by home users and businesses to monitor energy consumption [3].

SEGMeter consists of two main boards: 1) an Arduino board [7] with an ATMEGA32x series microcontroller, which is connected to a set of sensors and calculates consumption information and, 2) a gateway board which has LAN and wifi network interfaces, and communicates with the utility server. The boards communicate with each other through a serial interface. The meter software is split between the two boards, with the communication unit running on the gateway board and the control unit on the Arduino board. The software running on the gateway board consists of about 1300 lines of code written in the Lua language (not counting the communication stack implementation). The software running on the Arduino board consists of about 1500 lines of C code (not including the Arduino libraries).

5.2 Practicality (RQ1)

Our formal model is based on an abstract model of smart meters, as described in Sec. 4.1. Hence, it does not factor in the implementation details of SEGMeter, and some of the attacks found by our model may not be applicable to it. This is because our formal model must be applicable to other implementations of smart meters as well.

In this RQ, we investigate which of the attacks found by the formal model are applicable to the SEGMeter. For each

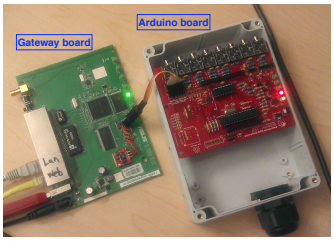

Figure 9: SEGMeter: our open source meter testbed. The Arduino board measures electricity consumption, and the gateway board communicates with the server.

of the attacks, we attempt to execute the attack on SEGMeter, and check if it results in an unsafe state on the meter. The results of this section show that the findings of the formal analysis result in real attacks on the SEGMeter

We need additional hardware to mount some of the attacks. We explain the reasons later in this section. We use widely available inexpensive, off-the-shelf hardware, namely an USB-to-Serial cable and a relay timer, which together cost less than \$50 [4]. These do not require advanced skills to work with (e.g, soldering or working with laser beam), and hence the attacks are easy to mount.

5.2.1 Rebooting meter

An attacker may reboot and restart execution at any point by cutting off the power to the meter. Smart meters may or may not have backup batter. Even if the meter has a backup battery, the attacker may disable it (see Sec. 3.2). As smart meters are deployed at homes and businesses, they must have mechanisms (both at the implementation and the design level) to handle these incidents without losing data. In our testbed, we observed that the system may reboot several times a day due to crashes. Therefore losing data in such incidents is an indication of bugs in the system that attackers may exploit.

We study the effect of rebooting execution by adding its action model (as defined in Sec.4.2), to the model of the smart meter. For this experiment, we define an unsafe state as one in which some of the consumption data is lost. In other words, state s_B, reachable from state s_A, is unsafe, if s_A contains some consumption data that is not included in s_B. Here we consider the states before data is submitted to the server. Below is an example of the search we perform on the model to find such unsafe states (simplified for clarity):

$$search\ sensor(N_1, M_1)\ sensor(N_2, M_2)\ sensor(N_3, M_3) \Rightarrow\ sensor(N_1, M_1)\ sensor(N_2, M_2). \quad (1)$$

The above search phrase considers 3 sensor channels for the meter, represented as $sensor(N_i, M_i)$. N_i indicates the channel index, and M_i indicates its corresponding measured energy. The search finds the paths where data are received from three sensor channels, but only two of them have been stored. This entails that the data measured by one of the sensor channels is lost, and not sent to the server.

Maude found 9 distinct groups of solutions for the cases where the system may face data loss as a result of system reboot. These solutions correspond to four meter components shown in Fig. 1, namely 1) receiving sensor data, 2) storing sensor data to the flash, 3) retrieving data from flash memory, and 4) submitting data to the server. In our experiments, we observed that in three of these components (1, 3, and 4), SEGMeter handles system reboot correctly without losing data. However, we found that component 2, namely storing data to flash memory, does not handle reboot correctly, and is vulnerable to attacks found by our model. In particular, storing data to flash memory lacks proper acknowledgment mechanisms which leads to data loss if the system is terminated at specific points in this component. We explain an example below.

[4]As of May 2016, from ebay.com

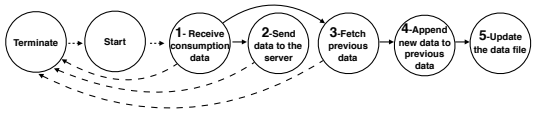

Figure 10: The abstract model for updating sensor data file.

Example of system reboot attack on SEGMeter: To understand this attack, we need to understand how consumption data is updated in our smart meter model. Fig.10 shows the state diagram of this process. In state 1, the meter receives new data from sensors. These data may be directly sent to the server (state 2), or be stored in a data file. The main reasons for storing data before sending them to the server are reduction of communication overhead, and handling temporary unavailability of connection to the server. When storing the data, the meter appends them to the previously stored data (states 3 and 4) and updates the data file (state 5). By letting the attacker reboot the system, our model produced paths from states 1, 3, and 4 of Fig.10, to the initial state of the system. In these paths, the meter receives new consumption data, but does not update the data file, and hence the data is lost when the meter is rebooted.

We profiled the paths of the meter, and found that SEGMeter updates its data files in time intervals of 30 seconds and 42 seconds. Also, we discovered that LEDs on the meter indicate the start time of transferring data and storing them. Leveraging these information, we used a programmable solid state timer to trigger the reboot. We used the timer to ensure we can mount the attack at the precise time for maximum damage. We programmed the timer and repeatedly applied the reboot in 30 second and 42 second time intervals. We found that in 14 out of 20 tries, the new data is lost (i.e., the attack was successful). Further, we found that in 3 cases the attacks had even more devastating consequences. In these cases, rebooting not only erases the new data, *but also wipes out all the previously stored data in the file from the system.* This happens as the meter has an implementation bug, where a file is opened in 'write' mode, as opposed to in 'append' mode and written back. This bug is dormant if there is no attack. Appendix . A.1 provides detailed explanation of this attack, and suggestions for mitigating it.

5.2.2 Dropping messages

One benefit of smart meters is that they enable the utility providers to adopt time-of-use billing. Therefore, smart meters periodically coordinate their clock with a time server, via time synchronization messages. In this attack, we were

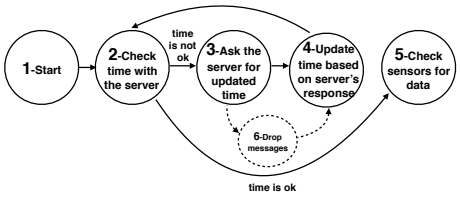

Figure 11: The abstract model for time synchronization. The dotted lines are added by the attacker.

able to successfully compromise time synchronization for the SEGMeter by dropping time synchronization messages.

An example of the search we perform in our model to find scenarios in which dropping time synchronization messages leads to an unsafe state is given below.

$$search \quad timeSyncRequest \quad \Rightarrow \quad incorrectTime. \quad (2)$$

The above search phrase explores the model to see if there are execution paths that result in the meter having incorrect date or time settings, in spite of sending time synchronization messages to the server. This leads to the consumption data having *incorrect* time stamps, which in turn leads to incorrect billing (with a time-of-use billing policy).

Example of message-dropping attack on SEGMeter: Fig. 11 shows the state diagram for time synchronization in the meter. In state 2, the meter sends its current time to the server, and receives a response indicating whether the current time is valid. If the time is valid, the meter goes to state 5, and starts the process of calculating and storing consumption data. Otherwise, it goes to state 3 where it sends a time adjustment request to the server. The server responds with a command to adjust the time on the meter (state 4), and the system checks whether the time adjustment was successful (by going back to state 2). If not, the above process is repeated until it is successful.

The attacker's actions resulted in creating extra states (shown by dotted lines, as state 6) between states 3 and 4 of Fig. 11, where messages are dropped (eliminated). Dropping the messages results in the time value to be *invalid* in our model (as no response is received from the server), and it does not transition to a state with *valid* time, which in turn results in incorrect time-labels for data.

In our lab setup, we have root access to the machine that routes the smart meter traffic to the campus gateway. This corresponds to access A3 in Table 1, and is in line with our threat model in Sec. 3.2. On that machine, we added an IPTables rule that targets the packets destined for the time server and drops them. We observed that this causes the smart meter to get stuck in an infinite loop and hang. As a result, the meter is prevented from recording new consumption data. We present details of mounting this attack, and suggestions for mitigating it in Sec. A.2.

5.2.3 Replaying messages

In the smart meter, replaying a messages involves transitioning to a state prior to sending the message, which may cause the system to malfunction. Below is an example of a search we perform on our model to check if replaying messages can lead to unsafe states.

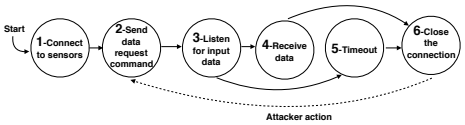

Figure 12: The abstract model for sensor communication

$$search \; ask-for-sensor-data \; \Rightarrow \; nullSocket \; N : sensorData. \quad (3)$$

In the above search, $ask-for-sensor-data$ represents a state where a data-request command is received by the microcontroller. In an attack free execution, the microcontroller sends the newly calculated consumption data, and the other end (i.e., the gateway board) receives the data. In the query, we are checking whether it is possible that the microcontroller sends new sensor data, while the other end of the connection is closed (as indicated by *nullSocket*). This would result in the data being lost as it would be removed from the microcontroller's memory after being sent, but not be recorded, as there is no receiver on the other end.

We found a successful instantiation of this attack on the SEGMeter that was identified by the formal model.

Example of replaying message attack on SEGMeter: Fig. 12 illustrates the state diagram of our model, when the microcontroller communicates with the gateway board of the meter. In states 1 and 2, the gateway board establishes a connection with the microcontroller and sends a data-request command. In state 3, the data storage component listens for input data. If there is any data available, it reads them (state 4). Otherwise, the connection times out (state 5). After all the data is received, or the data request times out, the connection is closed (state 6).

A replay attack makes the system directly transition to states of the model where a message is sent. In this case, such a transition represents jumping to state 2, as pointed to by a dashed arrow in Fig.12. This results in creating a path from state 6 to state 2, and sending the data-request message again after the connection is closed (in our model, socket will transition to a null socket). Going through the data-request transition while the state of the receiver socket is *null* in our model, results in the data *not* transitioning to the *received* state, and later, to the *stored* state. This attack would result in data loss.

We successfully mounted this attack on SEGMeter, using a laptop computer[5] and an USB-to-Serial cable. As a result of this attack, we were able to force the meter to delete the newly calculated data, without saving them. Note that we do not require root access to mount this attack, nor do we need to decrypt any of the messages. Sec. A.3 has more details on this attack, and suggestions for mitigating it.

Summary: We observe that many of the attacks found by the model checker apply to the SEGMeter, and that they result in exposing non-trivial corner cases and bugs in its implementation. Further, most attack can be carried out using inexpensive, off-the-shelf hardware components with little technical expertise on the part of the attacker.

[5]The attack can be carried out through a specialized microcontroller such as an Aurdino, and does not need a laptop.

Attacker action	Time (h)	Attacks Found
Dropping packets	0.002	12
Replay	0.005	845
System rebooting	1.9	6452

Table 2: Performance of model checker for different attacks

5.3 Performance (RQ2)

We measure the time taken to run the searches associated with each attacker action in Maude, along with the number of attack paths for each action found by the model in Table 2. As can be seen, the time varies widely from a few seconds to a couple of hours depending on the kind of attack and the attacker actions. As expected, the larger the state space explored by the search queries, the longer it takes for the search. The search for the effects of dropping packets takes the least time (7 seconds) as it only affects the messages sent/received between the meter components and the server, and as each message has only two states, namely dropped or unchanged. However, the search for the effects of system reboot takes about 2 hours as the system can be rebooted (or not), at every state in the state space of the model, which are much more numerous than messages.

Table 2 shows that when the attacker action affects a larger state space (such as system reboot), the number of paths to explore in the model is higher. However, we observed that many of the paths in the model represent the same attack, applied on different elements of the model (for example dropping different packets of time synchronization, or dropping such packets at different runs of the system). Therefore, although a search query may return hundreds of results, in most cases we only need to try one of them on the code to test whether it applies, as they are all mostly equivalent. This significantly reduces the number of attacks that need to be tested on the code.

Our results show that with a running time of a few hours, the model checker is able to analyze the model and find attacks on different execution paths of the model. Since the analysis is done offline prior to deployment, we do not expect the analysis time to be a bottleneck. Further, our formal model captures the design-level properties of smart meters, and not their implementation. Therefore, the size of the code does not affect the model checker's performance.

Another consideration in evaluating performance of the system is the time taken to successfully map an attack found by the formal model to the implementation. Based on our experience, this process was straightforward and took a few minutes for each attack (maximum duration was half an hour). We also developed a semi-automated tool for this purpose (Section 4). We acknowledge that we were very familiar with the SEGMeter's code and implementation. Because we target the smart meter's developers in our work, we expect them to be even more familiar with their code.

6. DISCUSSION

Applicability to other IoT devices: While we have focused on smart meters in this paper, the same ideas can be extended to other classes of embedded devices in IoT systems. The main requirement for doing so is to 1) identify the viable attacker actions specific to that system, and 2) based on the attacker actions, define an abstraction of the system that we can use to build an implementation-independent formal model. For example, AUTOSAR (AUTomotive Open System ARchitecture) is an open software architecture that provides the basic infrastructure for developing vehicular software. Similar platforms have been proposed for other classes of embedded systems, e.g., medical devices [1, 4]. Given the open and standardized architecture of these systems, we believe we should be able to take a similar approach for them. Extending this approach to these systems is a direction for future work.

Complexity of the attacks: We have demonstrated attacks resulting from dropping messages, replaying messages, and rebooting the system. An attacker is unlikely to mount a successful attack by taking random destructive actions, and needs to carefully time or coordinate her actions to mount these attacks. For actions such as dropping/replaying messages, the number of choices for an attacker are exponential, with respect to the number of messages. Also, the choices for rebooting the meter are exponential in the number of states, which run into thousands. We note that brute-force actions such as replaying *all* the messages, or dropping *all* the messages results in easy detection of suspicious activities (e.g., out of sync heartbeat messages or no heartbeat messages) either by the server or by reviewing the activity logs (manually or by automated tools). Therefore, attackers are unlikely to perform such actions.

Limitations: There are three limitations of our formal modeling approach as follows.

1. Scalability: Increasing the complexity of the model and the number of attacker actions increases the state space of the formal model, which in turn increases the time taken to generate attacks (proportionately). Intuitively, we do not expect the software running on embedded systems to have high complexity as they are normally running on devices with limited computational and memory resources. Therefore, we believe the technique can scale for many classes of embedded systems.

2. Model correctness: The correctness of our results depends on the correctness of the formal model. There are two aspects to correctness. First, there may be a mismatch between the design of the model and the specifications. We mitigate this by building a *single* model for the common features of smart meters, rather than a different model for every different meter. This allows us to refine potential flaws of the model over time by reusing and improving the model. The second aspect of correctness is implementation bugs in the model. We partially mitigate this limitation by using the executable engine of Maude to execute the model and see if, in the absence of attacker actions, it matches the real meter's behavior.

3. Abstraction level: Our model is built at the design level rather than at the implementation level of smart meters. This means that we may not be able to model low level actions for the attacker and discover attacks associated with specific implementations of smart meters (e.g., buffer overflows). Also, not all the attacks found at the abstract level may be mapped to the code. However, despite this limitation, we were able to find many attacks that apply to a real smart meter, SEGMeter.

7. CONCLUSION

IoT devices have gained wide adoption, and their security is an emerging concern. However, existing security techniques do not address their limitations and requirements.

In this paper, we analyze the security of smart meters, a widely-used IoT device. We build a formal model of smart meters using rewriting logic. We also formally define a set of attacker actions, and use model-checking to find all the possible sequences of the attacker actions that may transition the system to an unsafe state. Using the formal model, we were able to find attacks within a modest time of about two hours on the meter. We evaluated the attacks found on SEGMeter, an open source smart meter. We found that a sizeable subset of the attacks found map to real attacks, and can be carried out using commodity, inexpensive hardware that is easy to use, thereby demonstrating their practicality.

Acknowledgements

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Strategic Networks Grants programme for Developing next generation Intelligent Vehicular Networks and Applications (DIVA), and the Discovery Grants Programme.

8. REFERENCES

[1] Green hills medical devices platform. http://www.ghs.com/products/medical_platform.html.

[2] Information and technology standards, advanced metering infrastructure, government of ontario canada. http://www.decc.gov.uk/assets/decc/Consultations/smart-meter-imp-prospectus/1478-design-requirements.pdf.

[3] Monitoring office power consumption with a segmeter. http://www.anchor.com.au/blog/2012/11/monitoring-office-power-consumption-with-a-segmeter.

[4] Wind river medical devices platform. http://www.windriver.com/announces/platform-for-medical-devices.

[5] Najwa Aaraj, Anand Raghunathan, and Niraj K Jha. Analysis and design of a hardware/software trusted platform module for embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 8(1):8, 2008.

[6] Christel Baier, Joost-Pieter Katoen, et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.

[7] Arduino home page. http://www.arduino.cc.

[8] R. Berthier, W.H. Sanders, and H. Khurana. Intrusion detection for advanced metering infrastructures: Requirements and architecture directions. In *Smart Grid Communications (SmartGridComm), 2010*, pages 350 – 355, 2010.

[9] Robin Berthier and William H. Sanders. Specification-based intrusion detection for advanced metering infrastructures. *PRDC, IEEE*, 0, 2011.

[10] BinNavi. https://www.zynamics.com/binnavi.html.

[11] S. Brinkhaus, D. Carluccio, U. Greveler, D B. Justus, and C. Wegener. Smart hacking for privacy. In *28th Chaos Communication Congress*, Berlin, Germany, DEC. 2011.

[12] Eric J Byres, Matthew Franz, and Darrin Miller. The use of attack trees in assessing vulnerabilities in scada systems. In *Proceedings of the International Infrastructure Survivability Workshop*. Citeseer, 2004.

[13] Yuqun Chen, Ramarathnam Venkatesan, Matthew Cary, Ruoming Pang, Saurabh Sinha, and Mariusz H Jakubowski. Oblivious hashing: A stealthy software integrity verification primitive. *Work*, pages 400–414, 2003.

[14] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, and Carolyn Talcott. Maude manual (version 2.7). 2015.

[15] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All about maude-a high-performance logical framework: how to specify, program and verify systems in rewriting logic*. Springer-Verlag, 2007.

[16] Stéphanie Delaune, Steve Kremer, and Graham Steel. Formal security analysis of pkcs# 11 and proprietary extensions. *Journal of Computer Security*, 18(6):1211–1245, 2010.

[17] Department of Energy and Climate Change and the Office of Gas and Electricity Markets. Smart metering implementation programm, March 2011. http://www.ofgem.gov.uk/e-serve/sm/Documentation/Documents1/Design%20Requirements.pdf.

[18] Department of Energy and Climate Change and the Office of Gas and Electricity Markets. Smart metering âĂŞ response to prospectus consultation, March 2011. http://www.ofgem.gov.uk/Pages/MoreInformation.aspx?docid=56&refer=e-serve/sm/Documentation.

[19] Fbi: Smart meter hacks likely to spread. http://krebsonsecurity.com/2012/04/fbi-smart-meter-hacks-likely-to-spread/.

[20] K. Fehrenbacher. Smart meter worm could spread like a virus., 2010. http://earth2tech.com/2009/07/31/smart-meter-worm-could-spread-like-a-virus/.

[21] Eduardo Fernandez, Juan Pelaez, and Maria Larrondo-Petrie. Attack patterns: A new forensic and design tool. In *Advances in digital forensics III*, pages 345–357. Springer, 2007.

[22] Michael Gegick and Laurie Williams. Matching attack patterns to security vulnerabilities in software-intensive system designs. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[23] David Gries. *The science of programming*. Springer Science & Business Media, 2012.

[24] In-stat and ndp group company. http://www.instat.com/press.asp?ID=3352&sku=IN1104731WH.

[25] Hacking Medical Devices for Fun and Insulin: Breaking the Human. https://media.blackhat.com/bh-us-11/Radcliffe/BH_US_11_Radcliffe_Hacking_Medical_Devices_WP.pdf.

[26] Somesh Jha, Oleg Sheyner, and Jeannette Wing. Two formal analyses of attack graphs. In *Computer Security Foundations Workshop, 2002. Proceedings. 15th IEEE*, pages 49–63. IEEE, 2002.

[27] Hacking Humans. http://blog.kaspersky.com/hacking-humans/.

[28] Himanshu Khurana, Mark Hadley, Ning Lu, and Deborah A. Frincke. Smart-grid security issues. *IEEE Security & Privacy*, pages 81–85, 2010.

[29] Christoph Klemenjak, Dominik Egarter, and Wilfried Elmenreich. Yomo: the arduino-based smart metering board. *Computer Science-Research and Development*, pages 1–7, 2015.

[30] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway,

Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental security analysis of a modern automobile. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 447–462, Washington, DC, USA, 2010. IEEE Computer Society.

[31] Michael LeMay, George Gross, Carl A. Gunter, and Sanjam Garg. Unified architecture for large-scale attested metering. In *Proceedings of HICCS'07*, Washington, DC, USA, 2007. IEEE Computer Society.

[32] N. Lewson. Smart meter crypto flaw worse than thought, 2010. http://rdist.root.org/2010/01/11/smart-meter-crypto-flaw-worse-than-thought.

[33] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. *Electronic Notes in Theoretical Computer Science*, 4:190–225, 1996.

[34] Petr Matousek, Jaroslav Ráb, Ondrej Rysavy, and Miroslav Svéda. A formal model for network-wide security analysis. In *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the*, pages 171–181. IEEE, 2008.

[35] Bishop Matt et al. *Introduction to computer security*. Pearson Education India, 2006.

[36] Sjouke Mauw and Martijn Oostdijk. Foundations of attack trees. In *Information Security and Cryptology-ICISC 2005*, pages 186–198. Springer, 2006.

[37] P. McDaniel and S. McLaughlin. Security and privacy challenges in the smart grid. *IEEE S&P*, 2009.

[38] Stephen McLaughlin, Dmitry Podkuiko, and Patrick McDaniel. Energy theft in the advanced metering infrastructure. In *Critical Information Infrastructures Security*, pages 176–187. Springer, 2010.

[39] Stephen McLaughlin, Dmitry Podkuiko, Sergei Miadzvezhanka, Adam Delozier, and Patrick McDaniel. Multi-vendor penetration testing in the advanced metering infrastructure. In *Proceedings of ACSAC'10*, pages 107–116. ACM, 2010.

[40] Marino Miculan and Caterina Urban. Formal analysis of facebook connect single sign-on authentication protocol. In *SOFSEM*, volume 11, pages 22–28. Citeseer, 2011.

[41] Roberto Minerva, Abyi Biru, and Domenico Rotondi. Towards a definition of the internet of things (iot). *IEEE Internet Initiative, Torino, Italy*, 2015.

[42] Yilin Mo, Tiffany Hyun-Jin Kim, Kenneth Brancik, Dona Dickinson, Heejo Lee, Adrian Perrig, and Bruno Sinopoli. Cyber–physical security of a smart grid infrastructure. *Proceedings of the IEEE*, 100(1):195–209, 2012.

[43] Sibin Mohan, Jaesik Choi, Man-Ki Yoon, Lui Sha, and Jung-Eun Kim. Securecore: A multicore-based intrusion detection architecture for real-time embedded systems. In *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Washington, DC, USA, 2013. IEEE Computer Society.

[44] Farid Molazem and Karthik Pattabiraman. A model for security analysis of smart meters. In *WRAITS, Dependable Systems and Networks Workshops (DSN-W)*, 2012.

[45] Anderson Morais, Eliane Martins, Ana Cavalli, and Willy Jimenez. Security protocol testing using attack trees. In *Computational Science and Engineering, 2009. CSE'09. International Conference on*, volume 2, pages 690–697. IEEE, 2009.

[46] OllyDBG. www.ollydbg.de.

[47] Vijayakrishnan Pasupathinathan, Josef Pieprzyk, and Huaxiong Wang. Formal security analysis of australian e-passport implementation. In *AISC'08 Proceedings of the Sixth Australasian Conference on Information Security-(CRPIT Volume 81-Information Security 2008)*, volume 81, pages 75–82. Australian Computer Society, Inc, 2008.

[48] Miguel Correia Paulo Veríssimo, Nuno Ferreira Neves. Crutial: The blueprint of a reference critical information infrastructure architecture. In *Proceedings of ISC06*, August 2006.

[49] Ida Pro. https://www.hex-rays.com/products/ida/.

[50] Indrajit Ray and Nayot Poolsapassit. Using attack trees to identify malicious attacks from authorized insiders. In *Computer Security–ESORICS 2005*, pages 231–246. Springer, 2005.

[51] Dries Schellekens, Brecht Wyseur, and Bart Preneel. Remote attestation on legacy operating systems with trusted platform modules. *Sci. Comput. Program.*, 74:13–22, December 2008.

[52] Bruce Schneier. Attack trees. *Dr. DobbâĂŹs journal*, 24(12):21–29, 1999.

[53] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 1–16, New York, NY, USA, 2005. ACM.

[54] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M Wing. Automated generation and analysis of attack graphs. In *Security and privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 273–284. IEEE, 2002.

[55] Smart energy groups home page. http://smartenergygroups.com.

[56] Farid Molazem Tabrizi and Karthik Pattabiraman. A model-based intrusion detection system for smart meters. In *High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on*, pages 17–24. IEEE, 2014.

[57] smart meter testing framework Termineter. https://code.google.com/p/termineter/.

[58] Olivier Thonnard and Marc Dacier. A framework for attack patterns' discovery in honeynet data. *digital investigation*, 5:S128–S139, 2008.

[59] Arduino UNO. https://www.arduino.cc/en/main/arduinoBoardUno.

[60] Antti Valmari. The state explosion problem. In *Lectures on Petri nets I: Basic models*, pages 429–528. Springer, 1998.

[61] K. Zetter. Security pros question deployment of smart meters. *Threat Level: Privacy, Crime and Security Online*, March 2010.

[62] S. Zonouz, R. Berthier, and P. Haghani. A fuzzy markov model for scalable reliability analysis of advanced metering infrastructure. In *ISGT'12*, 2012.

APPENDIX

In the appendices, we provide more details about the attacks on SEGmeter, and then provide a brief overview of Maude.

A. MOUNTING THE ATTACKS ON THE SMART METER

In this section, we present the details of mounting the attacks explained in Sec. 5 on SEGMeter.

A.1 Rebooting the meter

In this section, we present the details of the attack introduced in Sec. 5.2.1.

We show the snapshot of the code in SEGMeter associated with updating data file in Fig.14. In line 2 (associated with state 3 of Fig.10), previously recorded data (called *all_data*) are read from the data file. In line 3 (associated with state 4 of Fig.10), current data and previous data are merged together. In line 5 (associated with state 5 of Fig.10), the data file is updated with the merged data.

The meter updates the data file in alternating 30 second and 42 second intervals. Smart meters follow a precise procedure for sampling data and calculating consumption, to ensure correct billing. The indicated timing is the result of this process. We measured these by profiling the software running on the meter. Although software profiling may not be feasible for an adversary, we observed that data transmission via serial interface and storing data, are indicated on SEGMeter by a flashing LED. Therefore, even someone who is able to observe the meter (access A1 in Table 1) may synchronize their operation of rebooting the meter with these time intervals.

We used a programmable solid state timer (Fig. 13) to trigger the reboot, to ensure we can mount the attack at the precise time for maximum damage. We placed the timer between the power source and the meter's power adapter. We programmed the timer and repeatedly applied the reboot in 30 second and 42 second time intervals. We found that in 14 out of 20 tries, the new data is lost. However, we found that in 3 cases the attacks not only erase the new data, *but wipe out all the previously stored data in the file*. The reason is that in line 4 of Fig. 14, the data file is opened in *write* mode (shown as 'w' in the code), which erases the contents of the file. This is not a problem in normal execution as the content of the file is read into memory (before overwriting it), and merged with the new data (line 3). However if we reboot the system right after line 4, the meter does not get the chance to write the in-memory data to the persistent storage. Rebooting the meter before the file has been closed results in losing a large portion of previously stored data [6].

Mitigation: For mitigating the attack that loses all data, the meter's implementation should be modified to open the data file in *append* mode. However, the meter is still vulnerable to losing *new data* received from the control unit. Another more fundamental fix is for the flash memory to send an acknowledgement message to the microcontroller after it stores the data. It is only after the reception of this confirmation message that the microcontroller should remove consumption data from its memory. Otherwise, it should re-send the data to the flash memory till the acknowledgement is received.

[6]We found a similar vulnerability on YoMo [29], another open source smart meter, suggesting that this is a common design pattern.

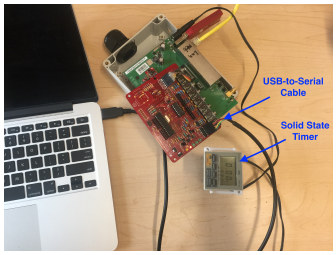

Figure 13: We used a programmable solid state timer for rebooting the system at precise times, and an USB-to-Serial cable to mount replay attack on the meter.

```
1.function update_node_list()
    // state 3
2. all_data = get_node_list()
    …
    // state 4
3. all_data =
     merge_tables(current_data,all_data)
    …
4. data_file = assert(io.open(dataFile, "w"))
    …
    // state 5
5. for key, value in pairs(node_list) do
    …
6.    data_file:write(data)
7. end
    …
8. assert(data_file:close())
    …
9.end
```

Figure 14: SEGMeter code for updating sensor data file. The comments are added by us to show the mapping with the states in Fig. 10.

A.2 Dropping messages

In this section, we present the details of the attack introduced in Sec. 5.2.2.

Figures 15a, 15b, and 15c show the SEGMeter code for time synchronization. Function *check_time()* (Fig.15b) corresponds to state 2 of Fig.11, and communicates with the server to verify whether the current time on the meter is correct. Function *set_time()* (Fig.15c) corresponds to state 4 of Fig.11 and requests a time from the server, and sets the meter's time to the server's time. Function *confirm_time_is_ok()* (Fig.15a) is the main function in charge of time synchronization. It calls *check_time()* in line 3 to verify whether the meter's time is correct. If the time is incorrect, it will call *set_time()* (line 5). This process is repeated in a 'while' loop until the time is set correctly for the meter.

The attacks found by our model correspond to dropping messages in line 2 in *check_time()* or *set_time()* functions. In our lab setup, we have root access to the machine that routes the SEGMeter traffic to the campus gateway. This corresponds to access A3 in Table 1 and is in line with our threat model in Sec. 3.2. In the specified machine, we used IPTables command to drop the desired packets. IPTables is a user-space firewall that is installed by default on all official Ubuntu distributions. One can use IPTables to add rules regarding actions on incoming and outgoing packets. We added a rule to drop the packets that were directed to the meter's time server. This rule is as follows:

$$iptables\ -A\ INPUT\ -d\ ADDRESS\ -j\ DROP$$

In the above rule, ADRESS is the time server's IP address and DROP is the action applied to the packets destined that

```
1.function confirm_time_is_OK()
2. while time_is_ok == false do
    ...
3.  time_is_ok = check_time()
4.  if (time_is_ok == true) then
5.   set_time()
6.   break
7.  end
8. end
9.end
```

(a) confirm_time_is_ok() function

```
1. function check_time()
2.  output = call_seg_api()
3.  if (output != nil) then
4.   result = string.find(output, ''ok'')
5.   if (search_result != nil) then
6.    return true
7.   end
8.  end
9.  return false
10.end
```

(b) check_time() function

```
1.function set_time()
2. output = call_seg_api()
3. if (output != nil) then
4.  time = extract_time(output)
    ...
5.  local executable=''date -s''..time
6.  os.execute(executable)
    ...
7. end
8.end
```

(c) set_time() function

Figure 15: Time synchronization code for SEGMeter.

```
1.function serial_talker()
    // state 1
2.  serial_client = socket.connect()
    …
3.  while (status != "closed") do
     …
     // state 2
4.   command = "(all_nodes (start_data))"
5.   serial_client:send(command .. ";\n")
     …
     // state 3
6.   status = serial_client:receive()
     …
7.  end
    …
    // state 6
8.  serial_client:close()
9.end
```

Figure 16: SEGMeter code for communicating with the sensors on the smart meter. The comments are added by us to show the mapping with the states in Fig. 12.

server.

We observed that after adding the above rule, the boolean variable *time_is_ok* in *confirm_time_is_ok()* will remain false. This will cause the code to get stuck in an infinite loop and hang. As a result, the meter is prevented from recording new consumption data. We note that this attack is applicable regardless of whether the packets are encrypted, as the attacker applies the DROP action only based on the IP address of the packets, and not the payload. It may be possible to detect this attack by monitoring time synchronization packets at the server. However, a cleverer implementation of the attack can allow some packets through to prevent raising suspicion from the server (we did not implement this however).

Mitigation: Developers should define an upper-bound for the number of tries for sending time synchronization messages, and keep track of the tries with a counter. If the counter value exceeds the bound, the meter should stop sending the messages and take other corrective actions. On further investigation, we found that such a counter actually exists in the SEGMeter code and its value is recorded in the logs. However no action is taken based on its value, likely due to an implementation bug. This example shows that our model can even find subtle implementation bugs in the meter's code.

A.3 Replaying messages

In this section, we present the details of the attack introduced in Sec. 5.2.3. Fig.16 shows a snapshot of the meter code associated with the states shown in Fig.12. In line 2 of Fig.16, the communication unit of SEGMeter creates a socket to communicate with the microcontroller (corresponding to state 1 of Fig.12). In lines 4 and 5 of Fig.16, the communication unit prepares a data-request command and sends it to the sensors (corresponding to state 2 of Fig.12). This data-request command is a simple string and is not encrypted. In line 6, the data storage component waits on the socket to receive any incoming data (corresponding to state 3 of Fig.12). In the end, the communication unit closes the connection (line 8, corresponding to state 6 of Fig.12).

To mount the attack, we used a USB-to-Serial cable (13) to connect our laptop to the sensor board via its serial interface (a 6 pin connector). This cable is a USB to TTL level serial interface converter and usually operates at the 5 Volt level. To establish the serial communication, we need to configure the communication parameters. These include: 1) the size of data in a frame (5-9 bits), 2) number of stop bits (1-2 bits), 3) availability of parity (0-1 bit), and 4) the baud rate. We used the default settings for the first three parameters, namely 8 bit data size, 1 stop bit, and no parity, which turned out to be the settings used in the meter. To select the baud rate, we tried the 10 common baud rates ranging from 300 to 230400 and within a few minutes were able to find the baud rate used, namely 38400bps. Thus, we could communicate with the meter through the USB-to-Serial cable.

To communicate with the meter, we sent the same message as in line 4 to the microcontroller. These packets contain the data-request command (shown in line 3). We note that for mounting the attack, the attacker does not need to be able to read the contents of the packet, but only needs to resend the packet to the microcontroller. We observed that as a result of replaying data request command, the microcontroller responds with the new sensor data and erases the data from its memory. However, these data are not received by the gateway board as its connection is closed (line 8 of Fig.16) and consequently, will not be recorded. This leads to incorrect billing to the attacker's benefit. Although we used our laptop to send commands to the meter, an attacker may use a device as simple as an Arduino Uno board [59], to avoid detection upon inspection. The board may even be placed and hidden inside the meter, as it has a small form factor. An Aurdino board costs about $25 on eBay (as of May 2016), and hence this attack is inexpensive.

Mitigation: We note that this attack will be successful even if messages are encrypted (which was not the case for the SEGMeter though). Hence, simple encryption will not mitigate the attack. To mitigate this attack, developers should include unique sequence numbers to requests so that they can be validated by the microcontroller. The entire message needs to be signed cryptographically to prevent attackers from modifying the sequence numbers.

B. MAUDE SYNTAX

In this section, we introduce the syntax of Maude for defining *sort*, *operation*, *equation*, and *rewrite* rules. A comprehensive manual on Maude can be found in Clavel et. al.[14]

Maude consists of *Modules*. They define a collection of *sorts*, operations on *sorts*, equations, and *rewrite* rules to change (rewrite) user inputs.

Sort: A *sort* is a category for value. It can describe any type, including 'integer' and 'list'. A sort is described with the keyword *sort*, and a period at the end:

$$sort\ real\ .$$

A *subsort* specifies a category, that is a subset of another *sort*. For example, 'integer' numbers are a *subsort* of 'real' numbers:

$$subsort\ integer < real\ .$$

Operation: Maude allows definition of operations on the defined *sorts*. An operation is defined using the keyword *op*, followed by the name of the operation , a colon, the names of the *sorts* that are input to the operation, an arrow ($\rightarrow$), the *sort* that is the output of the operation, and a period:

$$op\ \_ + \_ : integer\ integer \rightarrow integer\ .$$

Equation: In Maude, equations are used to define rules for the interpreter, so that it can simplify expressions. An equation is indicated by the keyword *eq*. For example, one rule of addition is that, 0 plus a number equals the same number:

$$eq\ 0 + N = N\ .$$

Rewrite rule: Rewrite rules define the transitions in the system. They are not equations as they apply only in one direction. They determine the changes in the states of the system. Rewrite rules are indicated by the keyword *rl*. For example, assuming we have a model of a networked system in which, under certain conditions, a 'socket' may be closed, we show:

$$rl[socket - rule] : open - socket \Rightarrow closed - socket\ .$$

In the above formula, the term in the bracket is the name of the rewrite rule.