A Model-Based Intrusion Detection System for Smart Meters

Farid Molazem Tabrizi and Karthik Pattabiraman
Department of Electrical and Computer Engineering,
University of British Columbia (UBC), Vancouver
{faridm, karthikp}@ece.ubc.ca

***Abstract*—Smart meters have gained wide adoption as an integral part of the smart grid. However, their security remains problematic as many attacks are discovered against them. Smart meters are embedded devices that are constrained in terms of computing power and memory. They are also deployed on a large scale which imposes specific requirements (e.g., no false positives) on any IDS developed for them.**

In this paper, we propose a model-based technique for building intrusion detection systems (IDS) for smart meters, that takes these constraints into account. We implement our IDS on an open source smart meter platform. We show that our IDS imposes little performance overhead, even under severe memory constraints, and effectively detects a wide range of both known and unknown attacks. In comparison, existing IDSs incur unacceptable performance overheads on the meter.

Keywords: IDS, security model, selective monitoring

I. Introduction

Smart grids are replacing traditional power grids around the world. Studies estimate that over 75% of the US electric grids will be converted to smart grids by 2016 and the worldwide revenue of smart grids will surpass US$12 billion [14]. Unlike traditional grids, smart grids use Advanced Metering Infrastructure (AMI) also known as smart meters, that provide two-way communication capabilities between the end user and the utility provider.

Unfortunately, the rapid deployment of smart grids has resulted in the deployment of smart meters without adequate security and reliability planning [15, 23]. Current estimates indicate that in the US alone, $6 billion is lost by providers due to fraud [20] and the damage resulting from cyber attacks is estimated to be $100 million in 2009 [9]. The financial and strategic benefits that would be accrued from attacking smart metering devices makes security of smart meters an important issue as many vulnerabilities and attacks have been discovered for these systems [7, 17, 21, 29].

Existing work on building security mechanisms for smart meters is limited to network IDS, and remote attestation techniques. For example, Berthier et al. [4] model the normal behavior of the communication of the meters and propose a specification-based IDS based on their model. However, they only detect abnormal network traffic, which may not be indicative of security attacks. Further, network IDSs also may have false negatives that allow attackers to bypass the IDS and attack the smart meter. LeMay et al. [16] propose a virtualization-based architecture for remote attestation of smart meters to ensure the integrity of the meter software and hardware. However, remote attestation techniques do not ensure that the software running on the meter does not have vulnerabilities that can be exploited by the attacker. In this paper, we propose a host-based intrusion detection system (IDS) for smart meters. Smart meters have specific requirements that make existing host-based intrusion detection systems (IDSs) unsuitable for them. These requirements include low performance and memory overheads, scalability to large number of meters, and ability to handle both known and unknown attacks (see Sec. II-B). Our goal is to build a host-based IDS for smart meters that satisfies these requirements. To do so, we use a generic model of a smart meter's software (proposed in our prior work [22]) to extract the security critical parts of the code, and to selectively protect these parts from attacks. Using the model enables us to capture the high level specifications of the software, and tailor the IDS to its characteristics, thus achieving high efficiency without compromising on coverage for the most important attacks.

After capturing the behavior of the software using the model, our IDS monitors the software and detects any deviation from its specified behavior. Such deviations can be caused by an attacker exploiting a vulnerability in the meter software and modifying the normal execution path of the software running on the smart meter. *To our knowledge, ours is the first host-based IDS that is tailored for the requirements and characteristics of smart meters.*

In this paper, we make the following contributions:

- Identify the unique requirements of IDSs for smart meters, and show that existing IDSs do not satisfy them.
- Design an IDS based on a generic model of a smart meter's operations to identify security critical parts of the code, and selectively monitor the operations in the security critical parts. Our IDS is derived in a semi-automated fashion, based on annotations added by the developer to the smart meter software.
- Implement our IDS for SEGMeter, an open source smart meter from Smart Energy Group [24].
- Evaluate our IDS in terms of its performance, memory overhead, detection latency, and detection capabilities. Our results show that our IDS has a performance overhead of 4% with a detection latency of about 10 seconds, incurs no false positives, and is able to detect a wide range of known and unknown attacks (Section V). In comparison, other host-based IDSes have overhead of higher than 160% for the smart meter.

While our host-based IDS has been implemented for a

specific smart meter (SEGMeter), we base it on a generic specification of smart meter behaviour [22]. Unfortunately, most manufacturers of smart meters do not make their code and specifications publicly available, so we are unable to demonstrate our system on other smart meters.

II. Background and Related work

A. Smart meter architecture

A smart meter is a networked embedded system. It receives data regarding power usage through analog front end sensors. This component receives analog data (for example, electric currents), converts it to digital data and passes it to the microcontroller. The microcontroller unit calculates consumption based on the input data. Smart meters are equipped with flash memory to store both data and event logs on the meter. They also have a real-time clock for the meters to provide time-of-use billing services. Finally, meters are equipped with network interfaces to communicate with the server and other devices.

B. Constraints

Unlike general purpose computers, smart meters are low end embedded computing devices. Therefore, they are constrained in terms of computing power, memory, cost, etc. We list the key constraints that IDSs must satisfy to address the requirements and constraints of smart meters:

1) Performance: Smart meters are low end embedded devices. Memory and computation overhead are therefore significant limiting factors. For example, the smart meter that we use has only 16 MB of RAM available, and runs a 240 Mhz processor.
2) No false positives: False positive happens when an IDS reports an intrusion when no actual intrusion has occurred. Smart meters are deployed in large scale and communicate with the utility server. Even a very small false positive rate will result in rapid aggregation of false alarms which imposes large overhead on the utility provider's server. IDSs typically target the false positive rate of 1% [13]. However, even a false positive rate of 0.5% over a full-day audit trace in an area where 500,000 smart meters are deployed, means handling 2500 daily false positives for the service provider. Therefore, it is important to have no false positives.
3) Software modification: Software running on smart meters must meet specific requirements regarding the platform it is running on. For example, the software must read consumption data from sensors in real-time and maintain a two-way communication with the server. Any IDS solution that modifies the software and changes the real-time behavior of the software is not acceptable for smart meters.
4) Low cost: Since smart meters are deployed on a large scale (on the order of millions), any extra cost for individual meters incurs large financial overhead to the whole system. For instance, adding a 10 dollar memory chip to the meters adds up to 10$ million extra cost when the meters are deployed for one million entities. Hence, cost is an important constraint for smart meters, and indirectly affects their performance.
5) Coverage of known attacks: Smart meters are essentially small computers and many of the attacks developed for desktop and server systems may be mounted on them e.g., buffer overflow attacks. An IDS for smart meters must be able to detect these known attacks.
6) Coverage of unknown attacks: Smart meters are fairly new systems and are currently being deployed around the world. Therefore, the attack vectors are not well-studied. Considering this, any IDS developed for smart meters must have the ability to detect unknown attacks.

C. Related work

In this section, we survey prior work on IDSs and discuss their applicability with regard to the constraints. One approach to design IDS decision engine is based on AI techniques. For instance prior work [12, 27] uses Hidden Markov Models to design the decision engine of IDSs. Support Vector Machines [13] and Neural Networks [19, 25] have also been used to detect anomalous behavior of the system. Statistical techniques have been used to handle large amount of audit data [28]. These techniques are suitable for modeling complex software and detecting unknown attacks. However, the major problem with these techniques is that they incur significant false positives (ranging from 0.5% to 11% for the above techniques) which makes them unsuitable for smart meters.

Another class of techniques uses static analysis to build IDSs [8, 26]. These techniques statically build a model of the software based on the program's source code. The audit data is compared against this model to identify any deviation from the normal behavior of the system. Since the model is built based on static analysis of the program's code, these techniques do not have false positives. This is so because static analysis techniques are conservative, and over approximate the set of program behaviors. However, these techniques incur high overheads, often exceeding the capabilities of the smart meter, as we show below.

Wagner et al. [26] propose a system call-based IDS that uses static analysis to derive the system call sequences. They propose three approaches, namely non-deterministic push down automaton (NDPDA), call-graph and digraph. In the call-graph technique they build a non-deterministic finite automaton (NDFA) of the system call sequences. One problem with this representation is that of impossible paths, which results in high false negative rates i.e., missed detections. In the NDPDA approach, they use full system call traces of the program and they remove impossible paths from the call graph model. However, as we show in Sec. V, using full system calls to build the model imposes unacceptable overhead on smart meters due to memory constraints. The third approach is the digraph technique which models the possible k-consecutive system calls. The running time for precomputing the possible sequences is exponential with respect to k and hence incurs high performance overhead.

Giffin et al. [8] propose the Dyck model to build a context

Technique	C1	C2	C3	C4	C5	C6
Dyck		X			X	X
NDPDA		X		X	X	X
HMM/NN/SVN	X		X	X	X	X
Statistical techniques	X		X	X	X	X

Table I
HOW EXISTING APPROACHES FOR INTRUSION DETECTION MEET OR DO NOT MEET THE SIX CONSTRAINTS FOR SMART METERS

sensitive IDS. They offer higher detection precision through adding context sensitivity to the model and thereby, excluding impossible paths from the model. Impossible paths are non-legitimate sequences of the system calls that are accepted by the context insensitive model of the software. In their technique, they add extra null system calls to the code, before and after system calls to make the model context sensitive. The addition of null calls has two consequences: 1) increasing the run-time overhead due to increased number of executed null calls, and 2) increasing the size of the model as the extra null calls must be included in the model. Increasing the size of the model results in increased memory overhead. This is specially a problem for smart meters for which memory is a constraint. For example, we inserted null calls for the SEGMeter software, our open source smart meter to mimic the operations of the Dyck technique. The process resulted in about 46% increase in the number of generated system calls during the run-time of the software. Later in Sec. V we show that modeling the existing system call traces already imposes an unacceptable overhead on the SEGMeter due to memory constraints. Therefore, the 46% increase in the size of system calls makes this solution (and similar solutions) impractical for smart meters.

Table I shows how existing IDS techniques meet the constraints of smart meters. We see that no system is able to meet all the six constraints, thereby necessitating a new IDS for smart meters.

III. THREAT MODEL

For a general purpose computer, the attacker is typically interested in taking control of the machine for malware distribution purposes without necessarily controlling the software running on the system. However, for smart meters, malicious users directly target the software running on the meter, as this is what brings them economic value.

In this paper, we assume that the goal of the adversary is to change the normal execution path of the software running on the meter to achieve economic benefits, for example paying less money for consumed power. To achieve this goal, the adversary may exploit the potential vulnerabilities in the smart meter and mount code injection attacks that add malicious code to the software and change its behavior to suit their ends. This may be done through buffer overflow attacks, man-in-the-middle attacks, etc. mounted on, for instance, the network interfaces of the meter where command/data is sent/received. The attacker changes the input so that some extra commands are executed, or some some of the commands (calculating/storing consumption data for example) are skipped. We are not considering the cases where the attacker mounts an attack without injecting any code (for instance through only changing the arguments of software system calls). Utility service providers typically have mechanisms in place to detect abnormality in data submitted through the meters [18]. Therefore, to elude detection, the attacker's changes must be small and subtle. This rules out large scale changes such as completely replacing the meter software with the attacker's own software, unless they are able to mimic the operation of the meter's software. We do not consider attacks on the privacy of smart meters and on the meter's availability, i.e., denial of service attacks (DoS).

IV. APPROACH

In this section, we introduce our IDS for smart meters and explain how we address the constraints discussed in Sec. II-B in our design. As in other IDSs [6, 8, 11, 26, 27], we use system calls as the input to the IDS. This allows the IDS to access raw data of the interaction between programs and the operating system.

The IDS is based on a high-level model of the software running on the meter. Based on the model, we identify the events that represent the core behavior of the software, and therefore are important from the security point of view. Our IDS selectively monitors the system calls based on the model, thereby detecting the important attacks, while satisfying the constraints of smart meters in terms of the computation and memory overhead.

Building a model for general purpose computers' operations may be infeasible given their vast state space of operations. However, unlike general purpose computers, smart meters are designed to carry out a *specific* set of operations. As a result, it is possible to model the activities of the smart metering software based on its architecture and use cases. Further, there are standard specifications available for smart meters, for example [5]. We use an *abstract model* derived from such specifications, introduced in our prior work [22]. However, in that work, we used the model to drive security attacks on smart meters, rather than to implement defensive measures for the meters. In this paper, we use the model for deriving the signatures of our IDS.

Based on the abstract model, we build a concrete model of our specific smart meter. The concrete model of the meter captures all the procedures implemented for that specific meter and the control flow between them. To build the concrete model of a specific instance of a meter, we map the abstract model to the implementation of the meter, using programmer defined tags/annotations. Different procedures in the code are mapped to the corresponding blocks in the abstract model and the result represents the detailed activities performed by the specific smart meter.

We traverse the abstract model and for each activity block, identify the class of attacks that could target the activities (such as physical storage access, network communications, etc.). From the set S, which contains all system calls in the software, we create a subset P of system calls that represent the identified activities. Through a procedure that

we explain later, we assign a subset of system calls in P to blocks of the concrete model and generate a state machine of system calls that constitute the signature for the IDS. The IDS monitors the executed system calls, verifies them with the state machine and detects any deviation from the state machine model as an attack.

This model-based approach of selecting system calls and building the state machine enables us to monitor the entire state of the software (represented by the concrete model), while considering only the important system calls that are involved in the potential attacks against the meter. This is crucial to the feasibility of IDS designed for smart meters without compromising on the constraints (see Sec. IV-D).

We discuss each of the above steps in more detail below.

A. Abstract Model

As mentioned before, we use the abstract model proposed by in our prior work [22] to derive the signatures for our IDS. For completeness, we summarize the abstract model here. The abstract model of the smart meter is presented in Figure 1. Below, we briefly discuss the blocks of the abstract model presented in Figure 1. Operations 1 through 6 represent control procedures and operations 7 through 13 represent communication procedures. The arrows between the blocks represents the control-flow of the program.

Operation 1 through 6 are in charge of consumption calculation. This includes initialization of the sensors, processing the commands that are sent from the server (for example remote disconnect to cut off the power), reading consumption data from the sensors, calculating consumption information, and passing this information to the processes in charge of communicating with the server.

Operations 7 through 13 are in charge of handling communication activities. These activities include: checking network connectivity, receiving commands from the server, storing consumption information on physical storage, retrieving data from physical storage and sending them to the server, via the network interface.

B. Concrete model

We specialize the abstract model for specific instances of the meter software. This is called the *concrete model* [22]. To build the concrete model of the meter, we map each block in the abstract model to the corresponding functions in the meter software that implement the functionality of that block, in an automated fashion as we show below. Figure 2 shows a portion of the concrete model for blocks 6 and 7 of the abstract model (Figure 1). This concrete model is for the SEGMeter, our open source smart meter from Smart Energy Groups [24]. In block 6, *powerOutputHandler* saves consumption information such as power, energy, and energy channel, into a buffer. *sendMessage* passes these information through serial communication to *ser2net*, which is a specialized process to pass data between procedures of SEGMeter. In block 7, *getData* receives information from *ser2net*, *validateMessage* verifies formatting of the message, and *readMessage* processes the content of the message.

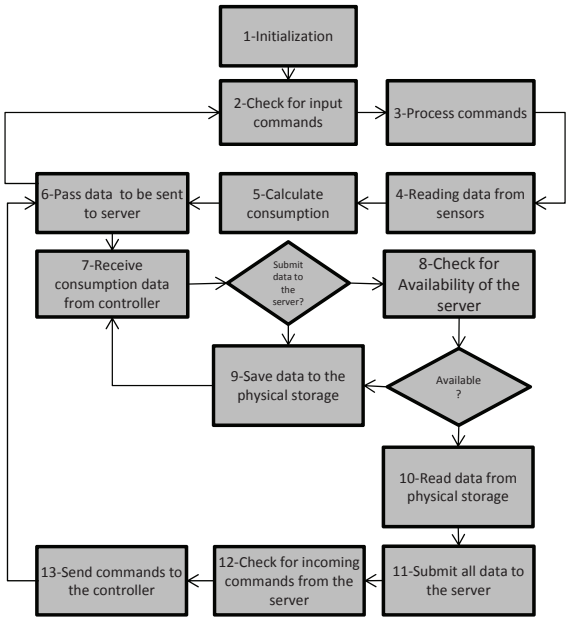

Figure 1. Abstract model for the smart meter

Thus, the concrete model may have multiple procedures for each block of the abstract model.

To automate the process of creating a concrete model of a specific smart meter, we need semantic information provided by the developer. The developer of the smart meter software needs to add tags to the procedures in the code as per their functionality. We use these tags to automatically generate the concrete model. We design our tagging system so that it satisfies two main goals:

1) Ease of use: The process must be straightforward for the developer so that it is conveniently integrated with the design process.
2) Flexibility: The process of concrete model generation must be flexible to be applied to different implementations of the smart meter.

To achieve the above goals, we design a hierarchical tagging system, that lets the developer express the intended functionality of a procedure in terms of a n-tuple of common operations performed by the meter. Because smart meters typically perform a small set of operations, it is possible to assign unique labels for each of these activities, and map the procedures to the activities.The activities are represented by the tags. By using the tags, the developer does not need to directly map the procedures to the blocks of the abstract model. The hierarchical structure helps the developer do the process step by step and to refine the state space iteratively. This system also provides the flexibility to generate the concrete model for a wide range of different implementations, as one procedure may have multiple tags associated with it.

The code developer adds the tags as comments at the beginning of each procedure. Our custom model generator parses the code and based on the assigned tags, places the procedures in appropriate blocks. Further, the model

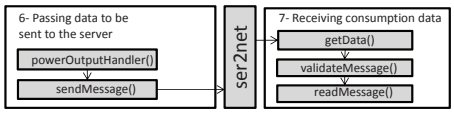

Figure 2. Concrete model for portion of the abstract model

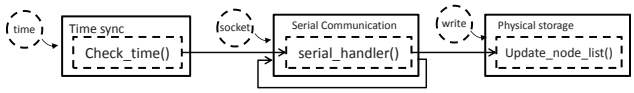

Figure 3. The state machine based on the concrete model: here the sequence *time socket* write* represents the flow of this small section of the concrete model.

generator creates the call graph of the procedures based on which the control flow between the blocks of the concrete model is determined. Due to space constraints, we do not present the format of the tags or the grammar we defined for the annotations.

C. System call selection and State Machine Generation

The abstract model and concrete model of smart meter together identify the system calls that are likely to be involved in attacks, and hence must be monitored. To identify the key system calls that are likely to be involved in security attacks, we first traverse the abstract model step by step and according to the functionality of each block, find if there are attacks that target those functionalities. This process is done manually, but using a well defined algorithm as we explain below (this process can be automated in future work).

Our goal is to find *P*, the set of system calls to be monitored. We use the classification proposed by Hansman et al. [10] to find the attacks that correspond to each block. Examples of attacks classes include spoofing, sniffing, etc. We assume *S* to be the set of all system calls generated by the smart meter software. For each functionality targeted by an attack in the classificiation, we add the system calls in *S* that represent those functionalities to the set *P*. For instance, there is a message passing step (for passing consumption data) between blocks 6 and 7 in the abstract model (Figure 1). The message passing procedure is vulnerable to a man-in-the-middle attack and data spoofing. The system calls representing message passing include *connect*, *send*, and *recv*. Therefore, we add these system calls to the set *P*.

Once we find the priority set *P*, we need to construct the state machine for the system calls in *P*. We follow the procedure below.

1) For each block of the concrete model, identify all the system calls in *P* that are associated with that block.
2) From the system calls associated with each block, select the ones relating to the specific functionalities of that block (for instance, *connect* and *socket* system calls for server communication blocks, *read* and *write* system calls for storage blocks, etc.)
3) Until all the blocks of concrete model are covered do
   a) Pick an unmarked system call *s* that appears in the maximum number of the blocks of the concrete model. This results in reducing the size of *P* while providing high coverage across all blocks of the concrete model.
   b) Mark system call *s*
4) Output all the marked system calls

After assigning the system calls to the blocks of the concrete model, we build a state machine that represents the concrete model of the smart meter.

Figure 3 shows a small section of the concrete model where the consumption data is received from the sensors through serial communication and stored in the physical storage. Three procedures of the code, namely *checkTime()*, *serialHandler()*, and *updateNodeList()* are represented by three system calls: *time*, *socket*, and *write*. The procedures are executed one after another and *serialHandler()* executes in a loop. Therefore, the regular expression corresponding to these blocks will be *time socket* write*. After identifying the system calls corresponding to the blocks of the concrete model using the procedure above, we build the expression through backward traversal of the concrete model and extraction of the patterns that correspond to the paths.

D. Discussion

In this section, we discuss how our IDS addresses the constraints outlined in Sec. II-B.

1) Performance: To build an efficient IDS, we have to be able to select a subset of system calls that are representative of the behavior of the software. Our technique enables us to select limited system calls that efficiently cover the important components of the concrete model regarding the potential attacks against the smart meter. In section V, we show the overhead benefits of having a compact model for smart meters that are memory-constrained devices.
2) No false positives: The concrete model of the smart meter is built based on the static analysis of the code for a specific model of the meter. Therefore, similar to other static analysis approaches [8, 26], our technique has no false positives.
3) Software modification: The only modification we make to the code is the addition of tags to the procedures to automatically generate the concrete model of the software. It is important to note that the tags are treated as comments in the code, and are read only by our model generator. Hence, they do not affect the execution of the code or its performance.
4) Low cost: Our technique is designed to build a model of software running on the smart meter that covers all the key components of the software but at the same time has a small size, and therefore, fits in the meter's memory. This means that it can be deployed on existing smart meters, without any new components.
5) Coverage of known attacks: Discussed in section V.
6) Coverage of unknown attacks: Discussed in section V.

Generalizability: The abstract model represents the main functionalities of the meters that are common among different instances of meters. These functionalities are defined in standard documents released by government departments in charge of deploying smart meters [1, 2]. As smart meters in each region must comply with these standards, we believe that the procedure of applying the abstract model to smart meters and designing the IDS, can be generalized to meters from different vendors. However, we have not been able to veriffy this generalizability of our technique as we do not have access to the code of other smart meters.

V. Evaluation

A. Experimental setup

We implement our solution for the SEGMeter, an open source smart meter from smart energy groups [24] (Fig. 4). SEGMeter consists of two main boards: 1) an Arduino board [3] with ATMEGA32x series microcontroller which is connected to a set of sensors, receives sensor data, and calculates consumption information and, 2) a gateway board with Broadcom BCM3302 V2.9 240MHz CPU and 16 MB RAM. The gateway board has LAN and Wifi network interfaces, and communicates with the utility server. It runs OpenWrt, which is a Linux distribution for embedded devices. OpenWrt allows the system developer to customize the device according to the limited resources.

The meter software consists of about four thousand lines of code and it is written in C++ and the Lua language. The C++ code resides on the Arduino board and the Lua code resides on the gateway board. The boards communicate with each other through a serial interface.

Our IDS runs on the gateway board and has two major components. The first components starts when the smart meter boots up and attaches *strace* to the process we are monitoring. *strace* intercepts every system call made by the process and logs it to a file. The second component of the IDS runs every T seconds, reads the log file, compares it with the model, clears the log, and raises an alarm if a mismatch occurs. We have set $T = 10seconds$ in our experiments. For SEGMeter, our IDS is built based on 29% of total system calls. Examples of the system calls selected in our IDS include: open, write, recv, send, connect, read, close, ioctl. Other system calls such as utime, fstat, chdir, _newselect, pause, time, gettimeofday, etc. were not selected through our process explained in Sec. IV. We manually added the tags required to generate the concrete model of the meter (Sec. IV). The entire process took about 6 hours.

In our experiments, we first measure the performance overhead of our IDS under memory constraints that are typical for smart meters. We compare the performance of our IDS to existing techniques that use the full system call trace of the software to monitor its operation. We call this IDS the *full trace IDS*. Then we evaluate the detection performance of our technique for both unknown and known attacks. Finally, we discuss the effect of detection latency, T, on the performance of our IDS.

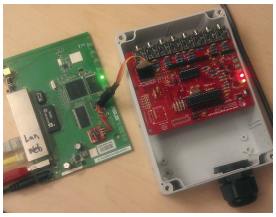

Figure 4. SEGMeter: our open source meter testbed. The board on the left is the gateway board in charge of communicating with the server, and the board on the right is the Arduino board that receives consumption data from the sensors.

B. Results

Performance overhead: As stated earlier, an IDS developed for smart meters should be able to operate under severe memory constraints, and still achieve reasonable overheads. The SEGMeter has a total of 12MB RAM available. To study the effect of memory constraints, we added dummy processes to take up memory with no processing overhead. We run our model-based IDS and the full-trace IDS under these constraints.

We measure the performance overhead as the ratio of the time taken to read the system call log, analyze it, and compare it against the model, to the time taken by the smart meter software to produce the trace. Note that we do not perturb the software in anyway except to run *strace* on it, which has minimal overhead (less than 1%). Table II shows the results.

As can be observed from the table, the performance overhead for the full-trace IDS is considerably higher than that incurred by our IDS. Even when the entire memory of the SEGMeter is available to the IDS, the overhead for the full-trace IDS is over 100%, while our IDS has only 4% overhead. This means that for 10 second logs, the full-trace IDS takes more than 10 seconds to perform detection and hence, falls behind the software. Further, increasing the the time between analyzing the logs does not help with the overhead of full-trace IDS as the size of the log will also increase (we have verified this for 10, 20, 30 second logs).

The main reason for poor performance of the full-trace IDS is that it significantly increases the size of the model compared to our IDS, and hence causes thrashing i.e., page faults due to the working set size not fitting in memory. This can be confirmed by the fact that the overhead of the full-trace IDS increases as the memory available decreases from 12 MB to 6 MB. Considering the long lifetime of smart meters (over 15 years), this overhead can be a major problem, as memory requirements may change substantially in this time frame. On the other hand, the performance overhead of our IDS remains constant as the memory capacity is decreased, even down to 6 MB.

Unknown attacks: As stated in the threat model (Sec. III), the goal of the attacker is to change the behavior of the software in a way that is stealthy and not easily detectable. To do this, the attacker needs to intercept the program when it is running and inject code to change the

	12 MB	9 MB	6 MB
Full-trace IDS (10s)	165.2%	214.6%	315.1%
Model-based IDS (10s)	4.0%	4.0%	4.0%

Table II
RUNNING TIME OVERHEAD OF OUR MODEL-BASED TECHNIQUE AND COMPLETE-SYSTEM-CALLS TECHNIQUE.

Component	Random (%)	Popular system calls	Full trace (%)	Model-based (%)		
				Minimum	Average	Maximum
Server communication	32	36	92	59	62	63
Storage & retrieval	14	44	84	73	74	78
Serial communication	42	28	88	67	72	74
Total	29.3	36.0	88.0	67.4	69.6	71.7

Table III
CODE INJECTION DETECTION COVERAGE OF OUR TECHNIQUE VERSUS OTHER TECHNIQUES FOR EACH SUB-SYSTEM OF THE METER

flow of execution (e.g., through a buffer overflow attack). Therefore, to mimic unknown attacks, we developed a code injection procedure that works as follows.

For each injection, a random procedure of the software containing system calls is selected and a few (1 to 8) lines of code are copied and pasted in the same procedure. Only one injection is performed per run to increase the stealthiness of the attack. While the attacks generated may not achieve the attacker's desired goals, they serve as a means to evaluate the detection capability of our model against stealthy attacks that change the execution of the code. The attacks generated using the above technique are difficult to detect because (1) we do not introduce new system calls that would trigger IDSs that look for system calls outside the allowed set, and (2) the copy-pasted code belongs to the original software and would hence be missed by IDSs that do not consider the system call ordering in their signatures.

To perform the injections, we divided the components of the software into three categories: 1) server communication: the functions that communicate with the utility server, 2) storage and retrieval: the functions that process consumption data, back it up, read it from flash memory, etc., and 3) serial communication: the functions that receive and process consumption data through serial communication interface from procedures in charge of consumption calculation. Using the above procedure, we created 50 injections in each category, adding up to a total of 150 injections.

In addition to the full-trace IDS introduced earlier, we compare our IDS with two other IDSs, namely (1) random, and (2) most-popular. In the random IDS, we picked the system calls to monitor randomly from the set of all the system calls in the software. In the most-popular IDS, we picked the system calls that have the highest frequency in the code. The expectation is that the higher the frequency of a system call, the better coverage it potentially provides. In both cases, we chose the same number of system calls to monitor as our model-based IDS, for fairness.

The detection results are shown in Table III. The random IDS provides an average coverage of 29.3%. and the most-popular IDS provides an average coverage of 36%. Our IDS provides an average coverage of 69.6%, while the full-trace IDS provides an average coverage of 88%. The reason that the full-trace IDS does not provide 100% coverage is that many of the procedures contain loops, and mutating the lines within a loop does not change the system call sequences.

When building the model-based IDS according to the steps provided in Sec. IV-B, it is possible that we have more than one option for assigning system calls to blocks of the concrete model to build the IDS. To study the effect of these options, we study the maximum and minimum coverage provided by the different sets of system calls chosen by our IDS. Note that the maximum/average/minimum results are calculated for each component by considering all possible combinations of the system calls chosen by our IDS for that component. As the table shows, the difference between the maximum and minimum values of total coverage is 4.3%. This shows that the effect of different combinations on the coverage of the IDS is negligible, as long as it is based on the procedures provided in Sec. IV-B.

Known attacks: To evaluate our IDS against real attacks on smart meters, we implemented two attacks introduced for smart meters in our prior work [22]. We also implemented two more attacks in this paper for the SEGMeter. To our knowledge, there is no publicly available dataset of attacks for smart meters at the system call granularity that we can use for our evaluation.

The goal of attacks in [22] is to help the attacker gain financial benefits from tampering with the meter software through changing the consumption data, etc. Further, we verified that none of these four attacks are detected by server side defences or other techniques developed on the meter. We briefly explain the attacks below:

1. *Communication interface attack*: The goal of this attack is to modify consumption data sent to the server. ser2net is a component of SEGMeter that receives consumption data from the Arduino board and passes it to the gateway board (Figure 2). We injected code to change the information flow from ser2net and modify the data before it is sent [22].

2. *Physical memory attack*: The SEGeter writes data to flash memory whenever the network is not available. We injected code to temporarily disable the network so that data is written to flash memory. This provides the opportunity for the attacker to modify data on the flash memory so when the data is retrieved, false data is sent to the server [22].

3. *Buffer full attack*: We changed the control flow of the communication between the gateway board and controller board to send fake 128 byte messages with a frequency of 30 times per second. This causes the receiver buffer on the controller board to always be full. Therefore, any legitimate command sent from the server will not be processed, including disconnection requests.

4. *Data omission attack*: We inject code that connects to the port 2000, which is the port through which the gateway board and the Arduino board communicate and exchange consumption data. The timeout for the communication is 10 seconds. Our attack code periodically (after the 10 second timeout) connects to port 2000 and reads available consump-

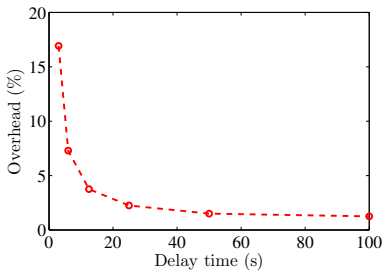

Figure 5. CPU overhead for different monitoring time values (T).

tion data. This data will be deleted by the controller board after it is read. Therefore, when the legitimate connection is established again, this data is not available to be sent to the server, and hence the server records lower consumption data than the legitimate consumption.

Our IDS was able to successfully detect all four attacks above. As explained in Sec. IV, we follow a comprehensive attack classification to identify security critical system calls for blocks of our model. All of the attacks we considered use one or more system calls that we identify as security critical in Sec. IV. Hence, our IDS successfully detects all the attacks. This reiterates the value of using model-based techniques to choose the system calls to monitor in the IDS.

Detection latency: As mentioned before, our IDS reads the generated system calls every T seconds and compares it with the model. Therefore, the value of T determines the maximum delay for detecting attacks. However, a smaller T results in higher performance overhead, as it is more intrusive.

Fig. 5 shows the variation of the performance overhead incurred by our IDS versus parameter T. We observe that at $T = 10s$, the performance overhead drops to 4.15% and we choose this value as the timeout T. We believe this delay is a reasonable trade-off between latency and overhead. Note that even with a delay of 1 second, our performance overhead is only 16%, which is 10x less than the overhead of the full-trace IDS (at 160%). This is the average performance overhead among different model-based IDSs that can be built. But the performance does not change significantly between different model-based IDSs (less than 2%) as the size of the model is the same for all of them.

VI. Conclusion

In this paper, we identified the constraints posed by smart meters for intrusion detection systems, and designed a model-based IDS that satisfies these constraints. We showed that our IDS incurs low performance overhead on our smart meter while providing reasonable detection coverage for both known and unknown attacks.

Future work will consist of improving the accuracy of our IDS for unknown attacks and extending our IDS for smart meters other than SEGMeter. We will also investigate techniques to automate the extraction of system call sequences from the concrete model.

Acknowledgment: This research was supported in part by the NSERC Strategic Networks Grants programme for Developing next generation Intelligent Vehicular Networks and Applications (DIVA) and Nokia Canada.

References

[1] 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.
[2] Uk department of energy, smart meter design document. http://www.decc.gov.uk/assets/decc/Consultations/smart-meter-imp-prospectus/1478-design-requirements.pdf.
[3] Arduino home page. http://www.arduino.cc.
[4] Robin Berthier and William H. Sanders. Specification-based intrusion detection for advanced metering infrastructures. *PRDC, IEEE*, 0, 2011.
[5] Department of Energy and Climate Change and the Office of Gas and Electricity Markets. Smart metering response to prospectus consultation, March 2011. http://www.ofgem.gov.uk/Pages/MoreInformation.aspx?docid=56&refer=e-serve/sm/Documentation.
[6] Eleazar Eskin, Salvatore J. Stolfo, and Wenke Lee. Modeling system calls for intrusion detection with dynamic window sizes. *DARPA Information Survivability Conference and Exposition,*, 1:0165, 2001.
[7] 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/.
[8] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Efficient context-sensitive intrusion detection. In *Proceedings of the 11th NDSS Symposium*, 2004.
[9] Electricity grid in u.s. penetrated by spies, 2009. http://online.wsj.com/article/SB123914805204099085.html.
[10] Simon Hansman and Ray Hunt. A taxonomy of network and computer attacks. *Computers & Security*, 24(1):31 – 43, 2005.
[11] Xuan Dau Hoang, Jiankun Hu, and Peter Bertok. A multi-layer model for anomaly intrusion detection using program sequences of system calls. In *Proc. 11th IEEE Intl. Conference on Networks*, pages 531–536, 2003.
[12] Jiankun Hu, Xinghuo Yu, D. Qiu, and Hsiao-Hwa Chen. A simple and efficient hidden markov model scheme for host- based anomaly intrusion detection. *Netwrk. Mag. of Global Internetwkg.*, 23(1):42–47, January 2009.
[13] Wenjie Hu. Robust support vector machines for anomaly detection. In *In Proc. ICMLA03*, 2003.
[14] In-stat and ndp group company. http://www.instat.com/press.asp?ID=3352&sku=IN1104731WH.
[15] Himanshu Khurana, Mark Hadley, Ning Lu, and Deborah A. Frincke. Smart-grid security issues. *IEEE Security & Privacy*, pages 81–85, 2010.
[16] 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.
[17] N. Lewson. Smart meter crypto flaw worse than thought, 2010. http://rdist.root.org/2010/01/11/smart-meter-crypto-flaw-worse-than-thought.
[18] Yao Liu, Peng Ning, and Michael K. Reiter. False data injection attacks against state estimation in electric power grids. In *Proceedings of CCS'09*, pages 21–32, New York, NY, USA, 2009. ACM.
[19] Abdel-Azim M, Abdel-Fatah A.I, and Awad M. Performance analysis of artificial neural network intrusion detection systems. In *In Proc. ICEE.*, 2009.
[20] P. McDaniel and S. McLaughlin. Security and privacy challenges in the smart grid. *IEEE S&P*, 2009.
[21] 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, New York, NY, USA, 2010. ACM.
[22] Farid Molazem and Karthik Pattabiraman. A model for security analysis of smart meters. In *WRAITS, Dependable Systems and Networks Workshops (DSN-W)*, 2012.
[23] Miguel Correia Paulo Veríssimo, Nuno Ferreira Neves. Crutial: The blueprint of a reference critical information infrastructure architecture. In *Proceedings of ISC06*, August 2006.
[24] Smart energy groups home page. http://smartenergygroups.com.
[25] Ahmed U and Masood A. Host based intrusion detection using rbf neural networks. In *In Proc. International Conference on Emerging Technologies.*, 2009.
[26] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings S&P'01*, USA, 2001. IEEE Computer Society.
[27] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *In IEEE S&P'99*. IEEE Computer Society, 1999.
[28] Nong Ye, Syed Masum Emran, Qiang Chen, and Sean Vilbert. Multivariate statistical analysis of audit trails for host-based intrusion detection. *IEEE Trans. Comput.*, 51(7):810–820, July 2002.
[29] K. Zetter. Security pros question deployment of smart meters. *Threat Level: Privacy, Crime and Security Online*, March 2010.