Design-Level and Code-Level Security Analysis of IoT Devices

FARID MOLAZEM TABRIZI and KARTHIK PATTABIRAMAN, University of British Columbia

The Internet of Things (IoT) is playing an important role in different aspects of our lives. Smart grids, smart cars, and medical devices all incorporate IoT devices as key components. The ubiquity and criticality of these devices make them an attractive target for attackers. Therefore, we need techniques to analyze their security so that we can address their potential vulnerabilities. IoT devices, unlike remote servers, are user-facing and, therefore, an attacker may interact with them more extensively, e.g., via physical access. Existing techniques for analyzing security of IoT devices either rely on a pre-defined set of attacks and, therefore, have limited effect or do not consider the specific capabilities the attackers have against IoT devices.

Security analysis techniques may operate at the design-level, leveraging abstraction to avoid state-space explosion, or at the code-level for ensuring accuracy. In this article, we introduce two techniques, one at the design-level, and the other at the code-level, to analyze security of IoT devices, and compare their effectiveness. The former technique uses model checking, while the latter uses symbolic execution, to find attacks based on the attacker's capabilities. We evaluate our techniques on an open source smart meter. We find that our code-level analysis technique is able to find three times more attacks and complete the analysis in half the time, compared to the design-level analysis technique, with no false positives.

CCS Concepts: • **Security and privacy → Formal methods and theory of security**; *Formal security models*;

Additional Key Words and Phrases: IoT, security analysis, model checking

ACM Reference format:
Farid Molazem Tabrizi and Karthik Pattabiraman. 2019. Design-Level and Code-Level Security Analysis of IoT Devices. *ACM Trans. Embed. Comput. Syst.* 18, 3, Article 20 (May 2019), 25 pages.
https://doi.org/10.1145/3310353

1 INTRODUCTION

Internet of Things (IoT) devices are networked, embedded computing devices that carry out specific operations. Smart meters in smart grids, modern car controllers, and implantable medical devices are examples of popular IoT devices that perform security-critical operations. The popularity and criticality of many of these devices make them a target for attackers, as many papers have demonstrated [5–7, 17, 25, 34, 36, 60]. However, most of these attacks were discovered in an ad-hoc or opportunistic manner, and may not be comprehensive. Developing a systematic

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.

Authors' addresses: F. M. Tabrizi and K. Pattabiraman, University of British Columbia, 2332 Main Mall, Vancouver, BC V6T 1Z4, Canada; emails: {faridm, karthikp}@ece.ubc.ca.

1539-9087/2019/05-ART20 $15.00
https://doi.org/10.1145/3310353

mechanism to analyze the security of IoT devices will help developers of IoT systems[1] discover the attacks, improve the design or implementation of the system, and find efficient ways to build security mechanisms to detect the attacks. This is the focus of this article.

Analysis of attacks against software systems may be performed via design-level techniques that leverage models of the system or code-level techniques that directly analyze the code. Examples of design-level techniques are attack trees [40, 55], attack patterns [26, 27], and attack graphs [18, 30, 53, 56]. 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. Examples of code-level techniques are static analysis and fuzzing [45]. Static analysis attempts to find vulnerability patterns in the code that may lead to vulnerabilities. Fuzzing searches for input data that may crash the system and, therefore, indicate a flaw in the implementation that can be exploited by attackers. design-level techniques leverage abstraction to avoid state-space explosion during analysis and, therefore, may have higher false positives or false negatives. On the other hand, direct code analysis may provide higher accuracy as they map directly to the code. However, such techniques may face state space explosion, especially with large code bases that exhibit complex behaviors.

Many classes of IoT devices, unlike general purpose machines, are designed to perform specific tasks and, as a result, their behavior follows a certain model regardless of implementation. Further, they often have significant resource constraints; hence, their software is generally smaller than software designed for general purpose computers. Based on these observations, we believe that IoT devices provide opportunities for *both* design-level and code-level security analysis. In this article, we propose a design-level and a code-level analysis technique for analyzing the security of IoT devices, and we compare the results. This provides insights regarding the applicability and effectiveness of each of these techniques in the IoT domain.

Developing security analysis techniques for IoT devices is challenging since, unlike remote servers, many IoT devices are installed at locations that are accessible by potential attackers. This provides extended accessibility for attackers, e.g., physically rebooting the device and accessing its internal/external communication interfaces. The attackers may use these extended accessibilities to interfere with the execution of the software running on the device. We call the attacks resulting from these accesses *software-interference* attacks. Unlike generic attacks, software-interference attacks are highly specific to each IoT device and, hence, need targeted analysis techniques to be found. Generic approaches to find security attacks [12, 45] either incur high false-positives, or are incapable of finding unknown attacks against the system when used for finding software-interference attacks. Finding unknown attacks is important for IoT devices as they are often deployed in critical scenarios, and it is difficult to patch them regularly. Therefore, IoT devices should be resilient to attacks that may arise in the field.

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 and businesses to calculate electricity consumption and to communicate with the utility server. It is estimated that the worldwide revenue of smart grids will soon exceed $12 billion [1]. The large-scale deployment of smart meters and the criticality of their operations make smart meters and their security an important concern, and there has been significant work on addressing this problem [32, 41, 61].

The main insight to analyze security of smart meters and finding software-interference attacks is that we perform security analysis against a set of the *attacker's actions* rather than pre-defined attacks. We define a set of actions that represent the attacker's capabilities. These are the actions that the attacker can take, e.g., dropping messages that are communicated via network interfaces.

[1]In this article, by an IoT system, we mean the software running on a networked, special-purpose, embedded device.

Note that these actions are not standalone attacks but rather the capabilities of the attacker that are building blocks of attacks. We define the attacker's actions in a way that they can be dynamically activated/deactivated at runtime. This allows us to consider all the scenarios in which an attacker may affect the system, during runtime, and tamper with the system. Therefore, we do not require the knowledge of existing attacks to perform the analysis.

We leverage the above insight to address the challenge of finding software interference attacks against IoT devices. We propose two techniques, one at the design-level and the other at the code-level. In our design-level approach, we build a formal model of smart meters, capturing their main functions. We also formalize the set of attacker's actions. Finally, 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 energy consumption data is less than zero is unsafe as it may result in incorrect billing.

In our code-level analysis technique, we define attacker's actions as code snippets and inject them into the source code of the system to transform it. We define the code snippets of the attacker's actions in a way that they can be dynamically activated/deactivated at runtime. This allows us to consider all the scenarios in which an attacker may affect the system during runtime. After transforming the code, we can leverage existing software analysis techniques, namely symbolic execution, to find executions paths that lead to a software-interference attack. In other words, we *symbolically* execute the transformed code. We search for *any* execution scenario in which the *activation* of attacker actions may result in violating security invariants. Any such solution represents a software-interference attack on the system.

In this article, we make the following contributions:

- We build a formal model of a smart meter in rewriting logic [38] using the abstract model of a smart meter presented in Molazem et al. [46], which represents the generic operations of a smart meter. We also develop a formal model of the attacker's actions for a generic smart meter, also in rewriting logic. The attacker may use a sequence of these actions to mount sophisticated software-interference 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 software-interference attacks found by our technique.
- We model attacker actions as code snippets and develop a framework to inject attacker actions directly in the code and via symbolic execution, finding all the ways an attacker may affect the code at runtime. This allows us to automatically find concrete software-interference attacks on the system given a set of attacker abilities.
- Using off-the-shelf, inexpensive equipment, we experimentally validate the software-interference attacks found on an open source smart meter: Smart Energy Groups Meter (SEGMeter) [2]. We find that the software-interference attacks found by our two techniques cause the meter to lose data and get stuck in an infinite loop, thereby allowing attackers to lower the meter's energy consumption fraudulently.
- Comparing the two techniques, the code-level analysis technique found over nine different types of software-interference attacks and over 50 ways to mount the attacks on the system. This was three times more than the attacks found by our design-level technique. Also, our code-level analysis technique incurs no false positives, while the design-level technique incurs an average of 50% false positives. Finally, the total analysis time of the code-level analysis technique is less than 1 hour, while the time for the design-level technique is close to 2 hours, on a regular desktop computer.

In this article, we focus on a single smart meter, SEGMeter, representing an IoT device, as we did not have access to the source code of any other IoT devices. However, we have designed our technique in a generalizable fashion; as such, we believe it can be applied to a wide range of IoT devices—the precise criteria for the applicability of our technique are outlined in Section 8.

2 RELATED WORK

Below, we discuss techniques for performing automated security analysis and their limitations.

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. [59] studied a large dataset of network attacks to find the common properties of some of the attacks. They developed a clustering tool and applied them on different feature vectors characterizing the attacks. Gegick et al. [27] encoded attacks in the attack database and used them in the design phase to identify potential vulnerabilities in the design components. Fernandez et al. [26] studied the steps taken to perform a set of attacks and abstracted the steps into attack patterns. They studied Denial of Service (DoS) attacks on Voice over IP (VoIP) networks and showed that their patterns can improve the security of the system at design time, and helped 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. Further, IoT systems are difficult to patch in the field and, hence, need to be resilient to even unknown attacks.

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. [18] developed attack trees for power system control networks. They evaluated the vulnerability of the system and provided counter measures for improvements. McLaughlin et al. [43] used attack trees for penetration testing of smart meters. Morais et al. [48] used attack tree models to describe known attacks and, based on the trees, developed fault injectors to test the attacks against the system. They tested 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. We use attacker actions to search through the set of all interactions an attacker may have with the system in order to tamper with it. Therefore, our technique may compliment the attack tree approach in finding viable attacks against 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. [56] and Jha et al. [30] proposed techniques for automatically generating and analyzing attack graphs for networks. They assumed that the vulnerability information for each node is available. Based on this information, they analyzed 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 [28]. For example, Matousek et al. formally verified security constraints on networks with

dynamic routing protocols [39]. Delaune et al. analyzed the security of PKCS#11, an API for cryptographic device [24]. Miculan et al. formally analyzed the security of Signle-Sign-On (SSO) authentication protocols for Facebook [44]. 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.

Fuzzing: Fuzzing is a sub-category of fault injection. However, it is widely used in industry for security evaluation; hence, we discuss it separately here. Fuzzing involves inserting random inputs to a program and evaluating their effects [45]. This process facilitates automated penetration testing. Some tools built based on fuzzing techniques include HP WebInspect [8], the IBM AppScan [9], and Acunetix web application security scanner [3]. Neves et al. presented a tool called AJECT for fuzzing the input to the servers, based on predefined attack patterns [49]. Fuzzing may uncover many of the existing bugs in the system. However, the effects of fuzzing may not necessarily be security-related. Therefore, these tools report a high percentage of false positives [50]. Also, embedded devices are exposed to a wide range of accesses (e.g., physical access) and attacker actions (e.g., physically rebooting the device and voltage manipulation) that may not easily be simulated by fuzzing techniques. Therefore, even though fuzzing is an effective technique for finding unknown attacks against software systems such as web applications, it is not enough for embedded systems with extended attack surfaces. Hence, our technique may complement fuzzing for such systems.

Program analysis: In recent work, Ivan et al. [51] found sensor-spoofing attacks against embedded systems via program analysis. They used symbolic execution to find sensor readings that lead to unsafe states in the program. Then, as an attacker, they generated inputs that produce those sensor readings. However, their work is limited to sensor spoofing and does not allow for finding attacks given a general model of the attacker (for instance, when an attacker can tamper with network communications, reboot system, etc.). Davidson et al. [22] proposed a symbolic execution engine based on KLEE [19] and specialized for the MSP430 family of 16-bit microcontrollers. They showed that this engine may be effectively used to analyze security of embedded systems for which a symbolic execution engine was not available. In this article, we used existing symbolic execution engines to enable our analysis technique and therefore, tools such as KLEE [19] can improve the effectiveness of our work and extend its applicability to a wider variety of platforms.

Power grid security: Prior work on smart grid security proposed techniques for modeling risks and threats associated with power systems, as well as detection of certain categories of attacks at runtime. Ray et al. [54] proposed different approaches for security risk management of smart power grids and discussed different threat and vulnerability modeling schemes. Sridhar et. al. [57] proposed an approach to assess security risks of cyber physical systems used in smart power grids by examining the dependencies between the cyber physical equipment and the power infrastructure. Rahman et al. [52] demonstrated the vulnerability of these systems to false data injection attacks. Lie et al. [37] modeedl false data injection in the power grid as a matrix separation problem and proposed two methods to address it. While these techniques are useful, they do not address the problem of discovering new attacks on the system, which is our focus. Moreover, our approach focuses on finding vulnerabilities *before* the system is deployed, while many of the above papers focus on runtime detection and mitigation of security attacks. Security analysis at development time helps harden the system and reduce the costs associated with runtime attack-mitigation.

Summary: IoT systems are being largely deployed in critical domains such as smart grids, homes, and modern cars. Therefore, their security is important. Existing techniques for analyzing attacks are useful. However, they have limitations when applied to IoT. Techniques such as fuzzing are general purpose and may be applied to IoT to find new attacks against them. However,

due to their generality, they fail to consider attacks that are specific to IoT devices due to special accesses the adversaries may have to the devices (e.g., physical access). Due to the criticality of many of IoT domains, creating security analysis techniques that are *tailored* for them, is especially important as such techniques are likely to have higher coverage than generic techniques.

We attempt to address this problem in this article and *complement* existing techniques when they are applied to IoT. We do this by incorporating *specific* accesses the attacker has to the device. We introduce an automated way to consider the *actions* an attacker may take against the system and violate its security properties. This allows for developing a security analysis that is *tailored* for a specific IoT device and its specific attacker model. However, our technique may be be specific to IoT and, hence, not as widely applicable as some of the existing techniques such as fuzzing.

3 BACKGROUND

3.1 Smart Meter

A smart meter is a networked device that measures electricity consumption and communicates with the utility 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, and so on, to form a Home Area Network (HAN). In each area, smart meters will be connected to a collector through a field area network (FAN). This collector gathers all data and communicates with the utility server through the 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

Attacker: In this article, we assume that the attacker may have read/write access to the communication interfaces of the smart meter. Therefore, the attacker may intercept data sent from the system and send (incorrect) data to the system. We also assume that the attacker may have physical access to the system and perform actions such as rebooting the system (e.g., powering it on and off). These are realistic assumptions as smart meters are installed in insecure 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 [58] allow communication via the serial interface and optical probe, and sending/replaying 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 [42]. Based on this, we assume an attacker may (1) drop messages communicated to the device's interfaces, (2) replay messages to the device's interfaces, and (3) reboot the device at any point in time. We consider reboot action to be a hard-reset, which completely clears the state of the system—this may not be possible in all systems.

Software Interference Attacks: The attacker stated above is designed based on the characteristics of IoT devices, which allow the users to physically interact with them and access their

communication interfaces. This is unlike attacks against remote servers where physical access is very limited. Therefore, we define a new term for these attacks. We call the attacks resulting from such interactions (e.g., the ones in the attack model above), which lead to corrupting data [20] of the software and/or violating its control flow [14], as software-interference attacks. We consider this category of attacks in this article. To the best of our knowledge, there is no existing term for these attacks, which is why we coined a new term.

We do not consider social engineering attacks, attacks on confidentiality, and DoS attacks. The reason is that these attacks are generally addressed by other techniques such as analysis of network infrastructure (DoS), analysis of cryptographic methods (confidentiality), and Human-Computer Interaction analysis/social studies (social engineering). Also, we do not consider that other meters monitor each other in the neighborhood-area network, as it is not a feature in the platforms we have been considering.

4 OVERVIEW

In this section, we provide an overview of the high-level steps of the two analysis approaches proposed in this article.

Design-level analysis: We follow a three-step process for design-level security analysis of smart meters. In 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, an abstract model for operations of smart meters was proposed [46]. This abstract model represents an implementation-independent model of the components of the meter, their operations, and their execution order. In this article, we express the abstract model formally in rewriting logic [38]. Rewriting logic lets us model all the operations (functions) of the system and the transitions between its states. In 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. In step 3, we compose the model of the smart meter, concurrently with the model of the attacker's 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. We map the execution paths to the implementation of the system to identify concrete attacks against 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.

Code-level analysis: We follow a two-step approach for code-level analysis of security of smart meters. These two steps correspond to steps 2 and 3 of the design-level analysis explained above. In step 1, we transform the code of the smart meter. An attack is a result of one or more actions taken by an attacker, e.g., dropping messages and replaying messages. The operations for the first step of our approach are shown in boxes 1 and 2 of Figure 5. In this step, we define a set of functions that represents an attacker's actions and inject these functions in the source code. We define these functions in such a way that they may be dynamically activated/deactivated during runtime. This allows us to consider all different ways an attacker may interact with the system. We call the code resulting from the injecting attacker's actions the *transformed* code. The operation of the second step is shown in boxes 3, 4, and 5 of Figure 5. In this step, the developer of the system (who wants to

analyze its security) may define a set of assertions that verify the security invariants of the system. The user of our technique injects these assertions in the transformed code. Our technique uses a symbolic execution engine to symbolically execute the transformed code and finds out whether there exist any execution flow in the code that violates the security assertions. The output of this step would be a set of inputs to the system that lead to violating security invariants. We translate this set of inputs to attacks that an attacker may mount on the original system to successfully interfere with the software's execution.

Comparison: In both techniques, we explore how the attacker's actions may affect execution paths of the system and transition it to a state where its correctness properties do not hold. However, there are two notable differences in the two approaches. First, the design-level approach analyzes execution paths with respect to the attacker's actions at an abstract level. Therefore, certain implementation details are not present in the analysis. The code-level analysis, on the other hand, performs the analysis directly on the code. Second, the design-level technique uses model-checking for finding attacks, while code-level analysis uses symbolic execution. These tools have been chosen as they facilitate analyzing the effect of attacker's actions on execution paths of the system. Model checking allows for exhaustive exploration of paths of a system specified in rewriting logic. Symbolic execution allows for exploring runtime behavior of the code without executing it.

Hypothesis: Abstraction typically reduces the state space of the system; hence, we hypothesize the design-level approach to be more efficient. On the other hand, abstraction may increase false positives as some results may not be applicable to the code. Therefore, we expect the code-level analysis to have fewer false positives compared to design-level analysis, although it may be slower. We test this hypothesis in this article.

5 DESIGN-LEVEL ANALYSIS

In this section, we explain the three-step approach highlighted in Section 4 for analyzing security of smart meters at the design level. We use rewriting logic [38] to formally model smart meters and attacker's actions. Rewriting logic is a flexible framework for expressing proof systems. We implement the formal model of the system in rewriting logic using Maude [21]. Maude is a tool that supports rewriting logic and enables the users to both execute rewriting logic rules and formally verify them. This allows us to execute the model to gain confidence before formally verifying it.

5.1 Formal Model

We use the abstract model of smart meters presented by Molazem et. al. [46] 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 communications unit of the meter also periodically

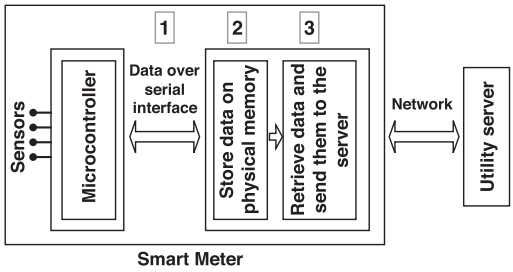

Fig. 1. In this section, we discuss and formalize the first execution path of the smart meter, shown in this figure.

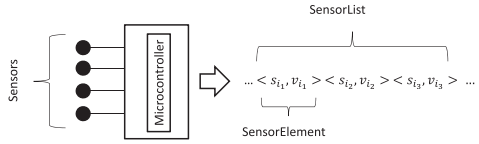

Fig. 2. SensorList is a series of SensorElements and is the result of microcontroller operations.

```
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
```

(a) Formal model of sensor data in Maude.

```
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
```

(b) Formal model of states of sensor data in Maude.

Fig. 3.

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 one part of the smart meter, namely passing the consumption data from the microcontroller to the communication unit of the meter. The corresponding paths are shown in Figure 1. For clarity and simplicity, we omit some details of the model—the formal model description for other parts may be found in our prior work [47].

5.1.1 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 Figure 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 Figure 3(a). In line 2 of Figure 3(a), 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, each of these types is called

```
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
```

Fig. 4. Formal model of the attacker actions.

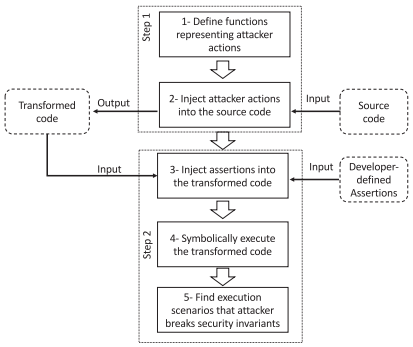

Fig. 5. Overview of code-level analysis technique to find software-interference attacks against the system.

a *sort*. Each sensor element is a tuple $< s, v >$ (as shown in Figure 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 built by putting a series of *SensorElement*s together. In line 8 of Figure 3(a), we define a common operation on the sensor data: *hasSensor*. This operation checks whether a stream of *sensorElement*s (i.e., *sensorList*) contains data associated with a specific sensor index.

After defining the sensor data in a smart meter, we present the rules that define their production in Figure 3(b). 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 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—the number of sensor channels depends on the specific model of the meter.

5.2 Attacker Model

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. It is possible to extend the set of attacker's actions to more sophisticated ones.

We present the formal rules for the attackers' actions in Figure 4. Dropping a message is defined in line 8 of Figure 4 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 at a random time.

Line 9 presents the general rule for rebooting the system. This action may correspond to simply rebooting the meter by unplugging it from the 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. The equation *before* in line 10 will return *true* if state p is a prior state in the system.

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 memory and allow transitioning to a data submission state while the socket is closed. Note that not all the unsafe states necessarily represent a feasible software-interference attack on the real smart meter. We discuss this in more detail in Section 7.

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 software-interference 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 $L = (r_1, r_2, \ldots, r_n)$, a sequence of rewrite rules $r_i, 1 \leq i \leq n$ that leads 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 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.

6 CODE-LEVEL ANALYSIS

In this section, we explain the two-step approach highlighted in Section 4 for finding software-interference attacks against the system at the code level. We model the attacker's actions in a generic way, which allows us, via symbolic execution, to exhaustively search for *all* the ways an attacker may interfere with execution of the software during runtime. Our technique finds the interactions that lead to breaking security invariants of the system and identifies these as software-interference attacks on the system.

6.1 Problem Formulation

We denote the program running on the device as P. We also assume that the attacker is capable of mounting a set of actions $A = \{a_1, a_2, \ldots, a_k\}$ on P. These actions may include dropping messages, and rebooting the system. Each of these actions interfere with execution of P and change its behavior. $V = \{v_1, v_2, \ldots, v_m\}$ denotes the set of security invariants that must hold true for P. We show execution of P with respect to input I and attacker's actions $A_i \subset A$, which satisfies the set of invariants $v \subset V$ as $hold(P_{I,A_i}, v) = true$.

We call $M(P)$, to be a transformation of P if the following conditions hold: (1) The set of security invariants for P is equivalent to the set of security invariants for $M(P)$; and (2) for any set of inputs

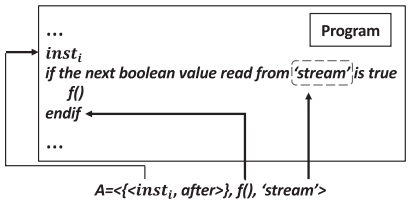

Fig. 6. High-level representation of injecting attacker's action in the code.

and attacker's actions on P that result in violating a subset of security invariants in P, there exists a set of inputs for $M(P)$ that results in breaking the same subset of security invariants and vice versa. More formally, we can write these conditions as follows:

$$\forall v \subset V \exists I, A_i \subset A \Rightarrow [hold(P_{I,A_i}, v) = false \iff \exists I' \Rightarrow hold(M(P)_{I',\varnothing}, v) = false] \quad (1)$$

Any modification to program P that satisfies the above two conditions is a valid transformation of P. In the first step, we want to find such a transformation. In the second step, we analyze $M(P)$ as a stand-alone program, finding execution instances that result in breaking its security invariants. These execution instances may be translated back to an execution instance of P and a set of attacker's actions that interfere with the software and lead to breaking the same security invariants in P.

6.2 Step 1: Code Transformation

To find software-interference attacks against the system, we mount the attacker's actions on the source code. This means that we insert code snippets to reset variables to *NULL*, reboot the system, or re-send messages to account for attacker's actions that drop messages, reboot, and replay messages. However, due to program structures such as branches and loops, there are many runtime possibilities to invoke attacker's actions within code paths. In this section, we discuss *how* we inject attacker's actions in the code so that we search all possible scenarios an attacker may perform at runtime to attack the system.

We define an attacker's action A as a triplet $A =< l, e, t >$. l denotes the instruction after which either *data* or *execution flow* changes due to attacker's action. In other words, l indicates the location in the code the attacker affects. e denotes the change in *data* or *control* as a result of the attacker's action. Examples of this include setting a value to *NULL* or jumping to an arbitrary location in code. t indicates the time the attacker's action occurs.

Figure 6 shows the high-level representation of attacker's action injection. We explain the procedure in more detail. To define each attacker's action, we define elements l, e, and t. For any action, l is a set of tuples $< s_i, p_i >$ where s_i is an instruction in the code, and p_i determines whether the attacker's action must be applied *before* or *after* s_i. We define wildcard "*" in place of s_i to denote that an attacker's action may be applied before/after *any* instruction. An example of this scenario is rebooting a device when an attacker has physical access to the system. For simplicity, here we consider the representation of the code in a high-level language (e.g., C, C++, Lua) as the granularity for l. We made this choice to simplify the implementation and decrease its performance overhead, while still keeping high granularity for l. However, our technique can also consider machine-level instructions as the granularity for l.

For example, if an attacker is capable of dropping messages sent to/from the device, the location l in which the software-interference attack manifests itself is before/after send/receive system calls. Therefore, we will have:

$$l = \{< send, before >, < receive, after >\} \quad (2)$$

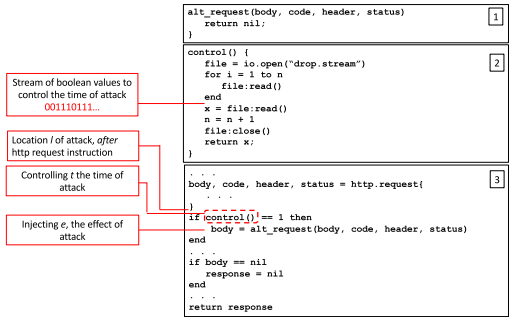

Fig. 7. "alt_request()" represents action of an attacker that is capable of dropping http response messages.

e represents the effect of the attacker's action at runtime. It may be any change in data and/or execution flow. Therefore, we define it as a function that implements the effect of an attacker's action. The arguments of the function are the return values of the instructions listed in l. The argument set may be empty if there exist no return values. For example, dropping a message before it is received may result in the return value of *NULL*. In this scenario, e will be a function taking one argument, which is the result of *receive()*, and sets it to *NULL*. We define t as a Boolean stream deciding the time attacker's action A occurs. A control function, at locations indicated by l, simply reads a Boolean value from the input stream, deciding whether the function associated with e must be invoked.

We explain an attacker's action injection further with an example. Figure 7 presents an example software-interference attack. The corresponding code is

$$A = \langle\{\langle http.reques,\ after\rangle\}, alt\_request(body, code, header, status), drop\_stream\rangle, \tag{3}$$

which allows malicious users to drop an http response. In this example, l indicates that the attacker's action is injected *after* "http.request" instruction. e indicates the pointer to function "alt_request"() that implements the effect of the attacker's action. t refers to file "drop.stream", which contains the Boolean values determining the times when the attacker's action must be invoked. In Figure 7, the code segment we inject an attacker's action into is presented in box 3. As determined by l and t, and e, we inject the functions "control()" and "alt_request()" after "http.request" instruction. At runtime, every time the execution flow reaches this location, a new Boolean value is read from "drop.stream" and is returned as the value of "control()". If the value is *true*, "alt_request()" is executed, which sets the response of "http.request" to *nil*. This is what happens when, in fact, no response is received (which is the case when the attacker drops the response). If the value read from the stream is *false*, no extra action is taken and the execution flow continues as normal. As the example shows, by injecting the function "alt_request()" at the right places, and using Boolean streams, we can represent all the ways in which an attacker may drop the "http" response to our system at runtime.

6.3 Step 2: Finding Software-interference Attacks

As we stated before, our goal is to find software-interference attacks on the system. This means finding actions an attacker must take, at certain times, so that security invariants of the system are violated. In the previous section, we injected code snippets, corresponding to the attacker's actions, in the code base. We also introduced streams of Boolean values into the input. The assignment of different Boolean values to the introduced streams represent different actions taken by the attacker at runtime, and covers the entire space of attacker's actions.

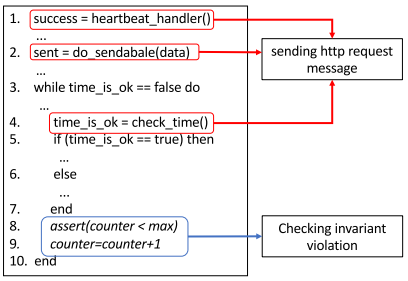

Fig. 8. By symbolically executing the code, we find out *how* attacker's actions may lead to violating security invariants in the code.

A sequence of attacker's actions leads to a successful software-interference attack if it violates one or more of the security invariants of the system. For every security invariant in the system, the user of our technique needs to introduce assertions in the code. These assertions verify the state of files, network ports, or values of sensitive data.

Given this setup, any set of input values that leads to violating the assertions corresponds to a successful software-interference attack that may be mounted on the system. We explain this with an example. Figure 8 presents an execution path in the code in which an attacker's action for dropping http response messages has been injected. In lines 1, 2, and 4, the function calls "heartbeat_handler()", "do_sendable()", and "check_time()" send and receive http requests and responses to/from the server. The piece of code that handles the http communication is shown in Figure 7. "heartbeat_handler()" sends heartbeat messages to the server to show that the meter is up and running. "do_sendable()" sends consumption data to the server. "check_time()" performs time synchronization with a server.

In lines 8 and 9, the user of our technique adds an assertion check to verify that the system is not stuck in the loop. Parameter "max" may be adjusted according to the acceptable delay. Assuming "max=5", assertion in line 8 of Figure 8 will be violated if we let the values read from "drop.stream' in Figure 7 be *false, false, true, true, true, true, true, true*. We note that changing either of the first two values to *true* (which leads to failure of heartbeat and data communication) may change the execution path from those considered in Figure 8 (for simplicity, the alternative paths are not shown).

By observing the values read from "drop.stream", we can infer what the attacker has to do to mount a software-interference attack that violates a security invariant. In this example, the attacker must let the heartbeat and data message as well as response pass through intact, and drop the response for only the time synchronization messages five times in a row.

To find the input values (including the Boolean streams) that lead to violating the security invariants, our system symbolically executes the code injected with attacker's actions. Symbolic execution analyzes the code to find the input values that lead to execution of different parts of the code. There are many existing symbolic execution engines that we can use for this purpose. We are interested in input values that result in violation of security invariants (assertions in the code). The values of Boolean streams in every set of input that leads to violation of security invariants determine the actions the attacker must take to mount a software-interference attack on the system.

7 EVALUATION

7.1 Research Questions

In this section, we evaluate our techniques for finding software-interference attacks at the design-level and the code-level for embedded systems. We lay out the following questions:

Table 1. Performance of Our Two Analysis Techniques for Different Attacks

	Design-level analysis		Code-level analysis	
	Time (m)	**Attacks Found**	**Time (m)**	**Number of injected actions**
Rebooting	114	6452	23	327
Dropping messages	0.12	12	4	6
Replaying messages	0.3	845	19	274

—**RQ1 (Performance):** Using our techniques, how long does it take to analyze the code?
—**RQ2.1 (Comprehensiveness of design-level analysis):** How many software-interference attacks does our design-level analysis technique find against a real smart meter?
—**RQ2.2 (Comprehensiveness of code-level analysis):** How many software-interference attacks does our code-level analysis technique find against a real smart meter?

7.2 Testbed

Smart meter: We evaluate our technique for finding software-interference attacks on SEGMeter, an open source smart meter from Smart Energy Groups [2]. SEGMeter consists of two main boards: (1) an Arduino board [10] with an ATMEGA32x series microcontroller, which is connected to a set of sensors and calculates consumption information, and (2) a gateway board that 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 1,300 lines of code written in the Lua language (not counting the communication stack implementation). The software running on the Arduino board consists of about 1,500 lines of C code (not including the Arduino libraries).

Analysis platform: We ran the formal analysis and code analysis on a Linux machine equipped with 16GB of RAM and 3.4GHz CPU. To run the symbolic execution, we use SymbolicLua [13], a dynamic symbolic execution engine for Lua (as the smart meter's code is written in the Lua language). SymbolicLuca uses Z3 [23] as its Satisfiability Modulo Theory (SMT) solver. We stubbed out the external dependencies such as server calls to create a stand-alone program for performing the analysis. For the sensor board's code that is in the C language, we used Clang's static analyzer, which has a symbolic execution engine [4]. To evaluate the formal model written in rewriting logic, we used Maude 2.7 [21] released for Linux64. Maude is a tool to run analysis and search queries on models written in rewriting logic.

7.3 Performance (RQ1)

In this section, we first present the performance results for our design-level analysis technique and then discuss the performance results of our code-level analysis technique.

7.3.1 Design Analysis. We measure the time taken to run the searches associated with each attacker's action in Maude, along with the number of attack paths for each action found by the model in Table 1. 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's 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 1 shows that when the attacker's 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 5.1). 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.

7.3.2 Code Analysis. We measure the time it takes to run the analysis on the transformed code. Table 1 shows the analysis time for three attacker's actions: dropping messages, replaying messages, and rebooting. The times shown in the table are rounded up to the nearest minute.

We also show the number of points where the attacker's actions are injected in the code. We find that the analysis time is correlated with the number of injection points. For example, the analysis time for *dropping messages* action is only 4 minutes, as the number of injection locations is small (six locations in the code). The reason is that based on the model for the action of dropping messages, the action is only injected after http and serial communication API calls. However, the number of injected attacker's actions for replaying messages and rebooting are far higher. The serial communication may receive data asynchronously; therefore, the attacker may replay messages at any time. The reboot action may occur at any time as well; hence, both these actions are injected after every instruction in the code.

This increases the number of states of the code; hence the analysis time increases to 19 minutes for replaying messages and 23 minutes for rebooting the meter.

The reboot operation (as explained in Section 7.4) jumps to the beginning of the code. This may increase the analysis time indefinitely. Therefore, we added a flag to limit the number of reboots by the symbolic executor to a single one. This is reasonable, as a single reboot resets the runtime memory of the system, and extra reboots will have a similar effect. It is worth noting that reboot operation is an extreme example of code transformation in terms of increasing the states of the code. Therefore, it is a measure of the worst-case time taken by our technique.

The total analysis time for our technique for all attacker's actions is less than an hour. This is acceptable as the analysis is performed offline and prior to the deployment of the system.

7.3.3 Comparison. The total analysis time at the design-level is about 2 hours, which is twice as long as code-level analysis. In our evaluation, we observed that symbolic execution engine (using its SMT solver) was able to drop many of execution paths in the code that were not viable. Symbolic execution tries each execution path only once, using symbolic values, which leads to

shorter analysis time. In contrast, the design-level analysis technique had to try each path multiple times, as it did not have the information from the code to prune them.

7.4 Comprehensiveness (RQ2)

In this section, we first present the results of finding software-interference attacks for our design-level analysis technique, and later, discuss the results of our code-level analysis technique with regard to RQ2.

7.4.1 Design Analysis (RQ2.1). Our formal model is based on an abstract model of smart meters. Hence, it does not factor in the implementation details of SEGMeter, and some 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 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

In our experiments, Maude found nine 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 Figure 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. Also, Maude finds two paths where dropping messages may lead to data loss, one of which was applicable to the meter code. In this scenario, dropping time-synchronization messages leads to the meter getting stuck in a loop and failing to record consumption data. Finally, Maude finds two paths where replaying messages leads to incorrect behavior in the model. Only one of these paths may successfully be instantiated on the meter. In this case, replaying request for sensor data leads to early submission of data; hence, the meter fails to record them. As an example, we explain how our design-analysis technique finds attacks with respect to the *reboot* action.

Rebooting Meter. We study the effect of rebooting execution by adding its action model (as defined in Section 5.2), 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):

$$search\ sensor(N_1, M_1)\ sensor(N_2, M_2)\ sensor(N_3, M_3) \Rightarrow\ sensor(N_1, M_1)\ sensor(N_2, M_2). \quad (4)$$

The above search phrase considers three 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.

We explain a concrete example of an attack path found in our model by applying the *reboot* action, which successfully maps to the meter's code. To understand this attack, we need to understand how consumption data is updated in our smart meter model. Figure 9 shows the state

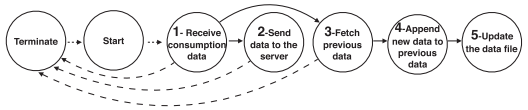

Fig. 9. The abstract model for updating sensor data file.

```
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
```

Fig. 10. SEGMeter code for updating sensor data file. The comments are added by us to show the mapping with the states in Figure 9.

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 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 produces paths from states 1, 3, and 4 of Figure 9, to the initial state of the system. In these paths, the meter receives new consumption data, but does not update the data file; hence, the data is lost when the meter is rebooted.

We explain the details of a reboot attack found on the meter. We show the snapshot of the code in SEGMeter associated with updating the data file in Figure 10. In line 2 (associated with state 3 of Figure 9), previously recorded data (called *all_data*) are read from the data file. In line 3 (associated with state 4 of Figure 9), current data and previous data are merged together. In line 5 (associated with state 5 of Figure 9), 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, anyone who is able to physically observe the meter may synchronize their operation of rebooting the meter with these time intervals.

We observed that rebooting the system after line 4 not only erases new data, *but in the worst case, all the previously stored data in the file are wiped out from the system.* The reason is that in line 4 of Figure 10, 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.[2]

[2]We found a similar vulnerability on YoMo [33], another open source smart meter, suggesting that this is a common design pattern, and likely a bug in the design.

```
function alt_http_resend(uri, method, msg,
                msg_len, content_type, response)
   n = io.read("*n")
   for i = 1, n do
      local body, code, headers, status =
         http.request{
         url = uri,
         method = method,
         headers = {
            ["content-length"] = message_length,
            ["content-type"] = content_type
         },
         source = ltn12.source.string(message),
         sink = ltn12.sink.table(response)
      }
   end
   return body, code, headers, status
end
```

```
function alt_serial_resend(address,
                                port, msg)
   serial_client =
      socket.connect(address, port)
   n = io.read("*n")
   for i = 1, n do
      serial_client:send(msg)
   end
   serial_client:close()
end
```

```
function alt_boot()
   resetVariables()
   goto start
end
```

```
function alt_http_receive(body, code, header,
                          status)
   return nil;
end
```

```
function alt_serial_receive(length)
   // returning nil for stream,
   // status and partial
   return nil, nil, nil;
end
```

Fig. 11. The functions defining dropping messages, replaying messages, and rebooting the system.

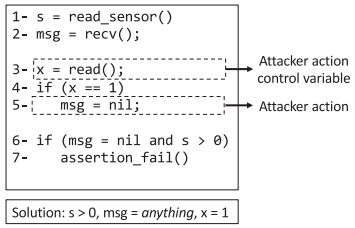

Fig. 12. Solution to an attack.

7.4.2 Code Analysis (RQ2.2). In this section, we explain the results of our code-level analysis technique for finding software-interference attacks. As mentioned earlier, we consider three actions for the attacker, namely rebooting the system, dropping messages, and replaying messages. The functions defining these actions are shown in Figure 11. The meter uses two different APIs for communication. One is via LAN, using http, and the other is via serial interface. Therefore, we have two sets of actions targeting data received through each of these interfaces. Function "alt_http_receive" targets http messages received via LAN, and function "alt_serial_receive()" targets messages received via serial interface. Functions "alt_http_resend" and "alt_serial_resend()" replay messages via LAN and serial interface, respectively. Function "alt_boot" represents rebooting the system. It calls a "resetVariables" function, which reset all the variables in the code to their initial values and then jumps to the beginning of the code. It is important to note that this way of representing reboot makes analysis via symbolic execution easier as it provides a continuous execution flow.

Our technique finds solutions for software-interference attacks as a series of symbolic input values to the system that satisfies certain conditions. An example of such a solution is shown in Figure 12. The solution in Figure 12 suggests that to reach line 7 where an assertion fails, the sensor input *s* may have any value greater than zero, received message *msg* may be any string, and the value of *x* that determines whether or not the attacker's action of dropping message should be invoked is 1 (indicating that the attacker's action should be invoked). This solution provides us with the input values and the precise time at which the attacker's action should be invoked during execution (i.e., between lines 2 and 6 in Figure 12). Therefore, we can mount the software-interference attack on a real system with relative ease and precision.

Our technique found 12 solutions where dropping messages, either communicated through the LAN or serial interface, leads to a software-interference attack against the system. These solutions represent three different types of software-interference attacks. These attacks lead to (1) overflowing sensor buffer, (2) losing new consumption data from sensors, and (3) getting stuck in the time-synchronization process. The first two attacks allow the attacker to pay lower amounts

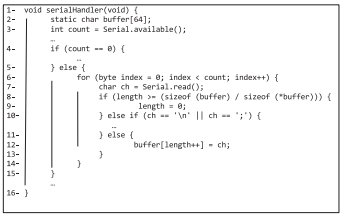

Fig. 13. Code snippet for reading data from the serial interface.

for electricity consumption. The third attack delays the meter, which may affect the recording of consumption data and/or processing server commands. Finally, our technique found 18 solutions where replaying messages results in breaking security invariants in SEGMeter. These solutions indicate two different types of attacks. The first type results in losing new consumption data, while the second type results in overflowing a data buffer and overwriting consumption data.

Below, we provide an example software-interference attack that our technique finds based on the attacker *replaying* messages.

Replaying Messages. Figure 13 shows the code snippet for "serialHandler()" function. The controller of the sensors calls this function when reading data via a serial interface. Received data will be stored in *buffer*, defined in line 2. The length of this buffer is set to 64, which is defined in line 2. This size is picked based on the maximum length of commands received by sensor controller. In line 6, "serialHandler" loops over available characters on the serial interface and adds the characters to the buffer in line 12. As a precaution, in line 9, "serialHandler" wraps around the buffer and starts over from index 0 if it goes over the size of the buffer. However, an attacker can exploit this feature and send data of the size *64 —sizeof(command)* to the sensor controller, after *command* string is sent to it. This leads to the controller thinking the size of the received data (as indicated by *length* in the code) is 0, and therefore, ignoring the command. Our technique finds this software-interference attack as symbolic execution discovers a path in which replaying messages leads to failure of an assertion that ensures that the command size is greater than 0. This assertion is presented in row 5 of Table 2. These assertions are extracted from functional and non-functional requirements of smart metering operations specified in the corresponding design documents [11, 31].

To mitigate this problem, the sensor controller may use two buffers for consecutive communications and alternate between them (i.e., dual buffering). This ensures the content of the buffers are processed before they are used again for receiving a message.

7.4.3 Comparison. Code-level analysis finds three times more software-interference attacks than design-level analysis. Moreover, analysis at the design level has false-positive rates of up to 75% (average of 50%). The reason for this high false-positive rate is that the design-level analysis is done on an abstract model of the meter. Not all the attacks on the abstract model may be applicable to the code. For example, of the four types of reboot attacks found on the design-level analysis, three of them were mitigated at the code level and, hence, did not apply to it.

The software-interference attacks found via code analysis, on the other hand, are a strict superset of the attacks found by design analysis, with no false positives. Although symbolic execution may potentially lead to false positives, we did not observe any in our evaluations, probably due to the simplicity of the meter's code.

While code-level analysis outperforms design-level analysis both in terms of time and accuracy, design-level analysis is still helpful as it allows developers to discover shortcomings of the design

Table 2. Assertions Used for SEGMeter

Security invariant	Assertions	How to define assertions
1—Data must be stored	—Check "save_data" flag is set —Check "seg_data.dat" file exists and is non-empty —Check "node_list.dat" exists and is non-empty —Check "site_token" has the latest value at startup	Find all data files in the code and make sure they are created/updated correctly
2—Data must be sent to communication board	—Check "talk_to_seg" flag is set; otherwise, the communication fails	Find the necessary condition for server communication in the code
3—Meter must meet its functional requirements in a loop	—Check http timeout is at most 5s; otherwise, delays operations —Check loop counters do not exceed maximum limit; otherwise, delays operations	Study the timing constraints described in the specification document and make sure they are satisfied in the code
4—Consumption data must be calculated correctly	—Check consecutive timestamps are within correct range —Check consumption data is within correct range —Check data size is smaller than size of buffer for storing data —Check buffer indices are within the range of buffer size	Check requirements of correct consumption calculation based on the specification document and make sure they are satisfied in the code
5—Commands must be received/processed correctly	—Check size of command buffers be at least the size of command strings —Check the size of commands be greater than 0 —Check buffer indices are within the range of buffer size	Find all the commands described in the specification and make sure enough buffer capacity and resources are assigned to each corresponding command string

early and avoid implementation mistakes that may lead to security attacks. This is important, as addressing the bugs *after* implementation significantly increases the development cost [15, 16].

7.5 Mounting the Attacks

To evaluate feasibility of the software-interference attacks found by our analysis techniques, we mount them on SEGMeter. In our evaluation, our techniques found several solutions for each attack category. Each solution represents an attack. To mount the attacks on the meter, we used commonly available hardware/software and inexpensive tools to perform rebooting and replaying/dropping messages. The total value of the hardware required for mounting the attacks was less than $50 USD (based on prices in 2017 on eBay.com), and the hardware was easily available.

To drop and/or send messages to SEGMeter via serial interface, we used a 6-pin-serial-to-USB cable. This way we interfaced our laptop with the meter and intercepted the traffic. The parameters needed to be set for serial communication are data size in the frame (5–9 bits), stop bits (1–2 bits), parity bits (0–1 bit), and baud rate (there are about 10 common baud rates). There are a limited number of available configurations, and we tested different configurations to arrive at the correct one: 8-bit data size, 1 stop bit, no parity, and 38,400bps baud rate. The solutions indicate the exact points in time where the attacker's actions must be applied. Having this information, we were able to estimate the timing of messages indicated in the analysis solutions via software profiling. Based on the timing estimation, we were able to mount the attacks targeting the serial interface.

To mount the attacks targeting http communication via LAN, we used IPTables, which is a user-space firewall installed on many Linux distributions by default. We selected one of the machines in our lab that route the traffic of SEGMeter and inserted IPTables rules that drop the messages specified in our solution. We used source and destination IP addresses to identity the messages; therefore, the attack was feasible regardless of encryption.

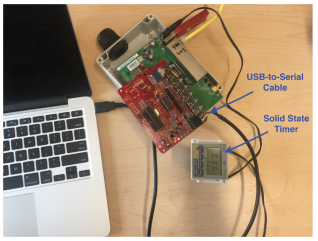

Fig. 14. We used a programmable solid state timer for rebooting the system at precise times and a USB-to-Serial cable to mount replay attack on the meter.

To mount the reboot attacks, we used a programmable solid state timer (Figure 14). This allows us to send reboot signals to at exact pre-programmed times. The solutions provide us with the exact execution points in the code at which the reboot must be applied. Having this information, we identified the timing of the reboot attacks via software profiling. Also, we took advantage of the LEDs on the meter that indicate the state of the meter (e.g., data communication via green flashes). We place the timer between the power source and the meter, and repeatedly reboot the meter in the calculated time intervals using it. We were able to successfully mount the reboot attacks on the meter with high probability in most of the trials (17 out of 20 trials).

8 DISCUSSION

8.1 Limitations

Generalizability: In this article, we consider finding *software-interference* attacks against IoT devices, and evaluated our techniques on a smart meter. We considered three actions, namely dropping messages, replaying messages, and rebooting the device as the attacker capabilities. Since we have not applied our technique on platforms other than the smart meter, its generalizability to other IoT devices is yet to be evaluated. Unfortunately, we did not have access to other candidate platforms, such as source code of other meters [29, 35] and smart car embedded systems. The characteristics of other IoT devices, e.g., code complexity, accessibility, and their behavior model, may affect the results of our technique. However, we note that we did not rely on any specific feature of smart meters to develop our technique. We rather relied on the characteristics of similar platforms in general. The main features that our technique relies on include: (1) small search space of the code—small size of the code for special-purpose IoT devices makes their analysis using our technique feasible; (2) physical accessibility—we are able to define a clear set of physical accesses the attacker has to the device; (3) single-purpose platform—since, unlike general-purpose machines, our target device is designed to carry-out specific tasks, we are able to build their formal model in our technique. If an IoT device has these three characteristics, we *expect* the same approach and attacker model to be applicable to them. However, further investigation is required to confirm this expectation.

Model correctness: The correctness of the results of our design-level analysis 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 verify that, in the absence of attacker actions, it matches the real smart meter's behavior.

Scalability: Increasing the complexity of the model and the number of attacker actions, as well as increasing the size of the code, increases the state space for model-checkers and symbolic execution engine. This, 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 embedded devices typically have limited computational and memory resources and perform a narrow range of functions. Therefore, we believe these techniques can apply to many classes of embedded systems.

Attacker actions: In the code-level approach, we assume that attacker actions that can always be represented as a code snippet. If an attacker action cannot be simply represented as a function, we cannot incorporate it in our model. For example, removing and corrupting flash memory and changing power voltage outside the program cannot be easily represented as a function in the code. The design-level approach does not suffer from this limitation, however.

8.2 Lessons Learned

One of the surprising results of our study is that the code-level technique surpasses the design-level technique for discovering software-interference attacks *both in terms of accuracy and scalability*. This shows that our initial hypothesis was incorrect, as we had expected code-level analysis to be more accurate (i.e., have fewer false-positives), but slower than the design-level analysis. This was not the case, however, even without considering the cost of translating in the model-based technique, as our technique had fewer paths to analyze at the code level (as many of the paths could be pruned as they were found to be infeasible paths). On the other hand, the design-level technique had to contend with a larger number of paths, as it could not easily determine which paths were infeasible. This seems to indicate that code-level techniques such as symbolic execution may be more effective for security attack analysis than design-level techniques for IoT devices.

9 CONCLUSION

IoT devices, unlike remote servers, are user facing; therefore, attackers have extended access to them. This may include physical access and access to internal communication interfaces. We call the attacks resulting from this extended level of access *software-interference* attacks. We introduced two techniques, one at the design-level and the other at the code-level, for automatically finding such attacks in IoT devices. We evaluated our techniques on SEGMeter, an open source smart meter. Although both techniques successfully discovered real attacks against the system, our code-level analysis technique proved to be more efficient and more accurate. The design-level technique found three types of attack on SEGMeter and completed the analysis in about 2 hours. On the other hand, the code-level analysis technique found nine different types of attacks, including the attacks found by the design-level approach, and took less than 1 hour. It also has no false-positives unlike the design-level approach, which incurred false-positive rates of up to 75% (average of 50%).

REFERENCES

[1] 2017. In-Stat and NDP Group Company. Retrieved from http://www.instat.com/press.asp?ID=3352&sku=IN1104731WH.
[2] 2017. Smart Energy Groups Home Page. Retrieved from http://smartenergygroups.com.
[3] 2017. Acunetix Web Application Security Scanner. Retrieved from http://www.acunetix.com/.
[4] 2017. Clang: A C Language Family Frontend for LLVM. Retrieved from https://clang.llvm.org/.
[5] 2017. FBI: Smart Meter Hacks Likely to Spread. Retrieved from http://krebsonsecurity.com/2012/04/fbi-smart-meter-hacks-likely-to-spread/.
[6] 2017. Hacking Humans. Retrieved from http://blog.kaspersky.com/hacking-humans/.
[7] 2017. Hacking Medical Devices for Fun and Insulin: Breaking the Human. Retrieved from https://media.blackhat.com/bh-us-11/Radcliffe/BH_US_11_Radcliffe_Hacking_Medical_Devices_WP.pdf.

[8] 2017. HP WebInspect. Retrieved from http://www8.hp.com/us/en/software-solutions/webinspect-dynamic-analysis-dast/index.html.
[9] 2017. IBM Security AppScan. Retrieved from http://www-03.ibm.com/software/products/en/appscan.
[10] 2017. Arduino home page. Retrieved July 31, 2017 from http://www.arduino.cc.
[11] 2017. UK Department of Energy, Smart Meter Design Document. Retrieved July 31, 2017 from https://www.ofgem.gov.uk/ofgem-publications/63541/smart-metering-prospectus.pdf.
[12] 2017. National Vulnerability Database. Retrieved from https://nvd.nist.gov/.
[13] 2017. SymbolicLua. Retrieved from https://github.com/kohyatoh/symboliclua.
[14] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*. ACM, 340–353.
[15] Barry Boehm and Victor R. Basili. 2005. Software defect reduction top 10 list. *Foundations of Empirical Software Engineering: The Legacy of Victor R. Basili* 426 (2005), 37.
[16] Barry W. Boehm. 1988. Understanding and controlling software costs. *J. Parametrics* 8, 1 (1988), 32–68.
[17] S. Brinkhaus, D. Carluccio, U. Greveler, D. B. Justus, and C. Wegener. 2011. Smart Hacking for Privacy. In *28th Chaos Communication Congress*. Berlin, Germany.
[18] Eric J. Byres, Matthew Franz, and Darrin Miller. 2004. The use of attack trees in assessing vulnerabilities in SCADA systems. In *Proceedings of the International Infrastructure Survivability Workshop*. Citeseer.
[19] Cristian Cadar, Daniel Dunbar, Dawson R. Engler, et al. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, Vol. 8. 209–224.
[20] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. 2005. Defeating memory corruption attacks via pointer taintedness detection. In *DSN 2005*. IEEE, 378–387.
[21] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. 2007. *All About Maude-a High-performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*. Springer-Verlag.
[22] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. 2013. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security Symposium*. 463–478.
[23] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
[24] Stéphanie Delaune, Steve Kremer, and Graham Steel. 2010. Formal security analysis of PKCS# 11 and proprietary extensions. *J. Comput. Secur.* 18, 6 (2010), 1211–1245.
[25] K. Fehrenbacher. 2010. Smart Meter Worm Could Spread Like a Virus. Retrieved from http://earth2tech.com/2009/07/31/smart-meter-worm-could-spread-like-a-virus/.
[26] Eduardo Fernandez, Juan Pelaez, and Maria Larrondo-Petrie. 2007. Attack patterns: A new forensic and design tool. In *Advances in Digital Forensics III*. Springer, 345–357.
[27] Michael Gegick and Laurie Williams. 2005. Matching attack patterns to security vulnerabilities in software-intensive system designs. *ACM SIGSOFT Software Eng. Notes* 30, 4 (2005), 1–7.
[28] David Gries. 2012. *The Science of Programming*. Springer Science & Business Media.
[29] itron. 2018. Retrieved from https://www.itron.com/.
[30] Somesh Jha, Oleg Sheyner, and Jeannette Wing. 2002. Two formal analyses of attack graphs. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*. IEEE, 49–63.
[31] Department of Energy and Climate Change and the Office of Gas and Electricity Markets. 2011. Smart Metering Implementation Programm. Retrieved July 31, 2017 from http://www.ofgem.gov.uk/e-serve/sm/Documentation/Documents1/Design20Requirements.pdf.
[32] Himanshu Khurana, Mark Hadley, Ning Lu, and Deborah A. Frincke. 2010. Smart-grid security issues. *IEEE Secur. Privacy* (2010), 81–85.
[33] Christoph Klemenjak, Dominik Egarter, and Wilfried Elmenreich. 2015. YoMo: The Arduino-based smart metering board. *Comput. Sci. Res. Dev.* (2015), 1–7.
[34] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. 2010. Experimental security analysis of a modern automobile. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP'10)*. IEEE Computer Society, Washington, DC, 447–462.
[35] landis. 2018. Retrieved from https://www.landisgyr.com/.
[36] N. Lewson. 2010. Smart Meter Crypto Flaw Worse Than Thought. Retrieved July 31, 2017 from http://rdist.root.org/2010/01/11/smart-meter-crypto-flaw-worse-than-thought.
[37] Lanchao Liu, Mohammad Esmalifalak, Qifeng Ding, Valentine A Emesih, and Zhu Han. 2014. Detecting false data injection attacks on power grid by sparse optimization. *IEEE Trans. Smart Grid* 5, 2 (2014), 612–621.

[38] Narciso Martí-Oliet and José Meseguer. 1996. Rewriting logic as a logical and semantic framework. *Electronic Notes in Theoretical Computer Science* 4 (1996), 190–225.
[39] Petr Matousek, Jaroslav Ráb, Ondrej Rysavy, and Miroslav Svéda. 2008. A formal model for network-wide security analysis. In *ECBS 2008*. IEEE, 171–181.
[40] Sjouke Mauw and Martijn Oostdijk. 2006. Foundations of attack trees. In *Information Security and Cryptology-ICISC 2005*. Springer, 186–198.
[41] P. McDaniel and S. McLaughlin. 2009. Security and privacy challenges in the smart grid. *IEEE S&P* (2009), 75–77.
[42] Stephen McLaughlin, Dmitry Podkuiko, and Patrick McDaniel. 2010. Energy theft in the advanced metering infrastructure. In *Critical Information Infrastructures Security*. Springer, 176–187.
[43] Stephen McLaughlin, Dmitry Podkuiko, Sergei Miadzvezhanka, Adam Delozier, and Patrick McDaniel. 2010. Multi-vendor penetration testing in the advanced metering infrastructure. In *Proceedings of ACSAC'10*. ACM, 107–116.
[44] Marino Miculan and Caterina Urban. 2011. Formal analysis of Facebook connect single sign-on authentication protocol. In *SOFSEM*, Vol. 11. Citeseer, 22–28.
[45] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (1990), 32–44.
[46] Farid Molazem and Karthik Pattabiraman. 2012. A model for security analysis of smart meters. In *WRAITS, Dependable Systems and Networks Workshops (DSN-W)*.
[47] Farid Molazem and Karthik Pattabiraman. 2016. Formal security analysis of smart embedded systems. In *Proceedings of the 2016 Annual Computer Security Applications Conference (ACSAC'16)*. IEEE Computer Society.
[48] Anderson Morais, Eliane Martins, Ana Cavalli, and Willy Jimenez. 2009. Security protocol testing using attack trees. In *2009 International Conference on Computational Science and Engineering, CSE'09*. Vol. 2. IEEE, 690–697.
[49] Nuno Neves, Joao Antunes, Miguel Correia, Paulo Verissimo, and Rui Neves. 2006. Using attack injection to discover new vulnerabilities. In *2006 International Conference on Dependable Systems and Networks (DSN)*. IEEE, 457–466.
[50] James Newsome and Dawn Song. 2005. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the 12th Network and Distributed Systems Security Symposium*. 3–4.
[51] Ivan Pustogarov, Thomas Ristenpart, and Vitaly Shmatikov. [n.d.]. Using program analysis to synthesize sensor spoofing attacks. In *Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security*. 757–770.
[52] Md Ashfaqur Rahman and Hamed Mohsenian-Rad. 2013. False data injection attacks against nonlinear state estimation in smart power grids. In *2013 IEEE Power and Energy Society General Meeting (PES)*. IEEE, 1–5.
[53] Indrajit Ray and Nayot Poolsapassit. 2005. Using attack trees to identify malicious attacks from authorized insiders. In *Computer Security–ESORICS 2005*. Springer, 231–246.
[54] Partha Datta Ray, Rajgopal Harnoor, and Mariana Hentea. 2010. Smart power grid security: A unified risk management approach. In *2010 IEEE International Carnahan Conference on Security Technology (ICCST)*. IEEE, 276–285.
[55] Bruce Schneier. 1999. Attack trees. *Dr. Dobb's Journal* 24, 12 (1999), 21–29.
[56] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M. Wing. 2002. Automated generation and analysis of attack graphs. In *Proceeding of the 2002 IEEE Symposium on Security and Privacy*. IEEE, 273–284.
[57] Siddharth Sridhar, Adam Hahn, and Manimaran Govindarasu. 2012. Cyber–physical system security for the electric power grid. *Proc. IEEE* 100, 1 (2012), 210–224.
[58] Smart meter testing framework Termineter. 2017. Retrieved from https://code.google.com/p/termineter/.
[59] Olivier Thonnard and Marc Dacier. 2008. A framework for attack patterns' discovery in Honeynet data. *Digital Invest.* 5 (2008), S128–S139.
[60] K. Zetter. 2010. Security pros question deployment of smart meters. *Threat Level: Privacy, Crime and Security Online* (March 2010).
[61] Berthier R. Zonouz, S. and P. Haghani. 2012. A fuzzy Markov model for scalable reliability analysis of advanced metering infrastructure. In *ISGT'12*.

Received October 2017; revised October 2018; accepted January 2019