2015 11th European Dependable Computing Conference

Flexible Intrusion Detection Systems for Memory-Constrained Embedded Systems

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

***Abstract*—Embedded systems are widely used in critical situations and hence, are targets for malicious users. Researchers have demonstrated successful attacks against embedded systems used in power grids, modern cars, and medical devices. This makes building Intrusion Detection Systems (IDS) for embedded devices a necessity. However, embedded devices have constraints (such as limited memory capacity) that make building IDSes monitoring *all* their security properties challenging. In this paper, we formulate building IDS for embedded systems as an optimization problem. Having the set of the security properties of the system and the invariants that verify those properties, we build an IDS that maximizes the coverage for the security properties, with respect to the available memory. This allows our IDS to be applicable to a wide range of embedded devices with different memory capacities. In our formulation users may define their own coverage criteria for the security properties. We also propose two coverage criteria and build IDSes based on them. We implement our IDSes for SegMeter, an open source smart meter. Our results show that our IDSes provide a high detection rate in spite of memory constraints of the system. Further, the detection rate of our IDSes at runtime are close to their estimated coverage at design time. This validates our approach in quantifying the coverage of our IDSes and optimizing them.**

Keywords: IDS, selective monitoring, memory constraints

I. INTRODUCTION

Embedded systems are becoming increasingly deployed in critical situations. For example, modern cars are equipped with embedded devices that control different parts of the car such as Anti lock Break System (ABS), and Adaptive Cruise Control (ACC). Millions of pacemakers are implanted around the world and in the power grid industry, the number of Advanced Metering Infrastructure (AMI), also known as smart meters, is projected to exceed 800 million by 2020 [38]. These embedded devices carry out critical tasks and hence, are potential targets for malicious users. They are also equipped with data and network interfaces which result in an increased attack surface. Researchers have successfully demonstrated vulnerabilities in all these classes of embedded systems [25, 28, 23, 22, 13, 46].

Intrusion Detection Systems (IDS) are widely deployed in general purpose computer systems to protect them from attacks. Based on the critical nature of the applications of embedded devices, building IDSes for these systems is a necessity. However, embedded devices have characteristics that make building IDSes for them challenging as follows:

- Limited memory capacity: An important component of an IDS is a model that represents the correct behavior of the software and/or a set of malicious behavior signatures. This model may occupy a large space in memory. However, many embedded devices have limited memory capacity and this makes a body of existing techniques inapplicable to them [37, 42, 15].
- Large scale deployment: This implies that the security mechanism should not have false positives. False positives occur when no actual attack has happened but an attack is reported by the IDS. Even a small rate of false positive for systems deployed on a large scale aggregates quickly. Further, these systems are deployed in mission critical settings where shutting them down on a false alarm is not a viable option.

Existing techniques for building IDSes for general computers are not suitable for embedded devices. For example, statistical techniques [43, 19, 40] have false positives that make them impractical for embedded devices that are deployed on a large scale. Techniques based on static analysis [42, 15] do not have false positives as they build a model by conservatively taking into account all possible code behavior. However, the size of this model may exceed the limited memory of embedded devices, and these techniques offer no systematic way to reduce the size of the model. For instance, Wagner et. al. [42] and Giffin et. al. [15] use system calls to build a model of the system. Reducing the size of the model by randomly removing some of the system calls results in a model that may no longer provide any guarantees on the coverage of the system behavior.

In this paper, we formulate the problem of building IDSes for embedded systems as an optimization problem. Given the set of the security properties of the system and the invariants that verify those properties, we build an IDS that maximizes the coverage for the security properties, with respect to the available memory. Because there exists no single metric for measuring security coverage [1, 2], our formulation is flexible in terms of using the desired coverage function. We also propose two coverage functions and build and evaluate IDSes using them. Our technique is a good fit for embedded systems as we can customize the IDS for embedded devices based on their memory capacities, and maximize the coverage for them. Also, our technique is based on static analysis and hence, has no false positives. *To the best of our knowledge, we are the first to build an IDS with guaranteed coverage under memory*

978-1-4673-9289-1/15 $31.00 © 2015 IEEE
DOI 10.1109/EDCC.2015.17

constraints. We make the following contributions in this work:

- We formulate the problem of building an IDS for embedded devices as an optimization problem. Using this formulation, we maximize the security coverage for embedded devices under a given memory constraint. This makes our solution applicable to a wide range of embedded devices with different memory constraints.
- Given the security properties of the system and the available memory, we automate the process of generating the IDS that maximizes the coverage of security properties.
- We propose two metrics for defining security coverage, and design two IDSes based on them: *MaxMinCoverage* IDS, and *MaxProperty* IDS. We implement these IDSes for SegMeter, an open source smart meter. Smart meters are important components of smart grids that measure real-time energy consumption and communicate with the utility provider over the network.
- We evaluate our IDSes based on their detection rates, compliance of their detection rates with their estimated coverage at design time, and their performance overheads. Our results show that using only a small portion of the memory required to build a complete IDS, *MaxMinCoverage* IDS can effectively distribute the coverage among all security properties with a minimum of 70% detection rate, while *MaxProperty* IDS can provide 100% detection rate for 5 out of 10 security properties of the system. Also, our results show that 1) the detection rate of our IDSes at runtime are very close to their estimated coverage at the design time, 2) they have a detection latency of 10 seconds, and 3) they incur less than 7% performance overhead on the system.

II. Background and Related work

Techniques for building IDS include broad categories. Here we discuss their limitations in addressing the requirements of embedded devices.

Hardware-based solutions: Hardware-based techniques rely on specialized hardware components that measure execution time, keep secret keys, etc. Timing Trace Module (TTM) and Trusted Platform Module (TPM) are examples of such hardware components [34, 31]. While using custom hardware is efficient in terms of performance, it is not suitable for low-cost embedded devices that are deployed on large scale as they result in added per-device cost and an increase in update costs [24].

AI-based solutions: There are many papers on building IDSes using Hidden Markov Models, Neural Networks, and Support Vector Machines. Warrender et. al. [43], Hu et. al. [18], and Hoang et. al. [17], use Hidden Markov Models to design the decision engine of IDSes. Neural Networks have been used to classify intrusions and detecting them [32]. Support Vector Machines [19] have also been used to model the correct behavior of the software so that deviation from the normal behavior is detected [40]. Statistical techniques have also been used to detect abnormal behavior of the software by analyzing large volume of audit data [45]. AI-based techniques are suitable for modeling the behavior of large and complex software and detecting unknown attacks. However, their false positives make them a poor fit for embedded devices. Many embedded devices are not directly interacting with users and hence, false alarms cannot be handled by individual users. In this case, even a small false positive rate of 0.5% can aggregate quickly to an overwhelming 5000 false alarms for 1000 units running 10 applications each.

Static analysis-based solutions: Static analysis techniques build a model of the software based on the source code. These models are conservative representations of the software and consequently, have no false positives. Wagner et. al. [42] propose a technique to build a model of the software based on the system call-graph. They show that the call-graph technique results in a high false negative rate and they improve their technique by introducing non-deterministic pushdown automaton (NDPDA). In NDPDA, they build an extensive model of the software based on the system calls. However, as shown in our prior work [37], this technique results in a high memory overhead. Giffin et. al. [15] propose the Dyck model to build an IDS based on static analysis of the software and its system calls. In their work, they added context sensitivity to the model, removing some of the false negative inaccuracy that exists in the static analysis techniques. While this improves the accuracy, it results in increasing the size of the model considerably. This is a huge disadvantage for embedded devices with small memory capacity.

Techniques developed for embedded devices: There are IDSes developed for embedded devices using the techniques discussed above [31, 8]. However, these IDSes are either designed for the communication link, or assume the presence of very specific architecture or hardware modules. For example, Mohan et. al. [31] propose an IDS for embedded systems equipped with multicore processes running on a Hypervisor. A secure core is dedicated to running the IDS which monitors the controller that is running on the other core. This work can be applied only to select embedded devices as the system must be equipped with a multicore processor as well as a Timing Trace Module (TTM). Also, it is designed only for higher end embedded devices that have enough resources to run a hypervisor. Berthier et. al. [7, 8], provide guidelines to build IDSes for smart meters. They propose a specification-based IDS for network communications of smart meters. This work is focused on building a network-based IDS and does not protect the host if the attacks are not detected by the network-based IDS. In our prior work [37], we developed an IDS for memory-constrained embedded devices, where we reduce the size of the IDS by considering a subset of system calls of the system. However, we do not optimize the coverage of the IDS based on the available memory. Also the process of building the IDS in [37] is not automated, and is specific to the smart meter we chose. Closse et. al. propose a tool for verification of real-time behavior of embedded systems [10]. However, they do not consider security properties in their work, nor do they discuss means to address memory limitations of embedded devices.

Remote attestation: Remote attestation techniques [35, 9, 26] use a remote server to verify the security of a system.

Information from the system is periodically sent to the server and is processed remotely. However, using this technique for run-time verification results in high communication overhead which increases energy consumption. This is a limiting factor for many classes of embedded devices. It also requires the devices to be connected to the remote server at all times. These limitations make this technique unsuitable for embedded devices that are deployed on a large scale in bandwidth and energy constraint environments.

Runtime verification: A class of papers are dedicated to monitoring and analyzing the system during run-time [29, 4, 30, 16] for safety properties. Basin et. al. [4] and Gunadi et. al. [16] propose a new specification language based on temporal logic for defining and monitoring safety and security policies. Maggi et. al. [29, 30] propose an LTL-based run-time monitoring of business processes to verify their compliance with certain constraints. Unlike us, they do not offer any mechanism for optimizing coverage for security invariants of the system within the memory constraints of embedded devices.

Summary: Existing IDS techniques are not designed to take into account the memory limitations of the platform on which they are being deployed. For embedded devices, this is a limiting factor. In this paper, we propose a technique that given the security invariants of the system, automatically builds an IDS that maximizes coverage of the invariants within the available memory. The coverage function in our technique is flexible and can be defined by the user. Therefore, our technique produces a customized IDS for every device, with respect to its memory capacity, and the security invariants provided by the user.

III. Threat Model

In this paper, we assume the goal of the adversary is to compromise the functionality of the embedded device. This means that they either prevent a critical functionality of the embedded device from being executed (e.g., data not being sent to the utility server, brake not being applied in a car), or functionalities being executed incorrectly (e.g., incorrect operations executed by the device). The first set of attacks target the availability properties, while the second set of attacks target the integrity properties of the device. To achieve this goal, the adversary may exploit the potential vulnerabilities in the embedded device resulting in changes to the normal execution path of the software. We do not consider attacks against confidentiality properties of the system in our threat model.

IV. Approach

In this section, we explain our approach for building an IDS for embedded devices, for optimizing the coverage of the security properties of the system with respect to its memory constraints.

We use examples from a smart metering platform to explain our solution. Hence, we first present a brief introduction of smart meters. After that, we present the problem formulation, and then an overview of our solution followed by a detailed explanation.

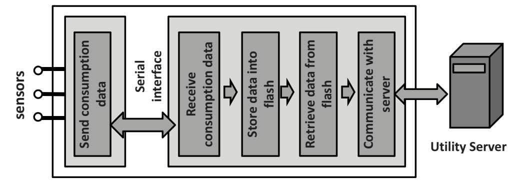


Fig. 1. An execution path of smart meter operations used as a running example.

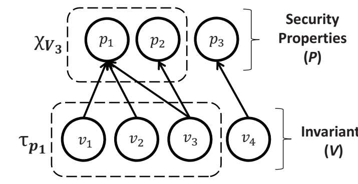


Fig. 2. Formulation of the IDS as an optimization problem.

A. Platform

Smart meters are networked embedded systems. They are key components of the smart grid that will be installed at homes and businesses, calculate power consumption, and provide a two-way communication with the utility provider. It is predicted that by 2016, the worldwide revenue of smart grids will surpass US$12 billion [21]. The large scale deployment of smart meters in the world, and the criticality of their operations make them an interesting testbed for us to evaluate our work. A smart meter receives power usage data through analog front end sensors. This component receives analog data (electric current), converts it to digital data and passes it to a microcontroller which calculates power consumption. Smart meters are equipped with flash memory to store both data and event logs on the meter. They are also equipped with network interfaces to communicate with the server and other devices. They communicate with the utility provider's server to receive commands and send consumption data.

We explain a subset of the operations on the meter as shown in Fig. 1. This figure is based on the abstract model of smart meter operations presented in our prior work [37]. The operations shown in Fig. 1 do not cover all the components of the meter such as command execution, timing synchronization, heartbeat communication, etc. However, they present an important execution path that we use as a running example in this section. As shown in Fig. 1, in the smart meter, raw data are received via sensors attached to the meter. The sensor controller calculates electricity consumption based on sensor data and, through a serial interface, passes them to a data management module to store them on the flash memory. Later, data is loaded from the flash memory and sent to the utility provider's server. The meter also periodically polls for input commands from the server to execute them.

B. Problem Formulation

In this section we formulate the problem of building an IDS as an optimization problem. Fig. 2 illustrates the formulation. We define the following terms in the formulation:

- We let Σ denote the set of all the system calls that may be executed by the software (on a given OS).
- The set $P = \{p_1, p_2, ..., p_m\}$ is the set of security properties of the system. These properties define the integrity and availability properties of the system that should be preserved so that the system is secure. Examples of such properties are *integrity of data storage/retrieval* and *availability of network communication*.
- τ_{p_i} indicates the set of invariants that must hold, so that property p_i is verified. Each of the invariants in τ_{p_i} is defined over system calls Σ. We explain the invariants in sec. IV-D1.
- The set $V = \cup_{i=1}^{m} \tau_{p_i} = \{v_1, v_2, ..., v_n\}$, denotes all invariants in the system that must hold so that the correctness of all $p_i \in P$ is verified.
- We define $\chi_{v_i} = \{p_j \in P | v_i \in \tau_{p_j}\}$. Intuitively, χ_{v_i} denotes the set of security properties for which, verification of v_i is a necessary condition.
- $D(V')$ indicates an IDS monitoring the set of invariants $V' \subseteq V$. The IDS is a state machine defined over Σ. We will explain how to build this state machine later.
- We denote the size of IDS $D(V')$ by $|D(V')|$ and define it as the number of the states in the IDS. We let M denote the upper bound for the size, determined by the memory constraints of the system.
- $U : 2^V \rightarrow R$ is a coverage function defined over the subset of invariants. Different subsets of invariants may have a different value depending on their role in verifying the security properties. Users may be interested in selecting invariants so that they verify a specific property, or selecting invariants in a way that they are evenly distributed among different properties. We assume that the coverage function is monotonic. Formally, we define monotonicity as the following: $\forall x, y \subset V \;\; x \subset y \Rightarrow U(x) < U(y)$. The monotonicity assumption is reasonable as we expect to have improved coverage when making the set of monitored properties larger.

Our goal is to build an IDS based on the subset of invariants $V' \in V$ so that the 1) the size of the IDS is smaller than or equal to the available memory M, and 2) $U(V')$ is maximized. Formally, we define our goal as the following:

- We find $D(V'), V' \subseteq V$ and $|D(V')| \leq M$ such that $\forall V'' \subseteq V, |D(V'')| \leq M \Rightarrow U(V'') \leq U(V')$.

In our formulation, the user can define their own coverage function U. Later in this section, we introduce two coverage functions and explain their rationale.

C. System Overview

The high level steps that we take to build the IDS are shown in Fig. 3. We briefly explain the steps below:

- Steps 1-2: The goal of the first two steps is to define the set of high level invariants that verify the security properties of the system. We call these *abstract* invariants. These invariants verify the *availability* and *integrity* properties of the system from the security perspective. We use Linear Temporal Logic (LTL) [12] to define these invariants. LTL is a logic to express temporal relations among the monitored entities at runtime. We extract abstract invariants from the Software Design Document (SDD) which presents the architecture, data, and functional specification of all the components of the system (see [20]). For instance, execution paths of the system, similar to what was shown in Fig. 1 are presented in the SDD. SDDs are often used as implementation guidelines and for defining gray box and functional test cases [3, 6]. We define abstract invariants once for an entire class of embedded devices, and if necessary, tailor them for different models of an embedded device.
- Steps 3-4: We convert the abstract invariants into *concrete* invariants. Concrete invariants capture the code-level behavior of the system. To achieve this, we substitute the propositional variables of abstract invariants, which are functions described in design document, with the code-level functions that implement them. Later, through static analysis, we substitute these functions with the patterns of system calls that are executed within the functions. Hence, in the concrete invariants, system calls of the code are used as the propositional variables.
- Step 5: After building the concrete invariants, we build a Buchi automaton [14], to monitor the concrete invariants. However, the state space can be large (exponential in the worst case) and exceed the available memory capacity of the device. Therefore, we select invariants that maximize the coverage (U) of security properties of the system. Many definitions for security coverage have been suggested [1, 2]. Therefore, we designed our technique so that it is not bound to a single definition for the coverage, and we enable the user to define their own coverage function. We also propose two approaches for defining the coverage and build IDSes based on them. Our IDS checks the system calls' execution at run-time with a Buchi automaton. If the behavior of the system deviates from the set of monitored invariants, it reports it as an intrusion that has been detected.

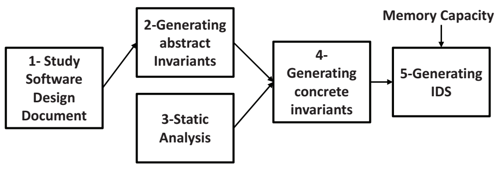

1- Study Software Design Document
2-Generating abstract Invariants
3-Static Analysis
4- Generating concrete invariants
5-Generating IDS
Memory Capacity

Fig. 3. The overview of the steps in building the IDS.

D. Details

In this section, we provide the details on how to build the invariants for security, and convert them into an IDS.

1) Invariants: A system is secure, if it achieves and sustains its mission according to a security policy. We model the security policy of the system by defining the set of invariants V that must be preserved during run time. These invariants verify the fundamental security properties of *availability* and *Integrity* [33] of the system. For example, integrity of consumption calculation, integrity of communication to server, availability of data storage/retrieval, and availability of server communication are security properties of a smart meter. Breaking any of these properties may result in loss of data and/or incorrect billing. Therefore, the user defines the invariants (set V) that verify correctness of all the properties $p_i \in P$.

We use Linear Temporal Logic (LTL) to define the invariants V, verifying properties P. Temporal logic is a formalism for describing properties of a system over time. LTL describes the properties of runs of a system, which makes it suitable for verifying the status of a system during run time. LTL introduces the operators $\bigcirc$(next), $\Box$ (globally or always), $\Diamond$ (eventually or in the future), and U (until) [44]. We use these operators to define the invariants. We define the invariants in two levels. First, we define them at a high level and call them *abstract* invariants. Then, we convert them to detailed, implementation specific invariants and call them *concrete* invariants.

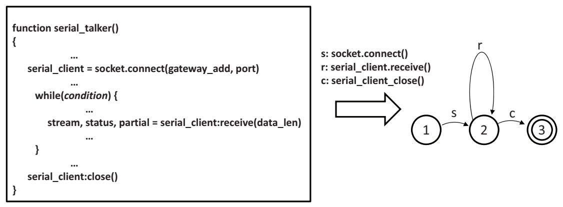
2) Abstract invariants: Abstract invariants capture the availability and integrity requirements of the system according to the software design document (SDD) [20]. This document presents the architecture, functional specification, and control flow of all the components of the system. We consider all the execution paths (from sequence diagrams for instance) that form a security property $p_i \in P$, and specify the invariants that should hold so that the path is correctly executed. For instance, the operations depicted in Fig. 1 presents a portion of the procedural flow of the smart meter in its SDD. The electricity consumption data is received via sensors, passed through serial communication interface and then stored in the flash memory for backup purposes. Later the data is retrieved from the flash memory and sent to the utility server. This operation may be compromised at different stages by an attacker making the functions or the resources unavailable (e.g., killing the running process, changing the execution path through corrupt input). Therefore we need invariants to make sure that these operations are not compromised. Using LTL, we formulate the invariants that address the availability and integrity properties of the system as shown in Table I.

3) Concrete invariants: In the next step, we convert the abstract invariants into *concrete invariants*. We substitute the functional descriptions in the abstract invariants with the code-level functions. Later, using static analysis, we substitute these functions with the automaton representing the system-call-based regular expressions that are executed within those functions.

For example, Fig. 4 shows parts of *serial_talker* function used in our smart meter testbed. This function is written in the Lua language on our smart meter platform, and implements receive_consumption_data part of Fig. 1. As the name suggests, *serial_talker* listens to a specific port and through a serial interface, receives consumption data sent from the sensor controller. The automaton resulting from static analysis of this function is shown in Fig. 4 and is described as (sr^*c). Although LTL does not include the star operation (as in regular languages), we can add these automaton connectives to it [44], without changing its semantics.

Invariant description	Invariant formulation	Targeted property
v_1: Consumption data received from serial interface must eventually be stored in the flash memory	$\Box(receive\_consumption\_data \Rightarrow \Diamond store\_data\_into\_flash)$	p_1: Availability of storage/retrieval
v_2: Data stored in the flash memory must eventually be retrieved	$\Box(store\_data\_into\_flash \Rightarrow \Diamond retrieve\_data\_from\_flash)$	p_2: Integrity of storage/retrieval p_1: Availability of storage/retrieval
v_3: Data retrieved from flash memory must eventually be sent to the server	$\Box(retrieve\_data\_from\_flash \Rightarrow \Diamond communicate\_with\_server)$	p_3: Integrity of network communication p_4: Availability of network communication
v_4: Storing data on the flash memory must be after data is received from serial interface	$\neg(\neg store\_data\_into\_flash U (\neg store\_data\_into\_flash \wedge \neg receive\_consumption\_data))$	p_2: Integrity of storage/retrieval
v_5: Retrieving data from flash memory must be after data is stored on the flash memory	$\neg(\neg retrieve\_data\_from\_flash U (\neg retrieve\_data\_from\_flash \wedge \neg store\_data\_into\_flash))$	p_2: Integrity of storage/retrieval
v_6: Sending data to the utility server must be after the data is retrieved from the flash memory	$\neg(\neg communicate\_with\_server U (\neg communicate\_with\_server \wedge \neg retrieve\_data\_from\_flash))$	p_3: Integrity of network communication

TABLE I
ABSTRACT INVARIANTS FOR AN EXECUTION PATH OF SMART METER.



```
function serial_talker()
{
            ...
    serial_client = socket.connect(gateway_add, port)
            ...
      while(condition) {
                ...
          stream, status, partial = serial_client:receive(data_len)
                ...
      }
            ...
    serial_client:close()
}
```

Fig. 4. We convert functions into automaton connectives through static analysis and then into concrete invariants. The labels on state transitions correspond to the system calls executed in the code.

E. Building the model

Having all the concrete invariants, the next step is to build a state machine that verifies the invariants. Given that the invariants are presented as LTL formulas, we build a Labeled Generalized Buchi automaton $\Im = \{Q, \Sigma, \delta, S, F\}$ that has an accepting language equivalent of the invariants [14]. In this formulation, Q is the set of states of the automaton, Σ is the alphabet of the input for the automaton, $\delta : Q \rightarrow 2^Q$ is the transition function, S is the initial state and F is the set of designated states. This automaton will be our IDS $D(V')$, defined in Sec. IV-B. $V' \subseteq V$ indicates what subset of invariants we will use in building the automaton. A word of infinite length is accepted by the automaton $\Im$ if it has a run that visits a designated state infinitely often. The details of building a generalized Buchi automaton from an LTL formula are provided in Gastin et al. [14]. Each state of $\Im$ contains

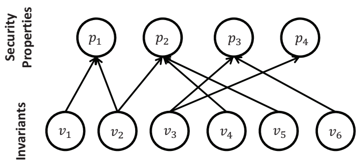


Fig. 5. The invariants v_1 to v_6 verify the properties p_1 to p_4.

labels indicating what sub-formulas of I are satisfied in that state.

The state space of the state machine monitoring all the invariants may be exponential in the worst case. Considering the limited available memory, we may not be able to build a state machine that includes all the invariants. Therefore, given the upper bound M for the size of the IDS, we select the invariants that maximize the coverage function U, as stated in sec. IV-B. The role of the coverage function is to *quantify* the coverage provided by the selected invariants. Our solution is structured in a way that the user can use their own coverage function in the formulation. Here, we define two metrics and build two coverage functions based on them.

We base our IDSes on the metrics for security coverage suggested by the Computer Security Devision of National Institute of Standard and Technology (NIST) [1], and Computer Emergency Response Team (CERT)'s guide on measuring software security in its annul research report [2]. We call these IDSes *MaxMinCoverage* IDS and *MaxProperty* IDS. We explain them below.

1) MaxMinCoverage IDS: In this approach, our goal is to maximize the minimum *property coverage* provided for each of the security properties of the system. We calculate *property coverage* for each of the individual properties $p_i \in P$ based on the portion of invariants verifying p_i that are included in the set of selected invariants to build the IDS. We provide the formal definition of *property coverage* and the coverage function of the IDS for MaxMinCoverage IDS, and then explain it with an example.

For IDS $D(V')$, the *property coverage* of property $p_i \in P$, is defined as the percentage of the invariants in V verifying p_i, that are included in V' (and hence, included in the IDS $D(V')$). Note that to fully cover a property, all invariants that are mapped to it need to be checked. Based on this, we denote *property coverage* of p_i with respect to IDS $D(V')$ by $\pi_{D(V')}(p_i)$ (expressed as percentage):

$$\pi_{D(V')}(p_i) = \frac{|\tau_{p_i} \cap V'|}{|\tau_{p_i}|} \times 100;$$

The coverage function U is defined as the smallest *property coverage* provided for any of the properties $p_i \in P$. Formally, we denote the coverage function of IDS $D(V')$ as:

$$U(V') = min\{\pi_{D(V')}(p_i) | p_i \in P\}.$$

We explain the above formulation with an example. In Fig. 5, we show six invariants v_1 to v_6, defined in Table I, and the corresponding security properties for the smart meter. As shown in Fig. 5, invariants $\{v_1, v_2\}$ verify availability of storage/retrieval (p_1). Invariants $\{v_2, v_4, v_5\}$ verify integrity of storage/retrieval (p_2). Invariants $\{v_3, v_6\}$ verify integrity of server communication (p_3). Finally, invariant v_3 verifies availability of server communication (p_4). Formally, we have $\tau_{p_1} = \{v_1, v_2\}$, $\tau_{p_2} = \{v_2, v_4, v_5\}$, $\tau_{p_3} = \{v_3, v_6\}$, and $\tau_{p_4} = \{v_3\}$.

In this example, we assume that we have the memory capacity to monitor no more than four invariants. The size of the generated IDS will depend on the invariants' structure, but in this example, we ignore this for simplicity. Selecting the invariant set $\{v_1, v_2, v_3, v_4\}$ will result in *property coverage* of 100% for p_1, 66% for p_2, 50% for p_3 and 100% for p_4. Hence, the coverage provided by the IDS (as determined by the smallest *property coverage*) will be 50%. We can easily verify that this is the best coverage we can obtain in this example.

The benefit of the coverage function used in this IDS is that it distributes the coverage among all the security properties as opposed to monitoring only some of the properties and leaving the others completely vulnerable. This is important if the user does not have any other resources to strengthen the security of the unmonitored properties.

2) MaxProperty IDS: In this IDS, our goal is to maximize the number of the properties that are completely verified by the IDS, as opposed to the MaxMinCoverage IDS that tries to distribute the coverage among *all* the properties.

We use the same definition for *property coverage* as we used for MaxMinCoverage IDS. We denote *property coverage* of p_i with respect to IDS $D(V')$ by $\pi_{D(V')}(p_i)$:

$$\pi_{D(V')}(p_i) = \frac{|\tau_{p_i} \cap V'|}{|\tau_{p_i}|} \times 100$$

Based on this, the coverage function U is the number of properties with the *property coverage* of 100%:

$$U(V') = |\{p_i \in P | \pi_{D(V')}(p_i) = 100\}|.$$

We consider the example from previous section. Again, for simplicity in this example, we assume that we have the memory capacity to monitor no more than four invariants. In Fig. 5, selecting all the invariants $\{v_1, v_2, v_3, v_4, v_5, v_6\}$ will result in *property coverage* of 100% for all the properties p_1, p_2, p_3, and p_4. By looking at Fig. 5, we notice that by selecting invariants $\{v_1, v_2, v_3, v_6\}$ we can have *property coverage* of 100% for properties p_1, p_3, and p_4. We can see that there are no other four invariants that can achieve *property coverage* of 100% for three properties. Based on this, if we only have capacity for monitoring four invariants, the set $\{v_1, v_2, v_3, v_6\}$ gives us the best result in terms of the number of properties.

The benefit of the coverage function used in this IDS is that knowing that some properties are 100% covered, the user may be able to apply other techniques (e.g., code hardening) on the uncovered components of the code.

3) Building the IDS: Fig. 6 shows the process of selecting the invariants to build the Buchi Automaton. To build MaxMinCoverage IDS or MaxProperty IDS we need to pass their corresponding coverage function U (defined in Sec. IV-E1 and Sec. IV-E2) to the algorithm. In general, the

Building the IDS

```
1.  // Input: V[] Invariants, P[] Security properties
1.  //        M IDS size constraint, U The coverage function
1.  // Output: A the IDS automaton
1.  invariantSet = {}
1.  bestUtility = 0
1.  Graph g[][]
1.  bool mark[]
2.  for i = 1 to size of V
3.    for j = 1 to size of P
4.      if (V[i] $\in \tau_{p[j]}$)
5.        g[i][j] = 1
6.  for all S[i] $\subset$ V[i], in the ascending order {
7.    if for any j, V[i] $\subset$ V[j] and mark[j] == true
8.      continue
9.    A, mem = calculateBuchi(S[i])
10.   if (mem $\leq$ M)
11.     mark[i] = true
12.     Graph h[][]
13.     for all pairs of j, k V[j] $\in$ S[i] and $p_k \in \chi_{V[j]}$
14.       h[j][k] = 1
15.     if (bestUtility < U(g,h)) {
16.       bestUtility = U(g,h)
17.       stateMachine = A
18.     }
19. }
20. return stateMachine
```

Fig. 6. Algorithm for building the IDS.

user may define their own coverage function U and use it as an input to the algorithm.

In lines 2, 3, and 4 of the algorithm, we build a bipartite graph g. One part of the graph are the nodes representing the security properties, and the other part of the graph are the nodes representing the invariants that verify those properties. $g[i][j] = 1$ if and only if $V[i] \in \tau_{p[j]}$. In line 6, we go through the subsets (represented by array s[]) of the invariants V[] in descending order of their size (the number of invariants they include). In line 7, we check if we have already calculated an automaton within the constraint M, using a superset of $S[i]$. If so, we do not process $S[i]$ as it will not give us any better results. This is because we assume that the coverage function is monotonic (see Sec. IV-B). In line 9, we pass the set of currently selected invariants ($S[i]$) to the function *calculateBuchi*. This function calculates the Buchi automaton for the set of invariants we pass it. The details of building a Buchi automaton is in [14] and we do not discuss it here. If the size of Buchi automaton does not exceed M (line 10), we mark $S[i]$ (line 11) to make sure that we do not process any of its subsets in the future, as their result will not be better than $S[i]$ (due to the monotonicity of the coverage function). Then, we build the vertex-induced subgraph h of graph g (lines 13, and 14). Subgraph h represents the invariants and properties monitored by the Buchi automaton. We pass both the original graph g and the subgraph h to the coverage function U in line 15. By comparing the original graph and the subgraph, function U calculates the coverage. We compare the result with the best coverage we have calculated so far. Whenever we achieve a better coverage within the memory constraint, we update the Buchi automaton (line 17). In the end, we return the Buchi automaton that provides the best coverage whose size does not exceed M (line 20).

The worst case time complexity of this algorithm is exponential, with respect to the number of the invariants and the sizes of the invariants. The reason is that, in the worst case, we have to go through all the subsets of invariants and build a Buchi automaton based on them (which may be exponential with respect to the size of the invariants). However, to generate the IDS, the algorithm is executed offline and only once. In our experiments, the size of the invariants is around 30. Therefore, the algorithm is not a significant performance bottleneck.

V. Case study

In this section, we discuss the results of running MaxMin-Coverage IDS and MaxProperty IDS on a smart meter platform and address the following research questions:

- **RQ1 (Detection):** What portion of the attacks against SEGMeter can be detected by MaxMin IDS and Max-Property IDS?
- **RQ2 (Relationship between estimated coverage and attack detection):** How close is the attack detection rate of MaxMinCoverage IDS and MaxProperty IDS at runtime, to their estimated coverage at design time?
- **RQ3 (Effect of increasing memory constraints on the IDS):** How does increasing memory constraints affect the detection rate of MaxMinCoverage IDS and MaxProperty IDS in terms of their compliance with their estimated coverage at design time?
- **RQ4 (Performance overhead):** What is the performance overhead of MaxMinCoverage IDS and MaxProperty IDS on the system?

A. Experimental setup

We use SEGMeter, an open source smart meter as a test bed to evaluate our technique for building IDS for embedded devices [36] (Fig. 7). SEGMeter has 6 channels of energy measurement temperature sensing and a reed relay control. It also has the capability to support Zigbee for wireless communication. The meter consists of two boards that communicate via a serial interface. The sensors are installed on a board with ATMEGA32x series microcontroller which calculates data consumption. The communication interfaces are installed on the Arduino board [5] with Broadcom BCM3302 V2.9 240MHz CPU and 16 MB RAM. It has LAN and WiFi network interfaces, and communicates with the utility server. It also runs OpenWrt, a Linux distribution for embedded devices, which is optimized in terms of size and performance. A portion of the meter software runs on the Arduino board, and the rest of it runs on the microcontroller board.

Our IDSes run on the Arduino board. As we are using system calls as propositional variables in our model, we use

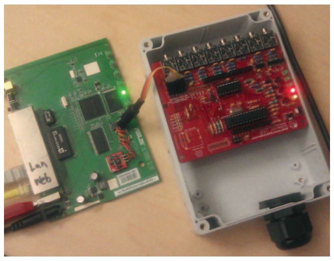


Fig. 7. SEGMeter: our open source meter testbed. The board on the left is the gateway board in charge of communicating with the server, and the board on the right is the Arduino board that receives consumption data from the sensors.

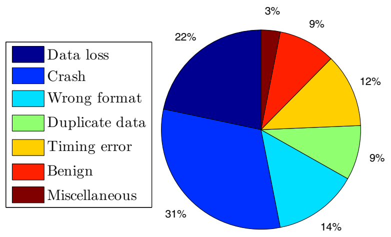


Fig. 8. We have categorized the fault injection results into 7 categories. The statistics of occurrences of each of these categories are presented in this figure.

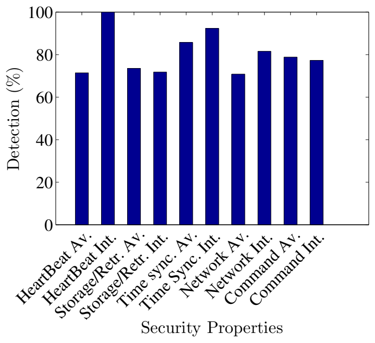


Fig. 9. The detection results of MaxMinCoverage IDS built using **4MB** of memory, for attacks against different security properties of SEGMeter.

strace to create the run-time trace of the software. At the boot-up, strace attaches to the process we are monitoring, intercepts the system calls and logs them, along with the time and argument information. The second component of our IDS runs in periods of 10 seconds, processes the log file and compares it to the model of the system. Later in Sec. V-C we explain why the 10 second time interval is chosen.

B. Experiments

An attacker may tamper with the meter by exploiting the potential vulnerabilities in the system. A successful attack results in changing the operational flow of the software by either executing unintended operations (e.g. deleting data), or by skipping legitimate operations (e.g. storing data).

We use fault injection to simulate the attacks that result in changing the normal behavior of the smart meter. The reason for using fault injection is that we do not have access to any database of known attacks against the smart meter software. Also, smart meters are relatively new platforms and we are interested in evaluating them against *unknown* attacks as well. Other papers have adopted similar strategies to study effects of attacks. For example, Tseng et. al. [39] have used fault injection to mimic the impact of malicious attacks against power grid embedded devices such as Real Time Automation Control (RTAC).

We perform the fault injection at the code level using code mutation. Each mutation involves flipping a single branch condition. This way, we simulate the effect of a large number of attacks by changing the operational flow of the software. This process may not result in skipping/executing the *exact* operations that the attacker intends to. However, we are interested in evaluating our IDSes in detecting *any* changes in the execution flow of the software both due to known and unknown attacks. As we are using fault injection for simulating attacks, we use the term attacks, in place of faults, in the rest of this section.

We created 226 mutations of the code (for 226 branches). These mutations result in breaking the security properties of the meter (unless they are benign). Based on our observations, the effect of the attacks can be categorized into 7 groups, as shown in Table III. The statistics of occurrences of each of the categories are presented in Fig. 8. Crashing is the most common effect of our attacks. The reason is that in many cases, changing the execution path of the SEGMeter software results in operating on *NULL* data, which results in a crash. Also, we list the availability and integrity properties of SEGMeter in Table II. These properties are affected as a result of attacks.

C. Results

SEGMeter is equipped with 16MB of RAM, and about 12MB is used by the OS and the meter software. This leaves 4MB for the IDS. We derived 34 invariants verifying 10 security properties of the smart meter (these security properties are shown in Table II). Building an IDS for monitoring all the invariants results in memory requirement of over 7MB which is not feasible for our smart meter platform. Therefore, in the first step, we build MaxMinCoverage IDS and MaxProperty IDS using 4MB of memory and evaluate them. We had to limit the number of the generated states of the IDS to less than 55000 states to meet the 4MB memory requirement. We note that given the number of states, memory requirements may vary depending on the programming language used. However, the memory used is directly correlated with the size of the state machine. Using our test machine (3.4GHz Intel CPU and 8GB of RAM), we were able to build the IDSes in less than 6 hours in total. We note that this process is offline and done once for each embedded device. Hence, it is not a significant

Property	Description
Heartbeat Availability	Generation of heartbeat messages at specific time intervals.
Heartbeat Integrity	Correctness of generation of heartbeat messages in terms of length, content, etc.
Storage/Retrieval Availability	Availability of data storage and retrieval modules of SEGMeter, when requested.
Storage/Retrieval Integrity	Correctness of storage and retrieval operations to guarantee integrity of data.
Time sync. Availability	Ability of SEGMeter to synchronize itself with the server timer, whenever required.
Time sync. Integrity	Correctness of synchronization results.
Network Availability	Availability of network modules (e.g., when sending/receiving data) upon request.
Network Integrity	Correctness of networking operations, to guarantee the communication integrity.
Command Processing Availability	Ability of SEGMeter to process server commands, upon receiving them.
Command Processing Integrity	Correctness of operations when processing and executing commands.

TABLE II
AVAILABILITY AND INTEGRITY PROPERTIES OF SEGMETER.

Fault injection results	Description
Data loss	Losing data before it is completely processed, e.g., clearing flash memory data before sending it to the server.
Crash	Terminating software execution due to exceptions, e.g., operating on *nil* data in the Lua language.
Wrong formatting	Changing execution path of the software resulting in incorrectly formatted consumption data.
Duplicate data	Skipping certain checks (e.g., whether data is stored) resulting in redundant operations.
Timing error	Failing to execute certain operations (e.g., connectivity check) within a predetermined time frame.
Benign	Causing no harm when changing the execution flow of the system, e.g., failing to print debug messages.
Miscellaneous	Other faults, e.g., random cases of failing to send heartbeat messages.

TABLE III
EFFECTS OF FAULT INJECTION ON THE SMART METER SOFTWARE.

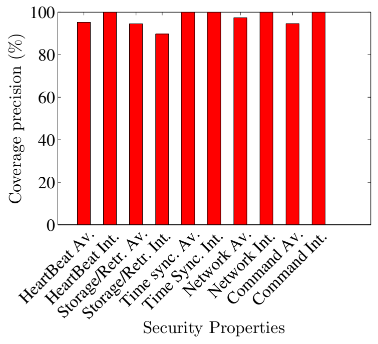


Fig. 10. The accuracy of MaxMinCoverage IDS built using **4MB** of memory for different security properties of SEGMeter.

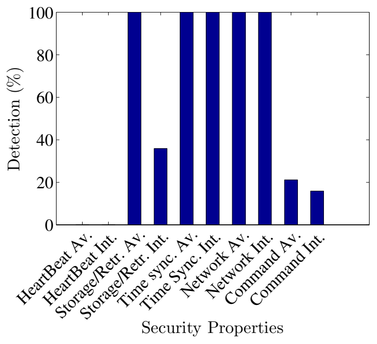


Fig. 11. The detection results of MaxProperty IDS built using **4MB** of memory, for attacks against different security properties of SEGMeter.

performance bottleneck.

RQ1(Detection): We evaluate the attack detection results of MaxMinCoverage IDS and MaxProperty IDS. We run our IDSes against the attacks explained in Sec. V-B. We evaluate the *detection* as below:

$$detection(\%) = \frac{number\ of\ detected\ attacks}{total\ number\ of\ injected\ attacks} \times 100$$

To better understand the results of our IDSes, we present their detection separately for each individual security property. Detection results of MaxMinCoverage IDS is presented in Fig. 9. As explained in Sec. IV-E1, the goal of MaxMinCoverage IDS is to maximize the minimum *property coverage* among all the security properties. Therefore, in practice we expect that the smallest detection for a security property to be close to the average detection among all the security properties. Our results verify this expectation. Fig. 9 shows that the smallest detection rate for a security property is 70%, while the average detection among all security properties is 79.5% (i.e., there is only 9.5% difference between the average and minimum detection).

The detection results for MaxProperty IDS are presented in Fig. 11. For MaxProperty IDS (see Sec. IV-E2), the number of the security properties that are completely covered (with *property coverage* of 100%) are maximized. We estimated MaxProperty IDS to provide *property coverage* of 100% for 5 security properties of the system. Our detection results in Fig. 11 verify this. The detection rate of MaxProperty IDS for

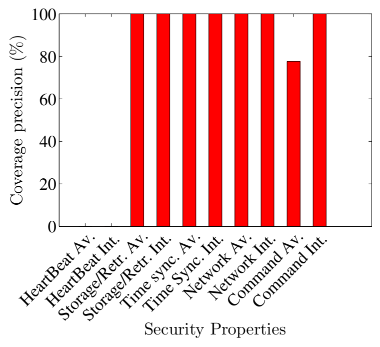


Fig. 12. The accuracy of MaxMinCoverage IDS built using **4MB** of memory for different security properties of SEGMeter.

the other security properties ranges from 0% to 35%.

RQ2 (Relationship between estimated coverage and attack detection): We evaluate the compliance of detection results of MaxMinCoverage IDS and MaxProperty IDS at runtime, with their estimated coverage at the design time. We define accuracy of our IDS for each security property of smart meter using the formula below:

$$accuracy(\%) = max\left(\frac{detection}{property\ coverage}, 1\right) \times 100$$

Accuracy indicates the ratio of detection rate of the IDS to the estimated coverage of the IDS. Higher accuracy indicates better compliance of the detection results of the IDS with its estimated coverage. In the above formula, *property coverage* indicates the estimated coverage provided by MaxMinCoverage IDS or MaxProperty IDS, for each of the security properties in the system (Sec. IV-E1 and Sec. IV-E2). The detection results of the IDS may be higher than its estimated coverage at design time (due to better performance than what we estimated) and hence, we have the term *max(detection / property coverage, 1)* to have the 100% accuracy as maximum. In the next paragraph we explain why the coverage at runtime may be even higher than the estimated coverage.

We study the accuracy of MaxMinCoverage IDS for each security property in Fig. 10. We observe that the smallest accuracy is 89%, and the average accuracy among all security properties is 97%. This indicates that detection of MaxMinCoverage IDS at runtime strongly complies with its estimated coverage at the design time. In practice, we observed that in many cases the detection rate is even higher than the estimated coverage (leading to the accuracy of 100%). The reason is that attacks may propagate in the system. Therefore, even if an attack breaking a security property is not initially detected, it may affect other security properties and be detected later. In other words, monitoring a security property helps with detecting attacks on other security properties as well.

We have presented the accuracy of MaxProperty IDS for all the security properties of the system in Fig. 12. Our results show that only one security property (command processing availability) has accuracy of less than 100% (which is 77%). This indicates that the detection results for MaxProperty IDS complies with its estimated coverage at design time. The 77% accuracy for one property (availability of command processing) is because the invariants verifying this property are larger in size (due to the complexity of the code) and lead to higher number of states compared to other invariants. Therefore, memory constraints result in lower coverage for this property. This leads to lower accuracy.

RQ3(Increasing memory constraints): To evaluate MaxMinCoverage IDS and MaxProperty IDS under memory constraints, we build these IDSes using 2MB of memory (severe memory constraint), and 1MB of memory (extreme memory constraint). To address these constraints we limit the number of states in our IDSes to less than 22000 states and 13000 states respectively. The detection results and accuracy for MaxMinCoverage IDS using 2MB of memory are shown in Fig. 14 and Fig. 15. These results show that severe memory constraints reduce the detection rate of MaxMinCoverage IDS to an average of 44.3%, which is expected due to the small available memory. However, even under severe memory constraints, the estimated coverage of MaxMinCoverage IDS is close to its runtime detection, resulting in an average accuracy of 92%. The detection results and accuracy of MaxProperty IDS using 2MB of memory are shown in Figures 16 and 17. These results show that, under severe memory constraints, MaxProperty IDS successfully provides 100% detection for three security properties of the system, and provides high average accuracy of 93%. This indicates that even under severe memory constraints, the detection results of MaxProperty IDS complies with its estimated coverage.

We further reduced the available memory to 1MB, which is an extreme memory constraint. In this case, as the number of invariants covered in our IDSes is very small, the detection rate and the accuracy are significantly reduced. MaxMinCoverage IDS provides an average detection rate of 19.8%, and MaxProperty IDS can provide 100% detection for only one security property. This is not surprising as 1MB is less than 15% of memory required to build an IDS monitoring the complete set of invariants of the system.

RQ4(Performance overhead): Our IDSes read and processes the system call log created by *strace* every T seconds. Smaller T results in faster detection of attacks (as the IDS is run more often), and smaller log files. We study the performance overhead of our IDSes, for different values of T, in Fig. 13. For $T = 10$ seconds, both MaxMinCoverage IDS and MaxProperty IDS (built using 4MB and 2MB of memory), incur less than 7% performance overhead on the system (we do not consider the overhead of IDSes that operate under extreme memory constraint of 1MB, as it results in low coverage). We observed that making the intervals too small (<1 second) results in high read/write operations, and higher performance overhead. Also, increasing the time interval beyond 10 seconds does not result in a significant performance gain. Based on this, we picked 10 seconds as the time interval for our IDSes. This keeps the size of the log file small (as the size of the flash memory is limited), results in fast detection of the attacks, and incurs low performance overhead on the system.

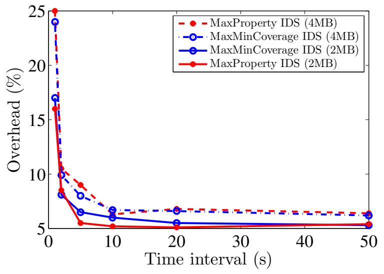


Fig. 13. Performance overhead of MaxMinCoverage IDS and MaxProperty IDS using different time intervals. Making the time interval too small will increase the performance overhead.

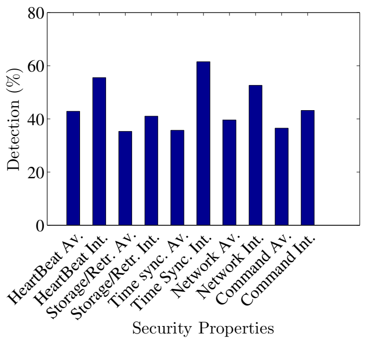


Fig. 14. The detection results of MaxMinCoverage IDS built using **2MB** of memory, for attacks against security properties of SEGMeter.

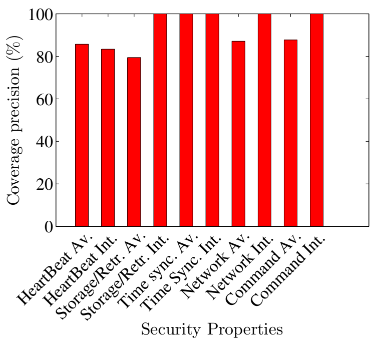


Fig. 15. The accuracy of MaxMinCoverage IDS built using **2MB** of memory, for security properties of SEGMeter.

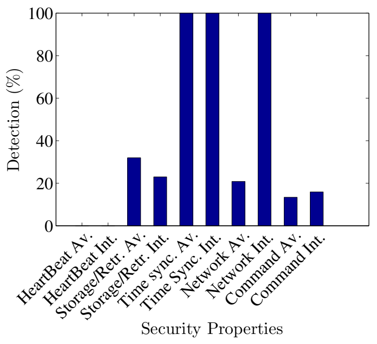


Fig. 16. The detection results of MaxProperty IDS built using **2MB** of memory, for different security properties of SEGMeter.

VI. Discussion

Generalizability: Using our technique, the user may define their own coverage function and use it to build an IDS. The security of the system is dependent on the validity of its security properties and invariants. Our formulation allows any monotonic coverage function defined over the graph of security properties P, and the set of invariants V, defined over single and linear executions, that verify those security properties. Examples of coverage functions other than the ones presented in this paper would be *number of selected invariants*, and *average number of invariants selected per security property*.

In our work, we are defining the invariants of the system using LTL. There are several variants of LTL with different expressive power and approaches for verification of the formulas [27, 41, 11, 47]. This work can be extended to use variants of LTL to benefit from their advantages.

Extreme memory constraints: Our results show that having extreme memory constraints may result in significant reduction in coverage of some of the security properties of the system. In these circumstances, our approach can still be useful. Providing quantifiable coverage enables the user to identify the components of the system that are not fully monitored and take further actions to address the limitations. For instance, they may apply code hardening techniques or require stricter code reviewing policies for certain components of system. Also, they may upgrade the available memory within the budget, to increase coverage.

Limitations: In this work, we are assuming all the invariants have the same priority. However, in practice, some properties of the system may be more critical than the others. Incorporating priorities of the security properties of the system is a direction for future work.

Also, we assume that all the attacks (simulated as faults) have the same likelihood of occurrence. Therefore, we treat them equally when calculating detection rate and coverage. In practice, some of the attacks may be more likely to occur, thereby, modifying our coverage results.

VII. Conclusion

Memory constraints of embedded systems make building IDSes for them challenging. In this paper, we formulated the problem of building IDS for embedded systems as an optimization problem. Given the memory constraints of the device, the set of security properties and the invariants of the system verifying those properties, we build an IDS within the memory constraints of the device that maximizes the coverage for the security properties. Our technique is flexible in terms of accommodating user-defined coverage functions. We also defined two coverage functions and built two IDSes based

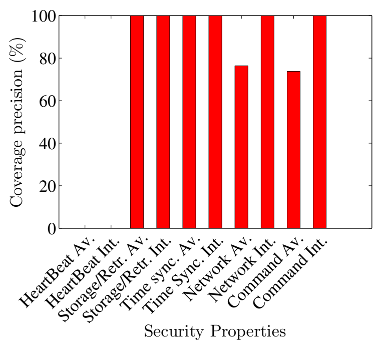


Fig. 17. The accuracy of MaxProperty IDS using **2MB** of memory, for different security properties of SEGMeter.

on them: MaxMinCoverage IDS and MaxProperty IDS. We showed that using 4MB of memory, MaxMinCoverage IDS is able to detect 80% of attacks against the security properties of the system. Also, MaxProperty IDS is able to provide 100% detection rate for 5 (out of 10) of the security properties of the system. Furthermore, our IDSes incur low performance overhead of less than 7% on the system, and the detection results of our IDSes comply with their estimated coverage at design time, even under severe memory constraints.

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

References

[1] NIST Special Publication 800-55 Revision 1, 2008.

[2] CERT research anual report, 2009.

[3] Pandya V. Acharya S. *International Journal of Electronics and Computer Science Engineering (IJECSE)*, 2013.

[4] David Basin, Felix Klaedtke, and Samuel Mller. Policy monitoring in first-order temporal logic. In *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin Heidelberg, 2010.

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

[6] John E. Bentley, Wachovia Bank, and Charlotte NC. Software testing fundamentals-concepts, roles, and terminology. Technical report, SAS Institute, 01 2008.

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

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

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

[10] Etienne Closse, Michel Poize, Jacques Pulou, Joseph Sifakis, Patrick Venter, Daniel Weil, and Sergio Yovine. Taxys: A tool for the development and verification of real-time embedded systems? In *Computer Aided Verification*, pages 391–395. Springer, 2001.

[11] Jean-Michel Couvreur. On-the-fly verification of linear temporal logic. In *FM99 Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 253–271. Springer Berlin Heidelberg, 1999.

[12] E Allen Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, 995:1072, 1990.

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

[14] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In *Proceedings of CAV'01*, volume 2102 of *Lecture Notes in Computer Science*, Paris, France, July 2001. Springer.

[15] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Efficient context-sensitive intrusion detection. In *Proceedings of the 11th NDSS Symposium*, 2004.

[16] Hendra Gunadi and Alwen Tiu. Efficient runtime monitoring with metric temporal logic: A case study in the android operating system. In *FM 2014: Formal Methods*, volume 8442 of *Lecture Notes in Computer Science*. Springer International Publishing, 2014.

[17] Xuan Dau Hoang, Jiankun Hu, and Peter Bertok. A multi-layer model for anomaly intrusion detection using program sequences of system calls. In *Proc. 11th IEEE Intl. Conference on Networks*, 2003.

[18] Jiankun Hu, Xinghuo Yu, D. Qiu, and Hsiao-Hwa Chen. A simple and efficient hidden markov model scheme for host- based anomaly intrusion detection. *Netwrk. Mag. of Global Internetwkg.*, 23(1):42–47, January 2009.

[19] Wenjie Hu. Robust support vector machines for anomaly detection. In *In Proc. ICMLA03*, 2003.

[20] 1016-1998-IEEE recommended practice for software design descriptions. http://standards.ieee.org/findstds/standard/1016-1998.html.

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

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

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

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

[25] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental security analysis of a modern automobile. SP '10, pages 447–462, Washington, DC, USA, 2010. IEEE Computer Society.

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

[27] Martin Leucker and Csar Snchez. Regular linear temporal logic. In *Theoretical Aspects of Computing ICTAC 2007*, volume 4711 of *Lecture Notes in Computer Science*, pages 291–305. Springer Berlin Heidelberg, 2007.

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

[29] FabrizioMaria Maggi, Marco Montali, Michael Westergaard, and WilM.P. van der Aalst. Monitoring business constraints with linear temporal logic: An approach based on colored automata. In *Business Process Management*, volume 6896 of *Lecture Notes in Computer Science*, pages 132–147. Springer Berlin Heidelberg, 2011.

[30] FabrizioMaria Maggi, Michael Westergaard, Marco Montali, and WilM.P. van der Aalst. Runtime verification of LTL-based declarative process models. In *Runtime Verification*, volume 7186 of *Lecture Notes in Computer Science*, pages 131–146. Springer Berlin Heidelberg, 2012.

[31] Sibin Mohan, Jaesik Choi, Man-Ki Yoon, Lui Sha, and Jung-Eun Kim. SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems. In *RTAS*, Washington, DC, USA, 2013. IEEE Computer Society.

[32] Mehdi Moradi and Mohammad Zulkernine. A neural network based system for intrusion detection and classification of attacks. In *AISTA*, 2004.

[33] M. McEvilley R. Ross, J.C. Oren. Systems security engineering: An integrated approach to building trustworthy resilient systems. NIST Special Publication 800-160, May 2014.

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

[35] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. SOSP '05, pages 1–16, New York, NY, USA, 2005. ACM.

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

[37] Farid Molazem Tabrizi and Karthik Pattabiraman. A model-based intrusion detection system for smart meters. In *HASE*, pages 17–24. IEEE, 2014.

[38] The smart meter revolution. https://m2m.telefonica.comy.

[39] Kuan-Yu Tseng, Daniel Chen, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Characterization of the error resiliency of power grid substation devices. In *DSN*, pages 1–8. IEEE, 2012.

[40] Ahmed U and Masood A. Host based intrusion detection using RBF neural networks. In *In Proc. International Conference on Emerging Technologies.*, 2009.

[41] MosheY. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer Berlin Heidelberg, 1996.

[42] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings S&P'01*, USA, 2001. IEEE Computer Society.

[43] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE S&P'99*. IEEE Computer Society, 1999.

[44] Pierre Wolper. Temporal logic can be more expressive. *Information and control*, 56(1):72–99, 1983.

[45] Nong Ye, Syed Masum Emran, Qiang Chen, and Sean Vilbert. Multivariate statistical analysis of audit trails for host-based intrusion detection. *IEEE Trans. Comput.*, 51(7):810–820, July 2002.

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

[47] Duo Zhang and Shi Gong Long. A theoretic approach to translation of linear temporal logic into an automaton. In *ICNC*, pages 1111–1115. IEEE, 2014.