ARTINALI: Dynamic Invariant Detection for Cyber-Physical System Security

Maryam Raiyat Aliabadi
University of British Columbia
Department of Electrical and Computer Engineering
Vancouver, BC, Canada
raiyat@ece.ubc.ca

Amita Ajith Kamath*
National Institute of Technology Karnataka
Departement of Computer Science and Engineering
Mangalore, India
amita.a.kamath@ieee.org

Julien Gascon-Samson
University of British Columbia
Department of Electrical and Computer Engineering
Vancouver, BC, Canada
julien.gascon-samson@ece.ubc.ca

Karthik Pattabiraman
University of British Columbia
Department of Electrical and Computer Engineering
Vancouver, BC, Canada
karthikp@ece.ubc.ca

ABSTRACT

Cyber-Physical Systems (CPSes) are being widely deployed in security-critical scenarios such as smart homes and medical devices. Unfortunately, the connectedness of these systems and their relative lack of security measures makes them ripe targets for attacks. Specification-based Intrusion Detection Systems (IDS) have been shown to be effective for securing CPSs. Unfortunately, deriving invariants for capturing the specifications of CPS systems is a tedious and error-prone process. Therefore, it is important to dynamically monitor the CPS system to learn its common behaviors and formulate invariants for detecting security attacks. Existing techniques for invariant mining only incorporate data and events, but not time. However, time is central to most CPS systems, and hence incorporating time in addition to data and events, is essential for achieving low false positives and false negatives.

This paper proposes ARTINALI, which mines dynamic system properties by incorporating time as a first-class property of the system. We build ARTINALI-based Intrusion Detection Systems (IDSes) for two CPSes, namely smart meters and smart medical devices, and measure their efficacy. We find that the ARTINALI-based IDSes significantly reduce the ratio of false positives and false negatives by 16 to 48% (average 30.75%) and 89 to 95% (average 93.4%) respectively over other dynamic invariant detection tools.

CCS CONCEPTS

• **Security and privacy → Intrusion detection systems**; **Software security engineering**; *Domain-specific security and privacy architectures*; • **Software and its engineering → Real-time systems software**;

*This work was done during a summer internship at UBC.

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106282

KEYWORDS

Multi-dimensional model, Security, Cyber Physical System, CPS, Software Engineering

ACM Reference format:
Maryam Raiyat Aliabadi, Amita Ajith Kamath, Julien Gascon-Samson, and Karthik Pattabiraman. 2017. ARTINALI: Dynamic Invariant Detection for Cyber-Physical System Security. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 13 pages.
https://doi.org/10.1145/3106237.3106282

1 INTRODUCTION

Cyber Physical Systems (CPSes) are being increasingly used in security-critical contexts such as smart medical devices [23, 34], smart grids [35], smart cars [9], and Unmanned Aerial Vehicles (UAV) [18]. Unfortunately, these systems are vulnerable to cyber attacks due to their interconnectedness and relative lack of protection. Many attacks have been demonstrated against CPSes such as cars [21], smart medical devices [23, 27], and smart meters [36].

Intrusion Detection Systems (IDSes) have been widely used to monitor computer systems and detect security attacks. Typical IDSes fall into one of three categories: *Signature-based*, *Anomaly-based*, and *Specification-based*. Signature-based detection techniques compare the runtime behavior of the system against known security attacks, and hence cannot detect unknown attacks [30]. The latter is especially important for CPSes as they are often difficult to patch or upgrade in the field. In contrast, both anomaly-based and specification-based techniques use a behavioral model of the system to compare with suspicious behaviors, and can detect unknown attacks. Anomaly-based techniques learn the system's behavior by observing its operations at runtime and formulating the model of the system, while specification-based techniques rely upon apriori knowledge of the systems' behaviour to detect attacks. Unfortunately, anomaly based systems incur considerable overhead to learn the system model at runtime, and also suffer from high rates of false-positives. These factors inhibit their use in CPSes which are often resource constrained, and operate autonomously for long

periods of time. Therefore, specification-based systems have been proposed as the best fit for CPS security [8].

Specification-based techniques build a model for a system based on its code and the specifications defined by the developer. However, there are often inconsistencies between what developers describe their system does, and what the system does in practice [11, 31]. Moreover, code alone does not provide information about the run-time behavior of the system in its operational environment. In contrast, dynamic analysis-based techniques provide an alternative way to understand the system by observing the run-time behavior. There has been a significant amount of work on using dynamic analysis to find likely invariants for program understanding and debugging [14, 16, 19, 22, 24, 25, 28, 32, 33, 42, 44]. These systems mine execution traces of the system for deriving invariants on the data values of the system, the events or both. However, we find that many of these systems incur significant false-positives and/or false-negatives, when used in the context of an IDS, which makes them challenging to deploy.

This paper introduces ARTINALI (A Real-Time-specific Invariant iNference ALgorIthm) for mining likely invariants through dynamic analysis in CPS systems, for specification-based IDSes. The main innovation in ARTINALI is that it incorporates time as a first-class notion in the mined invariants, in addition to the traditional data and event invariants. This is important for two reasons. First, most CPSes have real-time constraints, and hence their operational correctness depends on both logical correctness, and correct timing behavior [20, 41]. Hence, incorporating time is essential for detecting many common security attacks in these systems. Secondly, CPSes have predictable timing behaviors to a first order of approximation, and hence leveraging this predictability leads to higher accuracy (i.e., lower false-positives and negatives). However, incorporating time in dynamic invariant detection techniques increases the complexity of the learning due to the much larger state space that needs to be covered. To alleviate this issue, we break up the problem of learning invariants along the three dimensions into problems of learning invariants along two dimensions, namely data-events and events-time, and then combine them into data-events-time invariants. *To the best of our knowledge, ARTINALI is the first dynamic invariant detection system that mines invariants along the three dimensions of data, event, and time, and uses the mined invariants for intrusion detection.* Our contributions are:

- We designed ARTINALI, an algorithm that generates a multi-dimensional model for CPSes by mining invariants along the data, event and time dimensions (Section 3).
- We built an ARTINALI-based IDS prototype, and used it in the context of two emerging CPS systems, namely i) advanced metering infrastructures, and ii) smart artificial pancreas (Section 4).
- We evaluated the efficacy of ARTINALI for 6 targeted attacks on the two systems. We find that the ARTINALI-based IDS can detect all 6 attacks, while none of the other dynamic invariant detection systems do so (Section 5).
- We also evaluated our ARTINALI-based IDS prototype on the two systems, and compared it with several existing state of the art dynamic invariant detection techniques. Overall, we find that the ARTINALI-based IDS exhibits significantly lower false-negatives and false-positives for arbitrary attacks emulated by fault injection, compared to the other techniques (Section 6). Furthermore, it incurs about 32% performance overhead, which is comparable to other invariant detection techniques.

2 BACKGROUND

We first survey related work in the area of dynamic invariant detection and how ARTINALI differs from them. We then present a motivating example from smart meters to illustrate why we need data-time-event invariants like the ones generated by ARTINALI.

2.1 Related work

Dynamic analysis-based techniques that model the behavior of software systems can be categorized into four classes, based on the models that they generate: i) data invariants, ii) event relationships, iii) data and event relationships, and iv) time dependencies of events. Figure 1 shows the main dynamic analysis-based techniques, and where they fall along the data, event and time axes.

Daikon was the first dynamic analysis-based technique to derive (likely) invariants about data value relations [14], and falls into the first class of techniques. Daikon can be placed on the *data* axis as it produces a model for data constraints without taking into account the events or timing of the system. DySy [10], which uses symbolic execution to derive invariants, is another example of this class.

The second class captures the sequence of events within a progam's execution paths by inferring finite state machines from a set of traces. Relevant examples include Perracotta [44] and Texada [25], both of which derive temporal logic propositions, and capture sequences of events by tracking dynamic traces. These tools fall along the *event* axis since they only capture the constraints on event relations independent of data or timing information.

The third class of techniques generate integration models that capture the relationship between data and events. For example, the GK-Tail algorithm merges temporal specifications and data invariants into Extended Finite State Machine models [28]. It represents sequences of method invocations that are annotated with data, and is hence limited to classifying data invariants that arise among method calls. Quarry finds data invariants at each program point, and then finds temporal relationships between the invariants [24]. Neither technique considers timing information, however.

The fourth class consists of a single technique, Perfume, which is a specification mining tool designed for modelling system properties based on resource (time and storage) consumption [32]. It generates an integration model of event relations and their time constraints. Although Perfume considers time as a part of model, it does not consider the relationship between data and time.

Overall, none of the current techniques consider the interplay among time, events and data in formulating invariants, which we believe is an essential characteristic of CPS systems.

2.2 A Motivating Example

We consider an example of a smart electric meter to illustrate why the existing dynamic analysis-based techniques are often insufficient for capturing the key properties of a CPS system. We also use this as a motivating example to illustrate ARTINALI later.

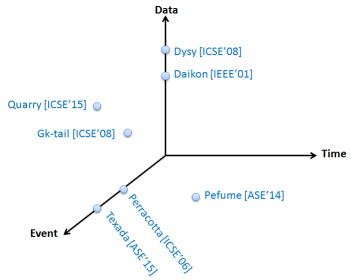

Figure 1: Scope of dynamic invariant detection techniques

We use an open-source smart meter called SEGMeter as one of the testbeds for our implementation and our evaluations (see Section 4.1). SEGMeter is composed of two main components: the meter component and a controller. The meter component is in charge of measuring and collecting power consumption data coming through its serial ports, and storing them in memory. The controller acts as the communication bridge between the meter board and the server, and is in charge of passing server commands to the meter board, as well as transmitting power consumption data to the server at specific time intervals. The `Serial-Talker()` function in the controller program of the smart meter is in charge of receiving power consumption data (at specific time intervals) and buffering them for billing calculation purposes (Lua code shown in Figure 2).

```
1  function serial_talker()
   .....
5     seg-data = get_data_timer()
6     if (seg-data == 'a') then
7       command = "(all_nodes (start_data))"
8       serial_client:send(command .. ";\n")
        ....
19       read(message)
        ....
23    end
24     if (seg-data == 'b') then
25      stream, status, partial = serial_client:receive(16768)
        ....
28      f:write("node_name = ", tostring(partial), "\n")
        ....
34    end
   ....
56 end
```

Figure 2: A snippet of Serial-Talker code for the SEGMeter

Figure 3 shows the execution path of `Serial-Talker()`. The argument `seg-data` can take two different values: `a` or `b`, in predetermined time intervals. The sequence of events that are invoked in this function varies based on the value passed in argument (`seg-data`). If `a` is passed (line 6 in Figure 2), then the program emits the event `send`, followed by `read`. Alternatively, if `b` is passed to the function (line 21 in Figure 2), then the program emits events `receive` and `write` respectively.

We examined the invariants inferred by the different dynamic analysis-based tools for this example. Daikon infers the values of variable `seg-data` within `Serial-Talker()` during normal execution as the set $\{a, b\}$, namely `seg-data:[a,b]`. A typical temporal specification miner such as Texada identifies the legal sequences of events, e.g., G(send $\rightarrow$ XF read), which means that upon event `send` happening, it is always followed by event `read`. The invariant inferred by Perfume *(send → receive, 0.1, 3.6)* complement the

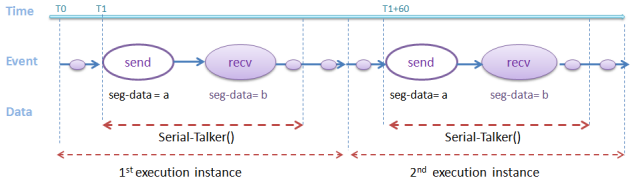

Figure 3: The state chart of Serial-Talker()

temporal invariant by adding time boundaries between events, i.e., send is followed by read within a time interval of 0.1 to 3.6 ms.

Assume that an adversary's goal would be to perform energy fraud and lower their energy bills. One possible attack would be for the attacker to tamper with the synchronization between the `send` and `receive` modes in the smart meter. As a result, a part of the energy usage would not be written to the memory buffer which is used for future energy usage calculations and billing. For instance, should the value `b` be passed to the function instead of `a`, then it would lead to the execution of `receive` and `write` instead of `send` and `read`; hence the billing information would be incorrect.

None of the above techniques can detect the attack as the incorrect occurrence of sequences are triggered by legal values of `seg-data` occurring at the wrong time (e.g., `seg-data (T1) = b`). More specifically, Daikon would notice a valid value for `seg-data`, Texada would notice a normal sequence of `receive` and `write` events, and Perfume would also observe valid time intervals between events `receive` and `write` within the executed path. Thus, none of them would detect the attack. Even if all three models are used jointly, they would still not detect the intrusion, as the different models either capture the legal data values, or the legal sequence of events with their time difference, but not the interplay among them. This interplay is essential for detecting the attack.

3 APPROACH

In this section, we introduce the security model that ARTINALI uses, and we explain its design. We first define our multi-dimensional model and the different classes of invariants. Next, we explain how to relate different dimensions to generate real-time data invariants. Finally, we present the ARTINALI workflow and algorithm.

3.1 Multi-dimensional model

We model a CPS in three dimensions, as follows:

Data refers to data values assigned to the variables of a program. It includes neither the timing of processes, nor the sequence and concurrency of processes.

Event refers to an action that a system takes to respond to an external stimulus.

Time refers to real-time constraints, and includes both the constraints on physical timing of various operations, and those where the system must guarantee response within a specified time frame.

We model the security policy of a CPS by inferring the set of *invariants* to be preserved during run time. An *invariant*, or interchangeably a *property*, is a logical condition that holds true at a particular set of program points. Like in prior work [14, 25, 44], we use the term *invariants* as a shortcut for *likely invariants*, which are

the properties that we *observe* to be true accross a set of dynamic execution traces. Corresponding to the dimensions defined above, we define six major classes of invariants that form the basis of the CPS model, as follows:

- *Data Invariant* captures the expected range of values of selected data variables during normal execution of program.
- *Event Invariant* captures common patterns in the system's events such as the order of the events' occurrence.
- *Time Invariant* captures the normal time boundaries (such as duration or frequency) of an event.
- *Data per Event(D|E) Invariant* captures the temporal relationship between data and events. It allows the IDS to check the validity of data invariants based upon events.
- *Event per Time (E|T) Invariant* captures the constraints over event and time. It represents the boundaries of transition time from one event to another in an event sequence.
- *Data per Time (D|T) Invariant* captures the relational constraints of time and data invariants. It represents the data invariant as a function of time.

3.2 Data-Event-Time Interplay

In a CPS, an event is defined as an instance of an action that leads to a change of condition [40] (e.g., message send/receive, sensor data reading, or activating insulin injection). Events have three key features. First, they reflect interactions between system components and observations rather than internal state. The second feature is the notion that events are separated in space and time [12, 13, 40], and hence, there is no concurrency among event executions. Moreover, in the case of dependent events (i.e., one event triggers the other), once the first event (task) is done, it triggers the execution of the second event. Thirdly, the locations in the code where events are triggered are usually *system calls* that are accessible by attackers. From a security perspective, events are important as they play the role of an input channel for malicious communication with the CPS. For instance, those points in which a new measurement is read from sensors, or actuation commands are sent to physical components, are more vulnerable to *spoofing attacks* [15].

Finding a direct relationship between time and data is challenging from both the *learning* and *detection* perspectives. Since time is a continuous phenomenon, we cannot define a sharp time for transitions in data values or changing states of the system; instead, a distribution of time values has to be learned. As execution time variations might be caused by differences in input sets or different execution flows, rather than malicious activities, the invariant inference technique should learn the normal time variations of the system. The IDS also has to distinguish legitimate time variations from any time deviation that indicates an intrusion.

To overcome these challenges, we leverage the event-based nature of a CPS, in which every event takes place in an unique timeframe. We discretize the *time* by the *events*, and use these for learning invariants. After doing so, we first examine the relationship between data and event dimensions to produce invariants that integrate event information with constraints on data values ($D|E$ invariants). Secondly, we discover the relational constraints over time and event dimensions to calculate the physical time boundaries of events, either independently (*time* invariants), or in relation to each other ($E|T$ invariants). Finally, we combine the result of the previous steps to infer $D|T$ invariants.

In the following discussion, we illustrate how we infer the $D|T$ invariants given the conditional probability of having *data* D given *event* E invariant ($P(D|E)$), and given the conditional probability of having *event* E given *time* T invariant ($P(E|T)$).

Considering data D, event E and time T as random variables, equation 1 expresses the joint probability distribution of variables D, E and T. We rewrite it to obtain equation 2. From these two equations, we then derive equation 3, which expresses the probability of having D and E, given T.

$$P(D, E, T) = P(D, E|T) \cdot P(T) \tag{1}$$

$$P(D, E, T) = P(D|E, T) \cdot P(E|T) \cdot P(T) \tag{2}$$

$$P(D, E|T) = P(D|E, T) \cdot P(E|T) \tag{3}$$

Using the marginal probability mass function of D shown in equation 4, we formalize $P(D|T)$ (the probability of having D given T) in equation 5 as the sum of the probabilities of data D and event E_j given time T *for all events* E_j, which can then be rewritten as equation 6 (using equation 3).

$$P(D) = \sum P(D, E_j), \forall E_j \tag{4}$$

$$P(D|T) = \sum P(D, E_j|T), \forall E_j \tag{5}$$

$$P(D|T) = \sum P(D|E_j, T) \cdot P(E_j|T), \forall E_j \tag{6}$$

For example, assuming that at time T, event E_j occurs; and that upon E_j occurring, then variable D gets assigned a specific value. This implies that T is the cause of E_j, and that D is the effect of E_j. Thus, variable D is *conditionally independent* of time variable T given event E_j. Consequently, D and T are conditionally independent, and $P(D|E_j, T) = P(D|E_j)$. Hence, we can simplify the formulation of $P(D|T)$ as follows :

$$P(D|T) = \sum P(D|E_j) \cdot P(E_j|T), \forall E_j \tag{7}$$

According to the event-based semantics of CPS, any given event takes place in a unique time frame. This implies that two or more events cannot take place at the same time T; i.e., $P(E_j|T) > 0 \Rightarrow P(E_i|T) = 0, \forall E_i \neq E_j$. Given this assumption, we first rewrite equation 7 to obtain equation 8. Then, we simplify it to obtain equation 9, which captures the relationship between data D and time T by exploiting the relational constraints of both data and time *over the same event* E_j which takes place at time T.

$$P(D|T) = P(D|E = E_j) \cdot P(E = E_j|T) + \sum P(D|E_i) \cdot P(E_i|T), \forall E_i \neq E_j \tag{8}$$

$$P(D|T) = P(D|E = E_j) \cdot P(E = E_j|T) \tag{9}$$

In other words, *for a given event* E_j, *a* $D|T$ *invariant holds true (i.e., happens with a high probability) if and only if both the corresponding* $D|E$ *invariant and* $E|T$ *invariant hold true.*

3.3 ARTINALI Workflow

ARTINALI is a *dynamic analysis-based* technique that generates models of dynamic system behavior, and proposes a multi-dimensional

model based on the design concepts introduced in the previous section. Figure 4 shows the key blocks of ARTINALI's workflow.

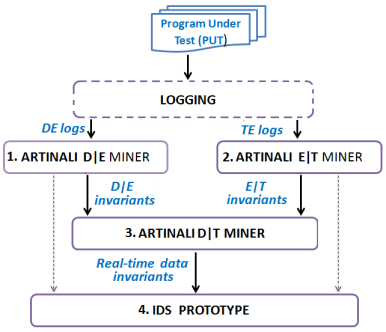

Figure 4: Work flow of ARTINALI

In order to generate logs for mining invariants, we manually instrument events and their associated data variables[1]. We assume that CPS source code is available, and that it can be modified to instrument events - this is reasonable as we envision our technique to be used by CPS developers (if not, one can use a binary instrumentation engine). In our technique, events are system calls. Hence, we capture all system calls as events. However, the user can optionally prune the space of events by specifying only the *important* system calls based on the system's requirements. We instrument the events' program locations by inserting calls to the *ARTINALI API functions* that we developed for collecting logs, before and after the event. During the attack-free runtime executions, these functions collect *data* and *time* information associated with the instrumented events in separate log files (namely *DE logs* and *TE logs*). The logged information is used as the basis for mining invariants.

Block 1. ARTINALI $D|E$ Miner The ARTINALI $D|E$ Miner learns invariants about the variable values, and how these values relate to a particular event in the system using a three-step process. First, the $D|E$ Miner takes the logged information, and groups them within each trace into distinct classes labeled with the events. It then merges classes across *DE logs*. Second, within each class, using the Frequent Item Set mining algorithm [17], it merges the data variables while calculating the level of the *confidence* and *support* for every variable. As in prior work [14], *support* is the fraction of traces in which the variable x within class E_j is seen, and *confidence* is the fraction of supported classes, where variable x is assigned to the same value(s).

Finally, the $D|E$ Miner infers the data invariants associated with each class (event). $D|E$ invariants are multi-propositional data invariants as they all hold true within the same observed event (at the same time). The $D|E$ invariants are stated in the form of $(E_i : d_1 = [], d_2 = [], ...d_n = [])$, where E_i denotes the name of $(i)th$ event, and $d_1 - d_n$ denote the range of concrete values of n data variables mapping to the event E_i. For the example in Section 2.2, the invariant *receive: seg-data=false, command=nil, status=[nil, time-out], len.partial>0* represents the data invariants that are valid during the event *receive*, and is hence a $D|E$ invariant.

[1]This is similar to what almost all other invariant detection techniques do, with the exception of DAIKON, which has an automated instrumentation engine.

Table 1: E|T and D|T Invariant Types

	E|T Invariant Type
Type I	$E_i(t) \rightleftharpoons E_i(t + \frac{1}{Freq_i})$
Type II	$E_j \rightleftharpoons E_i : \Delta t_{ji}\text{max}, \Delta t_{ji}\text{min}$
Type III	$E_i : \Delta t_i\text{max}, \Delta t_i\text{min}$
	D|T Invariant Type
Type I	$d_m(T_i \leq t \leq T_j) = []$
Type II	$d_m = [] \rightleftharpoons d_n = [] : \Delta t_{ji}\text{max}, \Delta t_{ji}\text{min}$

Block 2. ARTINALI $E|T$ Miner ARTINALI's $E|T$ Miner infers the $E|T$ invariants in four steps. First, it creates all consecutive event pairs within one trace annotated with their time differences. Second, it groups the pair of events that are labeled with the same pair name. Third, ARTINALI's E|T Miner looks for the pair-wise events that are observed in the same order within *TE logs*, and calculates their support. Finally, it merges the time variables within each class to calculate the time boundaries of the paired events, as well as the frequency and the average duration of every event execution. The E|T invariants are classified into three types, as shown in Table 1. Type I indicates that event E_i is repeated every $\frac{1}{Freq_i}$ seconds. Type II indicates that the pair of events E_i and E_j are repeated in all traces in the same order, and their time difference is bounded within $\Delta t_{ji}max$ and $\Delta t_{ji}min$. Type III indicates the maximum and minimum duration of event E_i. For the example in Section 2.2, $\texttt{send}(t) \rightleftharpoons \texttt{send}(t+60)$ showing the frequency of `send` occurrences in the system, and the invariant $\texttt{send} \rightleftharpoons \texttt{receive} : 14.3, 1.5$ representing the time boundary (between 1.5 and 14.3) and the logical ordering of the events (i.e., send before receive), are both examples of $E|T$ invariants.

Block 3. ARTINALI $D|T$ Miner According to the formulation described for $D|T$ invariants, ARTINALI combines the outputs of $D|E$ and $E|T$ miners to generate the real-time data invariants (D|T invariants). We define two types of data invariants (Table 1), and we explain each type using the example in Section 2.2.

Type I represents the distribution of valid data values of variable d_m within time slot $T_i \leq t \leq T_j$. For example, $\texttt{seg-data}(T_1 \leq t \leq T_2) = a$ means that the only valid value of variable `seg-data` is a during the time interval $T_1 \leq t \leq T_2$. Note that we differ here from Daikon data invariants (e.g. `seg-data=a,b`), as they only express the valid values of data invariants without considering the time.

Type II captures the relationship of data invariants between two consecutive events. As explained in previous section, every two consecutive events have a bounded time difference ($T_i + \Delta t_{ji}\text{min} \leq T_j \leq T_i + \Delta t_{ji}\text{max}$). As a result, the data invariants associated with those events have the same time difference. In other words, data invariant $d_j = []$ holds true *until* data invariant $d_i = []$ becomes true, while $\Delta t_{ji}\text{max}$ and $\Delta t_{ji}\text{min}$ specifies the time difference boundaries between those data invariants. In the previous example (Figure 3), ARTINALI $D|T$ Miner generates one invariant of this type, as follows: $\text{seg-data} = a \rightleftharpoons \text{seg-data} = b : 14.3, 1.5$; i.e., $\text{seg-data} = a$ holds true *until* `seg-data` is assigned value b, in a time interval ranging between 1.5 and 14.3 seconds.

Block 4. IDS Prototype As explained in the previous sections, the ARTINALI Miners derive three classes of invariants that comprise the final CPS model. The CPS model is used as an input to our

IDS prototype for monitoring attacks. Our IDS prototype consists of two components: the *Tracing module* and the *Intrusion detector*. The tracing module is in charge of collecting the required information from the program's execution and logging it. This module is the same as the ARTINALI Logger that instruments the code and collects logs, but with the difference that it is deployed on the production system. The collected information is fed to the intrusion detector, which periodically processes the log file and checks it against the invariants derived from the CPS model.

4 EXPERIMENTAL SETUP

This section first presents the details of two CPSes, and then the experimental procedure for evaluating the IDS on the two platforms. Finally, it presents the attack models that we considered for evaluation, followed by the evaluation metrics.

4.1 CPS platforms

We chose two CPS platforms as case studies to evaluate the efficacy of the invariants generated by ARTINALI and the other tools. Note that unlike generic applications, there are few publicly available open-source CPS platforms that are also security-critical. Furthermore, there is a significant amount of effort involved in setting up a CPS platform and generating execution traces from it. So we limit ourselves to 2 CPS platforms in the paper.

Advanced Metering Infrastructure (AMI) : Advanced Metering Infrastructure (AMI) systems are deployed on smart electric power grids. Smart meters are key components of AMI that provide a two-way communication with the utility provider[35]. The large scale deployment of smart meters and the discovery of many vulnerabilities in these systems [38, 43], make them good candidates to evaluate our work. A generic smart meter is composed of two main components, namely the meter and the controller. The meter component receives power consumption data through analog front end sensors, and stores them in the memory. The controller component is the communication bridge between the meter and the utility provider's server, passing server commands to the meter, and sending consumption data back to the server at specific time intervals (more details in [38]). We use SEGMeter [1], an open source smart meter to evaluate our IDS prototype. SEGMeter is implemented using the Lua language, and consists of 2500 lines of code (excluding libraries).

Smart Artificial Pancreas (SAP) : Diabetic patients are migrating from the traditional glucose meter and manual insulin injection systems to continuous glucose monitoring and autonomous insulin delivery devices [27], which are referred to as *Smart Artificial Pancreas (SAP)*. Since attacks to a SAP can threaten the patient's life, these systems are highly security-critical [34]. Hence, we selected SAP as our second case study to evaluate ARTINALI. The main building blocks of a generic SAP are a Continuous Glucose Monitor (CGM), an insulin pump, and a controller. The CGM samples the patient's blood glucose (BG) levels on a regular basis and sends it to the controller. The insulin pump is a wearable medical device that is used for automatic injection of insulin through subcutaneous infusion. The controller controls the closed loop in the SAP. It receives the measured BG from CGM, and issues the suitable actuation command for correcting the sugar level. We used Open Artificial Pancreas System (OpenAPS) [26], an open source SAP, as a second use case to evaluate our IDS prototype. OpenAPS implements the controller component of an SAP in JavaScript, and consists of 2000 lines of code (excluding libraries). We simulated a simple CGM and an insulin pump to close the loop, as we did not have access to a patient with a real insulin pump and glucose meter. OpenAPS provides a set of test cases that take different BG values as input and process them for calculating basal rate of insulin, which we use as a baseline for our experiments.

4.2 Experimental Procedure

Figure 5 shows the overall procedure that we follow. In addition to generating the CPS model using ARTINALI, we generate three other models (invariants sets) using Daikon, Texada, and Perfume for comparison purposes. We have made ARTINALI publicly available [4]. We downloaded the latest versions of these tools from their respective websites [2, 3, 5]. We do not run the instrumentation front-end of Daikon (i.e., Kvasir), as our goal was to generate data invariants based on the event traces we logged. We choose these three tools to represent the first, second and fourth classes of invariants as described in Section 2. We do not choose the tools in the third category, namely GK-tail and Quarry, as we use Daikon to find data invariants for the events that we identified in the system. Therefore, the invariants generated by Daikon cover the third class of invariants in our experiments (i.e., $D|E$ invariants).

There are 22 system calls in SEGMeter's code, and 4 system calls in the OpenAPS code. We consider all of them as events. Tables 2 and 3 present the types of invariants and the number of invariants generated by the three tools and ARTINALI for the SEGMeter and OpenAPS platforms respectively. As can be seen, ARTINALI generates invariants in the Time, $D|E$, $E|T$, and $D|T$ categories, while DAIKON, Texada and Perfume only generate invariants in the $D|E$, Event and $E|T$ categories respectively.

Because the format of the invariants generated by these other tools may be different from that expected by our IDS, we wrote scripts to convert the invariants to be in the format expected by the IDS interface. ARTINALI directly generated invariants in the proper format. In case a tool did not generate a certain kind of invariant (e.g., $D|E$), we leave that invariant file blank. The generated invariant sets are all fed into the IDS as inputs, and their efficacy is evaluated on different platforms.

We divide the experiment into a training phase and a testing phase for each system. We first obtain execution traces from the two platforms under normal operation, and randomly divide them into a set of training traces (*train*) and testing traces (*test*). We then choose different training set sizes for each invariant detection system to optimize the false positive (FP) and false negative (FN) ratios for that system. Finally, we evaluate the FP ratios of the invariants using the *test* traces, and the FN ratios using the attack models described in the next section.

The IDS is implemented in Python, and consists of about 1000 lines of code. Since the IDS is run on the CPS platform, which is often resource constrained, it is important to minimize its overheads. We measure the IDS's time and space overhead for the SEGMeter platform in Section 6. Because we run the OpenAPS platform in

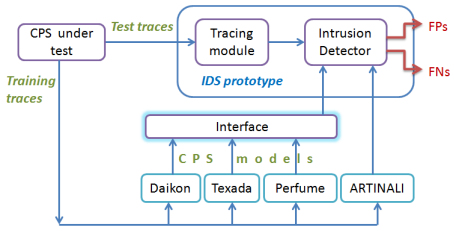

Figure 5: Overall experimental process of running the IDS

Table 2: Types and number of inferred invariants for SEGMeter across tools

	Event	*Time*	*D|E*	*E|T*	*D|T*
Daikon	-	-	24	-	-
Texada	158	-	-	-	-
Perfume	-	-	-	158	-
ARTINALI	-	12	24	37	24

Table 3: Type and number of inferred invariants for OpenAPS across tools

	Event	*Time*	*D|E*	*E|T*	*D|T*
Daikon	-	-	22	-	-
Texada	57	-	-	-	-
Perfume	-	-	-	57	-
ARTINALI	-	4	22	18	7

a simulator, as we did not have access to its hardware, we do not measure the IDS overheads on OpenAPS.

4.3 Attack Models

Traditionally, security techniques are evaluated using a small number of targeted (hand-crafted) attacks. Unfortunately, this is not sufficient for CPS systems for three reasons. First, CPSes are new systems for which there are few real attacks - hence they need protection from zero-day or unknown attacks. This is especially the case for security-critical CPSes such as smart medical devices. Secondly, unlike general computer systems, CPSes can be difficult to upgrade and patch frequently. Thirdly, there is no standard set of attacks on CPSes unlike general-purpose computer systems, so we would have to hand-craft individual attacks on these systems, which would potentially introduce bias in the evaluation.

Targeted Attacks To evaluate our IDSs against known attacks, we use three attacks for each system that we discovered based on manual analysis of the CPS (Section 5).

Arbitrary Attacks We use fault injection (i.e., mutation testing) to emulate arbitrary attacks. Fault injection has been used to study the effects of attacks in previous work [38]. Note that these are not complete attacks, but rather form the building blocks of attacks. We deploy different types of mutation in the program's code, as follows.

- *Data mutations*, which change the runtime values of data variables in the code;
- *Branch flipping*, which change the normal execution flow of the program by flipping branch conditions;

Table 4: The number of mutations in each attack category for SEGMeter and OpenAPS.

CPS	Attack category		
	Data mutation	Branch flip	Artificial delay
SEGMeter	35	76	45
OpenAPS	100	10	15

- *Artificial delay insertion*, which modify the normal timing behavior of the program.

Each of the above categories emulate different security issues. By performing data mutations, an attacker can change critical data in the program to their advantage. Such attacks can be accomplished by exploiting memory corruption vulnerabilities or race conditions in the program. Likewise, branch flipping can lead to illegitimate control flow paths being taken in the program, to accomplish the attacker's ends. Such attacks can occur due to code injection or sematic vulnerabilities. Finally, artificial delays can allow attackers to change the timing of the system's actions, and delay essential functions, or cause other functionality to be suppressed, again to their advantage. Through these mutations, we can emulate a wide variety of attacks, without a predefined target, thus avoiding bias and allowing modelling of hitherto unknown attacks.

Table 4 presents the number of mutations performed in each category for SEGMeter and OpenAPS. We manually seeded each of these mutations in the source code of the respective systems, by randomly sampling the corresponding program points in the program's code. While this could have been automated by a fault injection tool (e.g., LLFI [6]), the languages in which the two systems were implemented, JavaScript and Lua, were not supported by existing tools. Therefore, we had to perform mutations manually. However, we attempted to choose the program points randomly before performing the experiment to avoid biasing our evaluation.

After mutating the code, we can observe one of four outcomes.

- *Crash*, in which the program is aborted (exception);
- *Hang*, in which the program goes into an infinite loop or deadlocks;
- *SDC (Silent Data Corruption)*, in which the outcome of the program is different from a fault-free execution;
- *No corruption*, in which the outcome of the program does not show any observable impact with respect to fault masking or non-triggering faults. Internal states might however be corrupted.

Note that in the context of this paper, we are interested only in *SDC* and *No corruption* outcomes, as the *Crash* and *Hang* outcomes can easily be detected without an IDS. Therefore, we only present the results for the fault-injection experiments that resulted in the *SDC* and *No corruption* outcomes, and which need an IDS (these comprise about 75% of the outcomes on average).

4.4 Evaluation Metrics

Accuracy: We use three metrics to measure the effectiveness of our IDS from the accuracy point of view.

- *False Negative ratio (FN)*, which is the ratio of attacks that were undetected by the IDS to the total number of attacks;

- *False Positive ratio (FP)*, which is the ratio of execution traces that were (incorrectly) reported as attacks to the total number of normal traces;
- *F-Score(β)*, which is a computation of the harmonic mean of the true positive ratio (TP), FP and FN.

The variations of the argument β in F-Score(β) allow us to weigh the above metrics differently [37], and obtain different trade-off between FP and FN ratios based on the system requirements. A value of $\beta > 1$ weighs FNs higher, while a value of $\beta < 1$ weighs FPs higher. A value of $\beta = 1$ weighs them both equally. We hypothesize that FPs are more important in smart meters, as a false-alarm leads to added cost to the utility provider who needs to deploy service personnel to investigate the false alarm. An occasional FN may be acceptable in smart meters as the consequence is only a loss of revenue. In the OpenAPS, on the other hand, even a single FN can be fatal to the patient, while a FP may be acceptable if there are other checks in place to filter out FPs (e.g., patient intervention). Hence, for SEGMeter, we select F-Score(0.5), and for OpenAPS, we choose F-score(2) as our reference metric.

Overheads: In addition to the accuracy, we also measure the memory and performance overheads of the IDS.

Memory overhead is defined as the actual memory usage of the IDS. It depends on the size of IDS, the number of invariants that account for the CPS model, and the complexity of invariants (e.g., the invariant $E_j \rightleftharpoons E_i : \Delta t_{ji}\text{max}, \Delta t_{ji}\text{min}$ carries more information than the invariant $E_j \rightleftharpoons E_i$, and is hence more complex).

Performance overhead is the increase in execution time as a result of running the CPS on the target platform. This metric reflects the overhead of both the *tracing module* and the *intrusion detector*. Since CPSes run continuously for long periods of time, we measure the performance overhead per cycle, where a cycle refers to one full execution of the main loop of the CPS (both the SEGmeter and OpenAPS consist of a single main loop that runs continuously).

5 TARGETED ATTACKS

In this section, we discuss the potential targeted attacks and how we derive them for both platforms. We then evaluate the IDS seeded by ARTINALI and other tools against the attacks. Note that we used attack trees based on prior attacks against similar systems to generate the attacks to minimize bias and model realistic attacks. We found that ARTINALI was able to detect all the attacks, while none of the other tools do. This is because all the attacks involved violations of the interplay among data, events, and time.

5.1 AMI Attacks

Energy fraud is a major class of AMI attacks, and can result in *Power Consumption Data (PCD)* loss and improper billing [29]. We came up with an attack tree for energy fraud in AMI (shown in Figure 6), based on attacks introduced in previous work [29, 35, 39, 43]. There are three major branches in this tree, namely i) Measurement tampering, ii) Storage tampering, and iii) Network tampering. Corresponding to each branch, we developed the concrete attack actions as the leaves of the tree as follows.

Synchronization tampering (Blocks $A1 - A4$) occurs due to modification of the time of *send* and *receive* modes in AMI. We found that the communication between the AMI and the server is synchronized by a vulnerable function (*get-data-timer()*) in the controller unit. The controller frequently checks the time with the sever to decide when to request for data measured by the meter. If a malicious user modifies the time on the server, the controller will not receive data in the expected time, which leads to data loss, and improper calculation of final PCD.

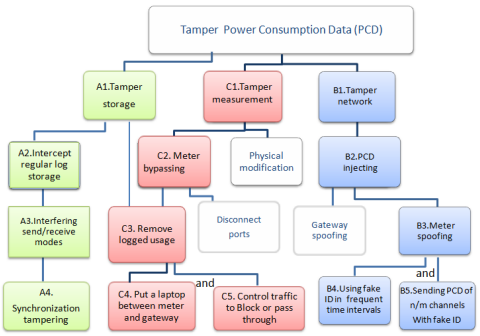

Figure 6: Attack tree for AMI

Table 5: ARTINALI invariants to detect the example attacks in AMI.

Attack	Detecting Invariant
Synchronization tampering	(1) send (T0+K · 60) ⇌ send (T0+(K+1) · 60), ∀ k≥0
Message dropping	(2) recv (T1) ⇌ recv (T1+1)
Meter spoofing	(3) node-name(T0+N · 60) = Node B, ∀ N≥0

Meter spoofing (Blocks $B1 - B5$): In a smart grid, AMIs communicate with the server using a unique name or ID. The controller unit is able to be connected to more than one meter, collects the PCDs, and send them along with the meter's ID to the server. As the controller cannot differentiate between normal and abnormal messages, it can be tricked by falsified inputs sent by an attacker instead of the meter. This attack is called *meter spoofing attack*. We found that spoofing the meter only requires the meter's ID that is printed on the meter's nameplate.

Message dropping (Blocks $C1 - C5$): An attacker may be able to drop the messages (i.e., a part of energy usage) after bypassing the meter and removing the logged PCD history. A simple way to mount this attack is to intercept the communication between the meter and the controller, and control what traffic to block and what to pass through (e.g., through a firewall). Hence, the blocked traffic would not be included in PCD calculations.

5.2 Detection of AMI attacks

We ran the ARTINALI-based IDS on the example attacks, and found that it detected all of them. Table 5 indicates the important invariants that are derived by ARTINALI, which detect the attacks presented in the previous section.

Synchronization tampering As synchronization tampering attack modifies the timing of *send* and *receive* operations of SEGMeter, we picked events *send* and *receive* as relevant events to explain this attack. We can see in row 1 of Table 5 that the ARTINALI invariant captures the sequence of these events during normal operation, i.e., *send* operation happens every 60 seconds, and *receive*

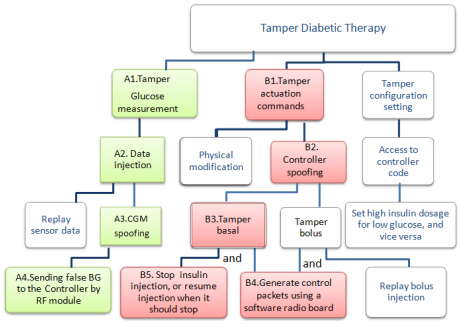

Figure 7: Attack tree for SAP

is repeated every 1 second. Thus, this invariant detects the attack as the timing of the events is violated by the attack.

Message dropping If we assume the attacker drops one or more messages from meter, the dropped messages will not be received at the expected time slots by the controller. As a result, the frequency of receiving messages in controller will change. This attack breaks the invariant number (2) in Table 5, which represents the time frequency of receive function which is 1003 milliseconds ($\cong 1sec$) within one full execution path. Thus this attack is also detected.

Meter spoofing To detect meter spoofing attack, we selected two *receive* events ($recvA$ and $recvB$) from two different meters (node A and node B) that are connected to the same controller, and analyzed the respective invariants. For example, *nodeName(T0+N*60) = Node B,* $\forall\ N\ \geq 0$ specifies that the valid value of *nodeName* at $T0 + N * 60$ is *Node B*. If the identity of node A is stolen by node B, it sends its messages every 60 seconds under the name *nodeA*. As a result, variable *nodeName* attached to event *recvB*, becomes *nodeA*. Thus, the invariant number (3) in Table 5 is violated.

5.3 SAP Attacks

Diabetic therapy tampering is one of the highest severity threats for patients, as it can result in death or severe health complications. We developed an attack tree for diabetic therapy tampering based on publicly available reports of attacks on SAPs [27, 34], in Figure 7. We consider three classes of attacks based on the tree.

CGM spoofing attack (Blocks $A1 - A4$) injects false into the communication channel between CGM and controller making the controller think that the glucose level is either higher or lower than it actually is. There are two ways that CGM spoofing can be accomplished. First, if the sensor data format is unknown, then a replay attack can be used. In this case, a sensor value read in the past can be re-sent (e.g., by a RF module [27]) to the controller. This would cause the controller unit to indicate an outdated glucose level rather than the actual one. Second, if the format of sensor data is known to hacker, she can send the false data at random time intervals to mislead the controller.

Basal tampering (Blocks $B1 - B5$) The basal tampering attack may be accomplished in two different scenarios. The attacker may issue a command for i) stopping the basal injection (e.g., $basal.rate = 0$) when it is required for patient, or ii) resume the basal injection ($basal.rate > 0$) when it has to be stopped. These attacks may be mounted using a software radio board that fully controls the SAP [27, 34], and transmits the malicious commands to the pump. To accomplish the attack, the attacker needs to spoof the PIN number of the controller, and the format of transmission packets - both of these can be done by an eavesdropping attack.

Table 6: ARTINALI invariants to detect example attacks in OpenAPS.

Attack	Detecting invariant
CGM spoofing	(1) read (t) $\rightleftharpoons$ read (t+5)
Stop basal injection	(2) (120 $\leq$ BG $\leq$ 485) $\rightleftharpoons$ (0.9 $\leq$ basal.rate $\leq$ 3.5) : 1.99, 0.464
Resume basal injection	(3) BG $\leq$ 75 $\rightleftharpoons$ basal.rate=0 : 1.99, 0.464

5.4 Detection of example attacks in SAP

We mounted the attack examples on the SAP system we considered (i.e., OpenAPS), and found that the ARTINALI-based IDS is able to detect all of them. There are four events in the SAP, namely 1) *send(BG)* or sending blood glucose by CGM , 2) *read(BG)* or reading BG by the controller , 3) *send(basal.rate)* or sending basal rate to pump by the controller, and 4) *recv(basal.rate)* or receiving basal rate by pump. We used these events as the basis for mining 51 invariants for OpenAPS's IDS model. Due to space constraints, we do not present all inferred invariants, but only those that detect the example attacks (Table 6).

CGM spoofing attack We selected *read(BG)* in controller as the relevant event, and analyzed the inferred invariants for this event to analyze CGM spoofing attack. Under normal conditions, the transmission of measured Blood Glucose (BG) to CGM occurs at deterministic, periodic times (e.g., every five minutes). This property is represented in our model as time frequency of event *read(BG)*, that is $read(t) \rightleftharpoons read(t+5)$. Using the above property, it would be possible to detect malicious sensor reading from any external source that performs replay attack or transmits wrong data at random time intervals to the controller as the frequency of reading data by controller would change.

Basal tampering attack As previously explained, the basal tampering attack may be accomplished in two different scenarios: i) stop basal injection ($basal.rate = 0$) when it is required, and ii) resume basal injection ($basal.rate > 0$) when it is not required. These attacks break the invariants shown in Table 6. The invariant number (2) indicates that if *BG* is higher than the normal range, the patient needs insulin (i.e., $basal.rate > 0$). However, the *stop insulin injection* attack makes the $basal.rate$ value to be 0, which breaks invariant number (2). Similarly, the invariant number (3) in Table 6 shows that for low BG ranges (e.g., $BG = 45$), the patient does not need insulin (i.e, $basal.rate$ must be 0), but *resume basal injection* attack sends a command ($basal.rate > 0$) to the SAP to inject insulin. As a result, invariant number (3) is violated.

6 EVALUATION

In this section, we present the results of the fault injection experiments to emulate arbitrary attacks, and the overhead measurements. We first present the research questions (RQs) we ask. We then address each of the RQs in a separate sub-section.

6.1 Research Questions (RQs)

RQ1. How do we choose the training set size to obtain the best F-Score(β) for each tool?

RQ2. What is the FN ratio incurred by the IDS using the invariants derived by ARTINALI and the other tools ?

RQ3. What is the FP ratio incurred by the IDS using the invariants derived by ARTINALI and the other tools?

RQ4. What is the memory overhead of the IDS when using the invariants derived by ARTINALI and the other tools?

RQ5. What is the IDS performance overhead when using the invariants derived by ARTINALI and the other tools?

6.2 RQ1. F-Score

As mentioned in Section 4, we obtain two sets of traces from each system, namely *train* and *test*. In this RQ, we ask what should be the optimal training set size for each system in order to maximize the corresponding F-Score values. To answer this question, we obtain a total of 40 training traces, and 50 test traces for each system. We then vary the training set size from 5 to 40, in increments of 5. We then run each of the invariant detection tools including ARTINALI on the same training set to derive invariants. We then measure the FP, FN, and F-Score values (0.5, 1, 2) for each invariant detection tool and system, as a function of the training set size.

Figures 8 and 9 show the distribution of the amount of false positives (FP), false negatives (FN) and the F-Score computed with $\beta = 0.5, 1, 2$ in relation to the amount of training traces, respectively for SEGMeter and OpenAPS, corresponding to each of the four invariant detection tools, including ARTINALI. As expected, as the amount of training traces increases, the FP ratio decreases, since a broader set of invariants are extracted; thus a lower amount of legitimate actions are flagged as potential attacks. A consequence is that more attacks are undetected (FN increases), as a more restricted set of invariants can lead to some attacks being undetected. Overall, an increase in the amount of training traces lead to an increase of the F-Score at first, then it stabilizes, at which point an *optimal* amount of training traces have been found (for a given values of β).

Tables 7 and 8 show the optimal amount of training traces (optimal F-Score) for each invariant detection tool, for SEGMeter and OpenAPS respectively. Recall that we choose F-Score(0.5) for SEGMeter and F-Score(2) for OpenAPS, and hence these are the F-score values we choose for the optimal number of traces. For example, in SEGMeter, a training set size of 20 results in the maximum value of the F-Score(0.5) value of ARTINALI, whereas for OpenAPS, a training set size of 15 results in the maximum value of F-Score(0.5). Likewise, we compute the optimal training set sizes for the three other tools on both platforms. These are the values of the training set sizes we use for deriving the invariants for each tool in the rest of this section. *In other words, we find the best configuration of each tool on each platform, and generate invariants using this configuration for comparing the corresponding IDSes .*

6.3 RQ2. False Negatives

In this section, we compare the variation in the FN ratio incurred by the IDS, using invariants extracted by ARTINALI and the other tools. Tables 7 and 8 also show the FN ratios for each tool for the SEGMeter and OpenAPS systems respectively. We observe that

Table 7: Optimal training set size for maximum F-Score(0.5) for SEGMeter across tools, and the FP and FN ratios.

	Daikon	Texada	Perfume	ARTINALI
F-Score(0.5)	0.721	0.78	0.813	0.898
Num of traces	30	30	35	20
FP (%)	23	15	15	12
FN (%)	57	60	38	2.3

Table 8: Optimal training set size for maximum F-Score(2) for OpenAPS across tools, and the FP and FN ratios.

	Daikon	Texada	Perfume	ARTINALI
F-Score(2)	0.604	0.62	0.686	0.952
Num of traces	30	20	15	15
FP (%)	21	16	22	13.5
FN (%)	61	61	39	2

overall, ARTINALI was able to detect around 97.5% of attacks, which means it has an average FN ratio of 2.5%. In contrast, in Perfume, Texada and Daikon the FN ratio was respectively 38.5%, 60.5% and 59% on average, across the two platforms. *Thus, the ARTINALI-based IDS reduces the ratio of false negatives by 89 to 95% (average 93.4%) over other dynamic invariant detection tools.*

Figure 10 and Figure 11 illustrate the FN ratio of the IDS for the three category of attacks (code mutations), as well as the aggregated FN ratio, for each tool, in both SEGMeter and OpenAPS.

Data mutations: ARTINALI exhibits the lowest FN rate for data mutations (2 to 3%). This is followed by Daikon, which provides a much lower FN ratio in *data mutation* attacks (15% in SEGMEeter and 17% in OpenAPS) than Perfume (53% in SEGMEeter and 78% in OpenAPS) and Texada (52% in SEGMEeter and 87% in OpenAPS). This is because DAIKON focuses on data invariants, while Texada and Perfume do not include data invariants in their model. However, the Daikon data model does not include other properties like ARTINALI does, resulting in much higher FNs than ARTINALI.

Branch flipping: Among the other three tools, ARTINALI has the lowest FN rate for branch flipping attacks (1%). Perfume, Texada and ARTINALI exhibit a lower FN ratio compared to Daikon for branch flipping attacks. As these attacks impact the order and sequence of the events in an execution instance, and Daikon does not have event invariants, it shows less sensitivity.

Artificial delay: Again, ARTINALI has a much lower FN ratio (2-3%) than all three tools for artificial delay attacks, followed by Perfume. This is because they both include time in their model. Nevertheless, Daikon and Texada are still able to detect attacks that impact data variables or alter the execution flow of the program.

Overall, the results support our hypothesis that a more comprehensive invariant model, such as ARTINALI, which can find invariants and their constraints along three dimensions, can detect a significantly larger amount of attacks (and hence has fewer FNs).

6.4 RQ3. False Positives

In this section, we compare the FP ratio incurred by our IDS when using the invariants derived by ARTINALI against the invariants generated by the other tools (Daikon, Perfume and Texada). The results are shown in Table 7 and 8 for the SEGMeter and OpenAPS

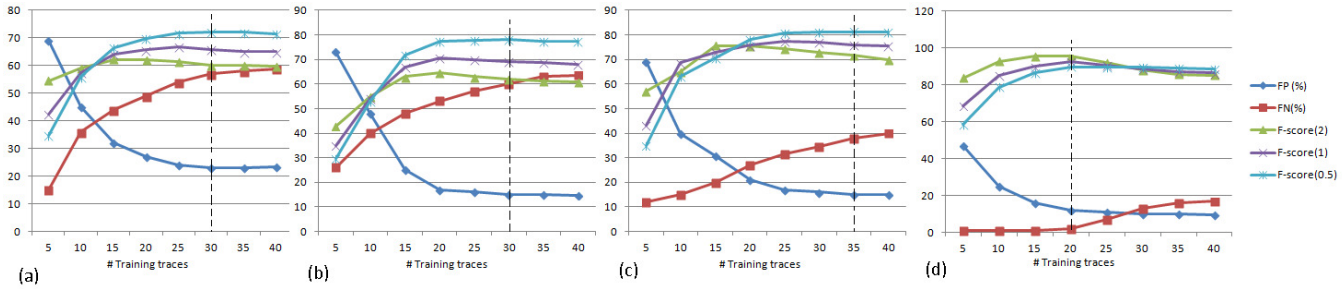

Figure 8: FN, FP and F-Score variations based on number of training traces for SEGMeter across tools; (a) Daikon, (b) Texada, (c) Perfume and (d) ARTINALI. X-axis is the training set size.

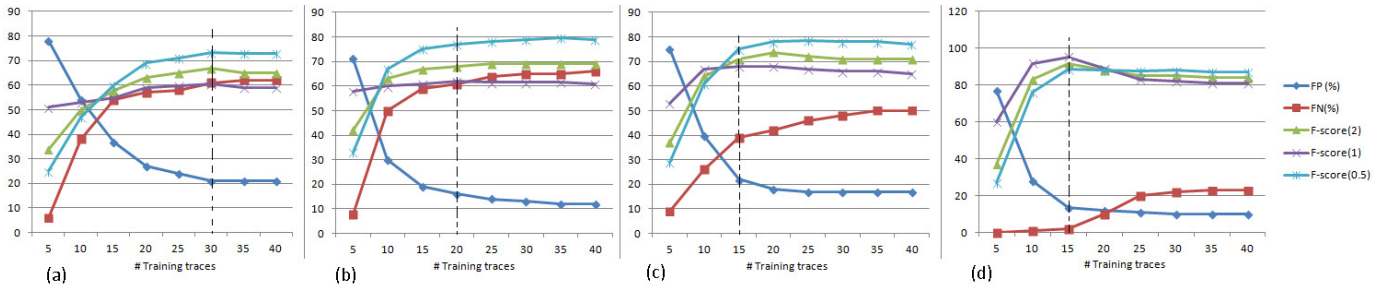

Figure 9: FN, FP and F-Score variations based on number of training traces for OpenAPS across tools; (a) Daikon, (b) Texada, (c) Perfume and (d) ARTINALI. X-axis is the training set size.

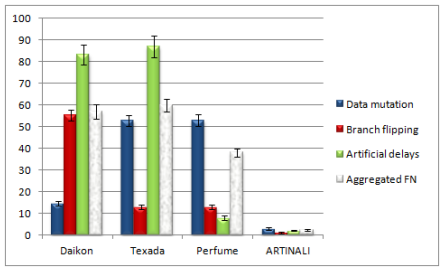

Figure 10: FN(%) of IDS for SEGMeter for different attack types across the tools. Error bars are shown for the 95% confidence interval.

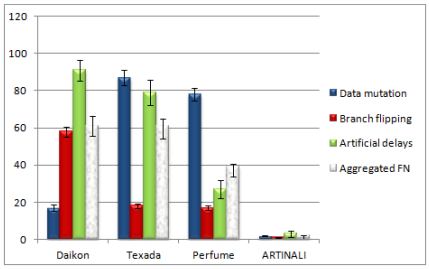

Figure 11: FN(%) of IDS for OpenAPS for different attack types across the tools. Error bars are shown for the 95% confidence interval.

systems respectively. We can observe that in both CPSes, the use of the ARTINALI-generated invariants lead to significantly less false positives compared to the invariants generated by the other tools. More precisely, ARTINALI provides a 20% to 48% improvement of the FP ratio for SEGMeter, a 16% to 39% improvement of the FP ratio for OpenAPS, and averagely 30.75% improvement of the FP ratio for both platforms over the other tools.

These results can be explained by the fact that ARTINALI leverages the correlations among data, event and time dimensions during correct system behavior to generate more stable invariants. ARTINALI infers event invariants that precisely describe the ordering of events in a sequence within an execution flow, and then associates data and time constraints to the events within every path ($D|E$ and $E|T$). Therefore, during normal operation, the system is unlikely to follow the same path with different associated data and time values in a given execution, which in turn, reduces the probability of false positives. Although the IDS uses the same traces for all tools, none of these tools other than ARTINALI look at the relational constraints of both data and time along the events' paths, resulting in a higher ratio of false positives.

While the FP ratio for the ARTINALI-based IDS is lower than the other tools, it is still high for both platforms. To reduce the FP ratio, one can deploy multiple variants of the code and switch to a different variant when an attack is detected. If the invariant is not violated in the second version, it may be a false positive. Another solution is to remove invariants that exhibit high FP ratios [7], but this may also increase the FN ratio.

Table 9: Memory and performance overhead of IDS, seeded by ARTINALI and the other tools, running on SEGMeter.

	Daikon	Texada	Perfume	ARTINALI
Memory usage (MB)	1.24	3.21	3.94	2.96
Tracing overhead(%)	22.6	13.4	18.8	23.3
Detector overhead(%)	4.7	10.3	13.3	8.3
Overall overhead(%)	27.3	23.7	32.08	31.6
Full cycle execution(s)	60.94	60.94	60.94	60.94
IDS execution time(s)	16.63	14.45	19.57	19.25

6.5 RQ4. Memory Overhead

We measured the memory consumption of our IDS running on the SEGMeter platform, using the invariants generated by different tools. We also calculated the number of invariants that ARTINALI and the other tools inferred for both platforms. Our results are shown in Table 9 ("Memory usage" row). Generally, invariants that involve two or more dimensions (e.g., $E|T$ invariants) carry more information than the invariants of one dimension (e.g., event invariants), and hence are more complex. We observe that the memory usage grows as the number and complexity of invariants increases. For example, the IDS consumes the maximum memory usage (3.94 MB) when it uses the Perfume-generated invariants, which straddle two dimensions, and have the maximum number of invariants (158 - Table 2). Overall, we find that the memory consumption of the IDS with ARTINALI-generated invariants is lower than those with Perfume or Texada-generated invariants, but higher than those with Daikon-generated invariants. However, the memory usage for all tools is much lower than the available memory in SEGMeter (16 MB).

6.6 RQ5. Performance Overhead

In this section, we discuss the performance overhead of our IDS running on the SEGMeter platform. Recall that the IDS consists of two components, namely tracing module and intrusion detector module. Table 9 (middle part) shows the overheads of the two modules separately for each tool. Each of these measurements is an average of the overhead of 10 execution traces for each tool, where an execution trace is defined as one complete execution of the meter's main loop. We find that ARTINALI and Perfume have the highest aggregate overhead, followed by Daikon, and then Texada. The difference in the overhead is due to the difference in the tracing module, which needs to collect both event and data/time information for ARTINALI and Perfume, compared with Texada (events only), and Daikon (data only).

In addition to the performance overheads, the IDS execution time should be lower than the execution time of the system's cycle, or else it will be unable to keep up with the system. We measure the raw execution times of a full cycle in Table 9 (last part). As can be seen from the table, the entire cycle takes about 60 seconds (1 minute). However, the execution of the IDS for each tool takes less than 20 seconds even in the worst case (for Perfume), which is only a third of execution time of the full cycle. Therefore, the IDS is not a bottleneck in any of the four systems, and is easily able to keep up with the system.

Note that the invariant mining process takes place offline, and hence does not contribute to the performance overhead of the IDS running on the CPS platform. Nonetheless, we measured the time to mine invariants using ARTINALI, on a standard desktop system (Intel core i7 processor with 32 GB RAM). We found that the time ranges from 8 to 96 seconds in SEGMeter, and from 6 to 36 seconds in OpenAPS. Though this overhead may be higher for larger systems, this process needs to be done only on a code update.

7 DISCUSSION

In this section, we first examine the threats to the validity of our experiments, followed by reflections on ARTINALI's generalizability.

Threats to Validity: An *external threat* to the validity is the limited number of CPS platforms considered (two). However, as we have mentioned, finding CPS platforms that are publicly available and security critical is a challenge. We have attempted to mitigate this threat by choosing two fairly diverse platforms, with different tradeoffs in terms of FP and FN ratios. We acknowledge that these platforms exhibit somewhat simple behaviors - however, many CPSes fall into this category [12]. An *internal threat* to validity is in our evaluation of the efficacy of the invariants for attack detection through fault injection experiments. While not necessarily representative of all security attacks, fault injection allows us to emulate the behavior of potential attackers without biasing the evaluation towards known vulnerabilities (at the time of the evaluation). We have attempted to mitigate this threat by using mutation operators that were used for emulating attacks in prior work [38]. Finally, a *construct threat* to validity is the evaluation metrics used for measuring efficacy. FP and FN ratios have however been used in a lot of prior work on intrusion detection, as have F-scores, and hence we do not believe this is a significant threat. Another potential construct threat is the choice of tools we use for comparing with ARTINALI, but we mitigated this to an extent by first systematically classifying the space of invariant detection techniques, and then choosing the tools in each category.

Generalizability of ARTINALI: ARTINALI relies upon two features, namely *event-based semantics*, and *conditional independence of time and data* (Section 3.2). Events are operations that involve interaction with the outside world. Event-based semantics implies that every event takes place in a unique time frame, and hence, there is no concurrency among event executions. Secondly, ARTINALI assumes an event occurs at a specific time interval, and subsequently, data variables are assigned to specific values. Thus, the time and data corresponding to a particular event, are conditionally independent regardless of the dependency among events. These two features are a common paradigm for CPSes, and hence ARTINALI can be generalized to other CPS platforms.

However, ARTINALI is not applicable to non-CPS platforms for two reasons. First, the non-concurrency of events does not hold in non-CPS platforms such as mobile phones. Secondly, CPS events have limited functionality, and hence inferring invariants for each event is straightforward. Unlike CPSes, in general real-time systems, tasks can be of unbounded complexity. Furthermore, general purpose computers with full preemptive (i.e., non-real-time) operating systems have a large space of behaviors. Therefore, learning invariants for such systems is challenging.

8 CONCLUSION

Cyber-physical systems (CPSes) are becoming increasingly subject to security attacks due to their interconnectedness and relative lack of protection. In this paper, we attempt to use dynamic invariant detection techniques to build intrusion detection systems for CPSes. Our key insight is that time is a first class constraint in CPS systems, and hence we incorporate time into the invariants, in addition to data and events. We devise an efficient algorithm for learning invariants over the three dimensions of data, events and time, and implement it in a tool called ARTINALI. We demonstrate the use of ARTINALI on two CPS platforms for intrusion detection. We find that ARTINALI has significantly lower false negatives and false positives than other dynamic invariant detection tools.

ACKNOWLEDGEMENT

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), and the MITACS.

REFERENCES

[1] 2011. Smart energy groups home page. http://smartenergygroups.com.. (2011).
[2] 2014. Perfume User Manual. http://people.cs.umass.edu/~ohmann/perfume/. (2014).
[3] 2016. Texada User Manual. https://bitbucket.org/bestchai/texada/. (2016).
[4] 2017. ARTINALI Invariant Detector. (2017). https://github.com/karthikp-ubc/Artinali
[5] 2017. The Daikon Invariant Detector User Manual. https://plse.cs.washington.edu/daikon/download/doc/daikon.html. (2017).
[6] Maryam Raiyat Aliabadi and Karthik Pattabiraman. 2016. FIDL: A Fault Injection Description Language for Compiler-Based SFI Tools. In *International Conference on Computer Safety, Reliability, and Security*. Springer, 12–23.
[7] Leonardo Aniello, Claudio Ciccotelli, Marcello Cinque, Flavio Frattini, Leonardo Querzoni, and Stefano Russo. 2016. Automatic Invariant Selection for Online Anomaly Detection. In *International Conference on Computer Safety, Reliability, and Security*. Springer, 172–183.
[8] Robin Berthier, William H Sanders, and Himanshu Khurana. 2010. Intrusion detection for advanced metering infrastructures: Requirements and architectural directions. In *Smart Grid Communications (SmartGridComm), 2010 First IEEE International Conference on*. IEEE, 350–355.
[9] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, and others. 2011. Comprehensive Experimental Analyses of Automotive Attack Surfaces.. In *USENIX Security Symposium*. San Francisco.
[10] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy: Dynamic symbolic execution for invariant inference. In *Proceedings of the 30th international conference on Software engineering*. ACM, 281–290.
[11] Barthélémy Dagenais and Martin P Robillard. 2010. Creating and evolving developer documentation: understanding the decisions of open source contributors. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 127–136.
[12] Patricia Derler, Edward A Lee, and Alberto Sangiovanni Vincentelli. 2012. Modeling cyber–physical systems. *Proc. IEEE* 100, 1 (2012), 13–28.
[13] John Eidson, Edward A Lee, Slobodan Matic, Sanjit A Seshia, and Jia Zou. 2010. A time-centric model for cyber-physical applications. In *Workshop on Model Based Architecting and Construction of Embedded Systems (ACES-MB)*. 21–35.
[14] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27, 2 (2001), 99–123.
[15] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2016. Security analysis of emerging smart home applications. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 636–654.
[16] Mark Gabel and Zhendong Su. 2008. Symbolic mining of temporal specifications. In *Proceedings of the 30th international conference on Software engineering*. ACM, 51–60.
[17] Gösta Grahne and Jianfei Zhu. 2005. Fast algorithms for frequent itemset mining using fp-trees. *IEEE transactions on knowledge and data engineering* 17, 10 (2005), 1347–1362.
[18] Ahmad Y Javaid, Weiqing Sun, Vijay K Devabhaktuni, and Mansoor Alam. 2012. Cyber security threat analysis and modeling of an unmanned aerial vehicle system. In *Homeland Security (HST), 2012 IEEE Conference on Technologies for*. IEEE, 585–590.
[19] Hengle Jiang, Sebastian Elbaum, and Carrick Detweiler. 2013. Reducing failure rates of robotic systems though inferred invariants monitoring. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*. IEEE, 1899–1906.
[20] Hermann Kopetz and Günther Bauer. 2003. The time-triggered architecture. *Proc. IEEE* 91, 1 (2003), 112–126.
[21] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and others. 2010. Experimental security analysis of a modern automobile. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 447–462.
[22] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. 2006. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 161–176.
[23] Neal Leavitt. 2010. Researchers fight to keep implanted medical devices safe from hackers. *Computer* 43, 8 (2010), 11–14.
[24] Caroline Lemieux. 2015. Mining temporal properties of data invariants. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 751–753.
[25] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. 2015. General LTL Specification Mining (T). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 81–92.
[26] Dana Lewis. 2015. Introducing the# OpenAPS project. (2015).
[27] Chunxiao Li, Anand Raghunathan, and Niraj K Jha. 2011. Hijacking an insulin pump: Security attacks and defenses for a diabetes therapy system. In *e-Health Networking Applications and Services (Healthcom), 2011 13th IEEE International Conference on*. IEEE, 150–156.
[28] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. 2008. Automatic generation of software behavioral models. In *Proceedings of the 30th international conference on Software engineering*. ACM, 501–510.
[29] Stephen McLaughlin, Dmitry Podkuiko, Sergei Miadzvezhanka, Adam Delozier, and Patrick McDaniel. 2010. Multi-vendor penetration testing in the advanced metering infrastructure. In *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 107–116.
[30] Robert Mitchell and Ing-Ray Chen. 2014. A survey of intrusion detection techniques for cyber-physical systems. *ACM Computing Surveys (CSUR)* 46, 4 (2014), 55.
[31] Gail C Murphy, David Notkin, and Kevin Sullivan. 1995. Software reflexion models: Bridging the gap between source and high-level models. *ACM SIGSOFT Software Engineering Notes* 20, 4 (1995), 18–28.
[32] Tony Ohmann, Michael Herzberg, Sebastian Fiss, Armand Halbert, Marc Palyart, Ivan Beschastnikh, and Yuriy Brun. 2014. Behavioral resource-aware model inference. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 19–30.
[33] Antonio Pecchia, Stefano Russo, and Santonu Sarkar. 2017. Assessing Invariant Mining Techniques for Cloud-based Utility Computing Systems. *IEEE Transactions on Services Computing* (2017).
[34] Jerome Radcliffe. 2011. Hacking medical devices for fun and insulin: Breaking the human SCADA system. In *Black Hat Conference presentation slides*, Vol. 2011.
[35] Florian Skopik, Zhendong Ma, Thomas Bleier, and Helmut Grüneis. 2012. A survey on threats and vulnerabilities in smart metering infrastructures. *International Journal of Smart Grid and Clean Energy* 1, 1 (2012), 22–28.
[36] Sean W Smith. 2009. Security and privacy challenges in the smart grid. (2009).
[37] Marina Sokolova, Nathalie Japkowicz, and Stan Szpakowicz. 2006. Beyond accuracy, F-score and ROC: a family of discriminant measures for performance evaluation. In *Australasian Joint Conference on Artificial Intelligence*. Springer, 1015–1021.
[38] Farid Molazem Tabrizi and Karthik Pattabiraman. 2015. Flexible intrusion detection systems for memory-constrained embedded systems. In *Dependable Computing Conference (EDCC), 2015 Eleventh European*. IEEE, 1–12.
[39] Farid Molazem Tabrizi and Karthik Pattabiraman. 2016. Formal security analysis of smart embedded systems. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 1–15.
[40] Carolyn Talcott. 2008. Cyber-physical systems and events. In *Software-Intensive Systems and New Computing Paradigms*. Springer, 101–115.
[41] Joachim Wegener and Matthias Grochtmann. 1998. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems* 15, 3 (1998), 275–298.
[42] Westley Weimer and George C Necula. 2005. Mining temporal specifications for error detection. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 461–476.
[43] Ye Yan, Yi Qian, Hamid Sharif, and David Tipper. 2012. A survey on cyber security for smart grid communications. *IEEE Communications Surveys & Tutorials* 14, 4 (2012), 998–1010.
[44] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of the 28th international conference on Software engineering*. ACM, 282–291.