The Journal of Systems & Software 180 (2021) 111016

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

ARTINALI++: Multi-dimensional Specification Mining for Complex Cyber-Physical System Security

Maryam Raiyat Aliabadi *, Mojtaba Vahidi Asl *, Ramak Ghavamizadeh

Faculty of Computer Science and Engineering, Shahid Beheshti University, Tehran, Iran

ARTICLE INFO

Article history:
Received 13 April 2020
Received in revised form 19 February 2021
Accepted 20 May 2021
Available online 8 June 2021

Keywords:
Program analysis
Specification mining
Intrusion Detection Systems
Cyber-Physical Systems
Security
Safety

ABSTRACT

Cyber-Physical Systems (CPSes) have been investigated as a key area of research since they are the core of Internet of Things. CPSs integrate computing and communication with control and monitoring of entities in the physical world. Due to the tight coupling of cyber and physical domains, and to the possible catastrophic consequences of the malicious attacks on critical infrastructures, security is one of the key concerns. However, the exponential growth of IoT has led to deployment of CPSes without support for enforcing important security properties. Specification-based Intrusion Detection Systems (IDS) have been shown to be effective for securing these systems. Mining the specifications of CPSes by experts is a cumbersome and error-prone task. Therefore, it is essential to dynamically monitor the CPS to learn its common behaviors and formulate specifications for detecting malicious bugs and security attacks. Existing solutions for specification mining only combine data and events, but not time. However, time is a semantic property in CPS systems, and hence incorporating time in addition to data and events, is essential for obtaining high accuracy.

This paper proposes ARTINALI++, which dynamically mines specifications in CPS systems with *arbitrary size and complexity*. ARTINALI++ captures the security properties by incorporating time as a substantial property of the system, and generate a multi-dimensional model for the general CPS systems. Moreover, it enhances the model through discovering invariants that represent the *physical motions* and *distinct operational modes* in complex CPS systems. We build Intrusion Detection Systems based on ARTINALI++ for three CPSes with various levels of complexity including smart meter, smart artificial pancreas and unmanned aerial vehicle, and measure their detection accuracy. We find that the ARTINALI++ significantly reduces the ratio of false positives and false negatives by 23.45% and 73.6% on average, respectively, over other dynamic specification mining tools on the three CPS platforms.

1. Introduction

CPSes have been investigated as a key area of research since they are the core of Internet of Things. CPSs integrate computing and communication with control and monitoring of entities in the physical world (Liu et al., 2017). Recently, they are increasingly deployed in many security-critical contexts such as smart medical devices (Leavitt, 2010; Radcliffe, 2011), surgical robots (Alemzadeh et al., 2016), smart grid (Skopik et al., 2012), smart cars (Checkoway et al., 2011), and Unmanned Aerial Vehicles (UAVs) (Javaid et al., 2012). Unfortunately, the pervasiveness and network accessibility of these systems and their relative lack of security measures make them attractive targets for attackers. Many attacks have been demonstrated against CPSes such as self-driving cars (Koscher et al., 2010), smart medical devices (Leavitt, 2010; Li et al., 2011), smart meters (Smith, 2009) and unmanned aerial vehicles (UAVes) (Samland et al., 2012).

Intrusion Detection Systems (IDSes) are used to monitor computer systems and detect security attacks. Typical IDSes fall into two major categories: *Signature-based*, and *behavior-based*. Signature-based IDSes identify attacks by matching real-time behavior against predefined *signatures* that model malicious activity. As signature-based IDSes rely on known *attack models (signatures)* they cannot detect unknown attacks (Mitchell and Chen, 2014). This is significantly important for CPSes since they are working autonomously for long periods of time, and hence are difficult to be interrupted for frequently patching or upgrading in the field. In contrast, behavior-based (or specification-based) systems use a *system model* to compare with suspicious behaviors. They watch a system's dynamic execution to identify suspect behavior and are able to detect both known and unknown attacks. As a result, specification based IDSes are proposed as the best fit for CPS security (Berthier et al., 2010; Goh et al., 2017; Bartocci et al., 2018). These techniques build a behavioral model for

* Corresponding authors.
E-mail addresses: m_raiyataliabadi@sbu.ac.ir (M.R. Aliabadi), mo_vahidi@sbu.ac.ir (M.V. Asl), r-ghavami@sbu.ac.ir (R. Ghavamizadeh).

https://doi.org/10.1016/j.jss.2021.111016
0164-1212/

system by defining a set of rules known as *invariants* or *mined specifications*. Invariants can be either discovered by *static analysis* or *dynamic analysis* of the program.

The static analysis-based specification mining techniques build a model for a system based on its code. These techniques are inherently conservative with low false positives, as they only generate specifications that are rigorously provable. However, merely considering code does not provide enough information about the real-time behavior of the system in its operational environment, which in turn, results in high false negatives. Furthermore, these techniques generate a big model, leading to overheads, often exceeding the resource constraints of a CPS. In contrast, dynamic analysis-based techniques provide an alternative way to understand the system by observing the run-time behavior. They log the key points of the program to peek into the actual program behavior at run-time, and infer a set of *likely invariants*.

There has been a significant amount of work on using dynamic analysis to infer likely invariants for program understanding, formal verification, test oracle, debugging and intrusion detection (Lemieux et al., 2015a; Thummalapenta and Xie, 2009; Carreon et al., 2019; Acharya et al., 2007; Bian et al., 2018; Chang et al., 2007; Kang et al., 2016; Li and Zhou, 2005; Liang et al., 2016; Livshits and Zimmermann, 2005; Lu et al., 2007; Tan et al., 2008; Ernst et al., 2007; Beschastnikh et al., 2015; Abrahamson et al., 2014; Yang et al., 2006; Lo and Maoz, 2008). These techniques mine execution traces of the system for learning invariants on the data values of the program, the events or both. However, we find that most of these techniques generate a system model that is neither sound nor complete, which makes them significantly challenging to deploy in the context of an IDS for CPS systems.

This paper introduces ARTINALI++ (A Real-Time-specific Invariant iNference ALgorIthm) for mining specifications through dynamic analysis in CPS systems with arbitrary size and complexity, for specification-based IDSes. The fundamental innovation in ARTINLI++ is that it creates a time-oriented model incorporating time specifications to the data and event specifications. This is necessary from two perspectives: First, CPSes interact with physical environment in a real-time fashion, and they have real-time constraints. Therefore, their operational correctness depends on both logical correctness, and correct timing behavior (Wegener and Grochtmann, 1998; Kopetz and Bauer, 2003). Hence, it is essential to incorporate time into the model for detecting many relevant security attacks in these systems. Secondly, as CPSes have predictable timing behaviors to a first order of approximation, exploiting this predictability leads to a higher IDS accuracy.

However, *complex CPSes* such as autonomous vehicles have specific intrinsic properties that need to be captured when building a system model. We define complexity as (i) having continuous movement in physical time and space, and (ii) working in multiple operational states. These two properties add new aspects to the system behavior, that are important from security perspective, and are addressed by ARTINALI++.

To the best of our knowledge, ARTINALI++ is the first dynamic specification mining technique that mines specifications along the four dimensions of data, event, time and physical motions for CPS systems having arbitrary size and complexity, and uses the mined specifications for intrusion detection. This paper is a substantially expanded and revised version of our paper published in FSE'2017 (Aliabadi et al., 2017). Our contributions are:

- We designed ARTINALI++, a technique that generates a multi-dimensional model for CPSes by mining invariants along the data, event, time and physical motion dimensions (Sections 4 and 5).
- We built an IDS prototype,[1] and used it in the context of three CPS systems with various levels of complexity, namely (i) advanced metering infrastructures, (ii) smart artificial pancreas and (iii) unmanned aerial vehicles (Section 6). We evaluated the efficacy of ARTINALI++ for 8 known attacks on the three CPS platforms. We find that ARTINALI++ is able to detect all 8 attacks (Section 7).
- We also evaluated our IDS prototype on the three platforms, and compared it with several existing state of the art dynamic specification mining techniques. Overall, we observe that the IDS exhibits significantly low false-negatives and false-positives for arbitrary attacks emulated by fault injection (Section 8). Furthermore, it incurs reasonable memory and performance overheads on the examined CPS systems.

We organize the rest of the paper in this way. Section 2 reviews the literature of attack detection techniques. In Section 3, we characterize the attack surface of a CPS, and introduce our attack model, accordingly. In Sections 4 and 5, we present ARTINALI++ technique for non-complex and complex CPS platforms. Section 6 introduces our case studies, followed by our experimental procedure. Finally, we explain the evaluation of our techniques against targeted and arbitrary attacks in Sections 7 and 8, respectively.

2. Related work

Below, we discuss the existing techniques for performing attack detection, and their limitations.

A. CPS security solutions: During recent years, CPS security has got a lot of attention due to ubiquity and criticality of these systems. Prior work proposes techniques for modeling threats associated with CPS systems, as well as detection of certain categories of attacks at run-time. Goh et al. (2017) propose an unsupervised learning approach for anomaly detection in the area of CPS. They evaluated their model on a water treatment plant using a set of known attacks. Carreon et al. (2019) model the normal timing for operations in software applications using cumulative distribution functions of timing sub-component within sliding execution windows. They presented a probabilistic formulation for estimating the presence of malware for individual operations by monitoring the internal timing of the different components of the system, and evaluated their model on a pacemaker using three malware scenarios. Deng et al. (2019) presents an anomaly-based IDS based on the fuzzy c-means clustering algorithm and the PCA algorithm for IoT networks. The PCA algorithm is used for feature extraction and reduction, which leads to a fast light weight IDS. Yoon et al. (2017) proposes a method for detecting anomalous executions using a distribution of system call frequencies in CPS systems. They use a cluster analysis to learn the legitimate execution contexts of CPS applications and then monitor them at run-time to capture abnormal execution paths. However, these approaches do not fully utilize the intrinsic code properties of CPS systems for attack detection. Yoon et al. (2013) propose a hardware-based IDS for CPS 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 is limited to the systems equipped with a multicore processor as well as a Timing Trace Module (TTM).

B. Artificial intelligence: During recent years, artificial intelligence, particularly, Deep Learning (DL) techniques are being widely used for detecting and classifying cyber attacks (Kim et al.,

[1] We upgraded our previously developed IDS (Aliabadi et al., 2017) to support ARTINALI++ features.

2016; Chawla et al., 2018; Chen et al., 2018; Aghakhani et al., 2018; An et al., 2019). For example, An et al. (2019) presents a deep reinforcement learning scheme to defend smart grid against data integrity attacks. Chen et al. (2018) presents HeNet, a deep learning approach to classify fine-grained control flow traces for malware detection. HeNet achieves high accuracy on detecting Return Oriented Programming (ROP) attacks against AdobeR Reader. An LSTM-based technique for anomaly detection was introduced in Kim et al. (2016). They capture the semantic meaning of each system call and its relation to other system calls. Moreover, they proposed an ensemble method that can better fit to IDS design by focusing on lowering false alarm rates. Previous work show that DL techniques are very accurate for modeling the behavior of large and complex software and detecting unknown attacks. However, as stated in Tange et al. (2019), Mohammadi et al. (2018) and Chalapathy and Chawla (2019), the adversary characteristics of CPS systems and DL techniques make their integration challenging. Therefore, CPSes can rarely host DL models due to their resource constraints.

C. Provenance: Provenance is a metadata describing the complete lineage of data and processes chain. Provenance is being beneficial in auditing, debugging, and forensics investigation. There is a considerable amount of work on provenance-based intrusion detection techniques. Nonetheless, the fusion of provenance with CPS security has not been completely explored yet (Suhail et al., 2016). Han et al. (2020) presents UNICORN, a host-based anomaly detection system that leverages whole-system data provenance to detect advanced persistent threats. UNICORN explores provenance graphs that provide rich contextual and historical information to identify stealthy anomalous activities with high accuracy. FRAPpuccino (Han et al., 2017) is another provenance-based approach for detecting unusual behavior in programs running on PaaS clouds. It uses a windowing approach to allow for efficient graph analysis. Palyvos-Giannas et al. (2018) presents GeneaLog, a fine-grained data provenance technique for data streaming applications. GeneaLog takes advantage of cross-layer properties of the software stack and incurs a minimal, constant size per-tuple overhead. To keep data traces of CPS systems, provenance can play a vital role as it solves many issues related to data trustworthiness, decision-making, data reconciliation and data replication. However, fine-grained data provenance is an intrinsically heavy operation that impose a notable time and space overheads, and processing costs. It is possibly afforded by high-end servers, but can be prohibitive for the resource-constrained CPSes.

D. Static analysis: Static analysis includes a family of techniques that analyze the source code to model the correct behavior of the system. For example, Späth et al. (2019) presented synchronized pushdown systems (SPDS), a new concept to context-sensitive, flow-sensitive, and field-sensitive data-flow analysis, and showed how this representation discover security vulnerabilities due to misusing Crypto APIs in Android apps and Maven Central repositories. Similarly, Wagner and Dean extract automaton model from source code for their FSM-based intrusion detection system (Wagner and Dean, 2001). They proposed an approach to build a model of the software system based on the system call-graph. They introduced a non-deterministic pushdown automaton (NDPDA), which builds an extensive model of the software system based on the system calls. NDPDA results in an improved accuracy. However, it is slow because of a high memory overhead. Giffin et al. (2004) proposed the Dyck model based on static analysis of the software and its system calls. In this work, they added context sensitivity to the model, removing some of the false negatives that inherently exists in the static analysis techniques. While this improves the accuracy, it results in increasing the size of the model noticeably. Static analysis-based techniques are inherently conservative with low false positives. However, these approaches generate a big model for complex CPS systems, often exceeding the resource constraints of the CPS. Moreover, as these techniques analyze the program without executing the program, they cannot provide adequate information about the real-time behavior of CPS system in its operational environment, which in turn, leads to high false negatives.

E. Dynamic analysis: 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, (iv) time dependencies of events and (v) data-event-time invariant inference. Fig. 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 (Ernst et al., 2001), and falls into the first class of techniques. DIDUCE (Hangal and Lam, 2002) is a dynamic invariant detection technique, that combines data invariant detection and checking in a single tool for fault diagnosis purposes. It is not only able to dynamically monitor and check the software, but also scales dynamic invariant detection to large programs. DySy (Csallner et al., 2008) is another example of this class, which combines the advantages of dynamic invariant inference using Daikon and static analysis using symbolic execution, that results in inferring more accurate invariant set than that of Daikon. All three tools can be placed on the *data* axis as they produce a model for data constraints without taking into account the events or timing of the system.

The second class captures the sequence of events within progam's execution paths via tracking dynamic traces, which are mostly represented in the form of *two-event temporal specifications*, or *finite state automata*. For example, Texada (Lemieux et al., 2015b) derives temporal logic propositions, and captures sequences of events via tracking dynamic traces. It mines traces for two-event rules of the form $G(a \Rightarrow XF(b))$, in which G, X, and F are Linear Temporal Logic (LTL) operators. Another relevant example is Yang et al. (2006), that captures the sequence of events within program's execution paths by inferring finite state machines via 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 (Lorenzoli et al., 2008). It represents sequences of method invocations that are annotated with data, and is hence limited to classifying data invariants that arise among method calls. Grant et al. (2018) introduced Dinv for inferring likely invariants, between variables at different nodes in a distributed system, and for checking these distributed invariants at runtime. Quarry finds data invariants at each program point, and then finds temporal relationships between the invariants (Lemieux, 2015). Neither technique considers timing information, however.

The fourth class consists of a single technique, Perfume, which is a specification mining tool designed for modeling system properties based on resource (time and storage) consumption (Ohmann et al., 2014). It generates an integration model of event relations and their time constraints. Although Perfume considers time as a part of the model, it does not consider the relationship between data and time.

The first four classes of dynamic specification mining techniques are useful for modeling the one or two aspects of a CPS. ARTINALI (Aliabadi et al., 2017), that is a single member of the fifth class of dynamic specification mining techniques captures

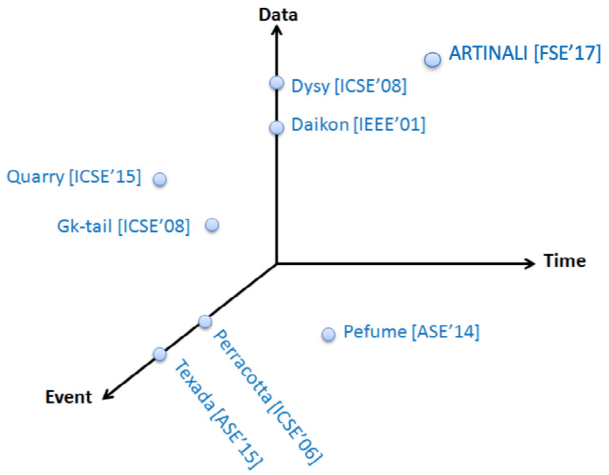

Fig. 1. Scope of dynamic invariant detection techniques.

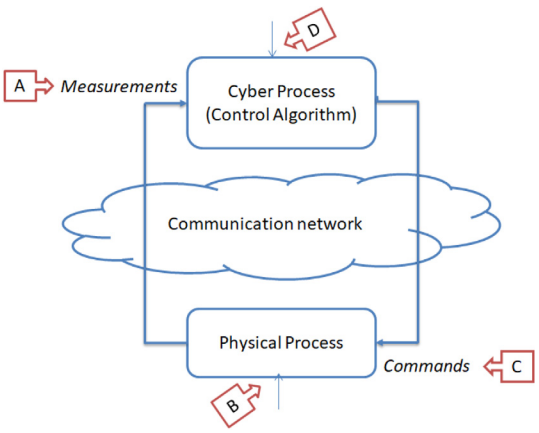

Fig. 2. CPS attack surface.

the system constraints over three dimensions of data, event and time, leading to a considerable increase in IDS accuracy. Using the inferred 3D model, ARTINALI indicated over 98% true positives on the faults injected into the control programs of two classes of smart systems. However, ARTINALI has been developed for non-complex CPS platforms, thus is not able to model the mobility aspects and distinct operational modes of complex CPS platforms.

Overall, the current specification mining techniques are useful for modeling one, two or three important dimensions of a system behavior. However, none of them consider the interplay among four dimensions of time, event, data and physical motions in formulating specifications, which we believe is a significant characteristic of complex CPS systems.

3. Characterizing the attack surface of a CPS

A CPS consists of a cyber unit (i.e., control program), and a physical unit connected by a communication channel. CPS's control programs keep collecting sensed data from physical environment, making decisions about the next state of physical process, and then issuing actuation commands to cope with the sensed data changes (Nuzzo, 2019). The sensing, decision-making, and actuation steps form a control loop which involves interactions between the cyber and physical domains in real-time, and each pass of such a control loop is called an *iteration*. As the interaction between the physical and the cyber domain increases, the physical unit becomes more susceptible to the software vulnerabilities/bugs that might exist in the cyber unit. Therefore, the safety and security of the overall system strongly depends on the CPS' control program.

3.1. Attack entry points

We have characterized the attack surface of a CPS, and provided a complementary model for attacker entry points presented by previous work (Lin et al., 2016; Kwon et al., 2013; Cardenas et al., 2008). According to Fig. 2, there are four likely entry points (marked as *A*, *B*, *C*, *D*) for attackers to penetrate the system. Type *A* attacks can be used to hide the real state of the physical process in order to trick the controller program to make a wrong decision about the next state of physical process, and to delay the detection of the attacks before the actual damage to the system (like what happened in *Stuxnet* (Chien et al., 2010)). Type *C* attacks can disrupt the system by directly compromising the control commands that are issued by controller. These attacks can jam the communication channel, or compromise the routing protocol. One example of this type is *SkyJack*, which hijacks other flying drones through de-authenticating the original controller (Kamkar, 2013). Type *B* include attacks that exploit the vulnerabilities in physical components. For example, Son et al. (2015) developed a way for attacking drones equipped with vulnerable MEMS gyroscopes using intentional sound noise causing drones to loose control and crash. Type *D* attacks aim to exploit the vulnerabilities in cyber unit to take over the dynamics of physical process. For instance, a recent work (Stevens et al., 2017) investigated an attack to smart facial recognition systems caused by exploiting a mis-classification bug (CVE-2016-1516) in the controller algorithm.

3.2. Attack model

We assume the adversarial goal is to compromise the functionality of CPS's control program. This means that she either prevents a vital functionality of the CPS from being executed (e.g., power consumption data not being sent to the utility server in smart meter, or location coordinates not being provided by a drone's GPS), or functionalities being executed improperly (e.g., insulin injection is resumed when it must be stopped in insulin pump, or attempt to brake leads to acceleration execution in a smart car). The first group of attacks (e.g. DoS attacks) targets the availability properties, while the second group (such as deception attacks) targets the integrity properties of the CPS. We do not consider those threats that compromise the privacy/confidentiality properties of the CPS in our model.

Accordingly, we assume that adversary is capable to penetrate into the CPS from one of the entry points *A*, *C* or *D*, and take deception and/or DoS attacks.

4. Approach

In this section, we introduce the security model that ARTINALI++ uses, and we explain it's 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.

4.1. Multi-dimensional model

In this section, we bring our definition for a multi-dimensional model and different classes of specifications. We model a CPS in four following dimensions:

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.

Motion refers to the physical movement of the system in time and space during run-time.

We model a CPS by inferring the set of *invariants* to be preserved during run time. An *invariant*, or interchangeably a *mined specification*, is a logical condition that we observe to be true across a set of dynamic execution traces. Corresponding to the dimensions we defined, we introduce eight main classes of invariants that form the basis of our CPS model.

- "*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, or the data invariant as a function of time.
- "*Derivative (DV) invariant*" captures the relational constraints over data, event, time and physical motion. It represents the boundaries of system continuous movements over time.
- "*State-aware invariant*" captures the relational constraints of any of the above categories of invariants over each operational state.

4.2. Event-data-time interplay

An event in a CPS is defined as an instance of an action that results in changing of a condition (Talcott, 2008) (e.g., glucose reading in smart medical devices, or activating insulin injection). Events have three important properties. First, they basically reflect interactions among system modules rather than internal states. The second property is that events are separated in space and time (Talcott, 2008; Derler et al., 2012; Eidson et al., 2010). Thirdly, the locations in the CPS code in which events are triggered are usually *system calls* that are reachable by attackers. From a security point of view, events are significant as they play the role of input channels for unwanted communications with the CPS. For example, those locations in which a new sensing data is read, or actuation commands are sent to control the physical components, are vulnerable to *spoofing attacks* (Fernandes et al., 2016).

It is challenging to find a direct relationship between data and time from both *learning* and *detection* perspectives. As time is a continuous phenomenon, we are not able to identify a sharp time for changing in data values or transitioning states of the system; instead, a valid time interval has to be learned. Variations in execution time might be sourced by various input sets or different execution flows, as well as malicious activities. Therefore, the invariant detection technique should learn both normal and abnormal time variations of the system. In other words, the IDS should be able to distinguish legal time variations from any time deviation that is a sign of an intrusion. To overcome these challenges, we exploit the event-based semantic of a CPS, where every event takes place in an unique time interval, and discretize the *time* by the *events* to learn the invariants. We first obtain the relationship between data and event dimensions to produce $D|E$ invariants, that integrate event information with constraints on data values. In the second step, we discover the relational constraints between time and event dimensions to compute the time boundaries of events, either in relation to each other ($E|T$ invariants), or independently (*time* invariants). In the last step, we integrate $D|E$ and $E|T$ invariants to infer $D|T$ invariants.

In the following mathematical derivation, we illustrate how we use the conditional probability of *data* D given *event* E invariant ($P(D|E)$), and the conditional probability of *event* E given *time* T invariant ($P(E|T)$) to derive the $D|T$ invariants.

As data D, event E and time T are considered as random variables, we expresses the joint probability distribution of variables D, E and T in Eqs. (1) and (2). We then derive Eq. (3) from these two equations to expresses the probability of 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 Eq. (4), we formalize $P(D|T)$ in Eq. (5) as the sum of the probabilities of data D and event E_j given time T *for all events* E_j. Then, using Eq. (3), we rewrite Eq. (5) as Eq. (6).

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

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

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

For example, considering that at time T, event E_j happens; and that upon E_j occurring, then variable D gets assigned specific value(s). This relationship indicates that D is the effect of E_j, and that T is the cause of E_j. As a result, data variable D is *conditionally independent* of time variable T given event E_j, which leads to the conclusion that D and T are conditionally independent (i.e., $P(D|E_j, T) = P(D|E_j)$. Therefore, we simplify the formulation of $P(D|T)$ as follows:

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

The event-based semantics of CPS implies that two or more events cannot take place at the same time T; i.e., $P(E_j|T) > 0 \Rightarrow P(E_i|T) = 0, \forall E_i \neq E_j$. Using the above assumption, we rewrite Eq. (7) to obtain Eq. (8), which is simplified to Eq. (9). Eq. (9) captures the relationship between data D and time T by exploiting the relational constraints of both time and data *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, *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 for a given event E_j.*

4.3. ARTINALI++ workflow

ARTINALI++ is a *dynamic specification mining* technique that generates models of dynamic system behavior, and proposes a *multi-dimensional* model. Fig. 3(a) and (b) show the key blocks

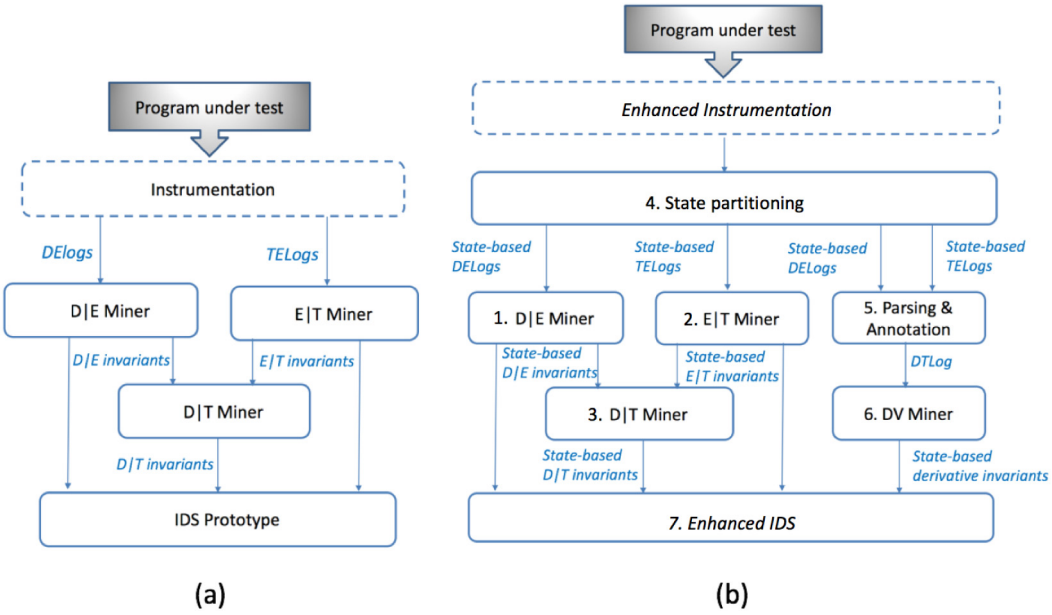

Fig. 3. Workflow of (a) ARTINALI (Aliabadi et al., 2017) and (b) ARTINALI++. ARTINALI is our previously developed technique, that mines specifications for non-complex CPS systems, and ARTINALI++ is the extended version of ARTINALI to support complex CPS systems.

of ARTINALI's (Aliabadi et al., 2017) and ARTINALI++'s workflows, respectively.

ARTINALI++ inherits three building blocks from ARTINALI including *1. D|E Miner*, *2. E|T Miner* and *3. D|T Miner*. We added new features to ARTINALI++ through designing of three new building blocks such as *4. State partitioning*, *5. Parsing and Annotation*, and *6. DV Miner*, and we upgraded *Instrumentation* and *IDS prototype* building blocks of ARTINALI to *Enhanced instrumentation* and *7. Enhanced IDS* in ARTINALI++, respectively, to support new features. In the following, we explain each building block.

4.4. Instrumentation

ARTINALI++ technique mines specifications at the granularity of *events* for non-complex CPSes. We instrument events and their associated data variables to collect logs. As we use the CPS model in the context of attack detection, we capture all system calls (which are accessible by attackers in our attack model) as events. However, events are flexibly user-defined in ARTINALI++ technique. Consequently, the user is able to optionally customize the level of granularity by selecting another type of events, or narrow down the space of events by specifying only the *critical* system calls based on the security preferences.

We instrument the pre-location and post-locations of events by inserting calls to the *ARTINALI++ API functions* that we developed for collecting logs. These functions collect *data* and *time* information associated with the instrumented events in separate log files (*DELog*, *TELog* and *DurationLog*) during the runtime. We collect the raw information about the events and the related data variables' assignments in *DELog* file. Similarly, *TELog* collects the raw information about the events and their timestamps. *DurationLog*, however, records the time before and after event occurrences. The logged information is used as the basis for mining specifications.

4.4.1. Block 1: D|E Miner

The *D|E* Miner learns invariants about the variable values, and how these values relate to a specific event in the system. It uses a three-step process to mine the data invariants that hold true upon events. In the first step, the *D|E* Miner takes the DELog information, and groups them within each trace into distinct classes, that are labeled with the events. The grouping is done to find the data values related to each event. Then, it merges classes across each training trace. Every resulted class includes the valid distribution of data values for a specific event within every execution trace. Secondly, using the Association Rules mining algorithm (Grahne and Zhu, 2005), it merges the data variables within all execution traces while calculating the level of the *confidence* and *support* for every variable within each class. In ARTINALI++, we define *Support* as the fraction of traces in which the variable x within class E_j is seen, and *confidence* as the fraction of supported classes, in which variable x is assigned to the same value(s). For example, if variable x is assigned to 1 in three out of four execution traces, the inferred rule, $x = 1$, will have 100% support and 75% confidence.

In the last step, the *D|E* Miner mines the data invariants, that are associated with each event (class). As *D|E* invariants hold true within the same observed event (at the same time), they are called *multi-propositional* data invariants. We state *D|E* invariants in the following form:

$$E_i : d_1 = [\alpha_1, \ldots, \alpha_p], d_2 = [\beta_1, \ldots, \beta_q], \ldots, d_n = [\sigma_1, \ldots, \sigma_k]$$

Where E_i denotes the name of $(i)th$ event, d_1 to d_n denote the name of data variables, and $[\alpha_1, \ldots, \alpha_p]$ to $[\sigma_1, \ldots, \sigma_k]$ denote the range of valid values of n data variables mapping to the event E_i.

We have chosen the above *D|E* invariant template as a common data invariant template. However, ARTINALI++'s *D|E* miner is extensible to all templates that Daikon uses for data invariant inference. We avoid using all Daikon templates for three reasons: First, data invariants inferred using various templates are overlapped (e.g., $2 \leq X \leq 5$, $Y \geq 5$, $Y \geq X$). Secondly, the larger number of invariants may lead to a higher rate of false positives in anomaly detection (Aniello et al., 2016). Thirdly, CPS has a limited memory capacity which makes a big model challenging to deploy for attack detection. Thus, a smaller set of rich and stable invariants is more desired for attack detection purpose in CPSes.

Fig. 4 shows the sample DELog and invariants that ARTINALI++ *D|E* miner generated for smart metering platform. For example, the invariant $receive : seg\_data = false, command = nil, status = time\_out, len.partial > 0$ represents four multi-propositional data invariants that are valid during the event *receive*, and is hence

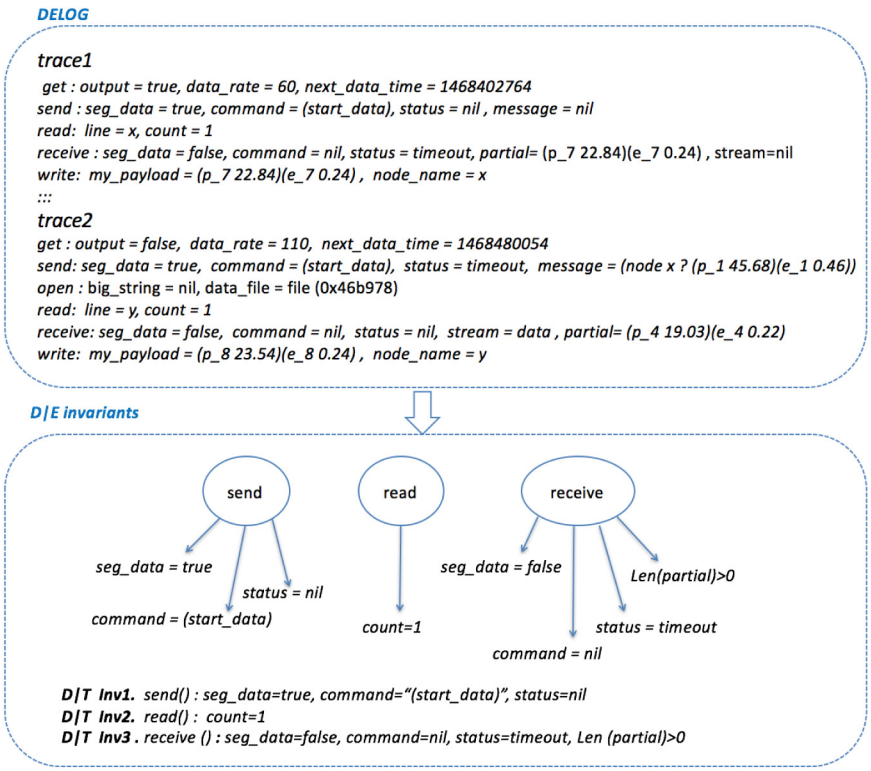

Fig. 4. Sample DELOG and $D|E$ invariants with 100% confidence and 95% support for smart metering platform.

a $D|E$ invariant. The invariants were generated with 100% confidence and 95% support for smart metering platform; Confidence and support are used to verify the correctness of a likely invariant, and hence the higher confidence/support thresholds lead to infer more reliable invariants.

4.4.2. Block 2: $E|T$ Miner

$E|T$ Miner infers the $E|T$ invariants, that are the time constraints among the events. $E|T$ invariants are mined in a four-step process. First, $E|T$ Miner creates all consecutive event pairs within one trace, and annotate them with their time differences. Second, it groups the consecutive event pairs, and label them with the same pair name within the trace. Third, using Association rules mining algorithm, E|T Miner looks for the pair-wise events that are observed in the same order within all training traces, and calculates their confidence and support. We define the threshold of 100% and 95% for confidence and support, respectively, which cause the $E|T$ miner to infer more solid rules. In the last step, $E|T$ Miner merges the time differences within each class to calculate the time boundaries of the paired events as well as the frequency of every single event. It also computes the duration boundaries of every event within all execution traces. The $E|T$ invariants are categorized into three types, as shown in Table 1. Type I indicates that event E_i is repeatedly seen every $\frac{1}{Freq_i}$ seconds within one single trace as well as all execution traces. Type II indicates that the pair of consecutive events E_i and E_j are repeated in all execution traces, and their time difference is bounded within $\Delta t_{ji}max$ and $\Delta t_{ji}min$. Type III indicates the duration boundaries of an event. We provided sample logs and $E|T$ invariants generated by ARTINALI++ for smart metering platform in Fig. 5. As seen, $\texttt{send}(t) \rightleftharpoons \texttt{send}(t+60)$ shows the average frequency of *send* occurrences in the system, and the invariant $send \rightleftharpoons \texttt{receive}$: 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.

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	$d_m(T_i \leq t \leq T_j) = [\alpha_1, \ldots, \alpha_p]$
Type II	$d_m = [\alpha_1, \ldots, \alpha_p] \rightleftharpoons d_n = [\beta_1, \ldots, \beta_q] : \Delta t_{ji}\text{max}, \Delta t_{ji}\text{min}$

4.4.3. Block 3: $D|T$ Miner

As described in the mathematical derivation of $D|T$ invariants (Section 4.2), ARTINALI++ combines the outputs of $D|E$ and $E|T$ miners to generate the $D|T$ invariants (aka real-time data invariants). As seen in Table 1, we design two types of $D|T$ invariants.

Type I states the distribution of valid concrete data values of variable d_m within time slot $T_i \leq t \leq T_j$. For instance, $\texttt{seg-data}(T_1 \leq t \leq T_2) = a$ means that the only valid value of variable `seg-data` is a within the time interval $T_1 \leq t \leq T_2$. Note that ARTINALI++ $D|T$ invariants differ from Daikon data invariants (e.g. `seg-data=a,b`), as the later only express the valid values of data invariants without considering the time.

Type II represents the relationship of data invariants between two consecutive events. As every two consecutive events have a bounded time difference ($T_i + \Delta t_{ji}\text{min} \leq T_j \leq T_i + \Delta t_{ji}\text{max}$), the data invariants associated with those events have the same time difference accordingly. Therefore, data invariant $d_j = [\alpha_1, \ldots, \alpha\_m]$ holds true *until* data invariant $d_i = [\sigma 1, \ldots, \sigma\_k]$ becomes true, while $\Delta t_{ji}\text{max}$ and $\Delta t_{ji}\text{min}$ specifies the time difference boundaries between these data invariants. Fig. 6 shows examples of two types of $D|T$ invariants that ARTINALI++ $D|T$ Miner generated for smart metering platform. As illustrated, one of the invariants of this type is as follows: seg_data $= a \rightleftharpoons$

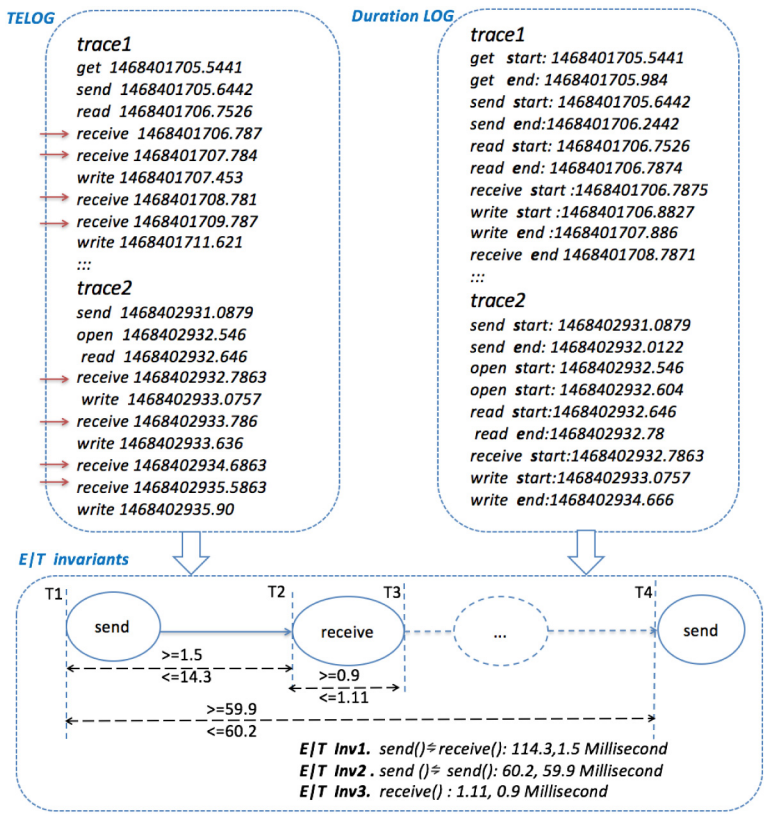

Fig. 5. Sample of TELOG, Duration-LOG and $E|T$ invariants with 100% confidence and 95% support for smart metering platform.

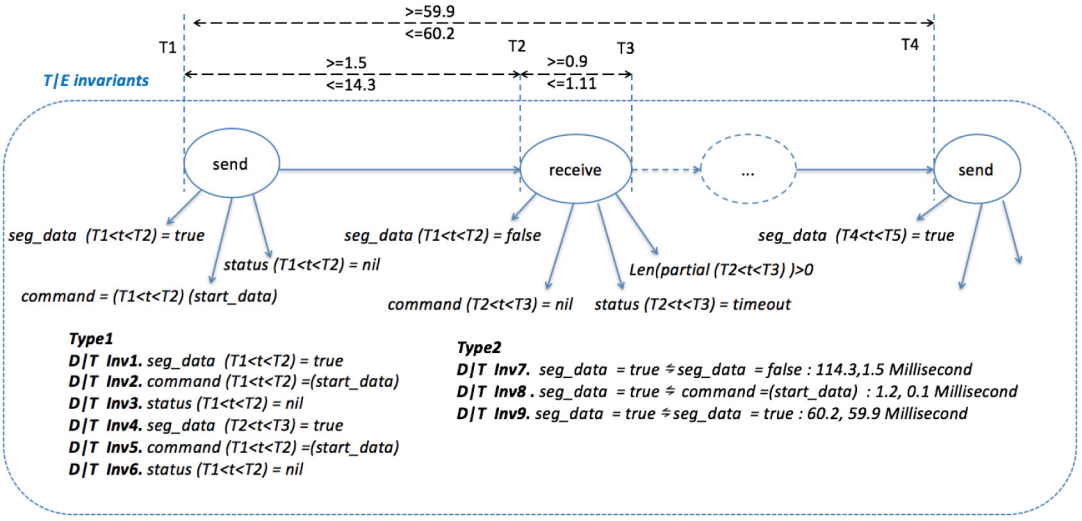

Fig. 6. Representation of sample $D|T$ invariants for smart metering platform.

seg_data $=$ b : 14.3, 1.5; i.e., seg_data $=$ a holds true until `seg_data` is assigned value b, in a time interval ranging between 1.5 and 14.3 s.

5. Specification mining for complex CPSes

ARTINALI++ technique relies upon two properties in CPSes, namely *event-based semantics*, and *conditional independence of time and data* (Section 4.2). The former implies that every event takes place in a unique time frame, and hence, there is no concurrency among event executions. The later assumes an event occurs at a specific time interval, and consequently, data variables are assigned to specific values. Thus, the time and data corresponding to a particular event, are conditionally independent apart from the dependency among events. These two properties are common paradigms for CPSes, and hence ARTINALI++ can be generalized to many CPS platforms. However, we are interested in expanding the generalizability of ARTINALI++ to more complex CPS platforms (such as autonomous vehicles).

5.1. Challenges

As the continuity of operation is of the utmost significance for complex CPSes including autonomous vehicles, it is essential to

dynamically monitor the system to learn its common behaviors and formulate specifications for detecting security attacks. However, deriving specifications for such systems using the current techniques is not a simple task as they are dealing with a set of challenges. In the following, we explain the existing challenges (C1-2) (Jiang et al., 2013; Schumann et al., 2015) for mining specification for these systems. We first provide more details about each challenge, then we propose our solution to the challenges.

5.1.1. C1: Various operational states

Autonomous vehicles operate in various operational states (e.g., UAV works in different flight states such as landing, taking off, flying, hovering, etc.), and behave differently in each operational state. As stated in Jiang et al. (2013) and Schumann et al. (2015), attempting to produce a system model (specifications) without differentiating between these states would result in a smaller set of more general specifications, but it would ignore many valuable specifications that only apply to one flight state. It results in a noticeable increase of false positives and false negatives in attack detection.

5.1.2. C2: Non-accessible parameters in code level

Invariant detection techniques generate specifications based on code level variables. They instrument the variables of interest and generate traces to mine specifications. We call the accessible variables *raw variables*. In complex CPS platforms, there are parameters that are significantly important for capturing the behavioral model of the system, while the specification mining tools cannot directly instrument them in the code. So, for such non-accessible parameters, we need to define derivative templates to derive specifications from continuous raw variables (as stated in Jiang et al., 2013). For instance, the variations of distance traveled over time results in velocity for the platforms that have movement in physical space and time like autonomous vehicles. To include such behaviors in the model, we need an automated way to calculate the derivative specifications, as well.

5.2. Solutions

ARTINALI++ is equipped with a set of features that overcome the challenges of specification mining for the complex CPS platforms. These features add new building blocks to ARTINALI (Fig. 3(a)). In the following, we explain the new building blocks additional to ARTINALI workflow for modeling complex CPS platforms (Fig. 3(b)).

Feature1. Automated inference of state-aware invariants: To address the first challenge discussed in previous section, we exploit the notion of conditional operational state invariants from Jiang et al. (2013). We enhanced our instrumentation strategy in ARTINALI++ and designed the *State partitioning* block to support this feature.

5.2.1. Enhanced instrumentation

ARTINALI++ technique is designed to work at two levels of granularity including *events* and *states* to mine specifications for complex CPSes. In addition to *event* level granularity, we define a second level of granularity in ARTINALI++ to infer invariants holding true upon certain operation states (so-called *state-aware invariants*). The *state* level granularity enables ARTINALI++ to analyze the behavior of autonomous vehicle in various operation states, independently. Using ARTINALI++ custom APIs, we instrument the pre-location and post-location of events in the program code to collect logs for the *event* level granularity. We also hook the operation state transition points in the code to enrich the logs by the operation state's labels for the *state* level granularity. The result is a set of log files including the events and their associated data and time information labeled with operational state.

5.2.2. Block 4. State partitioning

State partitioning block (block 4 in Fig. 3(b)) takes the collected logs and groups them into distinct partitions corresponding to distinct operational states. The partitioned logs are fed to ARTINALI++ miners for mining invariants. State partitioning block enables ARTINALI++ to capture invariants for every partition independently. It also provides predicates to differentiate invariants inferred for various operational states.

State-aware invariant inference is a very beneficial feature of ARTINALI++ for complex CPS platforms. From a qualitative perspective, this feature allows ARTINALI++ to capture a more specific model for each operational state other than a general model holding for all states. This capability results in a more sound and complete CPS model, and enables the IDS to generate more accurate results. To shed more light on the advantages of state-aware invariants, we take an example in the context of UAV systems. Typically, there exist four operational states in UAV system including taking off, flying, hovering and landing. During Flying state, drone flies from one point to the next target point via a direct path (aka, no angular rotation is required). Sudden deviations from such a straight path could indicate a suspicious behavior, while the same observation does not hold for sudden climbs of UAV during taking off state. Attempting to capture a general set of invariants that hold over all operational states would lead to ignore the important behavioral patterns of each state, and to miss the likely attacks.

Table 2 shows the sample of state-based invariants generated by ARTINALI++ for the UAV platform. As shown, when UAV is taking off or landing, the yaw variations is within specific boundaries (specifications (1) and (3)), while when it is flying from one target point to another one, there is not any variations in *yaw* parameter until it reaches the target .i.e., *distance* gets *zero* (specification (2)). Unlikely, in hovering mode, *yaw* varies regardless of *distance* value (specification(4)). Note that we use *Gen* label for the invariants that hold true in all four flight states. For example, the time boundaries between GPS sensor data is read (*GPS_read*) and flight controller issues the control command to actuators (*FC_send*) is maximum 1883 and minimum 541 ms (specification (5)).

In Section 8, we quantitatively show that how state-aware invariants are effective to increase the IDS accuracy.

Feature2. Automated inference of derivative invariants

The second feature of ARTINALI++ is the ability to automatically infer the derivative invariants. We implement this feature using Blocks 5 and 6 in Fig. 3(b).

5.2.3. Block 5. Parsing and annotation

In ARTINALI++, we enrich the traces with timing data for each spatial variable. Instrumenting of every single variable for collecting time stamps may lead to a huge runtime overhead, which is not a viable option for CPS system; instead, we obtain our required information from the existing logs using *parsing and annotation* block. ARTINALI++ API functions collect data and time information of instrumented events in separated log files (*TELog* and *DELog*). These files are used as inputs for the *Parsing and Annotation* block.

Parsing and annotation in ARTINALI++ is an automated process, and is not dependent on the CPS. ARTINALI++ APIs are written in such a way that collect the data, event and time information in a pre-defined format, allowing the parser to find the variables and their timestamps based on their location. For example, every line of *DELog* file has the following format:

$$Event: variable_1 = [], (*)variable_2 = [], \ldots, variable_n = []$$

Table 2
Sample state-aware specifications generated by ARTINALI++ for various operational states in UAV.

	Flight states	Specifications
(1)	Taking off	$Tak :: MinYaw \leq \Delta(yaw(t)) \leq MaxYaw$
(2)	Flying	$Fly :: distance \neq 0 \rightleftharpoons \Delta(yaw) = 0,\ distance = 0 \rightleftharpoons \Delta(yaw) > 0$
(3)	Landing	$Lan :: MinYaw \leq \Delta(yaw(t)) \leq MaxYaw$
(4)	Hovering	$Hov :: \Delta(yaw) > 0,\ distance = 0$
(5)	General	$Gen :: GPS\_read() \rightleftharpoons FC\_send() : 1883, 541$ ms

As seen, each line starts with the name of an *event* and is followed by colon (:). All variables are located after colon and are separated by comma. Spatial variables, however, have a tag (*). This tag is added at the time of instrumentation to the variable manually. Similarly, every line of *TELog* file starts with the name of an *event* and is followed by its time stamp. Parsing and annotation block uses these two files to parse the spatial variables (such as GPS coordinates), and associate each spatial variable instance to the time of its related event. These information are collected in a new log file named *DTLog*, which provides ingredients to infer derivative invariants.

5.2.4. Block 6. DV Miner

DV Miner is an ARTINALI++ miner that does not exist in ARTINALI technique, and is developed to mine derivative invariants. *DV Miner* is fed by *DTLog* and learns invariants about the boundaries of spatial variable values, and the associated velocities. We define two general derivative templates as follows:

The first template is used to calculate the boundaries of CPS movements over time among all training traces, where *SP* is a spatial variable value.

$$VelocityMin \leq \Delta SP(t)/\Delta t \leq VelocityMax \tag{10}$$

The second template is used to derive invariants representing variations of CPS continuous movements in each direction (X, Y, Z) that should follow a bounded threshold to be stable. For example:

$$LowerBoundX \leq \Delta X(t) \leq UpperBoundX \tag{11}$$

DV Miner uses a four-step process to mine the derivative invariants based on the above general templates. First, DV Miner finds all spatial variables (e.g., $SP_i(t)$) annotated with their time stamp within a trace. Second, using the formula in derivative templates, it calculates the directional movement of the vehicle in each time step (e.g., $SP_i(t_j) - SP_i(t_{j-1})$) and/or the related velocity (e.g., $SP_i(t_j) - SP_i(t_{j-1})/(t_j - t_{j-1})$). Third, it computes the average of directional movements ($Ave(\Delta SP_i(t))$ and/or velocities ($Ave(\Delta SP_i(t)/\Delta t)$ for each spatial variable within one trace. In the last step, *DV Miner* merges the average values within all training traces to calculate the average boundaries for directional movement (*LowerBoundX* and *UpperBoundX*) and the associated velocity (*VelocityMin* and *VelocityMax*). The most important step for capturing these invariants is finding the spatial variables based on the time (e.g., $SP(t_j)$), and calculating the time difference (e.g., $(t_j - t_{j-1})$, none of which are achieved by the other invariant inference tools such as Daikon, while ARTINALI++ addresses these issues.

From a qualitative view, derivative invariants model the behavioral patterns of vehicle physical movement. They allow the IDS to detect suspicious directional or angular movements, or speeds. Derivative invariants also show the variations of CPS continuous movements in each direction that should follow a bounded threshold to be stable. These invariants are beneficial to detect those attacks that target the stability of the autonomous vehicles that move in 3-D space, such as UAV and Maritime.

5.2.5. Block 7. Enhanced IDS

Our IDS performs two core functionalities including *Data monitoring* and the *Data analysis*. Data monitoring is the process by which an IDS observes system behavior and accumulates data logs. The IDS component that is in charge of Data monitoring is called *Tracing module*. Data analysis is the process by which an IDS periodically analyzes the collected logs and checks them against the specifications derived from the CPS's correct behavior. This process is performed by the *Intrusion detector* module of the IDS.

We upgraded both *Tracing* and *Intrusion detector* modules of ARTINALI to support ARTINALI++ additional functionalities. Similar to instrumentation procedure of ARTINALI++, *Tracing* module instruments the control program at two levels of granularity (including events and states), and collects logs, but with the difference that it is deployed on the production system.

The ARTINALI++ miners derive $4 * N$ classes of specifications that comprise our CPS model. They include $D|E$, $E|T$, $D|T$ and DV invariants that are captured for N diverse operational states. The IDS uses the CPS model and the information collected by *Tracing module* as inputs to *Intrusion detector* module. As the type of logs and CPS model are mainly different from those of ARTINALI-based IDS, we significantly upgraded the Intrusion detector to verify the security properties of the CPS.

6. Experimental setup

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

6.1. CPS platforms

We chose three CPS platforms with various levels of complexity as our case studies to evaluate the efficacy of the invariants generated by ARTINALI++. Our CPSes include two non-complex platforms (including advanced metering infrastructure and smart artificial pancreas) and a complex platform (unmanned aerial vehicle).

6.1.1. Advanced Metering Infrastructure (AMI)

As the key components of AMI, smart meters are deployed in smart grid, and provide a two-way communication with the utility provider (Skopik et al., 2012). The large scale deployment of smart meters and the discovery of many security vulnerabilities in these systems (Yan et al., 2012), make them suitable candidates to evaluate our technique. A generic smart meter is composed of two major components including the *meter* and the *controller* (or gateway). The *meter* receives power consumption data (PCD) through analog front end sensors, and buffers them in the memory. The *controller* plays the role of the communication bridge between the meter and the utility provider's server, passing server commands to the meter, and sending PCD back to the server at pre-defined time intervals. We use SEGMeter (Anon, 2011), an open source AMI platform as our testbed to evaluate our IDS. SEGMeter is implemented using the Lua language, and consists of 2500 lines of code excluding libraries.

6.1.2. Smart Artificial Pancreas (SAP)

Recently, diabetic patients are migrating from the traditional glucose measurement and manual insulin injection to continuous glucose monitoring and autonomous insulin delivery devices (Li et al., 2011), which are referred to as *Smart Artificial Pancreas (SAP)*. As attacks to a SAP can endanger the patient's life, these systems are highly security-critical (Radcliffe, 2011). Hence, we selected SAP as our second case study to evaluate our technique. A generic SAP is composed of a Continuous Glucose Monitor (CGM), an insulin pump, and a controller, that are commonly connected through a wireless network to form a real-time monitoring and response loop (Li et al., 2011). The CGM and insulin pump are wearable medical devices, in which the former measures the patient's blood glucose (BG) levels on a regular basis and sends it to the controller, and the later is used for automatic injection of insulin through subcutaneous infusion. It may deliver insulin in two doses: *bolus* and *basal*. Each type has specific injection time, rate, and dosage based on the patient's needs. The controller controls the closed loop in the SAP. It receives the measured BG from CGM, and sends a suitable actuation command to insulin pump for correcting the BG level. We used OpenAPS (Lewis, 2015), an open source SAP, as a second testbed to evaluate our IDS. OpenAPS implements the controller component of an SAP in JavaScript, and consists of 2000 lines of code excluding libraries.

6.1.3. Unmanned Aerial Vehicles (UAVs)

(UAVs) are essentially flying computers. They may be remotely controlled or can fly autonomously through software-controlled flight plans in their smart embedded systems that are operating in conjunction with on-board sensors and GPS. A generic drone consists of four main building blocks that are, one flight controller, multiple sensors and actuators, a wireless transmitter and a Ground Station (GS) controller. The flight controller is essentially a normal programmable micro-controller that has specific sensors on board. It receives control commands from the GS controller through the wireless transmitter, and calculates the new flight parameters of drone using data received from the sensors. We chose Ardupilot (Anon, 2012): an open source UAV platform as our testbed. It includes 5000 lines of code in C++ (excluding libraries), and works in 8 different operational modes. ArduPilot is used in a set of general-purpose autopilot systems, including, but not limited to, submarines, helicopters, multirotors, and aeroplanes. We use SITL (Software In The Loop) simulator that permits us to run Ardupilot with no hardware.

6.2. Experimental procedure

Fig. 7 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 downloaded the latest versions of these tools from their respective websites (Anon, 2017, 2016, 2014). 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. 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.

Table 3 presents the type and the number of invariants generated by the ARTINALI++ during training phase for the SEGMeter, OpenAPS and ArduPilot platforms, respectively. There are 22 system calls in SEGMeter's code, 4 system calls in the OpenAPS code and 58 system calls in the AduPilot code. We consider all of them as events, and generate specifications. We also generated 258 specifications including 29 derivative specifications for the ArduPilot platform. These specifications are also labeled for various flight states. The generated specification sets are fed into the IDS as inputs, and their efficacy is evaluated on different platforms.

Table 3
Types and number of mined specifications by ARTINALI++ for different CPS platforms.

	Time	*D|E*	*E|T*	*D|T*	*Derivative invariants*	State-aware invariants
SEGMeter	12	24	37	24	–	–
OpenAPS	4	22	18	7	–	–
ArduPilot	9	78	115	27	29	57

We divide the experiment into a training phase and a testing phase for each CPS platform. We first obtain execution traces from the three 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 specification mining tool to optimize the false positive (FP) and false negative (FN) ratios for that platform. 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 1500 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 in Section 8.

6.3. Attack models

Traditionally, security mechanisms are evaluated using a small number of targeted attacks. These are not enough for CPS systems as they are new systems for which there are few known attacks. Therefore, they need protection from zero-day attacks, as well. This is especially important for security-critical CPSes such as smart medical devices and autonomous vehicles.

Targeted Attacks To evaluate our IDS against known attacks, we target three attacks against the class of AMI and SAP each, and two attacks against the class of UAV, that we identified based on the literature. (Section 8).

Arbitrary attacks We use fault injection (i.e., code mutation) to emulate arbitrary (zero-day) attacks. Injected faults are not complete attacks. However, they form the building blocks of attacks. We inject different types of faults in the program's code, including *Data mutations*, *Branch flipping*, and *Artificial delay insertion*. Each of the above categories emulate different security issues. By performing data mutations, an attacker can change critical data in the program to their benefit. Such attacks can be crafted by exploiting memory corruption vulnerabilities or race conditions in the program. Likewise, branch flipping can result in illegitimate control flow paths being taken in the program, to accomplish the attacker's targets. Such attacks can occur due to code injection or semantic vulnerabilities. Finally, artificial delays can allow attackers to change the timing of the system's tasks, and delay important functions, or cause other functionalities to be suppressed, to their advantage. Through these mutations, we can emulate a wide variety of attacks, without a predefined target, thus avoiding bias and allowing modeling of hitherto unknown attacks.

Table 4 presents the number of mutations performed in each attack category for our CPS platforms. Overall, we performed 156,

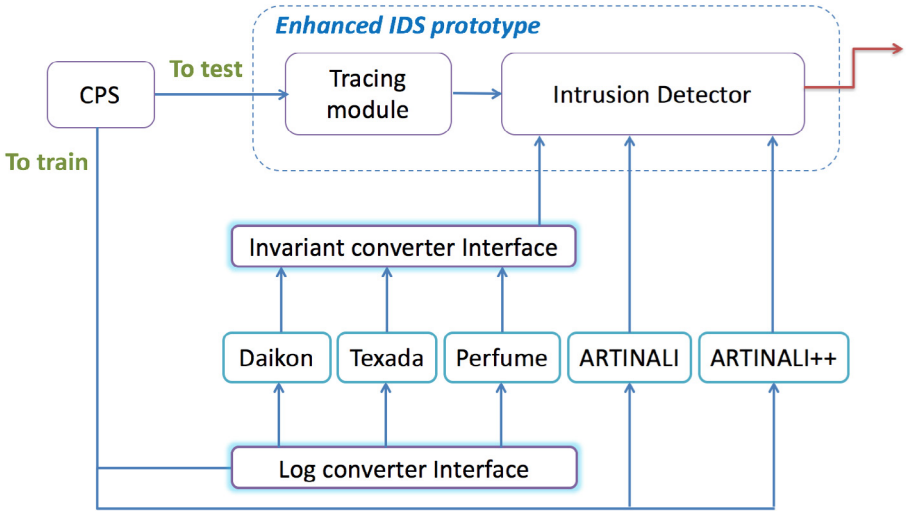

Fig. 7. Overall experimental process of running the IDS.

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

CPS	Attack category		
	Data mutation	Branch flipping	Artificial delay
SEGMeter	35	75	45
OpenAPS	100	10	15
ArduPilot	284	91	213

125 and 550 fault injections for SEGMeter, OpenAPS and ArduPilot, respectively. We injected each of these faults in the control program of the respective CPS platforms. After fault injection, 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.

Fig. 8 shows the outcome of our code mutation experiments across SEGMeter, OpenAPS and ArduPilot control programs, respectively. 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 need an IDS for the *SDC* and *No corruption* outcomes which comprise about 82% of the outcomes on average.

6.4. Evaluation metrics

Accuracy: We use three metrics to measure the effectiveness of IDS from the accuracy perspective.

- *False Negative ratio (FN)*, which is the ratio of attacks that were undetected by the IDS to the total number of injected attacks;
- *False Positive ratio (FP)*, which is the ratio of execution traces that were *improperly* reported as attacks to the total number of proper traces;
- *F-Score(β)*,[2] which is a computation of the harmonic mean of the true positive ratio[3] (TP), FP and FN.

The variations of the argument β in F-Score(β) enable us to put emphasis on the above metrics differently (Sokolova et al., 2006), and obtain different trade-off between FP and FN ratios based on the system constraints. For instance, a value of $\beta > 1$ emphasizes more on FNs, while a value of $\beta < 1$ emphasizes more on FPs. A value of $\beta = 1$ emphasizes them both, equally. We hypothesize that FPs are more critical in smart meters, as a false-alarm results in adding cost to the utility provider who needs to use service personnel to discover the false alarm. While an occasional FN may be tolerable 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 lead to patient intervention, and hence is more acceptable. Similarly, in ArduPilot, the continuity of operation is of the utmost significance, thus even a single FN may cause drone to crash or go off course, while FPs may only interrupt the flight and consume resources. Hence, for SEGMeter, we select F-Score(0.5), and for OpenAPS and ArduPilot we choose F-score(2) as our reference metrics.

Overheads: We 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 specifications that account for the CPS model, and the size of log files collected by *tracing module*. Performance overhead is the increase in execution duration as a result of running the IDS on the target platform. This metric sourced by both the *tracing module* and the *intrusion detector*. Since CPSes run continuously for long periods of time, we measure the performance overhead per iteration, where an iteration refers to one full execution of the main loop of the CPS.

7. Evaluation against targeted attacks

In this section, we discuss the potential targeted attacks and how we derive them for SEGMeter, OpenAPS and ArduPilot platforms. We then evaluate the IDS against the attacks. Note that we used attack trees based on prior attacks against each class of CPS platforms to minimize the bias and model realistic attacks.

[2] $F-Score(\beta) = \frac{(1+\beta^2)\times TP}{(1+\beta^2)\times TP+\beta^2\times FN+FP}$

[3] Represents the ratio of execution traces that were properly reported as attacks to the total number of proper traces.

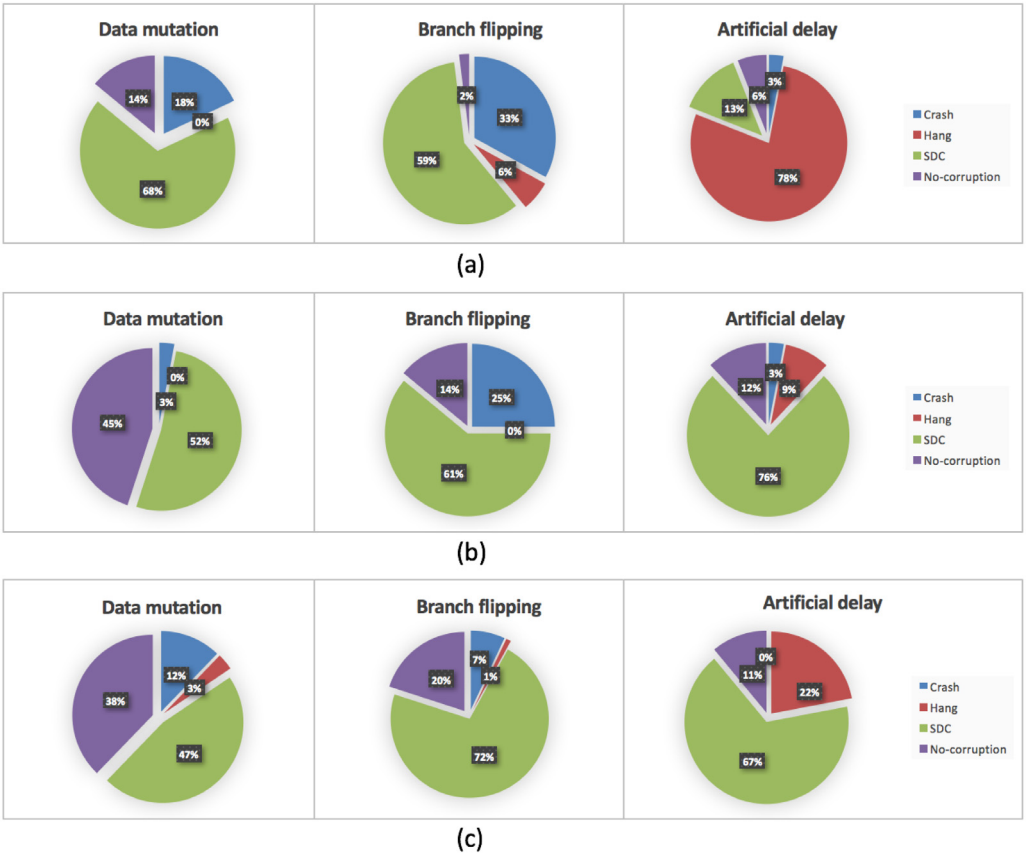

Fig. 8. Outcome of data mutation, branch flipping, and artificial delay attacks on (a) SEGMeter, (b) OpenAPS, and (c) ArduPilot control programs.

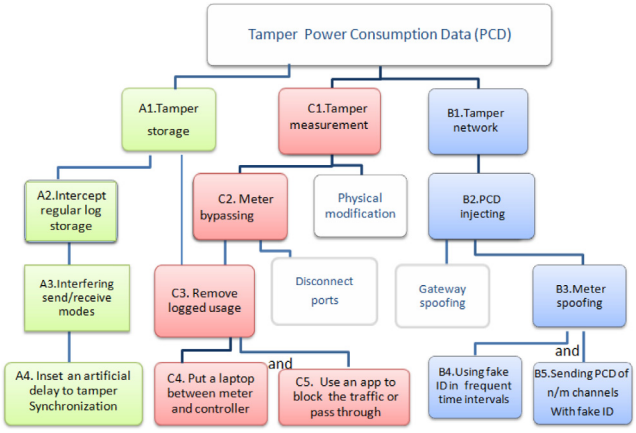

Fig. 9. Attack tree for AMI.

We found that IDS was able to detect all the attacks since they all involved violations of the interplay among data, events, and time.

7.1. AMI attacks

Energy fraud is considered as the major class of attacks against AMI, and can result in *PCD* loss and improper billing (McLaughlin et al., 2010). We illustrated an attack tree for energy fraud in AMI in Fig. 9, based on attacks introduced in the literature (McLaughlin et al., 2010; Skopik et al., 2012; Yan et al., 2012). Generally, there are three major branches in this tree, namely (i) Measurement tampering, (ii) Storage tampering, and (iii) Network tampering. We developed the concrete attack actions for each branch, as the leaves of the tree, which is described below.

Synchronization tampering (Blocks *A*1–*A*4**)** occurs due to modification of the time of *send* and *receive* modes in the smart meter (Aliabadi et al., 2017). We found a vulnerable function (*get-data-timer()*) in the control program of the smart meter, that synchronizes the communication between the meter and the server. The controller unit frequently checks the time with the server to decide when to request for the PCD measured by the meter unit. If the time on the server is maliciously modified, the controller unit will not receive PCD in the expected time, which results in data loss, and incorrect calculation of final PCD.

Meter spoofing (Blocks *B*1–*B*5**)**: In a smart grid, smart meters communicate with the server using a unique ID. The controller unit is able to be connected to more than one meter unit, collects the PCDs separately, and send them along with the meter's ID to the utility server. Since the controller cannot differentiate between normal and abnormal measurements, it can be tricked by malicious inputs sent by an attacker instead of a normal meter unit. This attack is called *meter spoofing attack*. We found that meter spoofing attack only requires the meter's ID that is printed on the meter's nameplate (Aliabadi et al., 2017). This potential vulnerability enables the attacker to use the meter's identity for energy theft.

Message dropping (Blocks *C*1–*C*5**)**: An attacker may be able to drop the measurements (i.e., a part of energy usage) after bypassing the meter unit and removing the historical PCD. This attack may be simply crafted through intercepting the communication between the meter and the controller units, and controls what traffic to block and what to pass through (e.g., through a firewall). Therefore, the blocked traffic would not be included in PCD calculations.

Table 5
Specifications generated by ARTINALI to detect the example attacks in AMI.

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

7.2. Detection of AMI attacks

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

Synchronization tampering Since synchronization tampering attack manipulates the timing of *send* and *receive* operations of smart meter, we picked events *send* and *receive* as relevant events to explain this attack. We can see in row 1 of Table 5 that the specification generated by ARTINALI++ captures the sequence of these events during normal operation, i.e., *send* operation occurs every 60 s, and *receive* is repeated every 1 s. Thus, this specification detects the attack as the timing of the events is violated by this attack.

Message dropping If message dropping attack happens, 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 specification number (2) in Table 5, which represents the time frequency of receive function (1003 ms $\cong$ 1*sec*) within one full iteration.

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

7.3. SAP attacks

Diabetic therapy tampering is the highest severity attack for diabetic patients, as it can lead to death or severe health complications. We developed an attack tree for diabetic therapy tampering based on publicly available work on SAPs (Radcliffe, 2011; Li et al., 2011), in Fig. 10. We derive three classes of attacks based on the tree.

CGM spoofing attack (Blocks *A*1–*A*4**)** injects false into the communication channel between CGM and controller unit, making the controller unit think that the glucose level is either higher or lower than the actual value. CGM spoofing may be accomplished in two ways: First, if the format of measured BG is unknown, then a replay attack may be used. In this case, a measured BG read in the past can be re-sent (e.g., by a RF module Li et al., 2011) to the controller unit. This would cause the controller unit to make decision based on an outdated sugar level rather than the actual one. Second, if the format of measured BG is known to attacker, she can inject the false data at random time intervals to mislead the controller.

Basal tampering (Blocks *B*1–*B*5**)** The basal tampering attack may be crafted in two different ways: The attacker may issue a command for (i) stopping the basal injection (e.g., $basal.rate = 0$) when it is needed for the patient, or (ii) resume the basal injection ($basal.rate > 0$) when it must be stopped. These attacks may be accomplished using a software radio board that fully controls the SAP (Radcliffe, 2011; Li et al., 2011), and transmits the malicious commands to the insulin pump. For this attack, the attacker needs to spoof the PIN number of the controller, and the format of transmission data both of which are possible through an eavesdropping attack.

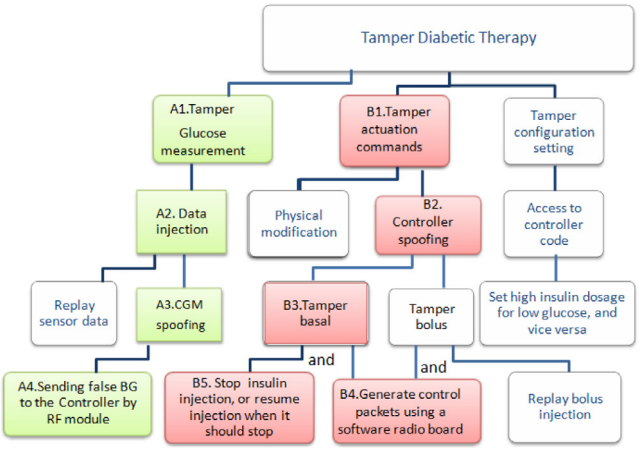

Fig. 10. Attack tree for SAP.

7.4. Detection of example attacks in SAP

We crafted the above attack examples on the OpenAPS, and found that the IDS is able to detect all of them.

There are four important events in the SAP, including (1) *send(BG)* or sending BG measurement 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 specifications for OpenAPS's IDS model. We show those specifications that detect the example attacks in Table 6.

CGM spoofing attack To analyze CGM spoofing attack, we chose event *read(BG)* in controller as the relevant event, and analyzed the mined specifications for this event. Under normal conditions, the transmission of measured BG to CGM occurs at deterministic time intervals (e.g., every five minutes). It is represented in our model as time frequency of event *read(BG)*, that is $read(t) \rightleftharpoons read(t + 5)$. Using the above specification, it would be possible to detect malicious sensor reading from any potentially malicious sources that perform replay attack or send false data at random time intervals to the controller.

Basal tampering attack The basal tampering attack may be accomplished in two different ways: (i) stop basal injection ($basal.rate = 0$) when it is needed, and (ii) resume basal injection ($basal.rate > 0$) when it is not needed. These attacks break the specifications shown in Table 6. The invariant number (2) shows 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 specification number (2). Similarly, the invariant number (3) in Table 6 shows that for low BG values (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 OpenAPS to inject insulin. Consequently, it causes the specification number (3) to be violated.

Table 6
Specifications generated by ARTINALI to detect example attacks in OpenAPS.

Attack	Detecting specification
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

7.5. UAV attacks

In this section, we discuss the potential known attacks against ArduPilot platform, and how IDS detects them. Flight Control Tampering (FCT) is considered as the most important threat for UAVs. We define the FCT attack as tampering with the data that comes into the drone from different sources and cause the drone to crash or go off course.

We developed an attack tree for FCT attack based on known UAV attacks from publicly available reports (Samland et al., 2012; Son et al., 2015; McCallie et al., 2011), and illustrated it in Fig. 11. Data comes into the flight controller from three entry points including the Sensors (such as GPS and gyroscopes), GS controller and Program setting. Considering three entry points, we consider tree main branches for the tree, starting with (i) Sensor data tampering, (ii) Actuation command tampering, and (iii) Program setting tampering. We have found several potential attack scenarios for FCT attack in UAV, and developed the concrete actions for each attack as the leaves of the tree. In the following, we analyze two attack examples, and explain how these attacks might be crafted by an attacker using the steps highlighted in the attack tree.

GPS Spoofing (**Blocks** A1–A5 in Fig. 11) Drone's GPS receivers are vulnerable to a number of different attacks including blocking, jamming, and spoofing (Warner and Johnston, 2003). The goal of these attacks is either to prevent a position lock (blocking and jamming), or to trick the receiver by false information so that it computes a wrong time or location (spoofing). GPS receivers can usually detect the blocking or jamming attacks because they have a loss of signal, but if it is combined by false data injection (GPS spoofing), it would be a more difficult attack to detect.

Drone hijacking (**Blocks** B1–B5 in Fig. 11) Drone hijacking attack is infusing false remote control commands between the drone and the GS controller. As a first step, attacker needs to disconnect the drone from original GS controller that may be done through WiFi de-authentication attack. A WiFi de-authentication attack is a type of DoS (Denial-of-Service) attack that targets communication between a user and a WiFi wireless access point. An attacker can send a wireless access point at any time, with a spoofed address for the victim. The only information that attacker requires is the victim's MAC address, which is available through wireless network sniffing. The attacker would be able to hijack the drone if he crafts WiFi de-authentication attack in conjunction with sending malicious actuation commands to drone.

7.6. Detection of UAV attacks

We emulated the UAV attacks on ArduPilot simulator, and observed that our IDS could monitor them. Table 7 shows the specifications generated by ARTINALI++, that detect the above attacks.

GPS Spoofing $E|T$ invariants in ARTINALI++ serve to characterize the event frequency and duration boundaries. For example, frequency of reading GPS coordinates should follow a certain threshold: *GPS-read(t)* $\rightleftharpoons$ *GPS-read (t+ 1/freq)*. We used this type of specification (shown in Table 7) to detect stale or false location data read by GPS sensors, which may direct the drone to the wrong location.

Drone hijacking During flying mode, drone flies from one point to the next target via a direct path, where no angular rotation is needed, i.e., there should not be any variations in *yaw* parameter until it reaches the target. Sudden deviations from such a straight path could indicate an unwanted maneuver. The same observation holds for sudden climbing movements of drone, i.e., there should not be any variations in altitude (H) until it reaches the target. If drone is hijacked, a different target point than the correct destination would be set up for it, that leads to a sudden change in *yaw* or/and H parameters. This attack can be detected by the specifications shown in Table 7.

7.7. ARTINALI vs. ARTINALI++

We compared the IDS capability for detecting known attacks on UAV platform when it is driven by ARTINALI and ARTINALI++ specifications. We observed that ARTINALI is able to infer specification (1) stated in Table 7, but not specifications (2) and (3). As a result, ARTINALI-based IDS is only able to detect GPS spoofing attack, while ARTINALI++ detects both GPS spoofing and Drone hijacking attacks on UAV system.

8. Evaluation

In this section, we bring 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.

8.1. 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 ARTINALI++ and the other tools across CPS platforms?

RQ3. What is the FP ratio incurred by the ARTINALI++ and the other tools across CPS platforms?

RQ4. How derivative invariants and state-aware invariants affect the accuracy of IDS?

RQ5. What is the memory overhead of the IDS seeded by each tool?

RQ6. What is the performance overhead of the IDS seeded by each tool?

8.2. RQ1. F-Score

As mentioned in Section 6, we obtain two sets of traces from each CPS platform, namely *train* and *test*. In this section, we are seeking the optimal training set size for each CPS platform 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 SEGMeter and OpenAPS platforms. We then vary the training set size from 5 to 40, in increments of 5. In ArduPilot, however, we obtain a total of 240 training traces, and 200 test traces, and vary the training set size from 20 to 240, in

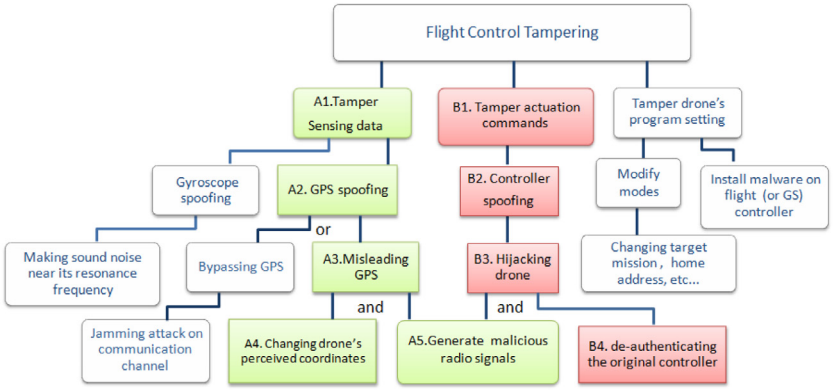

Fig. 11. Attack tree for flight control tampering attack.

Table 7
Specifications generated by ARTINALI++ to detect the example attacks in UAV.

Attack	Detecting specification
GPS spoofing	(1) Gen:: GPS-read(t) $\rightleftharpoons$ GPS-read (t) : 5 s, 4.3 s
Drone hijacking	(2) Fly:: distance $\neq 0 \rightleftharpoons \ \Delta$ yaw $= 0$ (3) Fly:: distance $\neq 0 \rightleftharpoons \ \Delta$ H $= 0$

increments of 20 in a 12-step procedure to find the maximum F-Score value. We observed that in ArduPilot, finding the optimal working point for the IDS needs a larger number of (training and test) traces than those of the first two platforms. The reason is that ArduPilot not only has much more lines of code, it also includes different working modes, both of which results in more diversity in execution paths. Therefore, to have a richer set of traces, it is necessary to create higher number of traces that traverse different paths in various modes. In the next step, We run ARTINALI++ on each training set for SEGMeter and OpenAPS, and ArduPilot to derive invariants. We then measure the FP, FN, and F-Score values (0.5, 1, 2) for each system, as a function of the training set size.

Fig. 12 shows 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, OpenAPS and ArduPilot. As expected, as the amount of training traces increases, the FP ratio decreases, since a broader set of specifications are extracted; thus a lower amount of legal actions are reported as potential attacks. A consequence is that a more restricted set of invariants can lead to some attacks being undetected (FN increases). Overall, an increase in the amount of training traces across all CPS platforms lead to increase of the F-Score at first, then it gets maximized, at which point an *optimal* amount of training traces have been found (for a given value of β).

Tables 8, 9 and 10 show the optimal amount of training traces (optimal F-Score) for each invariant detection tool, for SEGMeter, OpenAPS and ArduPilot respectively. Recall that we choose F-Score(0.5) for SEGMeter and F-Score(2) for OpenAPS and ArduPilot as reference metric, and find the optimal number of traces accordingly. For example, in SEGMeter, a training set size of 20 results in the maximum value of the F-Score(0.5) for ARTINALI++, whereas for OpenAPS, a training set size of 15 results in the maximum value of F-Score(2). In ArduPilot, however, a training set size of 100 results in the maximum value of the F-Score(2). Likewise, we compute the optimal training set sizes for the ARTINALI and the other three tools on all platforms. These are the values of the training set sizes we use for mining the specifications in the rest of this section.

In other words, we found the best configuration of each tool on each platform, and mined specifications using this configuration for addressing IDS optimization purposes for each CPS platform.

Table 8
Optimal training set size for maximum F-Score(0.5) for SEGMeter across tools, and the corresponding FP and FN ratios.

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

Table 9
Optimal training set size for maximum F-Score(2) for OpenAPS across tools, and the corresponding FP and FN ratios.

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

Table 10
Optimal training set size for maximum F-Score(2) for ArduPilot across tools, and the corresponding FP and FN ratios.

	Daikon	Texada	Perfume	ARTINALI	ARTINALI++
F-Score(2)	0.66	0.69	0.73	0.86	0.958
Num of traces	135	90	95	100	100
FP (%)	31	27	29.5	24	9.7
FN (%)	32	29	33.7	22.9	2.53

8.3. RQ2. False negatives

In this section, we compare the variation in the FN ratio incurred by the ARTINALI++, ARTINALI and the other tools for three CPS platforms. Tables 8, 9 and 10 show the FN ratios of each tool for the SEGMeter, OpenAPS and ArduPilot platforms, respectively. We observe that in overall, the IDS based on ARTINALI++ was able to detect more than 97.5% of attacks, which means it has an average FN ratio of 2.5% in all platforms. Our experiments show that ARTINALI's and ARTINALI++'s invariants for non-complex CPS platforms (including SEGMeter and OpenAPS) are the same, and hence the FN ratio of the IDS seeded by ARTINALI and ARTINALI++ on these two platforms remain the same. However, ARTINALI++ generates a richer set of invariants than ARTINALI for ArduPilot platform, that leads to 2.53% FN ratio, while ARTINALI incurs

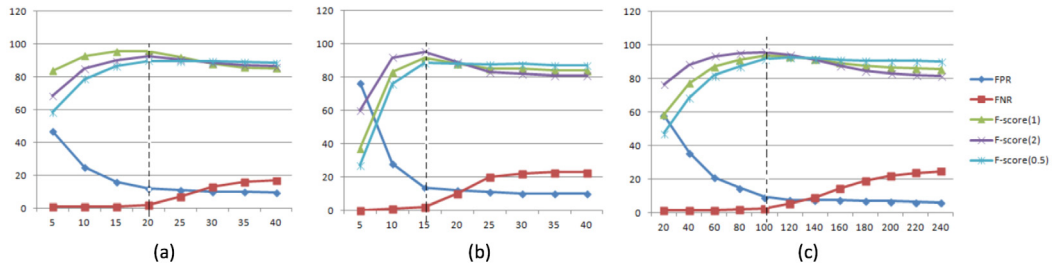

Fig. 12. FN, FP and F-Score variations based on number of training traces in ARTINALI++ for (a) SEGMeter, (b) OpenAPS and (c) ArduPilot. *X*-axis is the training set size and Y-axis is the percentage(%).

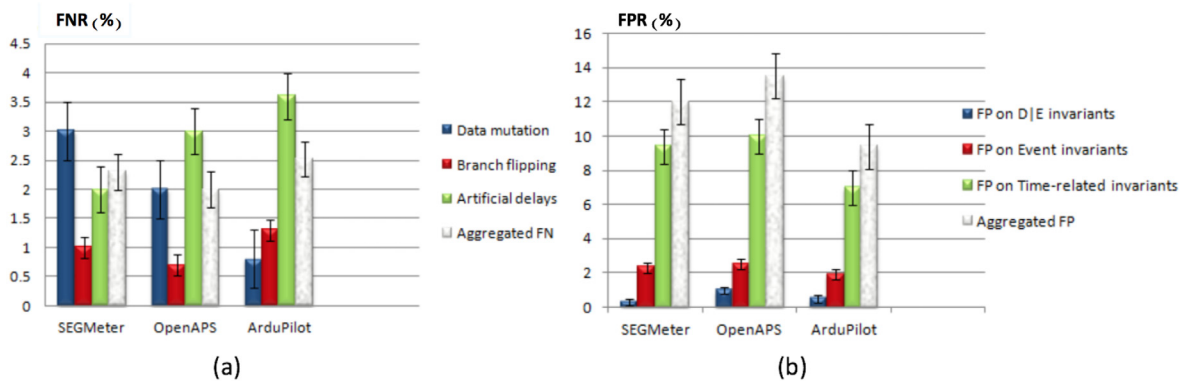

Fig. 13. (a) FN ratio(%) of IDS seeded by ARTINALI++ for different attack types, and (b) FP ratio(%) of the IDS seeded by ARTINALI++, across SEGMeter, OpenAPS and ArduPilot platforms. Error bars are shown for the 95% confidence interval.

22.9% FN ratio on ArduPilot system. In contrast, when the IDS is seeded by Perfume, Texada and Daikon the FN ratio was respectively 38.5%, 60.5% and 59% on average, across all platforms. Fig. 13(a) illustrates the FN ratio of the IDS for the three category of arbitrary attacks (fault injections), as well as the aggregated FN ratio, in SEGMeter, OpenAPS and ArduPilot platforms. As seen, the IDS exhibits 2%–3% FN ratio for data mutation attacks, lower than 1% FN ratio for branch flipping attacks, and 2%–3% FN ratio for artificial delay attacks.

Overall, the results support our hypothesis that a more comprehensive specification mining technique, such as ARTINALI++, which can find invariants and their constraints along four dimensions, detects a significantly larger amount of attacks (and hence has fewer FNs).

8.4. RQ3. False positives

In this section, we compare the FP ratio incurred by IDS for three CPS platforms using the invariants derived by ARTINALI++ against the invariants generated by the other tools (ARTINALI, Daikon, Perfume and Texada). The results are shown in Tables 8, 9 and 10 for the SEGMeter, OpenAPS and ArduPilot systems, respectively. The first observation is that ARTINALI++ incurs the same FP ratio as ARTINALI for the SEGMeter and OpenAPS systems, while improves it by 54% on Ardupilot system. We can also observe that in all CPSes, the use of ARTINALI++ invariants leads to significantly less false positives compared to the invariants generated by the other three tools (Daikon, Texada and Perfume). 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 54% to 78% improvement of the FP ratio for ArduPilot over the other tools.

Fig. 13(b) illustrates the FP ratio of the IDS seeded by ARTINALI++ for various categories of invariants that lead to false alarms, as well as the aggregated FP ratio, in three platforms. We grouped the FP-leading invariants in two classes: *time-related* invariants (such as $D|T$ and $E|T$ invariants) and *non-time-related* invariants (such as $D|E$ and event invariants). As can be seen in Fig. 13(b), the FP ratio incurred by non-time related invariants is less than 2.4% in all platforms, while the time-related invariants indicate 7%–9.6% FP ratio, and hence have the major contribution in generating false alarms. It can be sourced from the fact that time variations may be influenced by external attributes in a CPS operational environment, and hence providing the precise time properties that differentiate the correct timing behavior from the likely intrusions is more challenging.

8.5. RQ4. Effects of derivative and state-based invariants on IDS accuracy

We have also examined the effect of state-based and derivative invariants on the richness of the model generated by ARTINALI++ for ArduPilot platform. Using different sets of invariants, we defined four different models to feed the IDS. We considered the invariants generated by ARTINALI as the *base* model, and added derivative and state-based categories of invariants, inferred by ARTINALI++, one by one and altogether to the *base* model. So, our models include:

Model 1. Base, including Time, $D|E$, $E|T$ and $D|T$ invariants (i.e., ARTINALI invariants).
Model 2. Base + Derivative invariants.
Model 3. Base + State-aware invariants.
Model 4. Base + Derivative + State-aware invariants (i.e., ARTINALI++ invariants).

We separately seeded four models to IDS, and measured FP and FN ratios. Fig. 14 shows how each model improves the FN and FP ratios of IDS. We observed that derivative invariants reduce FP and FN ratios by 1.8% and 10.4%, respectively, that means they positively affect FN ratio much more than FP ratio. The insight

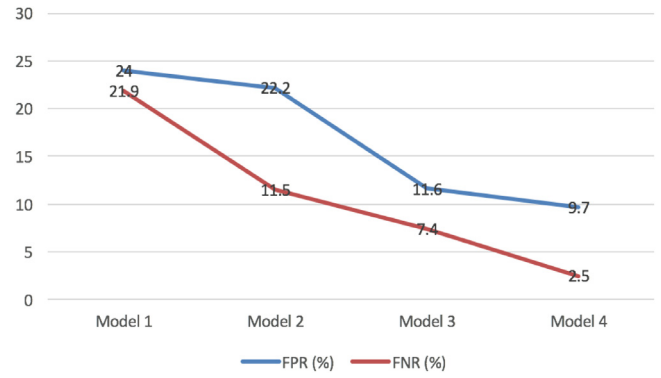

Fig. 14. Comparing FN and FP ratios of the IDS seeded by various models generated by ARTINALI++ for ArduPilot platform.

is that derivative invariants model a missing part of the system behavior (i.e., physical motions in space and time), that leads to a higher coverage (less FN ratio) in the IDS for ArduPilot. State-aware invariants, however, competitively improve the error ratios (14.5% and 12.4% reduction on FN and FP ratios, respectively). Furthermore, the state-aware invariants have more contribution than derivative invariants in the raising of the IDS accuracy for ArduPilot platform.

Overall, both derivative and state-based invariants significantly increase the IDS accuracy. The result supports our hypothesis that ARTINALI++ builds a richer model than that of ARTINALI for the complex CPSes such as ArduPilot platform.

8.6. 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 the three CPS platforms. Our results are shown in Table 11 ("Memory usage" row). Memory consumption depends on the size of IDS, the number of specifications that account for the CPS model, and the size of log files collected by tracing module. Generally, invariants that involve two or more dimensions carry more information than the invariants of one dimension, 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. The second source of difference in memory overhead is the difference in the amount of information that tracing module needs to collect, and it differs in each tool. For example, tracing module, needs to collect 4-D (data, event, time, and physical motion) information for ARTINALI++, and 3-D (data, event and time) information for ARTINALI, compared with Perfume (event and time), Texada (events only), and Daikon (data only). However, as instrumentation points and the mined invariants of ARTINALI++ are the same as those of ARTINALI on SEGMeter platform, the memory overhead remains the same.[4]

Overall, we find that the memory consumption of the IDS with ARTINALI++/ARTINALI invariants is lower than those with Perfume or Texada invariants, but higher than those with Daikon invariants. However, the memory usage for all tools is much lower than the available memory in SEGMeter (16 MB).

[4] Nevertheless, we believe that ARTINALI++ indicates a higher overhead on complex platforms due to more instrumentation points and higher number of specifications.

Table 11
Memory and performance overhead of IDS, seeded by ARTINALI++ and the other tools, running on SEGMeter.

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

8.7. RQ5. Performance overhead

In this section, we discuss the performance overhead of IDS running on the SEGMeter platform, which consists of an embedded micro-controller (Broadcom BCM3302 V2.9 240MHz CPU and 16 MB RAM) running Linux. Recall that the IDS consists of two components, namely tracing module and intrusion detector module. Table 11 (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. Our first finding is that the IDSes based on ARTINALI++ and ARTINALI indicate the same performance overhead on SEGMeter platform. This is due to the fact that both tools collect the same logs and mine the same set of specifications for smart meter. Our second finding is that ARTINALI/ARTINALI++ and Perfume have the highest aggregate overhead, followed by Daikon, and then Texada. The difference in performance overhead is due to the difference in the tracing module, which needs to collect different information for each tool.

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 11 (last part). As can be seen from the table, the entire cycle takes about 60 s (1 min). However, the execution of the IDS for each tool takes less than 20 s even in the worst case (for ARTINALI++/ARTINALI), which is much less than an 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. The specification mining process takes place offline, thus does not impact the performance overhead of the IDS running on the CPS platform. However, we measured the time to mine specifications using ARTINALI++, on a standard desktop system. We found that the time ranges from 8 to 96 s in SEGMeter, 6 to 36 s in OpenAPS and 31 to 178 s in ArduPilot. We observe that the time to mine specifications grows as the size of CPS's control program increases. However, specification mining is a one-time process and needs to be repeated only when the code is changed.

9. Threats to validity

An external threat to the validity is the limited number of CPS platforms considered (three). 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 three fairly diverse platforms, with different levels of complexity.

Another external threat to the validity is that the evaluation of proposed technique on complex CPS platforms is limited to only one system. We acknowledge that we have only used one class of complex CPS platforms (UAV) for evaluation purposes. However, UAV systems perfectly satisfy the requirements of complex CPS systems as they have multiple working modes as well as physical

motions. Moreover, implementation of specification mining capabilities for complex CPS platforms is not biased to UAV systems, and hence our technique can be generalized to any other complex system.

An internal threat to validity is using fault injection to evaluate our IDS. Fault injection does not necessarily represent all real attacks. However, it allows us to emulate the potential attacks without biasing the evaluation towards known exploits. We decreased this threat by using model-based fault injection used for emulating attacks in prior work.

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.

10. Conclusion

In this paper, we develop dynamic specification mining techniques to build intrusion detection systems for CPSes with arbitrary size and complexity. Our key insight is that time is a first class property in CPS systems, and hence we incorporate time into the specifications, in addition to data and events. We develop an efficient algorithm for mining specifications over the three dimensions of data, events and time, and implement it in a tool called ARTINALI++. To mine specifications for complex CPSes such as autonomous vehicles, we enhance the capabilities of ARTINALI++ to discover specifications that represent not only various operational states but also the physical motion in space and time. While the later is necessary for detecting suspicious velocity/accelerations, the former leads to a more sound and complete set of specifications for each operational state. We demonstrate the use of ARTINALI++ on three CPS platforms for intrusion detection. We find that they averagely indicate 97.7% detection accuracy across platforms, while incurring reasonable performance and memory overheads.

As future work, we plan to develop an optimization technique to maximize the attack detection accuracy while minimizing the overheads, and hence make the intrusion detection technique more scalable. Another potential direction for future work is to incorporate dynamic specifications for attack diagnosis and recovery. In particular, we plan to use Bayesian networks for fine-grained attack diagnosis through specifications, and perform system reconfiguration based on the results of the diagnosis.

CRediT authorship contribution statement

Maryam Raiyat Aliabadi: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data curation, Writing - original draft. **Mojtaba Vahidi Asl:** Conceptualization, Methodology, Formal analysis, Validation, Writing - review & editing, Supervision. **Ramak Ghavamizadeh:** Conceptualization, Methodology, Formal analysis, Validation, Writing - review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

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

References

Abrahamson, J., Beschastnikh, I., Brun, Y., Ernst, M.D., 2014. Shedding light on distributed system executions. In: Companion Proceedings of the 36th International Conference on Software Engineering. ACM, pp. 598–599.

Acharya, M., Xie, T., Pei, J., Xu, J., 2007. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. ACM, pp. 25–34.

Aghakhani, H., Machiry, A., Nilizadeh, S., Kruegel, C., Vigna, G., 2018. Detecting deceptive reviews using generative adversarial networks. In: 2018 IEEE Security and Privacy Workshops. SPW, IEEE, pp. 89–95.

Alemzadeh, H., Chen, D., Li, X., Kesavadas, T., Kalbarczyk, Z.T., Iyer, R.K., 2016. Targeted attacks on teleoperated surgical robots: Dynamic model-based detection and mitigation. In: Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on. IEEE, pp. 395–406.

Aliabadi, M.R., Kamath, A.A., Gascon-Samson, J., Pattabiraman, K., 2017. ARTINALI: dynamic invariant detection for cyber-physical system security. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, pp. 349–361.

An, D., Yang, Q., Liu, W., Zhang, Y., 2019. Defending against data integrity attacks in smart grid: A deep reinforcement learning-based approach. IEEE Access 7, 110835–110845.

Aniello, L., Ciccotelli, C., Cinque, M., Frattini, F., Querzoni, L., Russo, S., 2016. Automatic invariant selection for online anomaly detection. In: International Conference on Computer Safety, Reliability, and Security. Springer, pp. 172–183.

Anon, 2011. Smart energy groups home page. http://smartenergygroups.com.

Anon, 2012. ArduPilot (open-source drone project). https://github.com/diydrones/ardupilot.

Anon, 2014. Perfume user manual. http://people.cs.umass.edu/~ohmann/perfume/.

Anon, 2016. Texada user manual. https://bitbucket.org/bestchai/texada/.

Anon, 2017. The daikon invariant detector user manual. https://plse.cs.washington.edu/daikon/download/doc/daikon.html.

Bartocci, E., Deshmukh, J., Donzé, A., Fainekos, G., Maler, O., Ničković, D., Sankaranarayanan, S., 2018. Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In: Lectures on Runtime Verification. Springer, pp. 135–175.

Berthier, R., Sanders, W.H., Khurana, H., 2010. Intrusion detection for advanced metering infrastructures: Requirements and architectural directions. In: Smart Grid Communications (SmartGridComm), 2010 First IEEE International Conference on. IEEE, pp. 350–355.

Beschastnikh, I., Brun, Y., Abrahamson, J., Ernst, M.D., Krishnamurthy, A., 2015. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. IEEE Trans. Softw. Eng. 41 (4), 408–428.

Bian, P., Liang, B., Zhang, Y., Yang, C., Shi, W., Cai, Y., 2018. Detecting bugs by discovering expectations and their violations. IEEE Trans. Softw. Eng..

Cardenas, A.A., Amin, S., Sastry, S., 2008. Secure control: Towards survivable cyber-physical systems. In: Distributed Computing Systems Workshops, 2008. ICDCS'08. 28th International Conference on. IEEE, pp. 495–500.

Carreon, N., Gilbreath, A., Lysecky, R., 2019. Window-based statistical analysis of timing subcomponents for efficient detection of malware in life-critical systems. In: 2019 Spring Simulation Conference. SpringSim, IEEE, pp. 1–12.

Chalapathy, R., Chawla, S., 2019. Deep learning for anomaly detection: A survey. arXiv preprint arXiv:1901.03407.

Chang, R.-Y., Podgurski, A., Yang, J., 2007. Finding what's not there: a new approach to revealing neglected conditions in software. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis. ACM, pp. 163–173.

Chawla, A., Lee, B., Fallon, S., Jacob, P., 2018. Host based intrusion detection system with combined CNN/RNN model. In: Joint European Conference on Machine Learning and Knowledge Discovery in Databases. Springer, pp. 149–158.

Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., Kohno, T., et al., 2011. Comprehensive experimental analyses of automotive attack surfaces. In: USENIX Security Symposium. San Francisco.

Chen, L., Sultana, S., Sahita, R., 2018. Henet: A deep learning approach on intel® processor trace for effective exploit detection. In: 2018 IEEE Security and Privacy Workshops. SPW, IEEE, pp. 109–115.

Chien, E., Falliere, N., Murchu, L., 2010. W32. Stuxnet Dossier. Symantec security response.

Csallner, C., Tillmann, N., Smaragdakis, Y., 2008. DySy: Dynamic symbolic execution for invariant inference. In: Proceedings of the 30th International Conference on Software Engineering. ACM, pp. 281–290.

Deng, L., Li, D., Yao, X., Cox, D., Wang, H., 2019. Mobile network intrusion detection for IoT system based on transfer learning algorithm. Cluster Comput. 22 (4), 9889–9904.

Derler, P., Lee, E.A., Vincentelli, A.S., 2012. Modeling cyber–physical systems. Proc. IEEE 100 (1), 13–28.
Eidson, J., Lee, E.A., Matic, S., Seshia, S.A., Zou, J., 2010. A time-centric model for cyber-physical applications. In: Workshop on Model Based Architecting and Construction of Embedded Systems. ACES-MB, pp. 21–35.
Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D., 2001. Dynamically discovering likely program invariants to support program evolution. IEEE Trans. Softw. Eng. 27 (2), 99–123.
Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C., 2007. The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. 69 (1–3), 35–45.
Fernandes, E., Jung, J., Prakash, A., 2016. Security analysis of emerging smart home applications. In: Security and Privacy (SP), 2016 IEEE Symposium on. IEEE, pp. 636–654.
Giffin, J.T., Jha, S., Miller, B.P., 2004. Efficient Context-Sensitive Intrusion Detection. In: NDSS.
Goh, J., Adepu, S., Tan, M., Lee, Z.S., 2017. Anomaly detection in cyber physical systems using recurrent neural networks. In: 2017 IEEE 18th International Symposium on High Assurance Systems Engineering. HASE, IEEE, pp. 140–145.
Grahne, G., Zhu, J., 2005. Fast algorithms for frequent itemset mining using fp-trees. IEEE Trans. Knowl. Data Eng. 17 (10), 1347–1362.
Grant, S., Cech, H., Beschastnikh, I., 2018. Inferring and asserting distributed system invariants. In: Proceedings of the 40th International Conference on Software Engineering, pp. 1149–1159.
Han, X., Pasquier, T., Bates, A., Mickens, J., Seltzer, M., 2020. UNICORN: Runtime provenance-based detector for advanced persistent threats. arXiv preprint arXiv:2001.01525.
Han, X., Pasquier, T., Ranjan, T., Goldstein, M., Seltzer, M., 2017. FRAPpuccino: fault-detection through runtime analysis of provenance. In: 9th {USENIX} Workshop on Hot Topics in Cloud Computing, HotCloud 17.
Hangal, S., Lam, M.S., 2002. Tracking down software bugs using automatic anomaly detection. In: Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on. IEEE, pp. 291–301.
Javaid, A.Y., Sun, W., Devabhaktuni, V.K., Alam, M., 2012. Cyber security threat analysis and modeling of an unmanned aerial vehicle system. In: Homeland Security (HST), 2012 IEEE Conference on Technologies for. IEEE, pp. 585–590.
Jiang, H., Elbaum, S., Detweiler, C., 2013. Reducing failure rates of robotic systems though inferred invariants monitoring. In: Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on. IEEE, pp. 1899–1906.
Kamkar, S., 2013. SkyJack. https://samy.pl/skyjack/. [Online]. (Accessed 19 Dec 2013).
Kang, Y., Ray, B., Jana, S., 2016. APEx: Automated inference of error specifications for C APIs. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ACM, pp. 472–482.
Kim, G., Yi, H., Lee, J., Paek, Y., Yoon, S., 2016. LSTM-based system-call language modeling and robust ensemble method for designing host-based intrusion detection systems. arXiv preprint arXiv:1611.01726.
Kopetz, H., Bauer, G., 2003. The time-triggered architecture. Proc. IEEE 91 (1), 112–126.
Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., et al., 2010. Experimental security analysis of a modern automobile. In: 2010 IEEE Symposium on Security and Privacy. IEEE, pp. 447–462.
Kwon, C., Liu, W., Hwang, I., 2013. Security analysis for cyber-physical systems against stealthy deception attacks. In: American Control Conference (ACC), 2013. IEEE, pp. 3344–3349.
Leavitt, N., 2010. Researchers fight to keep implanted medical devices safe from hackers. Computer 43 (8), 11–14.
Lemieux, C., 2015. Mining temporal properties of data invariants. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 2. IEEE, pp. 751–753.
Lemieux, C., Park, D., Beschastnikh, I., 2015a. General LTL specification mining (t). In: Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on. IEEE, pp. 81–92.
Lemieux, C., Park, D., Beschastnikh, I., 2015b. General LTL specification mining (T). In: Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on. IEEE, pp. 81–92.
Lewis, D., 2015. Introducing the# OpenAPS project.
Li, C., Raghunathan, A., Jha, N.K., 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, pp. 150–156.
Li, Z., Zhou, Y., 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In: ACM SIGSOFT Software Engineering Notes, Vol. 30, No. 5. ACM, pp. 306–315.
Liang, B., Bian, P., Zhang, Y., Shi, W., You, W., Cai, Y., 2016. AntMiner: mining more bugs by reducing noise interference. In: Proceedings of the 38th International Conference on Software Engineering. ACM, pp. 333–344.
Lin, H., Alemzadeh, H., Chen, D., Kalbarczyk, Z., Iyer, R.K., 2016. Safety-critical cyber-physical attacks: Analysis, detection, and mitigation. In: Proceedings of the Symposium and Bootcamp on the Science of Security. ACM, pp. 82–89.
Liu, Y., Peng, Y., Wang, B., Yao, S., Liu, Z., 2017. Review on cyber-physical systems. IEEE/CAA J. Autom. Sin. 4 (1), 27–40.
Livshits, B., Zimmermann, T., 2005. DynaMine: finding common error patterns by mining software revision histories. In: ACM SIGSOFT Software Engineering Notes, Vol. 30, No. 5. ACM, pp. 296–305.
Lo, D., Maoz, S., 2008. Specification mining of symbolic scenario-based models. In: Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. ACM, pp. 29–35.
Lorenzoli, D., Mariani, L., Pezzè, M., 2008. Automatic generation of software behavioral models. In: Proceedings of the 30th International Conference on Software Engineering. ACM, pp. 501–510.
Lu, S., Park, S., Hu, C., Ma, X., Jiang, W., Li, Z., Popa, R.A., Zhou, Y., 2007. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In: ACM SIGOPS Operating Systems Review, Vol. 41, No. 6. ACM, pp. 103–116.
McCallie, D., Butts, J., Mills, R., 2011. Security analysis of the ADS-B implementation in the next generation air transportation system. Int. J. Crit. Infrastruct. Prot. 4 (2), 78–87.
McLaughlin, S., Podkuiko, D., Miadzvezhanka, S., Delozier, A., McDaniel, P., 2010. Multi-vendor penetration testing in the advanced metering infrastructure. In: Proceedings of the 26th Annual Computer Security Applications Conference. ACM, pp. 107–116.
Mitchell, R., Chen, I.-R., 2014. A survey of intrusion detection techniques for cyber-physical systems. ACM Comput. Surv. 46 (4), 55.
Mohammadi, M., Al-Fuqaha, A., Sorour, S., Guizani, M., 2018. Deep learning for IoT big data and streaming analytics: A survey. IEEE Commun. Surv. Tutor. 20 (4), 2923–2960.
Nuzzo, P., 2019. From electronic design automation to cyber-physical system design automation: A tale of platforms and contracts. In: ISPD, pp. 117–121.
Ohmann, T., Herzberg, M., Fiss, S., Halbert, A., Palyart, M., Beschastnikh, I., Brun, Y., 2014. Behavioral resource-aware model inference. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. ACM, pp. 19–30.
Palyvos-Giannas, D., Gulisano, V., Papatriantafilou, M., 2018. Genealog: Fine-grained data streaming provenance at the edge. In: Proceedings of the 19th International Middleware Conference, pp. 227–238.
Radcliffe, J., 2011. Hacking medical devices for fun and insulin: Breaking the human SCADA system. In: Black Hat Conference Presentation Slides, Vol. 2011.
Samland, F., Fruth, J., Hildebrandt, M., Hoppe, T., Dittmann, J., 2012. AR. Drone: security threat analysis and exemplary attack to track persons. In: Proceedings of the SPIE, Vol. 8301.
Schumann, J., Moosbrugger, P., Rozier, K.Y., 2015. R2u2: monitoring and diagnosis of security threats for unmanned aerial systems. In: Runtime Verification. Springer, pp. 233–249.
Skopik, F., Ma, Z., Bleier, T., Grüneis, H., 2012. A survey on threats and vulnerabilities in smart metering infrastructures. Int. J. Smart Grid Clean Energy 1 (1), 22–28.
Smith, S.W., 2009. Security and privacy challenges in the smart grid.
Sokolova, M., Japkowicz, N., Szpakowicz, S., 2006. Beyond accuracy, F-score and ROC: a family of discriminant measures for performance evaluation. In: Australasian Joint Conference on Artificial Intelligence. Springer, pp. 1015–1021.
Son, Y., Shin, H., Kim, D., Park, Y.-S., Noh, J., Choi, K., Choi, J., Kim, Y., et al., 2015. Rocking drones with intentional sound noise on gyroscopic sensors. In: USENIX Security. pp. 881–896.
Späth, J., Ali, K., Bodden, E., 2019. Context-, flow-, and field-sensitive data-flow analysis using synchronized Pushdown systems. Proc. ACM Program. Lang. 3 (POPL), 1–29.
Stevens, R., Suciu, O., Ruef, A., Hong, S., Hicks, M., Dumitraş, T., 2017. Summoning demons: The pursuit of exploitable bugs in machine learning. arXiv preprint arXiv:1701.04739.
Suhail, S., Hong, C.S., Ahmad, Z.U., Zafar, F., Khan, A., 2016. Introducing secure provenance in iot: Requirements and challenges. In: 2016 International Workshop on Secure Internet of Things. SIoT, IEEE, pp. 39–46.
Talcott, C., 2008. Cyber-physical systems and events. In: Software-Intensive Systems and New Computing Paradigms. Springer, pp. 101–115.
Tan, L., Zhang, X., Ma, X., Xiong, W., Zhou, Y., 2008. AutoISES: Automatically Inferring Security Specification and Detecting Violations. In: USENIX Security Symposium, pp. 379–394.
Tange, K., De Donno, M., Fafoutis, X., Dragoni, N., 2019. Towards a systematic survey of industrial IoT security requirements: research method and quantitative analysis. In: Proceedings of the Workshop on Fog Computing and the IoT, pp. 56–63.
Thummalapenta, S., Xie, T., 2009. Alattin: Mining alternative patterns for detecting neglected conditions. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, pp. 283–294.

Wagner, D., Dean, R., 2001. Intrusion detection via static analysis. In: Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on. IEEE, pp. 156–168.
Warner, J.S., Johnston, R.G., 2003. GPS spoofing countermeasures. Homeland Security J. 25 (2), 19–27.
Wegener, J., Grochtmann, M., 1998. Verifying timing constraints of real-time systems by means of evolutionary testing. Real-Time Syst. 15 (3), 275–298.
Yan, Y., Qian, Y., Sharif, H., Tipper, D., 2012. A survey on cyber security for smart grid communications. IEEE Commun. Surv. Tutor. 14 (4), 998–1010.
Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M., 2006. Perracotta: mining temporal API rules from imperfect traces. In: Proceedings of the 28th International Conference on Software Engineering. ACM, pp. 282–291.
Yoon, M.-K., Mohan, S., Choi, J., Christodorescu, M., Sha, L., 2017. Learning execution contexts from system call distribution for anomaly detection in smart embedded system. In: Proceedings of the Second International Conference on Internet-of-Things Design and Implementation, pp. 191–196.
Yoon, M.-K., Mohan, S., Choi, J., Kim, J.-E., Sha, L., 2013. SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems. In: 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium. RTAS, IEEE, pp. 21–32.

Maryam Raiyat Aliabadi is a Postdoctoral Fellow in the Faculty of Computer Science at University of British Columbia, Vancouver, Canada. She pursued her Ph.D. in the Faculty of Computer Science and Engineering at Shahid Beheshti University, Tehran, Iran and University of British Columbia, Vancouver, Canada. She received her B.Sc. in Electrical engineering from Shahid Beheshti University, and M.Sc. in Computer engineering from AZAD Tehran University. Her main research interests include security and reliability of Cyber-Physical Systems, program analysis, software testing and debugging.

Mojtaba Vahidi Asl is Assistant Professor in the Faculty of Computer Science and Engineering at Shahid Beheshti University, Tehran, Iran. He received his B.S. in Computer Engineering from AmirKabir University of Technology, M.S. and Ph.D. degree in Software Engineering from Iran University of Science and Technology. His research area includes program analysis, software testing and debugging and IOT.

Ramak Ghavamizadeh is Assistant Professor in the Faculty of Computer Science and Engineering at Shahid Beheshti University, Tehran, Iran. She received her BS in Computer Engineering from Jussieu University (Paris VII), M.S. and Ph.D. degree in Software Engineering from Orsay University (Paris XI) in France. Her research area includes algorithms and complexity, program analysis and software testing and debugging