5-2018

Learning from mutants: Using code mutation to learn and monitor invariants of a cyber-physical system

Yuqi CHEN

Christopher M. POSKITT
Singapore Management University, cposkitt@smu.edu.sg

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Learning from Mutants: Using Code Mutation to Learn and Monitor Invariants of a Cyber-Physical System

Yuqi Chen, Christopher M. Poskitt, and Jun Sun
Singapore University of Technology and Design
Singapore, Singapore
Email: yuqi_chen@mymail.sutd.edu.sg; {chris_poskitt, sunjun}@sutd.edu.sg

***Abstract*—Cyber-physical systems (CPS) consist of sensors, actuators, and controllers all communicating over a network; if any subset becomes compromised, an attacker could cause significant damage. With access to data logs and a model of the CPS, the physical effects of an attack could potentially be detected before any damage is done. Manually building a model that is accurate enough in practice, however, is extremely difficult. In this paper, we propose a novel approach for constructing models of CPS automatically, by applying supervised machine learning to data traces obtained after systematically seeding their software components with faults ("mutants"). We demonstrate the efficacy of this approach on the simulator of a real-world water purification plant, presenting a framework that automatically generates mutants, collects data traces, and learns an SVM-based model. Using cross-validation and statistical model checking, we show that the learnt model characterises an invariant physical property of the system. Furthermore, we demonstrate the usefulness of the invariant by subjecting the system to 55 network and code-modification attacks, and showing that it can detect 85% of them from the data logs generated at runtime.**

I. Introduction

Cyber-physical systems (CPS), in which software components and physical processes are deeply intertwined, are found across engineering domains as diverse as aerospace, autonomous vehicles, and medical monitoring; they are also increasingly prevalent in public infrastructure, automating critical operations such as the management of electricity demands in the grid, or the purification of raw water [1, 2]. In such applications, CPS typically consist of distributed software components engaging with physical processes via sensors and actuators, all connected over a network. A compromised software component, sensor, or network has the potential to cause considerable damage by driving the actuators into states that are incompatible with the physical conditions [3], motivating research into practical approaches for monitoring and attesting CPS to ensure that they are operating safely and as intended.

Reasoning about the behaviour exhibited by a CPS, however, is very challenging, given the tight integration of algorithmic control in the "cyber" part with continuous behaviour in the "physical" part [4]. While the software components are often simple when viewed in isolation, this simplicity betrays the typical complexity of a CPS when taken as a whole: even with domain-specific expertise, manually deriving accurate models of the physical processes (e.g. ODEs, hybrid automata) can be extremely difficult—if not impossible. This is unfortunate, since with an accurate mathematical model, a supervisory system could query real CPS data traces and determine whether they represent correct or compromised behaviour, raising the alarm for the latter.

In this work, using a high degree of automation, we aim to overcome the challenge of constructing CPS models that are useful for detecting attacks in practice. In particular, we propose to apply machine learning (ML) on traces of sensor data to construct models that characterise *invariant properties*—conditions that must hold in all states amongst the physical processes controlled by the CPS—and to make those invariants checkable at runtime. In contrast to existing unsupervised approaches (e.g. [5, 6]), we propose a supervised approach to learning that trains on traces of sensor data representing "normal" runs (the positive case, satisfying the invariant) as well as traces representing abnormal behaviour (the negative case), in order to learn a model that characterises the border between them effectively. To systematically generate the negative traces, we propose the novel application of code mutation (à la mutation testing [7]) to the software components of CPS. Motivating this approach is the idea that small syntactic changes may correspond more closely to an attacker attempting to be subtle and undetected. Once a CPS model is learnt, we propose to use statistical model checking [8] to ascertain that it is *actually* an invariant of the CPS, allowing for its use in applications such as the physical attestation of software components [9] or runtime monitoring for attacks.

In order to evaluate this approach, we apply it to Secure Water Treatment (SWaT) [10], a scaled-down but fully operational water treatment testbed at the Singapore University of Technology and Design, capable of producing five gallons of safe drinking water per minute. SWaT has industry-standard control software across its six Programmable Logic Controllers (PLCs). While the software is structurally simple, it must interact with physical processes that are very difficult to reason about, since they are governed by laws of physics concerning the dynamics of water flow, the evolution of pH values, and the various chemical processes associated with treating raw water. In this paper, we focus on water

flow: we learn invariants characterising how water tank levels evolve over time, and show their usefulness in detecting both manipulations of the control software (i.e. attestation) as well as detecting attacks in the network that manipulate the sensor readings and actuator signals. Our experiments take place on a simulator of SWaT due to resource restrictions and safety concerns, but the simulator is faithful and reasonable: it implements the same PLC code as the testbed, and has a cross-validated physical model for water flow.

Our Contributions. This paper proposes a novel approach for generating models of CPS, based on the application of supervised machine learning to traces of sensor data obtained after systematically mutating software components. To demonstrate the efficacy of the approach, we present a framework for the SWaT simulator that: (1) automatically generates mutated PLC programs; (2) automatically generates a large dataset of normal and abnormal traces; and (3) applies Support Vector Machines (SVM) to learn a model. We apply cross-validation and statistical model checking to show that the model characterises an invariant physical property of the system. Finally, we demonstrate the usefulness of the invariant in two applications: (1) code attestation, i.e. detecting modifications to the control software through their effects on physical processes; and (2) identifying standard network attacks, in which sensor readings and actuator signals are manipulated.

This work follows from the ideas presented in our earlier position paper [11], but differs significantly. In particular, the preliminary experiment in [11] was entirely manual, used (insufficiently expressive) linear classifiers, had a very limited dataset, and only briefly discussed how the invariants might be evaluated. In the present paper, we work with significantly larger datasets that are generated automatically, learn much more expressive classifiers using kernel methods, use a systematic approach to feature vector labelling, apply statistical model checking to validate the model, and assess its usefulness for detecting network and code-modification attacks.

Organisation. The remainder of the paper is organised as follows. In Section II, we introduce the SWaT water treatment system as our motivating case study, and present a high-level overview of our approach. In Section III, we describe in detail the main steps of our approach, as well as how it is implemented for the SWaT simulator. In Section IV, we evaluate our approach with respect to five research questions. In Section V, we highlight some additional related work. Finally, in Section VI, we conclude and suggest some directions for future work.

II. Motivation and Overview

In this section, we introduce SWaT, the water treatment CPS that provides our motivation for learning and monitoring invariants, and also forms the case study for this paper. Following this, we present a high-level overview of our learning approach and how it can be applied to CPS.

Fig. 1. The Secure Water Treatment (SWaT) testbed

A. Motivational Case Study: SWaT Testbed

The CPS that forms the case study for our paper is Secure Water Treatment (SWaT) [10], a testbed built for cyber-security research at the Singapore University of Technology and Design (Figure 1). SWaT is a scaled-down but fully operational raw water purification plant, capable of producing five gallons of safe drinking water per minute. Raw water is treated in six distinct but co-operating stages, handling chemical processes such as ultrafiltration, de-chlorination, and reverse osmosis.

Each stage of SWaT consists of a dedicated programmable logic controller (PLC), which communicates over a ring network with some sensors and actuators that interact with the physical environment. The sensors and actuators vary from stage-to-stage, but a typical sensor in SWaT might read the level of a water tank or the water flow rate in some pipe, whereas a typical actuator might operate a motorised valve (for opening an inflow pipe) or a pump (for emptying a tank). A historian records the sensor readings and actuator signals, facilitating large datasets for offline analyses [12].

The PLCs are responsible for algorithmic control in the six stages, repeatedly reading sensor data and computing the appropriate signals to send to actuators. The programs that PLCs cycle through every 5ms are structurally simple. They do not contain any loops, for example, and can essentially be viewed as large, nested conditional statements for determining the interactions with the system's 42 sensors and actuators. The programs can easily be viewed (in both a graphical and textual style), modified, and re-deployed to the PLCs using Rockwell's RSLogix 5000, an industry-standard software suite.

In addition to the testbed itself, a SWaT simulator [13] was also developed at the Singapore University of Technology and Design. Built in Python, the simulator faithfully simulates the cyber part of SWaT, as a direct translation of the PLC programs was possible. Inevitably, the physical part (taking

Algorithm 1: Sketch of Overall Algorithm

Input: A CPS S
Output: An invariant ϕ

```
Randomly simulate S for n times and collect a set of normal traces Tr;
Construct a set of mutants Mu from S;
Collect a set of positive feature vectors Po from Tr;
Collect a set of negative feature vectors Ne based on abnormal traces from Mu;
Learn a classifier φ;
Apply statistical model checking to validate φ;
if φ satisfies our stopping criteria then
    return φ;
Restart with additional data;
```

advantage of Python's scientific libraries, e.g. NumPy, SciPy) is less accurate since the actual ODEs governing SWaT are unknown. The simulator currently models some of the simpler physical processes such as water flow (omitting, for example, models of the chemical processes), the accuracy of which has been improved over time by cross-validating data from the simulator with real SWaT data collected by the historian [12]. As a result, the simulator is especially faithful and useful for investigating over- and underflow attacks on the water tanks.

The SWaT testbed characterises many of the security concerns that come with the increasing automation of public infrastructure. What happens, for example, if part of the network is compromised and packets can be manipulated; or if a PLC itself is compromised and the control software replaced? If undetected, the system could be driven into a state that causes physical damage, e.g. activating the pumps of an empty tank, or causing another one to overflow. The problem (which this paper aims to overcome) is that detecting an attack at runtime is very difficult, since a monitor must be able to query live data against a model of how SWaT is actually expected to behave, and this model must incorporate the physical processes. As mentioned, the PLC programs in isolation are very simple and amenable to formal analysis, but it is impossible to reason about the system as a whole without incorporating some knowledge of the physical effects of actuators over time.

B. Overview of Our Approach

Our approach for constructing CPS models consists of three main steps, as sketched in Algorithm 1: (1) simulating the CPS under different code mutations to collect a set of normal and abnormal system traces; (2) constructing feature vectors based on the two sets of traces and learning a classifier; and (3) applying statistical model checking to determine whether the classifier is an invariant, restarting the process if it is not. In the following we provide a high-level overview of how these three steps are applied in general. A more detailed presentation of the steps and their application to the SWaT simulator are given later, in Section III.

In the first step, traces (e.g. of sensor readings) representing normal system behaviour are obtained by randomly simulating the CPS under normal operating conditions, i.e. with the cyber part (PLCs) and physical part (ODEs) unaltered. To collect traces representing abnormal behaviour, our approach proposes simulating the CPS under small manipulations. Since we aim for our learnt invariants to be useful in detecting PLC and network attacks (as opposed to attackers tampering directly with the environment), we limit our manipulations to the cyber part, and propose a systematic method motivated by the assumption that attackers would attempt to be subtle in their manipulations. Our approach is inspired by mutation testing [7], a fault-based testing technique that deliberately seeds errors—small, syntactic transformations called *mutations*—into multiple copies of a program. Mutation testing is typically used to assess the quality of a test suite (i.e. good suites should detect the mutants), but in our approach, we generate mutants from the original PLC programs, and use these modified instances of the CPS to collect abnormal data traces.

In the second step, we extract positive and negative feature vectors from the normal and abnormal data traces respectively. Since an attack (i.e. some modification of a PLC program or a network attack) takes time to affect the physical processes, our feature vectors are pairs of sensor readings taken at fixed time intervals. While feature vectors can be extracted from the normal traces immediately, some pre-processing is required before they can be extracted from the abnormal ones: the mutations in some mutant PLC programs may never have been executed, or only executed after a certain number of system cycles, leading to traces either totally or partially indistinguishable from positive ones. To overcome this, we compare abnormal traces with normal ones obtained from the same initial states, discarding wholly indistinguishable traces, and then extracting pairs of sensor readings only when discrepancies are detected. With the feature vectors collected, we apply a supervised ML algorithm, e.g. Support Vector Machines (SVM), to learn a classifier.

In the third step, we must validate that the classifier is actually an invariant of the CPS. After applying standard ML cross-validation to minimise generalisation error, we apply statistical model checking (SMC) [8] to establish whether or not there is statistical evidence that the model is an invariant. In SMC, additional normal traces of the CPS are observed, and statistical estimation or hypothesis testing (e.g. the sequential probability ratio test (SPRT) [14]) is used to estimate the probability of the classifier's correctness. If the probability is high (i.e. above some predetermined threshold), we take that classifier as our invariant. Otherwise, we repeat the whole process with different randomly sampled data.

With a CPS invariant learnt, a supervisory system can monitor live data from the system and query it against the invariant, raising an alarm when it is not satisfied. This has at least two applications in defending against attacks. First, it can be used to detect standard network attacks, where packets have been manipulated and actuators are shifted into states

that are inappropriate for the current physical environment. Second, it can be seen as a form of code attestation: if the actual behaviour of a CPS does not satisfy our mathematical model of it (i.e. the invariant), then it is possible that the cyber part has been compromised and that ill-intended manipulations are occurring. This form of attestation is known as *physical attestation* [9, 15], and while weaker than typical software- and hardware-based attestation schemes (e.g. [16–19]), it is much more lightweight as neither the firmware nor the hardware of the PLCs need to be modified.

III. Implementing Our Approach

In this section, we describe in detail the main steps of our approach: (1) generating mutants and data traces; (2) collecting positive and negative feature vectors for learning a classifier; and (3) validating the classifier.

We illustrate each of the steps in turn by applying them to the SWaT simulator. We remark that our choice to use the SWaT simulator (rather than the testbed) has some important advantages for this paper. It allows us to automate each step in an experimental framework, with which we can easily investigate the effects of different parameters on the accuracy of learnt models. Furthermore, mutations can be applied and attacks can be simulated without the risk of damage, and the usefulness of learnt invariants can be assessed without wasting resources (e.g. water, chemicals) or navigating the policy restrictions of the testbed. Obtaining an invariant for the testbed can be achieved by re-running the trace collection phase on SWaT with optimised parameters for learning (see Section IV), or improving the accuracy of the physical model in the simulator to the extent that learnt classifiers can be validated as invariants of both the simulator and the testbed.

A. First Step: Generating Mutants and Traces

The first step of our approach is collecting the traces of raw sensor data that will subsequently be used for learning a CPS invariant. It consists of the following sub-steps: (i) fixing a set of initial physical configurations and a time interval for taking sensor readings; (ii) generating data traces that represent normal system behaviour; (iii) applying mutations and generating the (possibly) abnormal traces they produce.

Sub-step (i): Initial Configurations. In order to collect a set of data that captures the CPS' behaviour across a variety of physical contexts, a set of initial configurations should be chosen that covers the extremities of the sensors' ranges, as well as randomly selected combinations of values within them. A time interval for logging sensor readings (e.g. the historian's default) should also be chosen, as well as a length of time to run the CPS from each initial configuration.

Applied to SWaT. In the case of the SWaT simulator, since it only models physical processes concerning water flow, we collect traces of data from sensors recording the water levels in the five tanks. In particular, physical configurations are expressed in terms of the water levels recorded by these five sensors. The set of initial configurations we use in our experiments (see Section IV) therefore includes different combinations of water tank levels, including extreme values (i.e. tanks being full or empty). We choose to log the sensor values every 5ms, corresponding to the default time interval at which the simulator logs data. We fix 30 minutes as the length of time to run the simulator from each configuration, as previous experimentation has shown that the simulator's physical model remains accurate for at least this length of time.

Sub-step (ii): Normal Traces. To generate normal traces, we simply launch the CPS under normal operating conditions from each initial physical configuration, using the run length and time interval fixed in sub-step (i). The traces of sensor data should be extracted from the historian for processing in a later step.

Applied to SWaT. For our case study, we built a framework [13] around the SWaT simulator that can automatically launch and run the software on each of the initial configurations chosen earlier. Each run uses the original (i.e. unaltered) PLC programs, lasts for 30 minutes, and logs the simulated water tank levels every 5ms. These logs are stored as raw text files from which feature vectors are extracted in a later step.

Sub-step (iii): Mutants and Abnormal Traces. Next, we need to generate data traces representing abnormal system behaviour. In order to learn a classifier that is as close to the boundary of normal and abnormal behaviour as possible, we generate these traces after subjecting the control software to small syntactic code changes (i.e. mutations). These code changes are the result of applying simple mutation operators, which randomly replace some Boolean operator, logical connector, arithmetic function symbol, constant, or variable. To ensure a diverse enough training set, we generate abnormal traces from multiple versions of the control software representing a variety of different mutations.

Our approach for generating mutant PLC programs is summarised in Algorithm 2. Given a set of co-operating PLC programs, the algorithm makes a copy of all of them, and applies an applicable mutation operator to a single PLC program in the set.

Applied to SWaT. In the case of the SWaT simulator, our framework can automatically and randomly generate multiple mutant simulators. Note that each mutant simulator, built up of six PLC programs, consists of one mutation only in a PLC program chosen at random. Since the PLC programs are syntactically simple, we need only six mutation operators (Table I). Evidence suggests that additional mutation operators are unlikely to increase coverage [20], so our mutant simulators should be sufficiently varied.

To illustrate, consider the code in Listing 1, a small extract from the PLC program controlling ultrafiltration in SWaT. If the guard conditions are met, line 5 will change the state of the PLC to "19". This number identifies a branch in a case statement (not listed) that triggers the signals that should be sent to actuators. Now consider Listing 2: this PLC program

Listing 1
SNIPPET OF UNMODIFIED CONTROL CODE FROM PLC #3

```
1 if Sec_P:
2   MI.Cy_P3.CIP_CLEANING_SEC=HMI.Cy_P3.
        CIP_CLEANING_SEC+1
3   if HMI.Cy_P3.CIP_CLEANING_SEC>HMI.
        Cy_P3.CIP_CLEANING_SEC_SP or self
        .Mid_NEXT:
4         self.Mid_NEXT=0
5         HMI.P3.State=19
6 break
```

Listing 2
A POSSIBLE MUTANT OBTAINED FROM LISTING 1

```
1 if Sec_P:
2   MI.Cy_P3.CIP_CLEANING_SEC=HMI.Cy_P3.
        CIP_CLEANING_SEC+1
3   if HMI.Cy_P3.CIP_CLEANING_SEC>HMI.
        Cy_P3.CIP_CLEANING_SEC_SP or self
        .Mid_NEXT:
4         self.Mid_NEXT=0
5         HMI.P3.State=14
6 break
```

Algorithm 2: Generating Mutant PLC Code

Input: A set of PLC programs S
Output: A mutant set of PLC programs S_M

```
Let Ops be the set of mutation operators;
Make a copy S_M of the PLC programs S;
applied := false;
while ¬applied do
    Randomly choose a PLC P from S_M;
    Randomly choose a line number i from P;
    if some operator in Ops is applicable to line i then
        Apply an applicable operator to line i;
        applied := true;
return S_M;
```

TABLE I
MUTATION OPERATORS

Mutation Operator	Example
Scalar Variable Replacement	$x = a \rightsquigarrow x = b$
Arithmetic Operator Replacement	$a + b \rightsquigarrow a - b$
Relational Operator Replacement	$a > b \rightsquigarrow a \geq b$
Guard Valuation Replacement	$\texttt{if}(c) \rightsquigarrow \texttt{if(false)}$
Logical Connector Replacement	$a \texttt{ and } b \rightsquigarrow a \texttt{ or } b$
Assignment Operator Replacement	$x = a \rightsquigarrow x \mathrel{+}= a$

is identical to Listing 1, except for the result of a scalar mutation on line 5 that means the PLC would be set to state "14" instead. If executed, different signals will be sent to the actuators, potentially causing abnormal effects on the physical state—as might be the goal of an attacker.

Once we have our mutant simulators, we discard any that cannot be compiled. Of the mutants remaining, we run them with respect to each initial state for 30 minutes, logging the levels of all the water tanks every 5ms.

The current implementation of our mutant simulator generator for SWaT is available online [13], consisting of just over 200 lines of Python code. It applies mutations to the PLC programs by reading them as text files, randomly choosing a line, and then randomly applying an applicable mutation operator (Table I) by matching and substituting. This takes a negligible amount of time, so hundreds of mutant simulators can be generated very quickly (i.e. in seconds).

B. Second Step: Collecting Feature Vectors, Learning

At this point, we have a collection of raw data traces generated by normal PLC programs as well as by multiple mutant PLC programs. The second step is to extract positive and negative feature vectors from this data to perform supervised learning. It consists of the following sub-steps: (i) fixing a feature vector type; (ii) collecting feature vectors from the data, undersampling the abnormal data to maintain balance; (iii) applying a supervised learning algorithm.

Sub-step (i): Feature Vector Type. A feature vector type must be defined that appropriately represents objects of the data. For traces of sensor data, a simple feature vector would consist of the sensor values at any given time point. For typical CPS however, such a feature vector is far too simple, since it does not encapsulate any information about how the values evolve over the time series—an intrinsic part of the physical model. A more useful feature vector would record the values at fixed time intervals, making it possible to learn patterns about how the levels of tanks change over the time series.

Applied to SWaT. In the case of the SWaT simulator, we define our feature vectors to be of the form (π, π'), where π denotes the water tank levels at a certain time and π' denotes the values of the same tanks after d time units, where d is some fixed time interval that is a multiple of the interval at which data is logged (we compare the effects of different values of d in Section IV-B). Our feature vectors are based on the sliding window method that is commonly used for time series data [21].

Sub-step (ii): Collecting Feature Vectors. Next, the raw normal and abnormal data traces must be organised into positive and negative feature vectors of the type chosen in sub-step (i). Extracting positive feature vectors from the normal data is straightforward, but for negative feature vectors, we have the additional difficulty that mutants are not guaranteed to be *effective*, i.e. able to produce data traces distinguishable from normal ones. Furthermore, even effective mutants may

Algorithm 3: Collecting Feature Vectors

Input: Set of normal traces T_N and abnormal traces T_A, each trace of uniform size N
Output: Set of positive feature vectors Po; set of negative feature vectors Ne

Let S be the unmodified simulator;
Let t be the time interval for logging data in traces;
Let d be the time interval for feature vectors;
$x := 0$; $Po := \emptyset$; $Ne := \emptyset$;
foreach $Tr \in T_N$ **do**
  while $x + (d/t) < N$ **do**
    $\pi := \langle s_0, s_1, \dots \rangle$ for all sensor values s_i at row x of Tr;
    $\pi' := \langle s'_0, s'_1, \dots \rangle$ for all sensor values s'_i at row $x + (d/t)$ of Tr;
    $Po := Po \cup \{(\pi, \pi')\}$
    $x := x + 1$;
$x := 0$;
foreach $Tr \in T_A$ **do**
  while $x + (d/t) < N$ **do**
    $\pi := \langle s_0, s_1, \dots \rangle$ for all sensor values s_i at row x of Tr;
    $\pi' := \langle s'_0, s'_1, \dots \rangle$ for all sensor values s'_i at row $x + (d/t)$ of Tr;
    Run simulator S on configuration π for d time units to yield trace Tr';
    $\pi'' := \langle s''_0, s''_1, \dots \rangle$ for all sensor values s''_i at row d/t of Tr';
    if $\pi' \neq \pi''$ **then**
      $Ne := Ne \cup \{(\pi, \pi')\}$
    $x := x + 1$;
return Po, Ne;

not cause an immediate change. It is crucial not to mislabel normal data as abnormal—additional filtering is required.

Applied to SWaT. Algorithm 3 summarises how feature vectors are collected from the SWaT simulator and its mutants. Collecting positive feature vectors is very simple: all possible pairs of physical states (π, π') are extracted from the normal traces. For each pair (π, π') extracted from the abnormal traces, the unmodified simulator is run on π for d time units: if the unmodified simulator leads to a state distinguishable from π', the original pair is collected as a negative feature vector; if it leads to a state that is indistinguishable from π', it is discarded (since the mutation had no effect). In the case of SWaT, its simulator is deterministic, allowing for this judgement to be made easily. (For data from the testbed, some acceptable level of tolerance would need to be defined.)

Sub-step (iii): Learning. Once the feature vectors are collected, a supervised ML algorithm can be applied to learn a model.

Applied to SWaT. For the SWaT simulator, we choose to apply SVM as our supervised ML approach since it is fully automatic, with well-developed active learning strategies, and good library support (we use LIBSVM [22]). Furthermore, SVM has expressive kernels and has often been successfully applied to time series prediction [23]. Based on the training data, SVM attempts to learn the (unknown) boundary that separates it. Different classification functions exist for expressing this boundary, ranging from ones that attempt to find a simple linear separation between the data, to non-linear solutions based on RBF (we compare different classification functions for SWaT in Section IV-A). For the purpose of validating the classifier and assessing its generalisability, it is important to train it on only a portion of the feature vectors, reserving a portion of the data for testing. We randomly select 70% of the feature vectors to use as the training set, reserving the rest for evaluation.

We remark that SVM can struggle to learn a reasonable classifier if the data is very unbalanced. This is the case for the SWaT simulator: we have just one simulator for normal data, but potentially infinite mutant simulators for generating abnormal data. To ensure balance, we undersample the negative feature vectors. Let N_{Po} denote the number of positive feature vectors and N_{Ne} the number of negative feature vectors we collected. We partition the negative feature vectors into subsets of size N_{Ne}/N_{Po} (rounded up to the nearest integer), and randomly select a feature vector from each one. This leads to an undersampled set of negative feature vectors that is roughly the same size as the positive feature vector set.

C. Third Step: Validating the Classifier

At this point, we have collected normal and abnormal data, processed it into positive and negative feature vectors, and learnt a classifier by applying a supervised ML approach. This final step is to determine whether or not there is evidence that the learnt model can be considered a physical invariant of the CPS. It consists of the following two sub-steps: (i) applying standard ML cross-validation to assess how well the classifier generalises; and (ii) apply SMC to determine whether or not there is statistical evidence that the classifier does indeed characterise an invariant property of the system.

Sub-step (i): Cross-Validation. Our first validation method is to apply standard ML k-fold cross validation (with e.g. $k = 5$) to assess how well the classifier generalises. This technique computes the average accuracy of k different classifiers, each obtained by partitioning the training set into k segments, training on $k - 1$, and validating on the segment remaining (repeating with respect to different validation partitions).

Sub-step (ii): Statistical Model Checking. The second validation method applies SMC, a standard technique for verifying general stochastic systems [8]. The variant we use observes executions of the system (i.e. traces of sensor data), and applies hypothesis testing to determine whether or not the executions

provide statistical evidence of the learnt model being an invariant of the system. SMC estimates the probability of correctness rather than guaranteeing it outright. It is simple to apply, since it only requires that we can execute the (unmodified) system and collect data traces. It treats the system as a black box, and thus does not require a model [24].

Given some classifier ϕ for a system S, we apply SMC to determine whether or not ϕ is an *invariant* of S with a probability greater or equal to some threshold θ, i.e. whether ϕ correctly classifies the traces of S as normal with a probability greater than θ. Note that the *usefulness* of invariants is a separate question, addressed in Section IV-E. A classifier that always labels normal and abnormal data as normal, for example, is an invariant, but not a useful one for detecting attacks.

Applied to SWaT. In the case of the SWaT simulator, we generate a normal data trace from a new, distinct initial configuration, and collect the positive feature vectors from it. Next, we randomly sample feature vectors from this set, evaluate them with our classifier, and apply SPRT as our hypothesis test to determine whether or not there is statistical evidence that the classifier labels them correctly (setting the error bounds at a standard level of 0.05) with accuracy greater than some θ. If further data is required, we sample additional positive feature vectors from another distinct initial configuration. We remark that we choose θ to be the accuracy of the best classifier we train in our evaluation (Section IV-D). These steps are repeated several times, each with data from additional new initial configurations.

IV. Evaluation

We evaluate our approach through experiments intended to answer the following research questions (RQs):

- RQ1: What kind of classification function do we need?
- RQ2: How large should the time interval in feature vectors be?
- RQ3: How many mutants do we need?
- RQ4: Is our model a physical invariant of the system?
- RQ5: Is our model useful for detecting attacks?

RQ1–3 consider the effects of different parameters on the performance of our learnt models, in particular, the classification function (linear, polynomial, or RBF), the different time intervals for constructing feature vectors, and the number of mutants to collect abnormal traces from. We take the best classifier from these experiments, and assess for RQ4 whether or not there is statistical evidence that the model characterises an invariant of the system. Finally, for RQ5, we investigate whether or not the model is useful for detecting various different attacks that manipulate the network and PLC programs.

All the experiments in the following were performed on the SWaT simulator [13]. The mutation and learning framework we built for this simulator (as described in Section III) is available to download [13], and uses version 3.22 of LIBSVM [22] to apply SVM to our feature vectors.

A. RQ1: What kind of classification function do we need?

Our first experiment is to determine which of the main SVM-based classification functions—linear, polynomial (degree 3), or RBF—we should use in order to learn models with an acceptable level of accuracy. Intuitively, a simple model is more useful for human interpretation, but it may not be expressive enough to achieve high classification accuracy. First, we generate 700 mutant simulators, of which 91 are effective (i.e. led to some abnormal behaviour). From 20 initial configurations of the SWaT simulator, as described in Section III-A, we generate 30 minute traces (at 5ms intervals) of normal and abnormal data from the original simulator and mutant simulators respectively. From these data traces, we collect $1.68 * 10^6$ feature vectors with a 250ms time interval type, using undersampling to account for the larger quantity of abnormal data (see Section III-B). These vectors are then randomly divided into two parts: 70% for training, and 30% for testing. SVM is applied to the training vectors to learn three separate linear, polynomial, and RBF classifiers.

Table II presents a comparison between the three classifiers learnt in the experiment. We report two types of accuracy. The *accuracy* column reports how many of the held-out feature vectors (i.e. the 30% of the collected feature vectors held out for testing) are labelled correctly by the classifier. The *cross-validation accuracy* is the result of applying k-fold cross-validation (with $k = 5$) to the training set: this is the average accuracy of five different classifiers, each obtained by partitioning the training set into five, training on four partitions, and validating on the fifth (then repeating with a different validation partition). This measure helps to assess how well our classifier generalises. *Sensitivity* expresses the proportion of positives that are correctly classified as such; *specificity* is the same but for negatives. Across all four measures, a higher percentage is better.

From our results, it is clear that the RBF-based classifier far outperforms the other two options. While RBF scores highly across all measures, the other classification functions lag far behind at around 60 to 70%; they are much too simple for the datasets we are considering. Intuitively, we believe linear or polynomial classifiers are insufficient because readings of different sensors in SWaT are correlated in complicated ways which are beyond the expressiveness of these kinds of classifiers. Given this outcome, we choose RBF as our classification function.

B. RQ2: How large should the time interval in feature vectors be?

Our second experiment assesses the effect on accuracy of using different time intervals in the feature vectors. As discussed before, a feature vector is of the form (π, π') where π denotes the water tank levels at a certain time and π' denotes the levels after d time units. Intuitively, using these feature vectors, the learnt model characterises the effects of mutants after d time units. On the one hand, an abnormal system behaviour is more observable if this interval d is larger (as the modified PLC control program has more time to take effect).

TABLE II
COMPARISON OF CLASSIFICATION FUNCTIONS

type	accuracy	cross-validation accuracy	sensitivity	specificity
SVM-linear	63.34%	64.12%	66.44%	60.23%
SVM-polynomial	67.10%	68.32%	74.92%	51.67%
SVM-RBF	91.05%	90.99%	99.28%	82.82%

On the other hand, having an interval that is too large runs the risk of reporting abnormal behaviours too late and thus potentially resulting in some safety violation.

Table III presents the results of a comparison of *accuracy* and *cross-validation accuracy* (both defined as for RQ1) across classifiers based on 100, 150, ... 300 ms time intervals. SVM-RBF was used as the classification function, and abnormal data was generated from 700 mutants.

The results match the intuition mentioned earlier, although the accuracy stabilises much more quickly than we initially expected (at around 150ms time intervals). The time interval of 250ms has, very slightly, the best accuracy, so we continue to use it in the remaining experiments.

C. RQ3: How many mutants do we need?

Our third experiment assesses the effect on accuracy from using different numbers of mutant simulators to generate abnormal data. We are motivated to find the point at which accuracy stabilises, in order to avoid the unnecessary computational overhead associated with larger numbers of mutants.

Table IV presents a comparison of *accuracy* and *cross-validation accuracy* (both defined as for RQ1) across classifiers learnt from the data generated by 300, 400, 500, 600, and 700 *mutants*. Our mutant sets are inclusive, i.e. the set of 700 mutants includes all the mutants in the set of 600 in addition to 100 distinct ones. We also list how many of the generated mutants are *effective*, in the sense that they can be compiled, run, and cause some abnormal physical effect with respect to at least one of the initial configurations. We used SVM-RBF as the classification function, collecting feature vectors (see Section III-B) with a time interval of 250ms.

The results indicate that both accuracy and cross-validation accuracy start to stabilise in the 90s from 500 mutants (62 effective mutants) onwards. It also shows that with fewer mutants (e.g. 300 mutants / 23 effective mutants) it is difficult to learn a classifier with acceptable accuracy. Given the results, we choose 600 as our standard number of mutants to generate.

D. RQ4: Is our model a physical invariant of the system?

Our fourth experiment is to establish whether or not there is statistical evidence supporting that the learnt model is a *(physical) invariant* of the system, i.e. it correctly classifies the data in normal traces as normal with accuracy greater or equal to some threshold θ. We perform SMC as described in Section III-C, sampling positive feature vectors derived from a new and distinct initial configuration, setting the acceptable error bounds at a standard level of 0.05, and setting the threshold as $\theta = 91.04\%$ (i.e. the accuracy of the classifier learnt from 600 mutants and a feature vector interval of 250ms). Our implementation performs hypothesis testing using SPRT, randomly sampling feature vectors and applying the classifier until SPRT's stopping criteria are met. If the sampled data is not enough, we sample additional feature vectors from the traces of additional new initial configurations.

Our SMC implementation repeated the overall steps above five times, each with normal data derived from a different distinct initial configuration (falling within normal operational ranges). In each run, our classifier passed, without requiring data to be sampled from traces of additional configurations. This provides some evidence that the classifier is an invariant of the SWaT simulator. This is not surprising: in Section IV-A we found that the sensitivity of the classifier was very high (99.28%), i.e. the proportion of positive feature vectors that it classified as such was very high. Our SMC implementation evaluates for the same property but seeks statistical evidence.

E. RQ5: Is our model useful for detecting attacks?

Our final experiment assesses whether our learnt invariant is effective at detecting different kinds of attacks, i.e. whether it classifies feature vectors as negative once an attack has been launched. First, we investigate network attacks, in which an attacker is assumed to be able to manipulate network packets containing sensor readings (read by PLCs) and signals (read by actuators). Second, we investigate code-modification attacks (i.e. manipulations of the PLC programs), by randomly modifying the different PLC programs in the simulator and determining whether any resulting physical effects are detected. If able to detect the latter kind of attacks, the invariant can be seen as physically attesting the integrity of the PLC code.

Network attacks. Table V presents a list of network attacks that we implemented in the SWaT simulator, and the results of our invariant's attempts at classifying them. Our attacks are from a benchmark of attacks that were performed on the SWaT testbed for the purpose of data collection [12]. These attacks cover a variety of attack points, and were designed to comprehensively evaluate the robustness of SWaT under different network attacks. Of the 36 attacks, we implemented the 15 that could be supported by the ODEs of (and thus had an effect on) the SWaT simulator. The attacks are all achieved by (simulating) the manipulation of the communication taking place over the network, i.e. hijacking data packets and changing sensor readings before they reach the PLC, and actuator signals before they reach the valves and pumps. The attacks cover a variety of *attack points* in the SWaT simulator: these are documented online [10], but intuitively represent motorised

TABLE III
EFFECT OF INCREASING THE TIME INTERVAL ON STABILITY OF SVN-RBF FUNCTION

#time interval	accuracy	cross-validation accuracy
100	90.98%	88.68%
150	90.04%	90.01%
200	90.12%	90.08%
250	91.05%	90.99%
300	90.05%	90.99%

TABLE IV
EFFECT OF INCREASING THE NUMBER OF MUTANTS ON STABILITY OF SVN-RBF FUNCTION

#mutants	#effective mutants	accuracy	cross-validation accuracy
300	23	63.01%	81.91%
400	31	83.01%	89.01%
500	62	90.07%	89.08%
600	76	91.04%	90.89%
700	91	91.05%	90.99%

TABLE V
RESULTS: DETECTING NETWORK ATTACKS INVOLVING MOTORISED VALVES (MV), PUMPS (P), AND LEVEL INDICATOR TRANSMITTERS (LIT)

attack #	attack point	start state	attack	detected	accuracy
1	MV101	MV101 is closed	Open MV101	yes	89.67%
2	P102	P101 is on whereas P102 is off	Turn on P102	yes	90.01%
3	LIT101	Water level between L and H	Increase by 1mm every second	eventually	63.11%
4	LIT301	Water level between L and H	Water level increased above HH	yes	99.86%
5	MV504	MV504 is closed	Open MV504	yes	92.11%
6	MV304	MV304 is open	Close MV304	yes	88.01%
7	LIT301	Water level between L and H	Decrease water level by 1mm each second	eventually	56.97%
8	MV304	MV304 is open	Close MV304	yes	90.16%
9	LIT401	Water level between L and H	Set LIT401 to less than L	yes	89.36%
10	LIT301	Water level between L and H	Set LIT301 to above HH	yes	99.07%
11	LIT101	Water level between L and H	Set LIT101 to above H	yes	91.12%
12	P101	P101 is on	Turn P101 off	yes	92.06%
13	P101; P102	P101 is on; P102 is off	Turn P101 off; keep P102 off	yes	91.62%
14	P302	P302 is on	Close P302	yes	90.91%
15	LIT101	Water level between L and H	Set LIT101 to less than LL	yes	89.37%

valves (MV), pumps (P), and level indicator transmitters (LIT). The table indicates whether or not the invariant was able to *detect* each attack, and the *accuracy* with which it labels the feature vectors (here, this reflects the percentage of feature vectors labelled as negative *after* the attack has been launched). If the accuracy is high (above a threshold of 85%), we deem the attack to have been detected. Note that for attacks manipulating the sensor readings (LITs) read by PLCs, we assume that the *correct* levels are logged by the historian.

As can be seen, all of the attacks were successfully detected. For all the attacks except #3 and #7, this is with very high accuracy (around 90% and above). This is likely because these attacks all trigger an immediate state change in an actuator (opening/closing a valve; switching on/off a pump), either by directly manipulating a control signal to it, or indirectly, by reporting an incorrect tank level and causing the PLC to send an inappropriate signal instead (e.g. attack #4 causes the PLC to switch on a pump to drain the tank, even though the water level is not actually high). Attacks #3 and #7 are not detected initially, hence the lower accuracy (approx. 60%), because the sensor for the tank level is manipulated slowly, by 1mm per second. As a result, it takes more time to reach the threshold when the PLC opens a valve or switches on a pump, at which point the attack has a physical effect. If measuring from this moment onwards, Attack #3 would have an accuracy of 99.83% and #7 an accuracy of 99.72%—hence our judgements of detected *eventually*.

Overall, the results suggest that our invariant is successful at detecting network attacks when they lead to unusual physical behaviour, and thus might be useful in monitoring a system in combination with complementary defence mechanisms (e.g. for ensuring the integrity of the communication links).

Code modification attacks. Table VI presents the results of some code modification attacks, and our invariant's ability to detect them. Unlike for network attacks, there is no benchmark of code modification attacks to use for SWaT. In lieu of this, we randomly generated 40 *effective mutants* (distinct from those in our learning phase), each consisting of a single mutation to a PLC program controlling some *stage* of the SWaT simulator. We generated data from these mutants with respect to our 20 initial configurations, collected feature vectors, and applied our invariant. The table reports how many of the mutants were *detected* and with what *accuracy* (we determine whether a feature vector should be positive or negative analogously to how we labelled feature vectors derived from mutant traces). After grouping the attacks with respect to the PLC program they affect, we report both the average accuracy for *all* attacks as well as for only those that were *detected*.

Our invariant was able to detect 32 of the 40 mutants. Upon manual investigation, we believe the reason it was unable to detect the remaining mutants was because they generated data traces that were too similar to the normal behaviour of the system. Similar to our network attacks, when a code modification attack led to an unexpected change in the states of valves and pumps, the attack was detected. The results suggest that the invariant could be effective for physically attesting the PLCs, i.e. by monitoring the physical state of the system for any unexpected behaviours that could be caused by modified control code. Of course, an intelligent attacker may manipulate the code in a way that is not sufficiently captured by random modifications: seeking a more realistic attestation benchmark set is thus an important item of future work.

F. Threats to Validity

Finally, we remark on some threats to the validity of our evaluation:

(1) Our dataset is limited to a single system: the SWaT simulator;
(2) Data traces were generated with respect to a fixed set of initial configurations;
(3) We used randomly generated code modification attacks, rather than code modifications injected by an intelligent attacker.

Due to (1), it is possible that our results do no generalise to other CPSs. Because of (2), it is possible that normal but rarely occurring behaviours may have been missed in the training phase, and thus may be classified incorrectly by our invariant. These behaviours may also have been missed from the data traces used in the validation phase (SMC). Because of (3), it could be possible that our results do not apply to real code modification attacks designed by attackers with knowledge of the system.

V. Related Work

Anomaly detection has been widely applied to CPS in order to detect unusual behaviours (e.g. possible attacks) from their data [25–33]. Many of these approaches, however, require prior knowledge about the internals of the system—our technique avoids this and attempts to construct a model systematically and automatically.

The idea of detecting attacks by monitoring physical invariants has been applied to a number of CPS [34, 35]. Typically, however, the invariants are *manually* derived using the laws of physics and domain-specific knowledge. Moreover, they are derived for specific, expected physical relationships, and may not capture other important patterns hiding in the sensor data. Manual invariants have also been derived for stages of the SWaT testbed itself [36, 37].

Apart from monitoring physical invariants, the SWaT testbed has also been used to evaluate other attack detection mechanisms, such as a hierarchical intrusion detection system for monitoring network traffic [38], and anomaly detection approaches based on unsupervised machine learning [5, 6]. The latter approaches were trained and evaluated using an attack log [12] from the testbed itself. As our approach was evaluated on the SWaT simulator, an immediate and direct comparison with our results is not possible. However, we believe that our supervised approach would lead to higher sensitivity, and plan to do a proper comparison to confirm or refute this.

Mutations are applied by Brandl et al. [39], but to specifications of hybrid systems (rather than to the PLC programs themselves) in order to derive distinguishing model-based test cases that can be seen as classifiers. A discrete view of the system is used for generating test cases, with qualitative reasoning applied to represent the continuous part.

It is possible to obtain strong guarantees about the behaviour of a CPS by applying formal verification, but only with accurate enough models of the controllers and ODEs. With these, the CPS can be modelled as a hybrid system and a variety of established techniques can be applied (e.g. model checking [40], SMT solving [41], non-standard analysis [42], concolic testing [43], runtime model validation [44], or theorem proving [45, 46]). With discretised models of the physical part, classical modelling and verification techniques can also be applied, e.g. as demonstrated for some properties of SWaT [47, 48].

VI. Conclusion

We proposed a novel approach for automatically constructing invariants of CPS, in which supervised ML is applied to traces of data obtained from PLC programs that have been systematically mutated. We implemented it for a simulator of the SWaT raw water purification plant, presenting a framework that can generate large quantities of mutant PLC programs, data traces, and feature vectors. We used SVM-RBF to learn an expressive model, and validated it as characterising an invariant property of the system by applying cross-validation and statistical model checking. Finally, we subjected the simulator to 55 network and code modification attacks and found that the invariant was able to detect 47 of them (missing only 8 code modification attacks that had a limited effect on

TABLE VI
RESULTS: DETECTING CODE MODIFICATION ATTACKS

attack stage	# effective mutants	# detected	accuracy (detected)	accuracy (all)
PLC 1	8	5	99.82%	71.54%
PLC 3	20	17	99.89%	92.12%
PLC 4	4	4	99.29%	99.29%
PLC 5	5	3	99.43%	81.20%
PLC 6	3	3	99.87%	99.87%
summary	40	32	99.84%	88.20%

the water tank levels), suggesting its efficacy for monitoring attacks and physically attesting the PLCs at runtime.

Future work should seek to address the current complexity of the learnt invariants without reducing their effectiveness at detecting attacks, in order to bring them within reach of stronger validation approaches than SMC, e.g. symbolic execution [11]. It should also seek to make the approach more practical for real CPS such as the SWaT testbed (not just its simulator), by finding ways of reducing the amount of data that must be collected. One way we could achieve this is by applying mutations more effectively, reducing the amount of abnormal data we reject for being indistinguishable from normal traces. For example, we could use domain knowledge to focus the application of mutation operators to parts of the PLC code more likely to lead to useful abnormal traces. In future work we would also like to assess the generalisability of our approach by implementing it for other testbeds or simulators, especially those for applications other than water treatment. Finally, we would like to compare our supervised learning approach against some recently proposed unsupervised ones for SWaT [5, 6], in order to clarify whether or not the overhead of collecting abnormal data pays off in terms of the accuracy of the invariant and its ability to detect attacks.

ACKNOWLEDGMENT

We thank Jingyi Wang for assisting us with statistical model checking, and are grateful to Sridhar Adepu and the anonymous referees for their helpful comments and criticisms. This work was supported in part by the National Research Foundation (NRF), Prime Minister's Office, Singapore, under its National Cybersecurity R&D Programme (Award No. NRF2014NCR-NCR001-040) and administered by the National Cybersecurity R&D Directorate.

REFERENCES

[1] S. K. Khaitan and J. D. McCalley, "Design techniques and applications of cyberphysical systems: A survey," *IEEE Systems Journal*, vol. 9, no. 2, pp. 350–365, 2015.

[2] E. A. Lee, "Cyber physical systems: Design challenges," in *Proc. International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2008)*. IEEE, 2008, pp. 363–369.

[3] A. A. Cárdenas, S. Amin, and S. Sastry, "Research challenges for the security of control systems," in *Proc. USENIX Workshop on Hot Topics in Security (HotSec 2008)*. USENIX Association, 2008.

[4] X. Zheng, C. Julien, M. Kim, and S. Khurshid, "Perceptions on the state of the art in verification and validation in cyber-physical systems," *IEEE Systems Journal*, vol. PP, no. 99, pp. 1–14, 2015.

[5] J. Goh, S. Adepu, M. Tan, and Z. S. Lee, "Anomaly detection in cyber physical systems using recurrent neural networks," in *Proc. International Symposium on High Assurance Systems Engineering (HASE 2017)*. IEEE, 2017, pp. 140–145.

[6] J. Inoue, Y. Yamagata, Y. Chen, C. M. Poskitt, and J. Sun, "Anomaly detection for a water treatment system using unsupervised machine learning," in *Proc. IEEE International Conference on Data Mining Workshops (ICDMW 2017): Data Mining for Cyberphysical and Industrial Systems (DMCIS 2017)*. IEEE, 2017, pp. 1058–1065.

[7] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[8] E. M. Clarke and P. Zuliani, "Statistical model checking for cyber-physical systems," in *Proc. International Symposium on Automated Technology for Verification and Analysis (ATVA 2011)*, ser. LNCS, vol. 6996. Springer, 2011, pp. 1–12.

[9] T. P. Roth and B. M. McMillin, "Physical attestation of cyber processes in the smart grid," in *Proc. International Workshop on Critical Information Infrastructures Security (CRITIS 2013)*, ser. LNCS, vol. 8328. Springer, 2013, pp. 96–107.

[10] "Secure Water Treatment (SWaT)," http://itrust.sutd.edu.sg/research/testbeds/secure-water-treatment-swat/, acc.: January 2018.

[11] Y. Chen, C. M. Poskitt, and J. Sun, "Towards learning and verifying invariants of cyber-physical systems by code mutation," in *Proc. International Symposium on Formal Methods (FM 2016)*, ser. LNCS, vol. 9995. Springer, 2016, pp. 155–163.

[12] J. Goh, S. Adepu, K. N. Junejo, and A. Mathur, "A dataset to support research in the design of secure water treatment systems," in *Proc. International Conference*

on Critical Information Infrastructures Security (CRITIS 2016), 2016.

[13] "Supplementary material," http://sav.sutd.edu.sg/?page_id=3547, acc.: January 2018.

[14] H. L. S. Younes and R. G. Simmons, "Probabilistic verification of discrete event systems using acceptance sampling," in *Proc. International Conference on Computer Aided Verification (CAV 2002)*, ser. LNCS, vol. 2404. Springer, 2002, pp. 223–235.

[15] J. Valente, C. Barreto, and A. A. Cárdenas, "Cyber-physical systems attestation," in *Proc. IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS 2014)*. IEEE, 2014, pp. 354–357.

[16] T. Alves and D. Felton, "TrustZone: Integrated hardware and software security," ARM white paper, 2004.

[17] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative technology for CPU based attestation and sealing," Intel white paper, 2013.

[18] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, "On the difficulty of software-based attestation of embedded devices," in *Proc. ACM Conference on Computer and Communications Security (CCS 2009)*. ACM, 2009, pp. 400–409.

[19] A. Seshadri, A. Perrig, L. van Doorn, and P. K. Khosla, "SWATT: SoftWare-based ATTestation for embedded devices," in *Proc. IEEE Symposium on Security and Privacy (S&P 2004)*. IEEE, 2004, p. 272.

[20] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, no. 2, pp. 99–118, 1996.

[21] T. G. Dietterich, "Machine learning for sequential data: A review," in *Proc. Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR 2002) and Structural and Syntactic Pattern Recognition (SSPR 2002)*, ser. LNCS, vol. 2396. Springer, 2002, pp. 15–30.

[22] C. Chang and C. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, no. 3, pp. 27:1–27:27, 2011.

[23] N. I. Sapankevych and R. Sankar, "Time series prediction using Support Vector Machines: A survey," *IEEE Computational Intelligence Magazine*, vol. 4, no. 2, pp. 24–38, 2009.

[24] K. Sen, M. Viswanathan, and G. Agha, "Statistical model checking of black-box probabilistic systems," in *Proc. International Conference on Computer Aided Verification (CAV 2004)*, ser. LNCS, vol. 3114. Springer, 2004, pp. 202–215.

[25] L. Cheng, K. Tian, and D. D. Yao, "Orpheus: Enforcing cyber-physical execution semantics to defend against data-oriented attacks," in *Proc. Annual Computer Security Applications Conference (ACSAC 2017)*. ACM, 2017, pp. 315–326.

[26] Y. Harada, Y. Yamagata, O. Mizuno, and E. Choi, "Log-based anomaly detection of CPS using a statistical method," in *Proc. International Workshop on Empirical Software Engineering in Practice (IWESEP 2017)*. IEEE, 2017, pp. 1–6.

[27] M. W. Hofbaur and B. C. Williams, "Mode estimation of probabilistic hybrid systems," in *Proc. International Workshop on Hybrid Systems: Computation and Control (HSCC 2002)*, ser. LNCS, vol. 2289. Springer, 2002, pp. 253–266.

[28] ——, "Hybrid estimation of complex systems," *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, vol. 34, no. 5, pp. 2178–2191, 2004.

[29] S. Narasimhan and G. Biswas, "Model-based diagnosis of hybrid systems," *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, vol. 37, no. 3, pp. 348–361, 2007.

[30] F. Pasqualetti, F. Dorfler, and F. Bullo, "Cyber-physical attacks in power networks: Models, fundamental limitations and monitor design," in *Proc. IEEE Conference on Decision and Control and European Control Conference (CDC-ECC 2011)*. IEEE, 2011, pp. 2195–2201.

[31] A. Teixeira, D. Pérez, H. Sandberg, and K. H. Johansson, "Attack models and scenarios for networked control systems," in *Proc. International Conference on High Confidence Networked Systems (HiCoNS 2012)*. ACM, 2012, pp. 55–64.

[32] V. Verma, G. Gordon, R. Simmons, and S. Thrun, "Real-time fault diagnosis," *IEEE Robotics and Automation Magazine*, vol. 11, no. 2, pp. 56–66, 2004.

[33] F. Zhao, X. Koutsoukos, H. Haussecker, J. Reich, and P. Cheung, "Monitoring and fault diagnosis of hybrid systems," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 35, no. 6, pp. 1225–1240, 2005.

[34] A. Choudhari, H. Ramaprasad, T. Paul, J. W. Kimball, M. J. Zawodniok, B. M. McMillin, and S. Chellappan, "Stability of a cyber-physical smart grid system using cooperating invariants," in *Proc. IEEE Computer Software and Applications Conference (COMPSAC 2013)*. IEEE, 2013, pp. 760–769.

[35] T. Paul, J. W. Kimball, M. J. Zawodniok, T. P. Roth, B. M. McMillin, and S. Chellappan, "Unified invariants for cyber-physical switched system stability," *IEEE Transactions on Smart Grid*, vol. 5, no. 1, pp. 112–120, 2014.

[36] S. Adepu and A. Mathur, "Distributed detection of single-stage multipoint cyber attacks in a water treatment plant," in *Proc. ACM Asia Conference on Computer and Communications Security (AsiaCCS 2016)*. ACM, 2016, pp. 449–460.

[37] ——, "Using process invariants to detect cyber attacks on a water treatment system," in *Proc. International Conference on ICT Systems Security and Privacy Protection (SEC 2016)*, ser. IFIP AICT, vol. 471. Springer, 2016, pp. 91–104.

[38] H. R. Ghaeini and N. O. Tippenhauer, "HAMIDS: hierarchical monitoring intrusion detection system for

industrial control systems," in *Proc. Workshop on Cyber-Physical Systems Security and Privacy (CPS-SPC 2016)*. ACM, 2016, pp. 103–111.

[39] H. Brandl, M. Weiglhofer, and B. K. Aichernig, "Automated conformance verification of hybrid systems," in *Proc. International Conference on Quality Software (QSIC 2010)*. IEEE Computer Society, 2010, pp. 3–12.

[40] G. Frehse, C. L. Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "SpaceEx: Scalable verification of hybrid systems," in *Proc. International Conference on Computer Aided Verification (CAV 2011)*, ser. LNCS, vol. 6806. Springer, 2011, pp. 379–395.

[41] S. Gao, S. Kong, and E. M. Clarke, "dReal: An SMT solver for nonlinear theories over the reals," in *Proc. International Conference on Automated Deduction (CADE 2013)*, ser. LNCS, vol. 7898. Springer, 2013, pp. 208–214.

[42] I. Hasuo and K. Suenaga, "Exercises in nonstandard static analysis of hybrid systems," in *Proc. International Conference on Computer Aided Verification (CAV 2012)*, ser. LNCS, vol. 7358. Springer, 2012, pp. 462–478.

[43] P. Kong, Y. Li, X. Chen, J. Sun, M. Sun, and J. Wang, "Towards concolic testing for hybrid systems," in *Proc. International Symposium on Formal Methods (FM 2016)*, ser. LNCS, vol. 9995. Springer, 2016, pp. 460–478.

[44] S. Mitsch and A. Platzer, "ModelPlex: Verified runtime validation of verified cyber-physical system models," in *Proc. International Conference on Runtime Verification (RV 2014)*, ser. LNCS, vol. 8734. Springer, 2014, pp. 199–214.

[45] A. Platzer and J. Quesel, "KeYmaera: A hybrid theorem prover for hybrid systems (system description)," in *Proc. International Joint Conference on Automated Reasoning (IJCAR 2008)*, ser. LNCS, vol. 5195. Springer, 2008, pp. 171–178.

[46] J. Quesel, S. Mitsch, S. M. Loos, N. Arechiga, and A. Platzer, "How to model and prove hybrid systems with KeYmaera: a tutorial on safety," *International Journal on Software Tools for Technology Transfer*, vol. 18, no. 1, pp. 67–91, 2016.

[47] E. Kang, S. Adepu, D. Jackson, and A. P. Mathur, "Model-based security analysis of a water treatment system," in *Proc. International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS 2016)*. ACM, 2016, pp. 22–28.

[48] M. Rocchetto and N. O. Tippenhauer, "Towards formal security analysis of industrial control systems," in *Proc. ACM Asia Conference on Computer and Communications Security (AsiaCCS 2017)*. ACM, 2017, pp. 114–126.