Safety Envelope for Security*

Ashish Tiwari
Bruno Dutertre
SRI International

Dejan Jovanović
Thomas de Candia
SRI International

Pat Lincoln
John Rushby
SRI International

Dorsa Sadigh
Sanjit Seshia
U California Berkeley

ABSTRACT

We present an approach for detecting sensor spoofing attacks on a cyber-physical system. Our approach consists of two steps. In the first step, we construct a *safety envelope* of the system. Under nominal conditions (that is, when there are no attacks), the system always stays inside its safety envelope. In the second step, we build an *attack detector* monitor that executes synchronously with the system and raises an alarm whenever the system state falls outside the safety envelope. We synthesize safety envelopes using a modifed machine learning procedure applied on data collected from the system when it is not under attack. We present experimental results that show effectiveness of our approach, and also validate the several novel features that we introduced in our learning procedure.

Categories and Subject Descriptors

G.3 [**Probability and Statistics**]: Correlation and regression analysis; I.2.6 [**Artificial Intelligence**]: Learning; D.2.5 [**Software Engineering**]: Testing and Debugging—monitors

Keywords

Hybrid Systems; Invariants; Safety Envelopes; Security

1. INTRODUCTION

On 4 December 2011, an American Lockheed Martin RQ-170 Sentinel unmanned aerial vehicle (UAV) was captured by Iranian forces. It was speculated that a GPS spoofing attack was partly responsible for that incident, wherein the UAV was fed false GPS data to make it land in Iran at what the drone thought was its home base in Afghanistan [1]. Separately, modern automobiles were shown to have several attack surfaces that an attacker can use to compromise computers/networks within a car and inject sensor spoofing attacks [4, 9]. These incidents underline the need to enhance security of complex networked cyber-physical systems.

One way to improve security of any system would be to eliminate the attack vectors. However, the attack surface is growing swiftly as we increase complexity of these systems and introduce new features and capabilities into these complex systems. Inevitably, there will always be some channels available for an attacker to exploit.

An alternate approach for improving security would be to build systems that are resilient to attacks. A system that is resilient to attacks can be built by having a module that *detects attacks*. If all attacks can be successfully detected, the system can respond appropriately whenever attacked. For example, a controller can simply ignore inputs from a sensor that are marked as spoofed by an *attack detector*.

The goal of this paper is to build an *attack detector*. An *attack detector* is a "runtime monitor": it runs continuously, monitors the state of the system, and raises an alarm whenever it believes there is an attack; moreover, it also predicts which component of the system is misbehaving. Our focus here is on detecting sensor spoofing attacks: an attack where the adversary behaves as a sensor of the system, and sends/publishes spurious values for that sensor.

A good attack detector should have very low false positives (where the detector says there is an attack when there is none) and very low false negatives (where the detector says there is no attack when there is one). False negatives are clearly bad. False positives are bad too: they unnecessarily deteriorate the performance of the system.

What are the challenges in building a good attack detector? First, we may not have a *model* of a complex cyber-physical system that is simultaneously good and amenable to analysis. So, we have no information on the normal behavior of the system *a priori*. We do have access to the system, and hence, we have the ability to operate the system under normal conditions. Second, we have no knowledge of the noise characteristics of the sensors. We assume that we know informally the meaning of the value produced by a sensor, but we do not have any formal models of the sensor. Since sensors are noisy, a key challenge in building a detector is separating an attack from normal sensor noise: when a sensor output is unusually high/low, is that just the result of the system behavior and sensor noise behavior under normal conditions, or is that due to a spoofing attack?

*Supported in part by DARPA under contract FA8750-12-C-0284 and NSF grant SHF:CSR-1017483. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

HiCoNS'14, April 15–17, 2014, Berlin, Germany.

http://dx.doi.org/10.1145/2566468.2566483.

Note that an attack detector is just a *classifier*: it distinguishes system states reached when the system is operating under normal conditions from the states reached when the system is under attack. So, we use an *invariant* – an over-approximation of the reachable states – of the system *under normal conditions* as the classifier. We call this set the *safety envelope*. The safety envelope is represented as constraints. When the system is attacked, its state will most likely fall outside the safety envelope, and hence, we will detect it. The intuition is as follows: a system typically has several sensors, and under normal conditions, there are relationships that exist between different sensor values. For example, if a system has two velocity sensors, then under normal conditions, these two sensor values must be "equal". So, if one of the sensor is spoofed, the invariant that "the two sensor values must be equal" is violated and this violation indicates an attack. This is the basic idea, but in reality, it gets complicated because of sensor noise and different characteristics/behaviors of different sensors.

We present a new algorithm for synthesizing a safety envelope of a system using data gathered from normal operation of the system. Our algorithm is inspired by ideas from the field of machine learning. It is important to note that machine learning is broadly interested in learning patterns/generalizations from concrete data: a key aspect being that the procedure should *not* learn the noise, but only the underlying pattern after somehow ignoring noise. In our case, the situation is different. We are here *interested in learning the noise characteristics* of the sensors so that we can use the learnt normal noise characteristics to *differentiate normal behavior from a sensor spoofing attack*.

Our main contributions are: (a) a new algorithm for synthesizing safety envelopes from data, (b) several extensions and variants of the algorithm that incorporate key insights about invariant generation into the algorithm, and (c) experimental validation of the algorithm on a real robot.

Our approach has one drawback: if learning is performed in one environment (e.g., on road), but the system is then deployed in a completely different one (e.g., on sand, or with a heavy payload), then the attack detector, despite the generalization during learning, will likely give many false positives. This drawback can be potentially overcome by noting that the learning procedure is fast and automatic and we can learn (new mode) invariants on-the-fly (assuming we can be sure that the system is not under attack when learning).

2. PROBLEM FORMULATION

In this section, we formulate the *sensor spoofing detection problem*.

We have a networked cyber-physical system that interacts with its environment through a set of sensors and actuators. The sensor data can potentially be spoofed by an adversary. The goal is to detect such sensor attacks.

Formally, let X denote the state variables of the system whose dynamics are given by

$$\begin{aligned} \dot{X} &= f(X, c(Y), D) \\ Y &= g(X) \end{aligned}$$

where $c(Y)$ are values computed by the controllers, D are the external disturbances and Y are the variables denoting the measurements produced by the sensors on the system. For simplicity, assume that Y also includes outputs of *virtual sensors*; that is, values that can be computed, or inferred, from the real sensor readings.

We assume that we do *not* have access to the dynamical model f of the system and the sensor-values generating function g.

The data Y generated by the sensors can be spoofed by an adversary. For example, the adversary can add a constant offset to a variable $y_i \in Y$. Since the system dynamics are influence by Y, the adversary can potentially change the set of reachable states of the system.

The *sensor spoofing detection problem* seeks to identify such attacks, and if there is an attack (at any given time or time interval), then detect which sensor variable $y_i \in Y$ was attacked.

3. SAFETY ENVELOPES

In this section, we describe our high-level approach for solving the sensor spoofing detection problem.

Safety envelopes form the central concept in our approach. A safety envelope is simply *any over-approximation of the states of the system that are reachable under nominal conditions*. The safety envelope contains all system states that the system can reach when there are no sensor attacks. Thus, a safety envelope is just an *invariant* set for the system under normal operating conditions.

We use safety envelopes to detect sensor attacks at runtime. We continuously monitor the system, and if the system state falls outside the safety envelope, then we raise a "sensor attack detected" alarm.

In Section 4, we present a technique for computing a safety envelope for a cyber-physical system, which can be used to not only detect attacks, but also find the sensors that are being spoofed.

3.1 Why Safety Envelopes?

We learn safety envelopes from data collected during normal runs of the system. Since it uses an *over-approximation*, the safety envelope approach may appear inferior to an approach that computes the *best possible predictive model* from the data (using, say, Kalman filters). But, there is good reason why they are, in fact, *better* suited for attack detection: the actual system is often complex and linear templates are insufficient for describing the actual model. So, if we use fixed linear templates to learn the model, we will likely learn relations that are not physically meaningful, but are sufficient to "describe the data". Overfitting will occur. There will be lack of generalization. Consequently, either we will need a large amount of data to learn, or, if we use limited data to learn, we will get plenty of "false positives" since any behavior that is slightly different from the training set will be classified as an "attack". As a simple example, if we have distance and speed data for a robot moving at *almost constant* speed v, the best learnt model might predict that the speed of the robot remains set at v, whereas an "over-approximating" safety envelope might just learn the relationship between distance covered and speed, and generalize away from the actual speed. Furthermore, we do not really need a model, but just a good over-approximation to achieve our goal of detecting sensor attacks.

Safety envelopes are also models, but they are highly nondeterministic and abstract models of the system. When learning safety envelopes, we do not want to over-approximate

(abstract) too much: the more we abstract, the more false negatives are generated.

4. LEARNING SAFETY ENVELOPES

In this section, we describe a technique based on machine learning for synthesizing safety envelopes.

Recall that we are assuming that we do not have access to the model of the dynamics (f) of the system. However, we have access to *sensor data generated during some normal runs* of the system. We will use the data as a surrogate for the dynamical model.

Let t denote a sampling period and let

$$Y(0), Y(t), Y(2t), Y(3t), \ldots, Y((m-1)t)$$

denote the sequence of m samples of sensor data generated under nominal conditions. Let $Y(it)$ be a column vector with n entries. Consider the $(m \times n)$-matrix

$$\mathbf{Y} = \begin{bmatrix} Y(0)^T \\ Y(t)^T \\ \vdots \\ Y((m-1)t)^T \end{bmatrix}$$

Here $Y(it)^T$ denotes the transpose of $Y(it)$, and hence each $Y(it)^T$ is a row vector. Our algorithm for learning invariants is inspired by principal component analysis (PCA), but differs in one key aspect: we retain components corresponding to the small eigenvalues, and throw away components corresponding to large eigenvalues.

More formally, we present the pseudocode for our procedure `learnInvariant` that learns invariants from data in Figure 1. The input to the procedure is the $\mathbf{Y}$ data matrix defined above and the row vector Y of the n variable names. The procedure outputs a constraint (formula) that denotes the invariant set. The procedure computes the eigenvalues and eigenvectors of the $(n \times n)$-matrix $\mathbf{Y}^T\mathbf{Y}$. The matrix $\mathbf{Y}^T\mathbf{Y}$ is positive semi-definite, so all eigenvalues are non-negative reals. Each eigenvector $\vec{v}$ corresponding to a *small* eigenvalue defines a linear invariant $l \leq \vec{v}^T Y \leq u$, where (l, u) are the minimum and maximum of the values in the vector $\mathbf{Y}\vec{v}$. The eigenvectors corresponding to *large* eigenvalues are discarded. The procedure returns the conjunction of linear invariant corresponding to all the small eigenvalues.

The correctness of the procedure is obvious: the choice (l, u) of the lower- and upper-bounds guarantees that all the data points used to learn the invariant set clearly belong to the invariant set.

Why use small eigenvalues to define the invariant? This is because, in an ideal system (devoid of any sensor noise or errors), linear equational invariants will be defined by eigenvectors corresponding to 0 eigenvalues. For example, if we have two sensors – one giving the velocity v of a vehicle and the other giving the acceleration a of that vehicle, then we expect a linear relationship

$$v((i+1)t) = v(it) + \frac{(a(it) + a((i+1)t))}{2} t \qquad (1)$$

to exist between the velocity and acceleration readings of two successive time steps. Consider the $(m \times 2)$-matrix $\mathbf{Y}$ constructed using data for the two *virtual sensor* variables $v((i+1)t) - v(it)$ and $(a((i+1)t) + a(it))t/2$. *If* the vehicle was indeed moving with constant $\frac{da}{dt}$, and *if* the sensors for measuring v and a were accurate, then the (2×2)-matrix $\mathbf{Y}^T\mathbf{Y}$ would have an eigenvector (namely, $[1; -1]$) corresponding to eigenvalue 0 in the data matrix because:

$$\begin{array}{rrcl} \text{By assumption Equation 1} & \mathbf{Y}[1;-1] & = & \vec{0} \\ \text{therefore,} & \mathbf{Y^T Y}[1;-1] & = & \vec{0} \\ \text{therefore,} & (\mathbf{Y^T Y})[1;-1] & = & 0[1;-1] \end{array}$$

However, in reality, *since* the vehicle may not be moving with constant $\frac{da}{dt}$, and *since* the sensors for measuring v and a will necessarily be noisy, $\mathbf{Y}[1; -1]$ will not be identically equal to $\vec{0}$ (but it will be close to 0), and hence, the matrix $\mathbf{Y}^T\mathbf{Y}$ is unlikely to have 0 as an eigenvalue (but it will have an eigenvalue close to 0). So, *how to find the invariant in reality*? Rather than finding $\vec{v}$ such that $\mathbf{Y}\vec{v}$ is identically $\vec{0}$, we can find a $\vec{v}$ that minimizes the 2-norm of $\mathbf{Y}\vec{v} - \vec{0}$. Let us compute the 2-norm of $\mathbf{Y}\vec{v} - \vec{0}$.

$$\begin{array}{rcrcl} ||\mathbf{Y}\vec{v} - \vec{0}||_2 & = & ||\mathbf{Y}\vec{v}||_2 & = & (\mathbf{Y}\vec{v})^T(\mathbf{Y}\vec{v}) \\ & & & = & (\vec{v}^T\mathbf{Y}^T)(\mathbf{Y}\vec{v}) \\ & & & = & \vec{v}^T(\mathbf{Y}^T\mathbf{Y})\vec{v} \end{array}$$

If $\vec{v}$ is an *unit* eigenvector of $\mathbf{Y}^T\mathbf{Y}$ with eigenvalue λ, then the above expression evaluates to λ. Thus, among all eigenvectors, the eigenvector $\vec{v}^*$ corresponding to the smallest eigenvalue λ^* is the best candidate for defining equational invariant; see also illustration in Figure 2. We should remark here that the eigenvector $\vec{v}^*$ may *not* globally minimize $\vec{v}^T\mathbf{Y}^T\mathbf{Y}\vec{v}$, but it can be shown that *the vector which achieves global minimum approaches* $\vec{v}^*$ *as the ratio of any other eigenvalue to* λ^* *approaches* ∞. This is the justification for using small eigenvalues to construct the safety envelope in Procedure `learnInvariant` in Figure 1.

The next natural question to ask is *why throw away large eigenvalues?* The reason is *generalization*. From a few runs of the system, we wish to generate a safety envelope that is valid for *all* runs. Hence, we need to avoid finding invariants that "overfit" to the particular data being used to learn the invariants. Throwing away the eigenvectors corresponding to large eigenvalues is an effective way for generalizing.

```
1: procedure LEARNINVARIANT(Y, Y)
2:     Input: (m × n)-matrix Y of sampled sensor data
3:     Input: (n × 1)-column vector Y of variable names
4:     Output: Formula φ denoting the invariant
5:     compute the (n × n)-matrix Y^T Y
6:     compute the eigenvalues λ and corresponding eigen-
   vectors v⃗ of this matrix
7:     initialize invariant φ to True
8:     initialize templates to the emptyset ∅
9:     initialize λ_th to a small positive constant, say 0.2
10:    for each eigenvalue, eigenvector pair λ, v do:
11:        if λ > λ_th then continue
12:        else
13:            add v⃗ to (the set of) templates
14:        end if
15:    end for
16:    for (each template v⃗ in templates) do:
17:            set φ to (φ and min(Yv⃗) ≤ v⃗^T Y ≤ max(Yv⃗))
18:    end for
19:    return invariant φ
20: end procedure
```

Figure 1: Modified PCA for synthesizing invariants from data

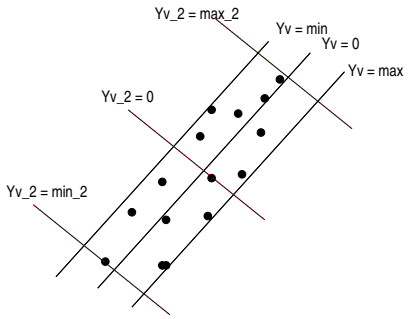

Figure 2: Illustrating modified PCA: The dark lines define the invariant $\texttt{min} \leq Y\vec{v} \leq \texttt{max}$ generated using the eigenvector $\vec{v}$ corresponding to a small eigenvalue; the dotted lines show the possible invariant $\texttt{min}_2 \leq Y\vec{v_2} \leq \texttt{max}_2$ that would be generated using the larger eigenvalue. We throw away the dotted invariant.

We illustrate the intuition with the same example as above. Recall that in the above example, the (2×2)-matrix $\mathbf{Y}^T\mathbf{Y}$ will have an eigenvector $[1; -1]$ corresponding to a small eigenvalue. Now, since $\mathbf{Y}^T\mathbf{Y}$ is symmetric, it has to be approximately equal to $[a, a; a, a]$. The other eigenvalue for this matrix will be approximately equal to $2a$ with $[1; 1]$ as the corresponding eigenvector. This eigenvector gives us the following expression over Y:

$$(v((i+1)t) - v(it)) + \frac{(a(it) + a((i+1)t))}{2}t$$

Any (lower or upper) bound for this expression is essentially equivalent to a bound for the acceleration. Thus, *if* we include a bound for this expression in the invariant, then our invariant will have a bound for the acceleration of the vehicle *based on the values of acceleration that were seen in the runs of the systems used to generate the data*. There may be other runs of the system where these bounds are violated. These runs will violate our invariant, and get wrongly classified as *attacks*. This is the reason for not using the large eigenvalues to define the safety envelope; see also illustration in Figure 2.

Our intuitive reasoning given above was confirmed in our experiments. Using the large eigenvalues in the invariant resulted in an increase in the number of *false positives* – that is, the system detected an "attack" when there was no real attack.

Procedure `learnInvariant` in Figure 1 is our high-level procedure for synthesizing a safety envelope for a system starting from data collected from runs of the system under nominal conditions. We enhance the basic procedure using some key new ideas that improve the quality of the computed safety envelopes. These improvements can be summarized as follows:

- **Learning for Hybrid Systems.** Rather than learning *one* invariant set, one can get better quality invariants if we learn the invariant set as a *disjunction* of smaller invariant sets. This is especially useful if the underlying system is multi-modal or hybrid – in which case, intuitively, we would like to have one invariant set for each mode of the hybrid system. Section 4.1 describes how to extend the basic `learnInvariant` procedure to *identify modes* and then learn the safety envelope as a disjunction of invariant sets for each mode.
- **Learning Relational Invariants at Multiple Time Scales.** Traditional invariants are subsets of the state space of the system, but often we are interested in *relational invariants*; that is, invariants between two successive states of the system. Relational invariants are a subset of the *square of the state space* of the system. An important parameter for defining relational invariants is the time duration between the successive states and increasing the time duration has implications on the quality of the invariants. Section 4.2.1 describes this aspect in detail.
- **Learning Using Virtual Sensors and Multiple Feature Vectors.** Using the raw data generated by the sensors for learning may not be beneficial, and one can improve quality of safety envelope using synthetic sensors – values that can be computed using the raw sensor values, usually using nonlinear transformers. Section 4.2.2 describes the benefits of using such virtual sensors and many different feature vectors for learning invariants.

We describe these three enhancements in the subsequent sections.

4.1 Learning Modes and Mode Invariants

In this section, we extend our basic learning procedure to also identify modes and invariants for these modes.

The learning procedure `learnInvariant` only learns polytopes as invariants; that is, conjunctions of linear inequality constraints. It identifies just one mode and one polytope invariant for that mode.

If a system operates in multiple modes (such as a hybrid system), then finding one polytope that includes reachable states of all modes can lead to severe over-approximation. Figure 3 provides an illustration of this phenomena. The figure depicts the reachable states for a 3-mode hybrid system. The reachable states in each of the three modes are shown by solid blob. An over-approximating polytope of the union of the three blobs is depicted using a dotted line, whereas polytope over-approximations of each of the three blobs is shown using dashed lines. The union of the dashed polytopes is a better over-approximation of the union of the blobs, compared to the dotted polytope. Procedure `learnInvariant` would find one dotted polytope. We wish to modify/extend it to find the three dashed polytopes.

The basic idea for inferring modes and mode invariants is as follows: instead of processing all data points (all rows in $\mathbf{Y}$) in one step (as is done in Procedure `learnInvariant`), we process rows of $\mathbf{Y}$ one at a time. If we ever find a row that does not belong to any of the mode invariants learnt so far, we use Procedure `learnInvariant` on the *next k rows of* $\mathbf{Y}$ to learn a new mode invariant. We then continue processing the rows in matrix $\mathbf{Y}$. Note here that we are implicitly assuming that the rows of $\mathbf{Y}$ are ordered by time and that the system possibly remains in the same mode during the time duration spanning the next k data points.

More formally, we present the pseudocode for the recursive procedure `learnModeInvariants` in Figure 4. The procedure has the same inputs as Procedure `learnInvariants`, and also one additional argument Φ, which is a set of mode

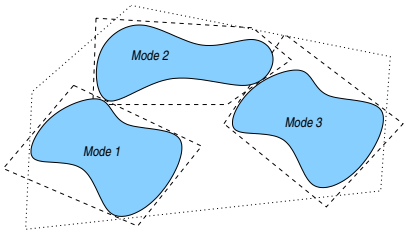

Figure 3: Learning safety envelope as a disjunction of smaller safety envelopes for each mode (dashed lines), versus as a single safety envelope for all modes (dotted line). The sum of the areas inside dashed lines is smaller than the area inside the dotted line, indicating the former is a better invariant.

1: **procedure** LEARNMODEINVARIANTS($\mathbf{Y}$, Y, Φ)
2: Input: $(m \times n)$-matrix $\mathbf{Y}$ of sensor data
3: Input: $(n \times 1)$-column vector Y of sensor variables
4: Input: Set Φ of mode invariants learnt so far
5: Output: Set Φ of (final) mode invariants
6: Fix parameter k to a positive number
7: **if** $\mathbf{Y}$ has zero rows **then**
8: return Φ
9: **end if**
10: Let $\mathbf{Y}[0]$ denote the first row vector of $\mathbf{Y}$
11: **if** $\mathbf{Y}[0]$ belongs to some formula $\phi \in \Phi$ **then**
12: update ϕ in Φ by including $\mathbf{Y}[0]$
13: update $\mathbf{Y}$ by removing first row
14: return `learnModeInvariants`$(\mathbf{Y}, Y, \Phi)$
15: **else**
16: slice $\mathbf{Y}$ as $\begin{bmatrix} \mathbf{Y1} \\ \mathbf{Y2} \end{bmatrix}$ s.t. $\mathbf{Y1}$ contains the first k rows of $\mathbf{Y}$
17: $\phi =$ `learnInvariant`$(\mathbf{Y1}, Y)$
18: $\Phi = \Phi \cup \{\phi\}$
19: return `learnModeInvariants`$(\mathbf{Y2}, Y, \Phi)$
20: **end if**
21: **end procedure**

Figure 4: Identifying Modes and Learning Mode Invariants

invariants (found so far). In the first call, this argument is the empty set $\emptyset$. Procedure `learnModeInvariants` is a wrapper over Procedure `learnInvariant`. It picks the next unprocessed data point, namely the first row $\mathbf{Y}[0]$ of the matrix $\mathbf{Y}$ (line 10), and then checks if the data point belongs to one of the existing mode invariants in Φ (line 11). If it does, then the first row of $\mathbf{Y}$ is removed (line 13) and Φ is updated (line 12, discussed more below), and the procedure recursively calls itself (line 14). If not, then the next k rows of $\mathbf{Y}$ are used to learn a new mode invariant using Procedure `learnInvariant` (line 16,17), and the procedure recursively calls itself on the updated Φ and unprocessed data $\mathbf{Y2}$ (line 19).

There are a few important undefined parameters and undefined functions in Procedure `learnModeInvariants`. The parameter k used in line 16 determines how many rows to use to construct a new mode invariant. We usually pick k to be at least twice the length of vector Y (length of the feature vector). Procedure `learnModeInvariants` also uses

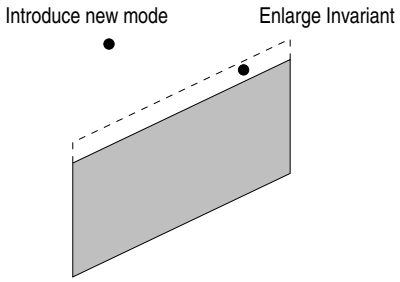

Figure 5: Introducing new mode versus enlarging an existing mode invariant depending on distance of new data point from the existing invariant.

a notion of *when a row* $\mathbf{Y}[0]$ *of data belongs to a multi-mode invariant* (line 11). We use a notion of distance of a point from a set for this purpose. Ideally, if this distance is zero, then the point belongs to the invariant set. But this interpretation is too strict and does not allow generalization of the learnt mode invariants – we need to generalize since we are now using *partial* data set to learn the initial mode invariants. Hence, even when the distance of a data point from an mode invariant set ϕ is nonzero, the data point may be considered as *belonging to the mode invariant set* ϕ *if the distance is less than some fixed small constant.* In such a case, we update ϕ by enlarging it to include the data point (line 12); see Figure 5 for graphical illustration of this process.

4.2 Picking the Feature Vector

One key aspect in designing a good learning procedure is identifying the feature vector. In our case, this is the vector Y that defines the columns of the data matrix $\mathbf{Y}$. In this section and the next, we present two ideas for defining columns of the data matrix $\mathbf{Y}$ that will yield useful invariants.

An "invariant" usually is a subset of the *state space* of the system. The state space of a system is defined as the set of all possible valuations for all state variables. For learning an invariant, the ideal choice for defining the feature vector (the vector Y) would be the state variables of the system. In our context, this would be the variables representing the outputs of all sensors in the system. However, this is a not a good choice for our learning-based invariant generation procedures. This is because the learning procedures find only *linear* expressions over the feature vector Y as invariants. And there may not exist linear relationships between the outputs of the sensors.

The first idea here is to use the learning procedure to find "relational invariants" rather than "invariants". This is helpful to find relationships between the *derivative* of a sensor output and other sensor outputs (Section 4.2.1). The second idea is to use "virtual sensors" in place of the raw sensors to define the feature set Y (Section 4.2.2).

4.2.1 Learning Relational Invariants at Multiple Time Scales

Let us assume we have n sensors, say $s_1, \ldots, s_n$. Let $s_j(it)$ denotes the output of the sensor s_j at time it for some natural number i and sampling period t. The natural choice of feature vector Y for learning invariants would be $s_1, \ldots, s_n$. For learning "relational invariants", we use a feature vector

of twice the length; namely

$$Y = [s_1; \ldots; s_n; s'_1; \ldots; s'_n]$$

where the prime variables denote the sensor value in the *next* sample. Thus, the matrix $\mathbf{Y}$ is given as follows:

$$\mathbf{Y} = \begin{bmatrix} s_1(0), & \ldots, & s_n(0), & s_1(t), & \ldots, & s_n(t) \\ s_1(t), & \ldots, & s_n(t), & s_1(2t), & \ldots, & s_n(2t) \\ s_1(2t), & \ldots, & s_n(2t), & s_1(3t), & \ldots, & s_n(3t) \\ & \vdots & & & \vdots & \end{bmatrix}$$

We note that using the feature vector Y defined above, we can learn (relational) invariants that relate the *change* in a sensor output (derivative of the signal) with other sensor outputs. For example, relational invariants can capture relationship between velocity and acceleration, and similarly between angular velocity and orientation. These relationships will be missed if we only use current velocity and current acceleration in the feature vector, since there will likely be no relationship between them.

In our experimental setup, different sensors published (time stamped) data at different rates – in a completely asynchronous manner. In this case, the value $s_1(it)$ is computed by averaging all the values published by sensor s_1 in the time interval $[it - t/2, it + t/2)$. The sampling period t becomes an important parameter that could determine what attacks are detected.

Noisy Sensor. If a sensor is really noisy, then the relational invariants computed using small values of t will be of poor quality, but those computed using large values of t will be more useful. Specifically, assume s_1 is noisy. If t is small and about the same as the publishing rate of s_1, then s_1 is likely to publish just one value in the interval $[it - t/2, it + t/2)$. So, averaging has minimal/no effect. As a result, due to the high noise, the range $[min(\mathbf{Y}\vec{v}), max(\mathbf{Y}\vec{v})]$ (line 17 of Figure 1) for any expression containing s_1 will be large. If s_1 is spoofed, it will be impossible to say if it is just noise or an attack. However, if we use a larger sampling period t, say 10 times larger, then s_1 is likely to publish about 10 values in the interval $[it-t/2, it+t/2)$. So, averaging will partially eliminate the effect of noise and the range $[min(\mathbf{Y}\vec{v}), max(\mathbf{Y}\vec{v})]$ will be small – indicating a better quality invariant that will be better in discriminating attacks (unless the attack looks just like noise).

Fast Dynamics Sensor. If a sensor is measuring a fast-changing attribute of the system, then the relational invariants computed using small values of t will be of high quality, but those computed using large values of t will likely be of poor quality. This is because averaging over the larger time period would give a value that is less likely to be the true value.

Slowly Drifting Attacks. Consider the scenario where an adversary spoofs a sensor in a way that the spoofed value remains inside the "noise envelope" at every time instance, but with a bias so that over a longer time horizon, the spoofed value has a considerable deviation from the real value. Relational invariants computed over longer time horizons can help detect such attacks, whereas relational invariants over small sampling period t will fail to detect such attacks.

4.2.2 Using Virtual Sensors and Multiple Feature Vectors for Learning

It is often the case that we have some knowledge of the physical interpretation of the value returned by all the sensors in a system. Moreover, we also know the relationships that should exist between all the physical quantities being measured by the various sensors. It is wasteful to not use that information for learning invariants.

We use 'virtual sensors' in the feature vector. These virtual sensors perform some computation on the raw data produced by the sensors and publish the result. The intuition is that there is more likely to be linear invariants that hold between the virtual sensors. Moreover, if we expect no linear invariants to hold for some raw sensor data, we drop it from the feature vector.

We provide a concrete example scenario now. Consider a system that has a GPS providing location information and an inertial measurement unit (IMU) providing the velocity data. We expect that the location and velocity should be related in some way. But this relationship is not linear. In this case, clearly the raw data provided by the GPS is not very useful as a feature in our feature vector. The distance travelled in the last t time units is more useful as a feature – since we expect it to be linearly related to the velocity. Therefore, we add to our feature vector a 'virtual sensor' that publishes the distance traveled in the last t time units. It computes this distance using the data published by the GPS in the last t time units. The function that computes this distance from the GPS coordinates is nonlinear and complex. But once we have the distance, we expect to find a *linear* relationship between distance and velocity. Furthermore, since the raw GPS longitude and GPS lattitude data is not useful, we remove both from the feature vector.

The final idea we use to define the feature vector is *creating multiple small feature vectors* rather than one large feature vector. This allows us to learn targetted invariants. Consider the system described above that has a virtual sensor providing "distance traveled" (`distance`) information and an inertial measurement unit (IMU) providing linear velocity (`vel`), angular velocity (ω), and orientation (θ) data. We expect a linear relationship to exist between the distance traveled and linear velocity, and a separate linear relationship to exist between the angular velocity and (change $\theta' - \theta$ in) orientation. To find these invariants, our feature vector Y would be

$$Y = [\texttt{distance}, \texttt{vel}, \omega, \theta, \theta']$$

However, since we expect invariants involving only a few of the variables (and not all), we can instead use *two different feature vectors* Y_1 and Y_2.

$$Y_1 = [\texttt{distance}, \texttt{vel}], \quad Y_2 = [\omega, \theta, \theta']$$

We then find invariant ϕ_1 using Y_1 and invariant ϕ_2 using Y_2. Finally, we return the *conjunction* ϕ_1 and ϕ_2 of the two learnt invariants.

4.3 Attack Detector

Once we have learnt the safety envelope (for all identified modes), we use it to detect attacks in the natural way. In fact, the pseudocode for monitoring the system is similar to that of Procedure `learnModeInvariants`: we construct the row vector defined by the virtual sensors, and check if the row vector belongs to some formula $\phi \in \Phi$, where Φ is the

learnt (multi-mode) safety envelope. If it does not, then the monitor raises an alarm.

How to detect which sensor is spoofed? Due to the use of virtual sensors and multiple feature vectors, as suggested in Section 4.2.2, we know exactly which invariants are violated by the new data. For each invariant, we know exactly which sensors were used to compute the virtual sensors used in that invariant. Thus, each violated invariant gives a set of candidate spoofed sensors. By just intersecting all such sets, we get a list of one or more sensors that are being most likely spoofed.

We can not only generate the candidate list of spoofed sensors, but also a confidence number on our prediction. Specifically, depending on the distance of the new data point from the invariant set, we can compute a confidence measure: if the new data point is really far away from the learnt invariant set, then we are sure that the system is under attack *with high confidence*, and if the new data point is outside, but relatively close to the learnt invariant set, then we predict that the system is under attack *with lower confidence*.

5. EXPERIMENTS

Our motivating case study was a customized LandShark. The LandShark is a fully electric unmanned ground vehicle (UGV) developed by Black I Robotics [2]. We focused on the following sensors and commands in our customized LandShark for purposes of learning a safety envelope and detecting spoofing attacks: (a) global positioning system (GPS), (b) wheel odometers, (c) inertial measurement unit (IMU), and (d) user input consisting of commanded speed and rotation. The software stack on the customized LandShark is based on robot operating system (ROS) (`www.ros.org`), which is a publish-subscribe middleware. All sensors publish *time-stamped* data at some specific frequency. Different sensors publish data at different rates.

The GPS sensor publishes the longitude, latitude and altitude. The wheel odometers publish the linear velocity, angular velocity, position, and orientation data: all in either 3-d cartesian coordinates, or as quaternions. The IMU publishes data for linear acceleration, angular velocity, and orientation. We use these sensor values to define a few virtual sensors; for example, a sensor that publishes values for the distance covered in the last sampling period; and also sensors that publish speed (not velocity) computed using different sources.

We use six feature vectors for learning invariants:
(1) $Y_1 = \{$ linear speed gv calculated using GPS, linear speed ov calculated using odometry $\}$,
(2) $Y_2 = \{$ linear acceleration oa calculated using odometry, linear acceleration ia calculated using IMU $\}$,
(3) $Y_3 = \{$ angular speed about z-axis owz calculated using odometry, commanded angular speed $cw\}$
(4) $Y_4 = \{$ ov, commanded linear speed cv $\}$
(5) $Y_5 = \{$ owz, change in orientation oo about z-axis, calculated using odometry $\}$
(6) $Y_6 = \{$ angular speed about z-axis iwz calculated using IMU, owz $\}$
Note that several features are "virtual sensors" whose values are calculated from the "actual sensors". Invariant generated using feature vector Y_i will be called *Type-i invariant* below.

We should remark here that we do not need to be completely confident that the values computed for the virtual sensors actually correspond to the names used for them above. For example, we observed that variables that appear to be measuring the same entity (such as, imu-angular-speed-about-z-axis and odometer-angular-speed-about-z-axis) turned out to be almost never equal in the data!

The landshark was run and data collected (in five different rosbags). We used three rosbags (about 800 seconds of real time) to learn the safety envelope. We then injected multiple attacks into each of the other two rosbags (about 200 seconds of real time each). We then tried to see if our attack detector based on the safety envelope (learnt from three bags) could detect attacks correctly in the other two bags.

5.1 Understanding the Results

We now present experimental evaluation of our learing-based attack detection procedure. Our algorithm has several parameters. In our default setting:
(a) we use a sampling period of 100ms,
(b) we use 20 samples to learn invariants of a new mode,
(c) we introduce a new mode whenever the distance of the next 3 data points from an existing invariant is more than 10% the length of the interval defining that invariant, and
(d) we ignore eigenvalues greater-than 0.2 when generating invariants.
In our experiments, we found that the performance of the attack detector was, surprisingly, mostly robust to minor changes in the values of these parameters. Hence, we will describe results using the default setting here.

In the default setting, our learning algorithm finds 5 modes. The first mode corresponds to the scenario where the landshark is stationary. The Type-1 invariant in this mode is

$$\text{Mode-1,Type-1}: \quad 1.00 * ov \in [0.00, 0.00] \quad \text{eigenvalue } 0.0$$

Providing physical intuition for the other modes is not as easy. The Type-1 (T-1) and Type-2 (T-2) invariants for the second mode (M-2) are:

$$\begin{array}{lcl} \text{M-2, T-1}: & 1.00 * ov \in [0.00, 0.02] & 0.0 \\ \text{M-2, T-2}: & -0.76 * oa + 0.65 * ia \in [-0.11, 0.06] & 0.04 \end{array}$$

The rightmost constant is the corresponding eigenvalue. Note that the eigenvalues are close to 0. The second mode corresponds, roughly, to scenario where the landshark is moving very slowly (the Type-1 invariant), the sensors (such as, the odometer acceleration and the IMU acceleration) are in close agreement (the Type-2 invariant), and there is little or no rotation (the Type-5 invariant, not shown above). Mode-2 was learnt by our procedure from data generated by an *accelerating* landshark (when the run of the landshark started). The last mode (Mode-5) is similar to Mode-2, but it was learnt from data generated by a *slowing down* landshark.

The two other modes identified by our learning algorithm cover the case when the landshark is in motion. A sample of their invariants is provided below:

$$\begin{array}{lcl} \text{M-3, T-1}: & -0.56 * gv + 0.83 * ov \in [-1.47, 0.11] & 0.10 \\ \text{M-3, T-2}: & -0.87 * oa + 0.49 * ia \in [-0.19, 0.39] & 0.06 \\ \text{M-4, T-1}: & -0.53 * gv + 0.85 * ov \in [-0.05, 0.04] & 0.01 \\ \text{M-4, T-2}: & -0.98 * oa + 0.20 * ia \in [-0.10, 0.09] & 0.03 \end{array}$$

In Mode-3, linear acceleration ia calculated using IMU is less-than twice the linear acceleration oa calculated using wheel odometry, whereas in Mode-4, ia is almost *five* times oa.

Note that one would expect that *oa* and *ia* are equal, and that *gv* and *ov* are also equal – because they represent the same physical quantity. But they are not equal in the data and the invariant captures the 'actual relationship' between them. The actual relationship is different from the expected relationship because
(a) sensors are noisy, and the noise need not be pure white noise, but it can have a certain characteristic (e.g. bias),
(b) sensors, in particular the IMU, is mounted on the landshark in a certain way, and this influences its reading,
(c) the values for the virtual sensors are computed making certain assumptions about the system (for e.g., the landshark is on a flat surface, or the landshark is moving with constant (linear or angular) acceleration), which can be wrong,
(d) the terrain and landshark's maneuvers influence the sensor behavior: for example, when the landshark is moving in straight-line without making any turns, the odometer is very accurate, but when the landshark rotates, the odometer data is less reliable. Similarly, the noise characteristics of the IMU data vary with the operating condition of the landshark.
It is difficult to model the parts (a)–(d) described above. But we need to have some model of them to be able to distinguish normal behavior from attacks. The learning algorithm can be seen as a way to learn the effect of these difficult-to-model aspects of a complex cyber-physical system.

5.2 Evaluating the Attack Detector

How good are the learnt invariants? We now evaluate the attack detector built using the above 5-mode invariant set that describes all the good states that the system can reach (when not under attack).

We take a new set of data (collected using another real run of the landshark), *which was not used for learning the invariant set*, and inject GPS, Odometer, and IMU spoofing attacks into the data. We insert attacks by just modifying the data published by the respective sensors. If a sensor publishes value $v(t)$ at time t, then we modify it to publish the value $v(t) + \texttt{offset}$ for a fixed offset, for all t in some predefined time interval $[t_0, t_1]$ (*translation attack*)[1].

We used the 5-mode invariant to detect GPS, odometer, and IMU spoofing attacks. The results are summarized in Table 1. The first column lists the attack: an attack name consists of the sensor being spoofed (e.g., gps, and the offset by which its value was translated). The table indicates if the attack was detected (Column 2); and if so, which invariants were violated (Column 3). Based on which invariants are violated, it is easy to predict the faulty/spoofed sensor (Column 4), and provide a confidence measure (Column 5) on the prediction.

As Table 1 shows, we easily detect GPS spoofing attacks where either the longitude or the latitude was translated by 0.001 (roughly $80m$). Similarly, translations to odometry velocity (by 2 m/s) and odometry angular velocity (by 3 rad/s) were also detected. Attacks on IMU linear acceleration were harder to detect – when the acceleration was translated by 10 m/s^2, we detected the attack but with low confidence. We were unable to detect spoofing on IMU angular velocity.

[1]We also tried some *drifting attack*, where the sensor publishes $v(t_0) + (\texttt{offset}/(t_1 - t_0) * (t - t_0))$ for all $t \in [t_0, t_1]$, but the results were the same as for translation with `offset` – most likely because we were not clever in designing the drifting attacks.

attack +offset	detected?	invariants violated	predicted sensor	confidence
gps +0.001	yes	1	GPS	1.0
Odo.v +2	yes	1,2	Odo	0.99
Odo.ω +5	yes	3,5	Odo	0.83
IMU.a +10	yes	2	IMU	0.58
IMU.ω +5	no			

Table 1: Detecting sensor spoofing using the learnt safety envelope: Col. 1 is the attack, Col. 2 says if it was detected, Col. 3 lists the invariants that are violated by the attack data, Col. 4 is the sensor predicted as under attack and Col. 5 lists the confidence number output by our tool.

Sensor	Offset	Detected?	Confidence
GPS	0.0001	yes	0.97
GPS	0.00001	no	
Odo.v	1	yes	0.87
Odo.v	0.1	no	
Odo.ω	3	yes	0.67
Odo.ω	1	no	
IMU.ω	100	no	
IMU.a	10	yes	0.58
IMU.a	5	no	

Table 2: Limits for detecting translation attacks: For each attack, we list the offset that was detected and the offset that was not detected.

5.3 Exploring the Limits

The results in Table 1 will change as we change the offset. So, we experimented and tried to find limits of what attacks can be detected.

The results are summarized in Table 2. For GPS, translations of either the longitude or the latitude by 0.00001 degrees (roughly $0.8m$) were not detected. Similarly, changing the odometry velocity by 0.1 m/s was not detected, and changing odometry angular velocity by 1 rad/s was also not detected. The limits for IMU acceleration were poor: we could not detect translations by 5 m/s^2. The attack detector performed worst for IMU angular velocity attacks. Even when it was translated by 100 rad/s, the attack detector failed to detect. *Why?* The reason is that the IMU on the landshark is a very very noisy sensor. The data generated by the IMU has plenty of high-frequency large spikes, which causes the invariants to become too liberal. In fact, in many cases, the eigenvalues of the $\mathbf{Y}^T\mathbf{Y}$ matrix when considering IMU angular velocity (i.e., when using feature vector Y_6) were larger than our threshold! So, our learning algorithm ignored the invariants involving IMU angular velocity (that is, there were no Type-6 invariants in many modes). As a result, attacks on IMU.ω were not detected.

What if we change the threshold that is used to decide if an eigenvalue is small enough? The result is that we start getting plenty of false positives. Several normal IMU spikes get classified as attacks.

Apart from the missing Type-6 invariants, Type-4 invariants are also missing in the list of *violated invariants* in Table 1. The reason is not entirely clear, but it appears that the commanded velocity is not directly related to the landsharks current velocity in a significant way, and hence the

discrimating power of the generated invariants is weak. In fact, just like Type-6 invariants, Type-4 invariants are missing from certain modes because the corresponding eigenvalues were all bigger than the threshold.

5.4 Single-mode versus multi-mode detectors

How does the procedure perform if we do not introduce modes and instead have just one large safety envelope? The procedure performs poorly compared to our baseline procedure in two ways:
(a) It produces more false negatives. The "one-mode" attack detector is unable to detect any GPS spoofing attacks. This is because the "one-mode" learning procedure looks at all the data to learn global invariants. It observes too much variance in the data involving the GPS, and it fails to find a small eigenvalue in the matrix involving the GPS.
(b) New false positives are observed: Our baseline procedure did not produce any false positives, but the "one-mode" procedure reports attacks when there were no attacks. This was a little surprising. On further investigation, we found that the false positives were caused by invariants whose eigenvalues happened to be just below the threshold of 0.2.
Nevertheless, we should mention that the one-mode attack detector performed as well as the baseline detector on odometry and IMU spoofing data. There are a few reasons for this: (1) Our test data, although separate from the training data, is similar to it because the landshark was performing similar maneuvers in both cases. The one-mode attack detector will likely give much more false negatives if it was evaluated on the actual landshark performing different maneuvers. (2) The landshark does not really have different operating modes. Modes are being used here just to improve precision. If the system really had modes (for e.g., if the training and test data had landshark moving on hard surface and on sand), then the difference in performance of one-mode and multi-mode detectors would be more pronounced.

5.5 Single versus Multiple Feature Vectors

How does the procedure perform if we do not introduce multiple feature vectors and instead have just one feature vector comprising of all real and virtual sensors?

When using just one feature vector, the learning algorithm produced a safety envelope consisting of 10 modes, with around 7 invariants in each mode. It is difficult to interpret the modes or the invariants in any way: all the invariants had nonzero coefficients for most of the Y variables. The performance of the attack detector based on the learnt safety envelope is shown in (left half of) Table 3. On GPS attacks, the performance matches the performance of the baseline attack detector: attacks that offset GPS longitude or latitude by 0.0001 (degrees) are detected, whereas attacks that offset GPS values by 0.00001 (degrees) are undetected. The performance of the new detector matches the performance of the baseline detector also on odometry linear velocity attacks: offset 1 is detected, but offset 0.1 is not. However, performance of the new detector is much worse on the odometry angular velocity attacks (offset 3 not detected) and on the IMU linear acceleration attacks (offset 10 not detected). The new detector performs better than the baseline detector on IMU angular velocity attacks: it detects offset 10 attacks, and fails to detect offset 5 attacks. Recall that the baseline detected had failed to detect all IMU angular velocity attacks, since the invariant involving IMU.ω was deemed to be of poor quality (and hence discarded) in the baseline detector. We believe this is reflected in the *false positives*: the new detector reported (about 3) false positives on (data from) a 200 second landshark run. Recall that the baseline detector had reported zero false positives on the same run.

Sensor	One Feature Vector Offset		$T = 500ms$ Offset	
	Detected	Undetected	Detected	Undetected
GPS	0.0001	0.00001	0.0001	0.00001
Odo.v	1	0.1	1	0.1
Odo.ω	-	3	3	1
IMU.ω	10	5	5	-
IMU.a	-	10	10	5^-
false positives	observed		not observed	

Table 3: Performance of attack detector when (a) using one feature vector consisting of all (virtual) sensor values (Columns `One Feature Vector`) and (b) using invariants that relate states 500ms apart. A - indicates we did not find a suitable value (due to limited experimentation).

5.6 Changing time scale for relational invariants

How does performance change if we use larger time scales for generating and monitoring invariants? We show the results in (right half of) Table 3. Recall from Section 5.1 that the baseline procedure used intervals that related states that were 100ms apart. We now use 500ms as the time interval, and 10 samples to learn invariants of a new mode. The other parameters were the same as for the baseline experiment. We also removed feature vector Y_4 from the list of feature vectors used to construct the invariants, since it was consistently observed to yield poor quality invariants (that were not discriminating and caused false positives in many different experiments – an indication that Y_4 is not a good feature vector.)

We had hypothesized that invariants over large time intervals will be effective in detecting clever *drifting attacks*, where the sensor value remains inside the "noise" term in every time step, but it has a "constant bias" so that over a longer time horizon, the spoofed value is significantly different from the true value. Since we did not generate such attacks, we are unable to experimentally confirm the above hypothesis. We had also hypothesized that invariants over large time intervals will be more effective in generating useful invariants for very noisy sensors. The baseline detector was unable to detect IMU angular velocity attacks for this reason. Using the larger time scale, the learning procedure found useful Type-6 invariants (involving IMU.ω). For example, in one of the modes (out of 5 that were learnt), the following Type-6 invariant was generated:

$$\text{M-5,T-6}: \quad -0.83 * iwz - 0.56 * owz \in [-0.05, 0.01] \quad 0.00$$

The resulting attack detector performed better than the baseline overall as Table 3 shows: it was able to detect translation attacks (by 5) on IMU.ω, and translation attacks (by 10 and by 5 too, but the latter with very low confidence) on IMU.a.

Finally, we note that, in all experiments, we used the same training data (generated from about 800 seconds of landshark run) and the same testing data (about 200 seconds of landshark run, which was disjoint from the training data).

6. RELATED WORK

In this paper, we detect sensor spoofing by constructing a safety envelope using learning techniques and then monitoring the system to detect if it stays inside the safety envelope. Broadly, our work is an instance of *anomaly detection* [3]. Our problem formulation and high-level approach are close to the work in [7], but our work falls in between those that assume availability of detailed models and those that use no information about the system.

The construction of safety envelopes is related to the field of system identification. In the area of hybrid systems identification, Paoeltti et al. [11] study multiple approaches for identifying switched affine and piecewise affine (PWA) models based on input, output observations. The system is typically represented in the form of the well-known *Switched AutoRegressive eXogenous (SARX)* model [8, 13], wherein the current output is represented as a linear combination of an extended regression vector and an error term, where the regression vector includes the observed inputs and outputs. The identification problem is solved by finding the parameters of the extended regression vector for each mode [11]. Our work also starts with the observed data. However, our goal is not to find models, but *relations or constraints* that are mode invariants. Note that our invariants can also depend on the current output observations. Unlike many system identification approaches, we do not disregard the noise term; on the contrary, we are interested in finding invariants that capture the usual noise characteristics. Our algorithm is not restricted to learning linear invariants: we can generate linear and non-linear invariants by appropriately choosing the feature vectors. Unlike work in identification, our work uses a novel method for learning mode invariants of a hybrid system that does not assume any *a priori* bound on the number of modes [11].

Liu et al. [10] study the problem of robust hybrid systems identification with unknown continuous fault inputs. They model every faulty mode as a discrete state in the estimation model. Although they consider faults (attacks) in their input data, and take the approach of decoupling the faults from the error dynamics, they make the strong assumption that dynamical models of the system (system matrices) are provided. Our work does not make any assumptions on the model that generates the input and output traces.

Fawzi et al. [6] study the problem of secure estimation and control for cyber-physical systems under attacks. The authors provide theoretical guarantees on the maximum number of errors that can be detected by a decoder function. In their approach, they assume the knowledge of system matrices, and the guarantees are based on the existence of extended observability matrix. In addition, the noise and attack models are not decoupled. In our work, we separate the learning phase from the detection phase; therefore, our invariants represent a realistic bound on the noise level, and we differentiate between noise and attack.

Our algorithm is inspired by the famous PCA method; however, we are interested in finding invariants instead of the "principal components". It is the dual of PCA in some sense. An attractive way to describe about our procedure – especially the invariant generalization aspect of Figure 5 – is to see it as performing abstract interpretation [5] on data, rather than on programs. We also note that the multi-mode invariant (that we are learning) is just a disjunctive invariant [12].

7. CONCLUSION

We presented a learning-based procedure for detecting sensor attacks in a cyber-physical system. We plan to improve the attack detector by using information learnt about transitions through modes and by experimenting with better attacks and other data sets.

8. REFERENCES

[1] Wikipedia, the free encyclopedia. `en.wikipedia.org/wiki/Iran-U.S._RQ-170_incident`.

[2] Black I Robotics. Basic operations manual, 2010. Revision 1.2, 12/1/10.

[3] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3), 2009.

[4] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. C. an, F. Roesner, and T. Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security*, 2011.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symp. on Principles of Programming Languages, POPL 1977*, pages 238–252, 1977.

[6] H. Fawzi, P. Tabuada, and S. Diggavi. Secure estimation and control for cyber-physical systems under adversarial attacks. *ArXiv e-prints*, May 2012.

[7] R. Fujimako, T. Yairi, and K. Machida. An approach to spacecraft anomaly detection problem using kernel feature space. In *Proc. 11th ACM SIGKDD Intl. Conf. on Know. Disc. in Data Mining*, 2005.

[8] F. Gustafsson. *Adaptive Filtering and Change Detection*. Wiley, Oct. 2000.

[9] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, , and S. Savage. Experimental security analysis of a modern automobile. In *Proceedings of the IEEE Symposium and Security and Privacy*, Oakland, CA, 2010.

[10] W. Liu and I. Hwang. Robust estimation algorithm for a class of hybrid systems with unknown continuous fault inputs. In *American Control Conference (ACC), 2010*, pages 136–141, 2010.

[11] S. Paoletti, A. L. Juloski, G. Ferrari-Trecate, and R. Vidal. Identification of hybrid systems a tutorial. *European Journal of Control*, 13(2-3):242–260, 2007.

[12] J. Rushby. Verification diagrams revisited: Disjunctive invariants for easy verification. In *Computer-Aided Verification, CAV '2000*, volume LNCS 1855, pages 508–520, 2000.

[13] R. Vidal. Recursive identification of switched ARX systems. *Automatica*, 44(9):2274–2287, 2008.