Palisade: A Framework for Anomaly Detection in Embedded Systems

Sean Kauffman[a], Murray Dunne[a], Giovani Gracioli[b], Waleed Khan[a], Nirmal Benann[a], Sebastian Fischmeister[a]

[a] *Real-Time Embedded Software Group, University of Waterloo, Canada*
[b] *Software/Hardware Integration Lab, Federal University of Santa Catarina, Brazil*

Abstract

In this article, we propose Palisade, a distributed framework for streaming anomaly detection. Palisade is motivated by the need to apply multiple detection algorithms for distinct anomalies in the same scenario. Our solution blends low latency detection with deployment flexibility and ease-of-modification. This work includes a thorough description of the choices made in designing Palisade and the reasons for making those choices. We carefully define symptoms of anomalies that may be detected, and we use this taxonomy in characterizing our work. The article includes two case studies using a variety of anomaly detectors on streaming data to demonstrate the effectiveness of our approach in an embedded setting.

Keywords: Anomaly detection, real-time embedded systems, software architecture, streaming-based architecture.

1. Introduction

Real-time embedded systems are part of modern daily life and are integral components of safety-critical devices in the automotive, aerospace, and medical device fields. Failure of these critical components may mean the loss of millions of dollars or even human lives. Increasingly, these systems require online anomaly detection to mitigate risks for operators from failures caused by faults and attacks.

Failures may be caused by logical faults, which can be avoided with good design principles, or execution faults, which can be difficult or impossible to detect at design time [1]. Execution faults may be caused by software errors such as memory leakage or deadlocks, or hardware errors like degraded sensors. Malicious attacks may also cause execution faults, exploiting weaknesses in hardware and software to cause unpredictable behaviors. As real-time embedded systems become more ubiquitous and connected, such attacks become more scalable and realistic [2]. Examples of attacks on embedded systems include disabling a car's braking system [3] and injecting a fatal dose of insulin in an insulin pump [4].

Detection systems must be able to identify faults and attacks quickly and accurately for their results to be useful. If an anomaly is missed it cannot be corrected, and if an anomaly is detected too late, the correction may be unable to prevent a failure. To use an analogy, anomalies such as faults and attacks can be thought of as diseases in a system. To prevent a disease from causing a system failure, it is necessary to understand the disease's symptoms, carefully watch for them, and act quickly if any are detected.

In this article, we introduce a taxonomy of anomaly symptoms and a framework for low-latency online de-

Email addresses: sean.kauffman@uwaterloo.ca (Sean Kauffman), mdunne@uwaterloo.ca (Murray Dunne), giovani@lisha.ufsc.br (Giovani Gracioli), wqkhan@uwaterloo.ca (Waleed Khan), njbenann@uwaterloo.ca (Nirmal Benann), sfischme@uwaterloo.ca (Sebastian Fischmeister)

tection of these anomaly symptoms called Palisade. Our system is designed to monitor remote, real-time embedded systems, and to provide a unified mechanism for responding quickly to faults and attacks. Palisade is also designed to be extensible to facilitate the incorporation of new anomaly detection algorithms to detect the symptoms of unforeseen anomalies.

The main contributions of this article are:

- We describe a comprehensive taxonomy of symptoms of anomalies that may occur in embedded systems and give examples of instances in the literature where they occur. This taxonomy is a valuable resource for anomaly detection work as it supplies a shared language with which researchers and practitioners can discuss their capabilities and requirements.
- We propose Palisade, a data streaming framework that supports online anomaly detection for embedded systems. We present its software architecture, design choices, included anomaly detectors, and two evaluations of its detection latency and extensibility.
- We evaluate the applicability of Palisade through two case studies: one using real data from an autonomous car and a second using data generated from an autonomous driving development platform. We show that, by integrating different anomaly detectors, Palisade is able to detect more anomalies than a stand-alone detector.

Palisade deviates from conventional anomaly detection frameworks in that it is designed to monitor remote, embedded systems and to provide online verdicts with low latency. Many anomaly detection algorithms have been proposed in recent years but the vast majority are designed to operate offline, by analyzing recorded traces [5, 6, 7, 8, 9, 10]. Offline anomaly detection is useful for post mortem analysis of problems, but not to prevent failures at runtime. To monitor remote systems, events and readings must be transmitted to detection algorithms using a data streaming architecture. Palisade uses the publish-subscribe interface from the Redis in-memory database [11].

Our contributions improve on the state-of-the-art of both taxonomies of anomalous behavior and anomaly detection frameworks. We propose a finer grained and more extensive taxonomy of anomalies from prior works, including symptoms of anomalies in an event series. Ours is also the first framework that unifies many algorithms into an online anomaly detection system that can be applied to remote, embedded systems.

The rest of this paper is organized as follows. Section 2 presents the notation used in the paper. Section 3 describes anomaly symptoms in both continuous signals and event series. Section 4 introduces and discusses the Palisade architecture. Sections 5 and 6 present the two case studies and performance results. Section 7 discusses a number of ways to evaluate the framework. Section 8 summarizes prior work related to this article. Section 9 concludes the paper.

2. Notation

We denote the set of all natural numbers by $\mathbb{N}$ and the set of all real numbers by $\mathbb{R}$. We write $A \times B$ to denote the cross product of A and B, and $A \to B$ to denote the set of total functions from A to B.

A sequence σ of n values is written $\sigma = [x_1, \cdots, x_n]$ where both x_i and $\sigma(i)$ mean the i'th item in the sequence. A subsequence of σ beginning at and including the ith index and ending at and including the jth index is denoted $\sigma_{[i,j]}$. A value x is in σ, denoted by $x \in \sigma$ iff $\exists\ i\ \in\ \mathbb{N}$ such that $\sigma(i) = x$. A *time series* is a sequence where each successive item of the sequence represents a sample taken at a regular time interval after its predecessor. The *alphabet* of a sequence denotes the set of possible items in a sequence and, given an alphabet Σ, the set of all possible finite sequences of its members is denoted Σ^*. Given an *alphabet* Σ, a *string* is of type Σ^*, meaning it is a finite sequence of members of Σ of length ≥ 0.

We use $\mathcal{N}$ to denote the type of *names*, which are also called *topics*, and are used as a kind of identifier. For clarity of notation, we use $\mathcal{C}_{lock}$ for the type of *dense clock time* which is defined as $\mathcal{C}_{lock} = \mathbb{R}$. The type of *values* is denoted by $\mathbb{V}$, which can be any of strings, numbers (natural or real), and Booleans. We denote the type of *maps* by $\mathbb{M} = \mathcal{N} \mapsto \mathbb{V}$, where a

map is a partial function from names to values with a finite domain. The empty map is given by $\varnothing$. An *event* is a three-tuple of type $\mathbb{E} = \mathcal{N} \times \mathcal{C}lock \times \mathbb{M}$, that is, it contains a name (a topic), a clock time, and a map. A sequence of events is called a *trace* (or *event series*) and is of type $\mathbb{T}$.

The arithmetic mean, sometimes just called the mean, of n values $\sigma = [x_1, \cdots x_n]$ is $\bar{\sigma} = \frac{1}{n}\Sigma_{i=0}^{n} x_i$. The mean of a sequence σ is denoted $\bar{\sigma}$. The variance of n values $\sigma = [x_1, \cdots, x_n]$ is $var(\sigma) = \frac{1}{n}\Sigma_{i=0}^{n}(x_i - \bar{\sigma})^2$ and the standard deviation (stdev) is the square root of the variance $stdev(\sigma) = \sqrt{var(\sigma)}$. The absolute value of a number n, denoted $|n|$ is n if $n \geq 0$ or $-n$ if $n < 0$.

The standard normal distribution is given as the probability density function $N(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$. The general normal distribution is the standard normal distribution with the domain stretched to a specified stdev and translated to a specified mean. We denote the general normal distribution as $\mathcal{N}(x|m, s) = \frac{1}{s}N(\frac{x-m}{s})$.

The *Discrete Fourier Transform (DFT)* of a time series is a sequence of the same length of the frequency components of the time series. Given a time series σ with length N, its DFT X is defined by $X_k = \Sigma_{n=1}^{N} \sigma_n e^{-\frac{2\pi i}{N} k(n-1)}$. We write $\mathcal{F}(\sigma)$ to denote the DFT of σ.

3. Anomaly Symptoms

This section logically groups the anomaly symptoms that are present in the observable outputs of a system. These symptoms represent the realization of a perturbation in an internal, unobserved state machine. These symptoms do not prove an anomaly by their mere presence, but an anomaly may cause one or more symptoms, hence the disease-symptom analogy. The list in this section is not exhaustive, but categorizes common anomaly symptoms.

This taxonomy relates to Mitre's Common Attack Pattern Enumeration and Classification (CAPEC) [12], in that both CAPEC and this taxonomy can be used to classify capabilities and behaviors. They differ substantially in that CAPEC describes possible attacks, while this section simply describes *symptoms* of attacks or other, non-malicious anomalies.

Many of the symptoms defined in this section are expressed in relation to a set of parameters. Examples include the constant factor c expressing the spike height for spike symptoms, or the difference ℓ expressing the difference required to define a level change symptom. For a given system, a user can determine these constants using a system simulation based on prior knowledge, or traditional parameter-finding approaches such as grid search [13, 14].

3.1. Continuous-Signal Anomaly Symptoms

For the purposes of anomaly symptoms, we define a continuous signal as a digitally sampled signal with a constant sample rate, represented here as a time series. This signal is expected to be the result of readings from a single sensor, not an amalgamation of many sources. The constant sample rate means that the sample time of each value is known from its index in the time series.

3.1.1. Spikes and S-waves

We define a Spike (Figure 1(a)) as a subsequence of contiguous samples that lie farther than a given number of standard deviations from the current mean of the signal. To account for signals with means that change over time, we consider the distance to the mean of a *window* of samples prior to the subsequence. More formally, given a time series y, a window size n, and a constant factor c, the subsequence $y_{[p+1,q]}$ is a spike iff

$$\forall y_t : p < t \leq q,\ |y_t - \bar{y}_{[p-n,p]}| > c \cdot stdev(y_{[p-n,p]})$$

We define S-waves (Figure 1(b)) as spikes with an additional deviation in the opposite direction immediately following the spike. S-waves can mimic spikes if the counter-spike is sufficiently dampened.

Example: A flooding attack over a vehicle Controller Area Network (CAN) may falsely indicate that the collision prevention system issued a command to engage the brakes [15]. Such an attack falls under the category of ⟨CAPEC-125: Flooding⟩ [12] and could be detected by monitoring for Spike anomalies in the volume of CAN packets.

3.1.2. Drifting

A Drift (Figure 1(c)) is a slow movement of the signal mean over a period of time. We consider only linear drift here; logarithmic and sub-linear drifts are rare, and higher order drifting encroaches on the definition of level changes or spikes. Mathematically, a continuous signal y is offset by tc, where t is the time index and c is a constant representing the slope of the drift. Formally, given a time series y, a nominal version of that time series $\hat{y}$, and a slope c, a subsequence $y_{[p,q]}$ has linear drift iff

$$\forall y_t : p \leq t \leq q,\ y_t = \hat{y}_t + tc$$

Example: An infrared combustible sensor, when functioning over the operational temperature limit, may drift or fail [16]. Such a failure could be detected by monitoring temperature readings for Drift anomalies.

3.1.3. Noise

Noise (Figure 1(d)), an expected component of any signal, is considered a symptom of an anomaly only when it is more pronounced than is typical. We define noise as a normally distributed offset around the true value of the signal. Given a time series y, some noisiness coefficient n and nominal time series $\hat{y}$, a subsequence $y_{[p,q]}$ is noisy iff

$$\forall y_t : p \leq t \leq q,\ y_t = \hat{y}_t + \mathcal{N}(0, n)$$

Where $\mathcal{N}(0, n)$ is a standard normal distribution centered at zero with standard deviation n.

Example: Compressed air in truck brakes may generate acoustical interference and cause metallic friction noise from track vehicles in ultrasonic sensors [17]. Brake failure could be detected by correlating Noise anomalies in ultrasonic sensors with air brake usage.

3.1.4. Clipping

We define Clipping as a loss of data at the extrema of a signal range (Figure 1(e)), where a signal is of a higher amplitude than is supported by the sensor or transmission medium. Thus a clipped signal can be represented by a series of identical samples at the maximum or minimum extent of the sample medium.

Example: A partially blinding attack on a camera of a vehicle by emitting light can hide objects [18]. This light can exceed the input range of the camera and would appear as clipped. This attack is an example of ⟨CAPEC-607: Obstruction⟩ [12]. Such blinding light attacks could be detected by monitoring for Clipping anomalies.

3.1.5. Loss

While Loss (Figure 1(f)) may more typically refer to high noise levels making it difficult to decode a signal, here we use loss to indicate a complete loss of a signal. Although trivially an anomaly, a total loss of signal may be a symptom of temporary network disruption without any more dangerous cause. We represent a total loss of signal as a sudden transition to a fixed sample value. This can be observed as a special case of Clipping, where the extrema of the signal are identical for a short time.

Example: An attack sending a large volume of request messages over the J1939 protocol increases the computational load of the recipient ECU until it is not able to perform regular activities like transmitting periodic messages [19]. Such an attack is an example of ⟨CAPEC-125: Flooding⟩ [12] and could be detected by monitoring CAN traffic for Loss anomalies.

3.1.6. Smoothing

We define Smoothing to be a reduction in the short term variance of a signal compared to recent history. Smoothing (Figure 1(g)) is the rarest of the symptoms presented here, with few natural causes. Given a constant k representing how far back the recent historical signal variance should be considered, and the factor threshold τ at which the signal is considered smoothed, we say a subsequence of n samples $y_{[t,t+n]}$ is smoothed iff

$$var(y_{[t,t+n]}) < var(y_{[t-(nk)-1,t-1]})\tau$$

Example: In an attack of a control system, the attacker may observe and record sensor readings and then continuously repeat the recorded values during the attack [20]. This is an example where the sensor values are smoothed. Such an attack falls under

the category of ⟨CAPEC-148: Content⟩ Spoofing [12]. Such spoofing attacks could be detected by monitoring sensor readings for Smoothing anomalies.

3.1.7. Amplification

Amplification (Figure 1(h)) is a simple gain on the target signal. For amplification of an original signal we multiply every sample by some factor. Given the magnitude of the amplification α, and an unamplified time series $\hat{y}$, a sample y_t is amplified iff

$$y_t = \alpha \hat{y}_t$$

Example: Analog to Digital Converters (ADCs) can be attacked by amplifying analog signals past the dynamic range of the device. These attacks can obscure other malicious behavior and damage hardware [21]. This type of attack is an example of ⟨CAPEC-153: Input Data Manipulation⟩ [12]. It could be detected by monitoring the analog signal for Amplification anomalies.

3.1.8. Level Change

A Level Change (Figure 1(i)) symptom is observed when the mean of a signal changes in a short period and then remains consistent at the new level. Slower changes may fall under drifting. Given a time series y, an acceptable minimum level change threshold ℓ, and a minimum number of samples the mean change must persist n, a level change has occurred over a window of w samples $y_{[t,t+w-1]}$ iff

$$|\bar{y}_{[t+w,t+w+n]} - \bar{y}_{[t-n-1,t-1]}| > \ell$$

Example: An attack that increases the amount of code execution will increase the power consumption of the Central Processing Unit (CPU), which can be observed as a Level Change [22, 23, 24]. Such an attack could be an example of ⟨CAPEC-175: Code Inclusion⟩, or ⟨CAPEC-242: Code Injection⟩ [12]. Many attacks with this profile can be detected by monitoring the power consumption of the CPU for Level Change anomalies.

3.1.9. Frequency Change

A Frequency Change (Figure 1(j)) occurs when the primary frequency of a signal changes over a short period. We say a Frequency Change occurs if the primary frequency in a sliding window moves more than some threshold over some time window. Given a time series y, a function P which extracts the frequency of the highest peak from a DFT (denoted $\mathcal{F}$), a threshold τ, and a minimum number of samples the frequency change must persist n, a subsequence of w samples $y_{[t,t+w-1]}$ experiences frequency change iff

$$|P(\mathcal{F}(y_{[t+w,t+w+n]})) - P(\mathcal{F}(y_{[t-n-1,t-1]}))| > \tau$$

It may be useful to consider more frequencies, but we restrict our definition to only consider the primary frequency for simplicity.

Example: An attack inserting a burst of light into a vehicle camera may change the tonal distribution (light frequency) of the captured images, resulting in misclassification [18]. This attack is an example of ⟨CAPEC-607: Obstruction⟩ [12] and it could be detected by monitoring captured images for Frequency Change anomalies.

3.1.10. Echo/Reflection

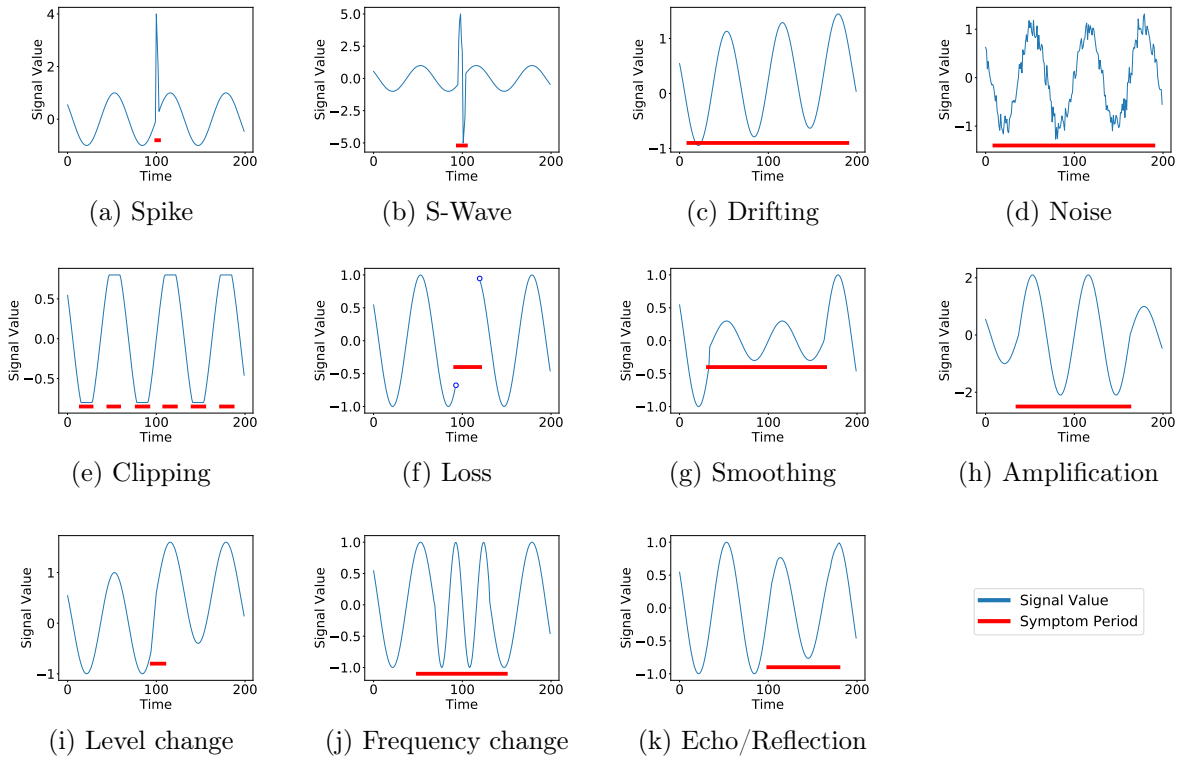
We consider an Echo (Figure 1(k)) to be a duplication of a previous series of samples on top of the underlying signal at a later position. A Reflection is identical to an Echo, except that the repeated signal is inverted. Given a time series y, an echo length e, an echo coefficient (the factor at which the echo is played back) q, and the nominal form of the time series $\hat{y}$, if we consider the subsequence $y_{[t,t+e]}$ as the origin of the echo, we say that the subsequence $y_{[t',t'+e]}$ exhibits Echo iff

$$y_{[t',t'+e]} = \hat{y}_{[t',t'+e]} + y_{[t,t+e]} \times q$$

Example: A relay attack on the original signal sent from the vehicle LiDAR creates fake echoes and can make real objects appear closer or further than their actual location, thus affecting the mission planning [18]. This attack is an example of ⟨CAPEC-586: Object Injection⟩ [12]. Such an relay attack could be detected by monitoring the LiDAR signal for Echo and Reflection anomalies.

3.2. Event-Series Anomaly Symptoms

Symptoms of anomalies also appear in event series, which are defined as a sequence of discrete events



(a) Spike (b) S-Wave (c) Drifting (d) Noise

(e) Clipping (f) Loss (g) Smoothing (h) Amplification

(i) Level change (j) Frequency change (k) Echo/Reflection

Figure 1: Time Series Anomaly Symptoms. The red line indicates the period of the described anomaly.

rather than a continuous signal. An event series represents a trace of the execution of an automaton where each event describes a state transition. Event series differ from discretized continuous signals in that their events are not required to occur at regular time intervals and they may carry more complex data than only a single real value.

3.2.1. Event Frequency Change

When events of the same name are periodic or semi-periodic, they have fairly consistent inter-arrival times and, by extension, frequency. When that frequency changes suddenly, it can be a symptom of a system anomaly. Similarly, when the frequency of all events in a trace change suddenly, it may be due to an anomaly.

The inter-arrival time of an event is the difference between the clock times of successive events of the same name. It can be thought of as the *period* of the event. More precisely, given a trace $T \in \mathbb{T}$, an event name $\mu \in \mathcal{N}$, and a non-empty time interval defined by the end points $t_1, t_2 \in \mathcal{C}lock : t_1 < t_2$, the inter-arrival time is defined as $interArrival(\mu, (t_1, t_2)) \triangleq ((\max t : (\mu, t, \cdot) \in S) - (\min t : (\mu, t, \cdot) \in S))/(|S|-1)$ where $S = \{(\mu, t, \cdot) \in T : t_1 \leq t \leq t_2\}$.

Event frequency measures how often events occur in a given time span. It is given as the inverse of inter-arrival time, or given a trace $T \in \mathbb{T}$, an event name $\mu \in \mathcal{N}$, and a non-empty time interval defined by the end points $t_1, t_2 \in \mathcal{C}lock : t_1 < t_2$, the event frequency of μ is given as $eventFreq(\mu, (t_1, t_2)) \triangleq 1/interArrival(\mu, (t_1, t_2))$.

A sudden change in event frequency, then, is when the first derivative of event frequency is high. A rapid change in event frequency can be found by taking the difference between successive time inter-

vals (or *windows*) in the trace. If the difference exceeds some threshold, then the change in event frequency may indicate an anomaly. Given a trace $T \in \mathbb{T}$, an event name $\mu \in \mathcal{N}$, a window size $w \in \mathcal{C}_{lock}$, and a threshold ϵ, an event frequency change may be defined as $\exists t_1, t_2, t_3 : \mathit{eventFreq}(\mu, (t_1, t_2)) - \mathit{eventFreq}(\mu, (t_2, t_3)) > \epsilon$.

Example: Lin and Siewiorek introduced their Dispersion Frame Technique (DFT) to predict hardware failures [25]. From analyzing the logs of file servers, they observed that there exists a period of an increasing rate of intermittent errors before most hardware failures. Many such failures could be detected by monitoring error reports for Event Frequency Change anomalies.

3.2.2. Unexpected Event

Most traces only contain events with a limited vocabulary of event names. While events themselves are unique, due to their varying clock times, the event names are repeated many times. When an event occurs in a trace with a name that has not come earlier in the trace, it may be a symptom of an anomaly.

Given a trace $T \in \mathbb{T}$, an unexpected event may be defined as an event $(\mu, t_1, \cdot) \in T : \nexists (\mu, t_2, \cdot) \in T$ where $t_2 < t_1$. That is, we can think of an unexpected event as the first event with a new name.

By this definition, however, most events at the beginning of a trace will be considered unexpected. To solve this problem, we can restate the definition in terms of the probability that an event occurs. Given a trace $T \in \mathbb{T}$ and a threshold ϵ, an unexpected event may be defined as an event $e \in T : \mathcal{P}(e \in T) < \epsilon$.

Example: Bellovin reported receiving broadcast packets meant for local networks, requests to unused ports, and requests to unoccupied addresses over the public Internet at AT&T in his classic whitepaper [26]. These types of requests are examples of ⟨CAPEC-169: Footprinting⟩ [12] and they could be detected by monitoring network traffic for Unexpected Event anomalies.

3.2.3. Periods of Silence

A period of silence in a trace is a segment of time where no, or few, events occur. Events may occur more-or-less frequently during the operation of a system as different behaviors result in different patterns in the trace. However, nominal system behavior usually results in *some* events appearing. When a period occurs where no events appear in the trace, it may be a symptom of an anomaly.

Given a trace $T \in \mathbb{T}$ and a minimum number of events ν, a period of silence may be defined as a non-empty time interval defined by the end points $t_1, t_2 \in \mathcal{C}_{lock} : t_1 < t_2$ where $|\{(\cdot, t, \cdot) \in T : t_1 \leq t \leq t_2\}| < \nu$.

The threshold for when a time interval is considered a period of silence varies from system to system. Some high priority tasks may monopolize system resources while not emitting any events. To solve this problem, we can restate the definition to specify a minimum length of the interval. Given a trace $T \in \mathbb{T}$, a minimum number of events ν, and a minimum length $\ell > 0$, a period of silence may be defined as a time interval defined by the end points $t_1, t_2 \in \mathcal{C}_{lock} : t_2 - t_1 \geq \ell$ where the interval meets the previous definition.

Example: Missing log messages can indicate problems and failures in High Performance Computing (HPC) logs that are too large for humans to manually analyze [27]. These types of failures can be detected by monitoring logs for Periods of Silence anomalies.

3.2.4. Sampled Value Anomaly Symptom

When an event trace includes sampled values from a continuous signal, those sampled values may include the same trace anomalies defined in Section 3.1. An event trace is not sampled at a fixed rate like a time series, however. To test for sampled value anomaly symptoms, it may be necessary to extrapolate the values between samples to approximate a continuous signal.

There are several popular methods for recovering a continuous signal from irregular samples. These algorithms include, for example, the projections onto convex sets (POCS) method [28], the Wiley/Marvasti method [29], the Sauer/Allebach Algorithm (also called the Voronoi method) [30], and the Adaptive Weights Method [31]. These methods generally construct an auxiliary signal from some sample values

and then obtain an initial approximation by applying a low-pass filter. The error between this approximation and the samples is then fed into an iterative algorithm which can recover the signal if the sampling density is high enough.

Example: Changes in byte frequency patterns in network payloads to the same host and port can be accurate predictors of network intrusions [32]. Such attacks can be detected by monitoring for Sampled Value Frequency Change anomalies.

4. Palisade Architecture

Palisade is motivated by the need to remotely detect a dynamic range of anomaly symptoms in an embedded system at the time they occur. We are further motivated by the desire to combine multiple anomaly detectors to leverage their different performance characteristics into a single, more reliable, detector. These motivations lead to the following requirements:

1. the anomaly detection must have low latency,
2. it must be easy to implement and maintain detectors,
3. the detectors must be able to be run on separate machines,
4. multiple detectors must be able to run in parallel on the same data, and
5. deployment of the system must be simple.

Requirement 1 is due to the need to respond to anomalies with enough time to mitigate their effects. Requirement 2 is important to any serious software framework, since it should always be a goal to reduce engineering costs. Requirement 3 allows Palisade to run on separate machines from the monitored system and to support anomaly detection appliances to plug into existing networks. It also facilitates horizontal scaling. Requirement 4 is related to Requirements 1 and 3, since operating multiple detectors in series over multiple machines would delay detection and require complex sequencing. Requirement 5 is to reduce the barriers to adoption of the system: if it is hard to install, no one will use it.

Palisade is designed as a set of distributed micro-services built around a data streaming architecture. These micro-services are implemented as three types of nodes: sources, processors, and sinks. Nodes are typically written in Python as Palisade provides a Python Application Programming Interface (API) to simplify their creation and maintenance. However, it is also possible to integrate existing tools written in other languages with Palisade. Figure 2 presents an overview of the Palisade architecture through a Unified Modeling Language (UML) information flow diagram [33]. Each node uses the publish-subscribe interface of the Redis data streaming infrastructure to receive and send data. We do not discuss the Graphical User Interface (GUI) in this article, as it is out-of-scope.

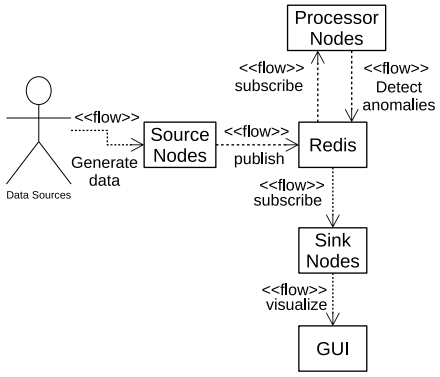


Figure 2: UML information flow of the Palisade architecture.

4.1. Data Streaming

To support Requirements 1, 3, and 4, we needed to find a distributed streaming architecture to transport information between different nodes. We evaluated the latency of four streaming frameworks while considering their inclusion in the Palisade architecture. Latency is defined as the time difference between the instant data is generated by a source and the instant it is received by a processor or a sink [34]. As a result of this experiment we chose Redis as the data streaming architecture for Palisade.

We designed a set of experiments to measure the latency of four data streaming frameworks: Redis [11], RabbitMQ [35], Kafka [36], and NATS [37]. We used two first generation Raspberry Pis (single 700 MHz

ARM6 core, 128 MB system RAM) for the subscriber and the publisher and a Raspberry Pi 2 (quad-core ARM Cortex-A7, 1G RAM) for the server. The clients and the server were synchronized using Precision Time Protocol (PTP), a network level time synchronization protocol capable of microsecond accuracy. We varied the transmitted message sizes (256 bytes, 1 KB, 100 KB, and 1 MB) and the publishing frequency (30 Hz, 60 Hz, and 100 Hz). Latency was measured by including the timestamp at which a message was sent within the message itself. The subscriber then noted the timestamp at which it received the message and subtracted the sending timestamp to find the latency. We ran each configuration of the experiment five times and extracted the average and standard deviation of the latency.

Table 1 shows the results for Redis latency. We note that mean latency increases as the packet size increases, as does the standard deviation. The throughput also increases when both frequency and packet size increase, reaching 6 MB/s at 1 MB and 100 Hz. Redis presents the lowest latency of the four data streaming frameworks. Refer to our previous work for a complete comparison [34].

Table 1: Redis latency results in seconds.

Freq. (Hz)/Packet size	256 B	1 KB	100 KB	1 MB
30	0.0125 ± 0.005	0.184 ± 0.005	0.0731 ± 0.035	0.315 ± 0.3
60	0.0218 ± 0.005	0.0213 ± 0.005	0.0461 ± 0.035	0.337 ± 0.3
100	0.0425 ± 0.005	0.0245 ± 0.005	0.0946 ± 0.035	0.365 ± 0.3

REmote DIctionary Server (Redis) is an open-source, in-memory, key-value database that provides a publish-subscribe interface. Redis clients publish data to channels using the REdis Serialization Protocol (RESP), and subscribers receive data in the same order it was published. Redis also supports integration with on-disk databases. Moreover, Redis has low memory consumption; in a 64-bit system, 1 million keys (hash values), representing an object with five fields, use around 160 MB of memory [11]. Redis provides a master-slave replication mechanism in which slave server instances are exact copies of master servers. To reduce the network round trip latency, Redis implements *pipelining*, making it possible to send multiple commands to the server without waiting for individual replies [11]. These replies are instead batched together into a single response.

4.2. Data Format

Formatting the data transmitted between Palisade nodes requires choices to be made between the relative importance of the requirements listed in Section 4. If Requirement 2 (implementation) and Requirement 5 (deployment) are more important, then a textual format is preferable due to its increased interoperability and human readability. If Requirement 1 (latency) is the most important one, then a simple, offset-based binary format is better because of its lower bandwidth requirements and parsing costs. Palisade handles this conflict by supporting both JavaScript Object Notation (JSON) and a custom binary format for data transmission between nodes.

JSON is a standardized data format [38] with support in most programming languages [39], which makes it easy to consume and produce messages compatible with Palisade from different tools. JSON is a textual format, meaning the data is represented as sequences of characters and, as such, must undergo some processing to convert to-and-from internal program state. While this does increase the processing requirements of JSON formatted data, it has the benefit of being human readable which substantially eases debugging. Formatting with JSON also means that new parameters may be added without breaking backward compatibility since a parser will ignore JSON object keys that aren't expected.

In contrast, an offset-based binary format needs to be strictly specified and any changes require updates to producers and consumers alike. Such a binary format has the advantage of transmitting less data and does not require parsing since its fields may be found by their memory offsets (like a C struct). Some binary formats, like Google's Protocol Buffers, are more complex and allow some modification without updating all uses. However these features always incur additional costs that must be considered. For example, it is only possible to extend a Protocol Buffer mes-

sage if field numbers were previously allocated for that purpose [40].

To support both sampled continuous value signals and discrete event series data, Palisade includes JSON formats for both time series and event series content. Palisade only supports a binary format for time series data. The time series binary format also enables integration with high throughput data sources such as a logic analyzer. Palisade does not currently support more complex binary formats that attempt trade-offs between extensibility and latency as there are no current use cases for such a format.

Figure 3 shows different JSON formats for event and time-series data. Time-series data (Figure 3(a)) contains the initial `timestamp` for the received message as well as a `frequency`, which is used to calculate the timestamp for each datum (by using the frequency and the timestamp of the first data). For the event-based format (Figure 3(b)), `timestamp` represents the instant in which the event is produced and the `data` field can be composed of arbitrary subfields (also in JSON format).

Figure 3(c) represents the binary format for time series. There are three fields in little-endian format. The first field is the millisecond component of the message time, followed by the two magic bytes set to zero, followed by the seconds portion of time, and the data in 16-bit signed two's complement integer format. In Figure 3(c), there are 100 samples, which makes the data field have 200 bytes (100 integers, 2 bytes each).

```
{
  "timestamp": 12345.5678,
  "data": [0.3, 0.1, -0.5],
  "frequency": 100000
}
```

(a) Time-series JSON format.

```
{
  "timestamp": 12345.5678,
  "data": {
    "somefield": 13.3,
    "someotherval": "test"
  }
}
```

(b) Event-series JSON format.

```
0x5D03      // 861 milliseconds
0x0000      // magic bytes
0x00100000 // 256 seconds
<data> // 200 bytes = 100 16-bit signed ints
```

(c) Time-series binary format.

Figure 3: Examples of data formats used in Palisade.

4.3. Source Nodes

Source nodes are responsible for streaming data into Redis. These nodes select and transmit data from a database, file, or an embedded system. Examples of source nodes are one that reads data from a relational database instance, a node that reads a Comma Separated Value (CSV) file, and a User Datagram Protocol (UDP) sniffer that reads packets from the network. Data might include system log entries, aggregate network states, or commands from an autonomous driving stack. Usually, a source node should read data from some data source, change it to JSON or binary Palisade formats, and publish it to a Redis channel.

4.4. Processor Nodes

Processors in Palisade are responsible for detecting anomalies and forwarding the results to sink nodes (see Section 4.5). Each processor implements a different anomaly detection algorithm.

All processors are sub-classes of the abstract *Processor* class. A processor object is instantiated by the *ProcessorLauncher* class, which associates the processor with an instance of the *DataSource* class, which supplies data from either Redis or an offline file source for unit testing purposes.

Individual processors must inherit from the *Processor* class and implement an interface used by the *ProcessorLauncher* to pass data from the *DataSource*. The relevant methods required are *configure* (which collected metadata) and *test_on_data* (which is invoked when new data is ready from the *DataSource*). Other optional methods that may be implemented by processors are: *begin* (called on startup), *end* (called on shutdown), *load_model* (called if a model is specified in *configure*) and *train* (used to build models where applicable).

All non-source nodes subscribe to a channel, named `command`. This is motivated by Requirements 3 and 5, that deployment must be distributed and simple.

Without this channel, each Palisade node would need to be individually controlled from a local interface. The `command` channel supports the control commands for nodes (such as restart and info (a status command)) the are used for general system maintenance.

Once a detector finds an anomaly, it publishes the timestamp at which the anomaly occurred, a note about its cause, and the source channel of the anomaly to a specific Redis channel. Figure 4 shows an example of the JSON data format for anomalies in Palisade. The `timestamp` field contains the time at which the anomaly occurred. The `anomaly` field is a measure of the confidence ($c \in \mathbb{R} : 0 < c \leq 1$) of the detector that an anomaly has occurred, where 1 represents 100% confidence. The `note` field is a textual description of the anomaly, and `channel` contains the Redis channel in which the anomaly happened.

```
{
    "timestamp": 12345.5678,
    "anomaly": 1,
    "note": "what happened",
    "channel": "input channel name"
}
```

Figure 4: JSON format used when an anomaly is detected.

4.5. Sink Nodes

Sinks are nodes that subscribe to Redis channels to perform final processing and do not publish their results. They usually serve as interfaces to other systems, such as GUIs, or alarm systems. Examples of sink nodes include the insertion of received data into databases, writing to I/O ports when an anomaly is received, and storing results into files.

4.6. Example Anomaly Detectors

Palisade currently includes more than 20 example anomaly detectors. All of these detectors are based on existing methods and are distributed with Palisade to provide out-of-the-box detection of the anomaly symptoms listed in Section 3. Tables 2 and 3 show detectors in Palisade and the anomaly symptoms they are capable of detecting. Table 2 shows the detectors for continuous-signal anomaly symptoms (see Section 3.1) while Table 3 shows the detectors for event-series anomaly symptoms (see Section 3.2). In both tables the left-most column shows the detectors while the top-most row shows the anomaly symptoms. The intersection of row and column contains a check mark (✓) if the detector is sensitive to the anomaly symptom and is blank otherwise.

For some detectors, the capability to detect an anomaly symptom depends on the magnitude of the symptom. For example, the spike detector can detect the noise symptom as long as the noise falls outside of the variance of the previous time window. Even though multiple detectors may be able to detect the same anomaly symptom, they complement each other by detecting it in different situations. The distributed nature of Palisade allows running multiple detectors in parallel, increasing the robustness of the system (we show such a situation in the Section 6).

Below is a brief description of each of the example detectors. For the more complex detectors, see the relevant citations for a detailed explanation of the algorithm.

4.6.1. Continuous Signal Example Detectors

- **Autoencoders** uses a neural network that can encode and then decode windows of non-anomalous time series [41]. The network is trained by comparing its input to its output and trying to find an encoding that minimizes the difference. Depending on the configuration, the encoding can be on the order of 10 bits for a window of hundreds of samples. When running, the detector encodes and then decodes its input time series using the same network it trained on nominal data. If the network does a poor job of producing output that matches its input (measured by a difference metric), then the detector concludes that the network must not have seen similar data during training, and therefore the input contains symptoms of anomalies.

- **Clip detect** checks for a number of contiguous identically valued symbols at the extremes of the range of a time series. The detector assumes that a sufficiently long sequence of such values is a symptom.

- **SAX + HMM** uses Symbolic Aggregate approXimation (SAX) to discretize a time series into a sequence of symbols [42]. The symbols are based on the distribution of a non-anomalous time series where a new symbol is generated based on thresholds in the learned distribution. The sequence of those symbols are used to build a Hidden Markov Model (HMM) approximating the sequence [43]. The input to the detector is encoded into symbols using the same distributions. Sequences that are sufficiently unlikely in the learned HMM are considered symptoms of anomalies.
- **Sixnum** tracks changes in six standard statistical metrics: mean, standard deviation, maximum, minimum, upper hinge, and lower hinge. Changes to these metrics above configurable thresholds are considered symptoms.
- **Spike detect** continuously computes the variance of the most recent window of a configurable number of samples. A new sample that falls too far outside the variance of the current window is considered a symptom of an anomaly.
- **Spectrum detect** stores a model of the frequency distribution calculated from the Fourier transform of a non-anomalous time series averaged over time. Windows of the input time series are transformed into the frequency domain and compared to the average frequency model. Deviations beyond a configurable distance metric are considered symptoms of anomalies.

4.6.2. Event Series Example Detectors

- **HMM** is similar to the SAX + HMM detector, except that the input is already composed of events that represent state change instead of continuous samples of a signal, so there is no need to transform them using SAX. Since events have a variable time between them, the HMM can consider the time between events, as well as the type of the event when evaluating the likelihood of the sequence against the learned model.
- **Nfer** is a recently introduced language and system for inferring event stream abstractions [44, 45] that utilizes a syntax based on Allen's Temporal Logic (ATL) [46]. `Nfer` transforms an event stream that represent state transitions into a series of intervals that represent state. These intervals create a hierarchy of abstractions that simplify human and machine trace comprehension. If an interval has been designated as anomalous, its generation is considered a symptom of an anomaly. Palisade supports both hand-written and mined `nfer` specifications [47].
- **SiPTA** uses the expected periodicity in events from embedded systems to apply signal processing techniques to compare the input traces to non-anomalous data. For more information, see Zedah et al.'s 2014 paper [48].
- **TPG** trains a Task Precedence Graph based on a non-anomalous event stream. This method exploits the periodicity of tasks executed in an embedded system. If the input event stream does not follow the learned graph, it is considered a symptom of an anomaly. For more information, see Iegorov and Fischmeister's 2018 paper [49].
- **Timed Regular Expression (TRE)** trains Timed Regular Expressions based on a non-anomalous event stream. If an event from the input stream of the detector violates the learned expressions, then it is considered a symptom of an anomaly. For more information, see Narayan et al.'s 2018 paper [50].

4.7. Fault Handling

Palisade is designed to detect anomalies in streams of data from remote systems, not in its own operation. However, Palisade includes some failure handling capabilities. All nodes in Palisade are monitored by a system service and restarted in the presence of a failure. Detector node failures are interpreted as though an anomaly has been reported by the failed detector, including failures to respond in a configurable time window. This can lead to false-positives, but it is a simple mechanism to alert operators to a situation that deserves their attention. Source node failures are interpreted as loss or period

Table 2: Example detectors and their detected continuous-signal anomaly symptoms.

Name	Autoencoders	Clip detect	Range check	SAX + HMM [42, 43]	Sixnum	Spike detect	Spectrum detect
Spike	✓		✓	✓	✓	✓	
S-Wave	✓		✓	✓	✓	✓	
Drifting	✓		✓	✓	✓		
Noise	✓				✓	✓	
Clipping	✓	✓			✓		
Loss	✓	✓	✓	✓	✓	✓	
Smoothing	✓			✓	✓		
Amplification	✓		✓	✓	✓		
Level change	✓		✓		✓	✓	
Frequency change	✓						✓
Echo/Reflection	✓						

Table 3: Example detectors and their detected event-series anomaly symptoms.

Name	HMM [43]	nfer [44, 45, 47]	SiPTA [48]	TPG [49]	TRE detector [50]
Event Freq. Change	✓	✓	✓	✓	✓
Unexpected Event	✓			✓	
Periods of Silence	✓	✓	✓	✓	✓
Sampled Value		✓			

of silence anomaly symptoms by the relevant detectors, and will be reported as anomalies. Sink node failure handling varies depending on the node, but many are obvious (GUI failures) or fail-warn (alarm systems).

5. Case Study 1: Autonomous Vehicle

We evaluated the performance and applicability of Palisade using the University of Waterloo's autonomous car as a case study. The vehicle was a 2016 Lincoln MKZ fitted with a range of sensor arrays including LiDAR, a Global Positioning System (GPS) receiver, Inertial Measurement Units (IMUs), cameras, and radars. Figure 5 shows an overview of the software and hardware organization in the vehicle. Sensors, such as LiDAR and cameras, produce data that is the input of the autonomy software stack. The output of the autonomy stack (control commands) is sent, for example, to actuators controlling the steering and brakes. Two Renesas automotive computers were installed on the vehicle to run the autonomous driving software. Each computer was equipped with two Systems-on-Chips (SoCs) with multiple ARM CPU cores and a single ASIL-D certified Micro-controller Unit (MCU).

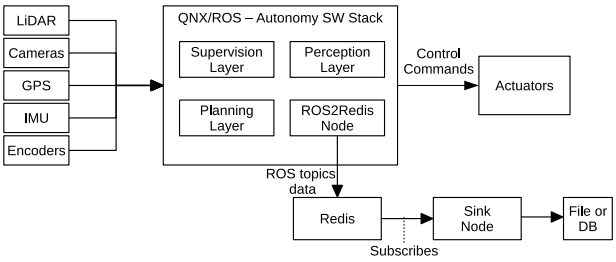


Figure 5: Overview of the software/hardware organization in the autonomous car.

The autonomous driving software was built using Robot Operating System (ROS). ROS is an open source framework for robotic application development in C++ and Python for POSIX-based Operating Systems (OSs). ROS employs the concept of nodes (a process that performs computation), which provides modularity and development isolation. ROS nodes operate on a periodic loop, are event-driven, or both (they publish data at different frequencies into topics). Like Palisade, ROS uses a publish-subscribe model to communicate between nodes.

We implemented a new ROS node, named ros2redis (see Figure 5), to receive messages published to ROS topics and republish them to Redis channels. A Palisade sink node received the command and sensor data and stored them in a database. The stored data were obtained from several ROS topics with different publish frequencies. For instance, GPS information was published at a frequency of 50 Hz, while throttle and gear reports were sent at 20 and 10 Hz, respectively. We logged the data during several autonomous driving sessions and then replayed the recorded data

as input to Palisade.

In the next sections, we present the results of running Palisade with the collected data from the autonomous car in two scenarios: gear flip-flop and autonomy mode flip-flop.

5.1. Gear Flip-Flop

The autonomous vehicle in the case study reports its current gear in messages published to the `_vehicle_gear_report` topic. The values reported in the data item `_vehicle_gear_report_cmd_gear` reflect the requested shift position of the automatic transmission, and the values in `_vehicle_gear_report_state_gear` reflect the reported shift position. Both values are encoded as integers with the following mapping: {0: No change, 1: Park, 2: Reverse, 3: Neutral, 4: Drive, 5: Low}.

The autonomous driving software sends gear change requests over an Internet Protocol (IP) network to a separate controller that interfaces with the vehicle. The controller then converts the requests into CAN messages that the vehicle transmission understands, and also converts messages from the transmission back into IP packets sent to the driving software. Messages on the `_vehicle_gear_report_cmd_gear` channel are only sent when the software requests a *change*. The request is repeated over a short interval until a message is received on the `_vehicle_gear_report_state_gear` channel reporting that the new gear has been reached. Conversely, the transmission regularly reports its current gear on the `_vehicle_gear_report_state_gear` channel regardless of whether or not it has recently changed.

The reports of the current gear from the transmission exhibit the Sampled Value Anomaly Symptom of Noise. When the value of the gear is stable, the signal appears to be nominal. However, when the gear is changing, the signal varies wildly before finally stabilizing on the correct gear. While it is not clear if intermediate gear values should be reported by the transmission during a gear change, it is clear that the transition should be approximately linear. The fluctuations in the signal amount to noise, and are a symptom that an anomaly has occurred in either the transmission itself or in the CAN controller.

Figure 6 shows a brief sample of the two channels over a period when a gear change into *Drive* was requested. The `_vehicle_gear_report_cmd_gear` messages (the blue points) begin at zero, indicating no change is requested, then change to four to request a change to *Drive*, then switch back to *No change* once the gear change is complete. The `_vehicle_gear_report_state_gear` value (the red line) demonstrates the *Noise* Sampled Value Anomaly Symptom as it transitions from *Park* to *Drive*.

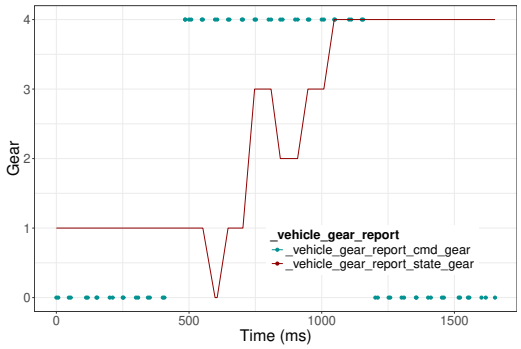


Figure 6: Gear flip-flop anomaly example.

We used `nfer` to detect the Gear Flip-Flop anomaly. To highlight the flexibility of Palisade, we implemented two different integrations with `nfer`. The first built Redis and JSON support directly into the C implementation of `nfer`, and the second used a Python processor node to call the `nfer` Python API. The advantage of building support directly in C is its execution speed, while the advantage of calling the tool through its Python API is its simplicity: the Python `nfer` processor is 42 lines of code.

Our `nfer` specification for detecting the Gear Flip-Flop anomaly is given in Figure 7. The specification contains one rule which defines conditions which, each time they are met, cause a new interval abstraction to be produced with the associated label (topic), timestamps, and data items also being specified by the rule. The rule says that a new interval abstraction should be published to the `gear_flip_flop`

topic when there are three messages published to the `_vehicle_gear_report` topic within 200 time units (ms) such that the first and third specify the same gear state, but the second specifies a different gear state.

Line 1 specifies the topic of the resulting message, while Lines 2 and 3 give the topics on which the tool will listen for events. Lines 2 and 3 also partially specify the temporal relationship of those events, and assigns shorthand identifiers (g1, g2, and g3) to each of the events. The **where** clause, from Lines 4 through 9 specifies further conditions that must be met to produce a new interval abstraction. The **map** clause, from Lines 10 through 16 specifies the data items associated with the published abstraction so that consumers of the message can access the details of the anomaly.

```
1  gear_flip_flop :−
2    g2:_vehicle_gear_report during
3           (g1:_vehicle_gear_report before
4            g3:_vehicle_gear_report)
5    where
6      g1._vehicle_gear_report_state_gear =
7            g3._vehicle_gear_report_state_gear &
8      g3.begin − g1.end < 200 &
9      g1._vehicle_gear_report_state_gear !=
10           g2._vehicle_gear_report_state_gear
11   map {
12     first_gear → g1._vehicle_gear_report_state_gear,
13     anomalous_gear → g2.
             _vehicle_gear_report_state_gear,
14     anomaly → 1,
15     note → "The gear changed and reverted within 200 ms",
16     channel → "_vehicle_gear_report"
17   }
```

Figure 7: `nfer` specification for detection of Gear Flip-Flops.

We ran `nfer` using our Gear Flip-Flop specification with the *minimality* restriction disabled (--full in the tool) and using the *window* optimization with the window size set to 200. Disabling the minimality restriction causes all matched intervals to be reported instead of the default of only reporting the shortest (minimal) intervals [44]. The window optimization restricts reported intervals to those shorter than the window size, instead of the default of no restriction on interval length [45]. The results are reported in Section 5.3.

5.2. Autonomy Mode Flip-Flop

The autonomous vehicle reports its current autonomy state in messages published to specific topics, such as `_vehicle_brake_report` and `_vehicle_throttle_report`. When the data item `enabled` is true, the car is running in autonomy mode. When `enabled` is false, the vehicle is controlled by the human driver.

Once enabled, autonomy mode should remain enabled, as indicated by the `enabled` data item, for the duration of a trip. However, in track testing, we found instances when the vehicle would switch from autonomy enabled back to autonomy disabled during a lap, giving the driver control of the vehicle. We discovered that the Dataspeed CAN module of the autonomous vehicle would disable autonomy mode if no command was given to the vehicle for at least 80 ms. This timeout was unexpected and caused an unsafe driving condition. When the timeout occurs, and autonomy mode is disabled, we call the condition Autonomy Mode Flip-Flop.

We used TREs to monitor the Autonomy Mode Flip-Flop anomaly. Regular expressions provide a declarative way to express patterns for a system specification. TREs define timing constraints in a regular expression [50]. For instance, a regular expression specification of the form "state a is followed by state b" can be modified to "state a is followed by state b within t time units" to obtain a TRE specification. The TRE processor uses a manual specification of a TRE to monitor and detect anomalies. To integrate with Palisade, we added Redis support to the C implementation of the TRE detector.

The TRE specification used for detecting the Autonomy Mode Flip Flop anomaly is the following

$$((\langle P.S\rangle\,[0,3])||(\langle S.P\rangle\,[0,3]))+$$

P and S above represents the two Autonomy Mode Flip-Flop states - Autonomy Enabled and Disabled respectively. The above specification can be translated as the occurrence of patterns where the flip-flop changes from enabled state to disabled state or vice versa within 3 time units.

5.3. Comparison with Siddhi

To evaluate Palisade's rules-based capabilities, we matched the `nfer` processor against the Complex Event Processing (CEP) system, Siddhi. We developed a query in the Siddhi query language to find all instances of the gear flip-flop anomaly described in Section 5.1, then modified the `nfer` specification in Figure 7 to exactly match the results from Siddhi. By first configuring Siddhi, we were able to keep its query short and natural, while the specification for `nfer` to exactly match its result was more complex. The purpose of this choice was to avoid biasing the test results against Siddhi. The data and configuration used for this comparison is available at [51]. In this test, Palisade detected anomalies over 35 times faster than Siddhi and with much lower variance in the latency.

Siddhi is a good choice to compare with the `nfer` processor of Palisade because it is a specification-based stream-processing framework. Both Siddhi and `nfer` require the user to write rules in a declarative language to generate facts from a stream of input events. Both languages are complex enough to define queries for the Gear Flip-Flop anomaly: `nfer` is Turing complete when circular references are permitted [45] and Siddhi is likely Turing complete, although no complexity analysis is available [52]. Like Palisade, Siddhi supports data streaming frameworks where sources and sinks may be remote from the processor. While Siddhi supports cloud-based installations, it can also be installed locally and deployed where internet connections are not available. The ability to install the software locally was important both for automotive use cases and for our ability to accurately measure the tool's detection latency. Siddhi is also easy to install, which is a requirement for Palisade that many CEP systems fail to meet.

We used Siddhi and the `nfer` Palisade processor to independently monitor events sent over a network. We ran the test on a desktop computer with an Intel I5-5200U 2.7 GHz processor and 8 GB of memory running Linux 3.10.0. For Palisade, we streamed the data over Redis and for Siddhi, as it did not support Redis, we sent the data directly over HTTP. For the data, we chose a period of about 68 minutes during which 73,079 events occurred. To simulate an online environment, we delayed publication between messages for the same period as the difference in their timestamps. For example, the difference between the timestamps of sequential messages A and B was 75 ms, we would publish message A and then delay 75 ms before publishing B.

Figure 8 contains part of the Siddhi query to detect the Gear Flip-Flop anomaly. In the query, Line 1 matches the input stream when three events occur within 200 ms where the first and third event report gear zero but the second reports a different gear. If there is a match, the code in Line 2 is a select statement for the data to be published, while Line 3 sends the data on the output stream over HTTP where we record the capture. The omitted portions of the query repeat Lines 1-3, but replace the gear value of zero in the first line with the other possible gears. For more information on the Siddhi query language, see the Siddhi documentation [53] and previous publication [52].

```
1  from every e1=dataInputStream[gear == 0], e2=
       dataInputStream[gear != 0], dataInputStream[gear!=e1
       .gear]*, e3=dataInputStream[timestamp - e1.
       timestamp < 200 and gear==e1.gear and timestamp >
        e1.timestamp]
2  select   e1.timestamp as Timea, e1.gear as Geara, e2.
       timestamp as Timeb, e2.gear as Gearb, e3.timestamp
       as Timec, e3.gear as Gearc, e3.pySendTime as
       pySendTime, eventTimestamp() as
       siddhiSendTimestamp
3  insert into outputDataStream ;
```

Figure 8: Partial Siddhi specification for detection of Gear Flip-Flops.

To measure round-trip detection latency, we added the time of publication to each event and then copied those values to the output when a gear flip-flop anomaly was detected. A second script monitored the results and recorded the timestamp when the anomaly report was received. We then subtracted the passed-through publish timestamp of the message that triggered the anomaly report (the most recent message) from the timestamp when the anomaly report was received.

The comparison in Table 4 shows that, in this case, Palisade detects anomalies over 35 times faster than Siddhi. In the table, lower lower values are better for

both the mean and stardard deviation of the latency. Not only is Palisade's mean detection latency much faster, but the standard deviation of its latency is also about 2.4% of that of Siddhi.

Round-trip latency	**Siddhi**	**Palisade**
Mean (lower is better)	60.6 ms	1.58 ms
Standard deviation	20.9 ms	0.50 ms

Table 4: Results of comparing Palisade with Siddhi

6. Case Study 2: ADAS-on-a-Treadmill

Advanced Driver-Assistance Systems (ADAS)-on-a-Treadmill is a research platform of the Real-time Embedded Software Group of the University of Waterloo [54]. The platform mimics the movement of car on a straight road using a treadmill. A model car with on-board ADAS features emulates various driving conditions, such as Adaptive Cruise Control (ACC), Lane Keeping Assistance (LKA), Lane Departure Warning (LDW), and Forward Collision Detection and Avoidance (FCDA). As in the autonomous vehicle case study, ADAS algorithms are implemented on top of ROS. We have integrated Palisade with the ADAS-on-a-Treadmill platform and run four anomaly detection algorithms (spike, clipping, loss, and range) in two different scenarios (GPS spoof and dead spot). The next sections describe each scenario and present an evaluation.

6.1. GPS Spoof Attack

The GPS spoof scenario simulates an attack on the vehicle positioning system. In this scenario, one car moves from one side of the treadmill to the other on the Y-axis, while keeping the same position on the X-axis. The GPS spoof attack changes the car's Y position by fooling the controller into correcting for an inaccurate reading. In a real autonomous vehicle, such an attack could result in an accident, for instance. We performed three GPS spoof attacks in a five minute period and ran two Palisade detector nodes, spike and clipping.

The spike detector keeps each data point in a buffer. Whenever the buffer is full and new data is received, the detector discards the oldest data point. Then, it compares the value of the newly received data with the mean and standard deviation (std) of the data in the buffer. If the received value is greater or smaller then the std multiplied by a constant plus the mean, then an anomaly is reported. Once an anomaly is detected, the detector waits for a period before starting to compute the mean again. The buffer length considered was 50, the std multiplicative constant was 4.6, and the waiting period was 8 seconds. These are all configurable parameters in the detector. We chose these parameters because they output anomalies without false positives (we discuss the choice of detectors parameters in Section 7).

The clip detector also buffers incoming messages to avoid detection jitter. When the clip detector fills its buffer, it counts how many data points in the buffer are within a configured distance of new values (buffer value + distance $>$ received value *and* buffer value - distance $<$ received value). When there are ten or more matching data points within the interval, a clipping anomaly is reported. The buffer length used in the experiment was 60 and the distance parameter was 0.0005 (difference in the received Y-axis position).

Figure 9 shows the car's Y position when the attacks were performed and the output from the two detectors. The spike detector identifies two out of the three anomalies, while the clip detector finds all three inserted anomalies. The first anomaly is identified quickly by the spike detector because there is a sharp jump in the received Y position. We also ran both detectors with the same parameters using a dataset without anomalies, and neither detected any anomalies, as expected.

6.2. Dead Spot in a Platooning Formation

This scenario simulates a situation where a lighting inconsistency in the environment causes "dead spots" on the conveyor, causing any car that drives into the spot to lose its positioning data. This is inspired by a real experience running the University of Waterloo autonomous car at CES 2018. In this scenario, two cars drive on the treadmill in a platoon formation. A dead spot is inserted in the treadmill and a command is issued to move the first car forward on the X-axis.

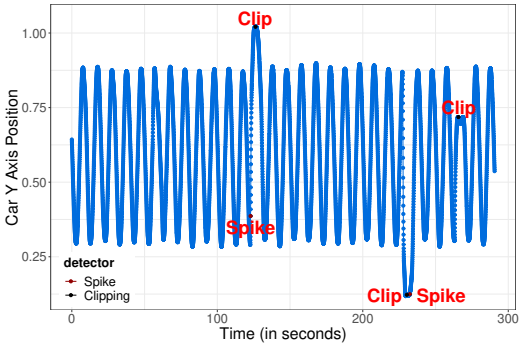
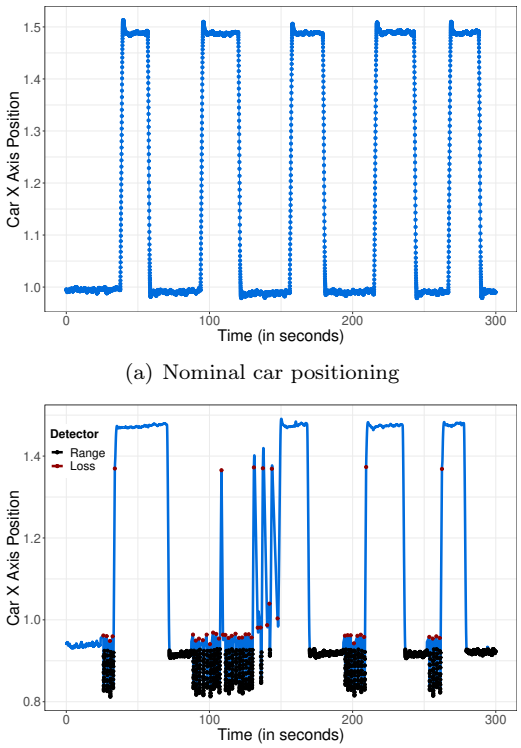


Figure 9: Car positioning with four inserted anomalies and the anomaly detection points (Spike and Clipping detector).



(a) Nominal car positioning

(b) Car positioning with dead spots

Figure 10: Car positioning with inserted dead spots and the anomaly detection points (Loss and Range detector).

The car then moves to the desired point, stays for a while, and returns to its original position. We ran the experiment for five minutes and inserted four dead spots while executing two Palisade detector nodes, loss and range check.

The loss detector keeps track of the average period of message reception and, when a data point takes longer to arrive than the average multiplied by a constant factor, an anomaly is reported. The detector checks for anomalies after a minimum number of samples is received. In the experiment, we set the minimum number of samples to 30 and the constant factor to four. The range detector tests whether a data point is between a maximum and minimum value. If a received data point is outside the range, then an anomaly is reported. We trained the range check with a dataset without anomalies.

Figure 10(a) illustrates the expected behavior (without anomalies) of the platooning formation for the first car. For instance, around 100 seconds into the experiment a command to move the first car forward is issued, which causes it to move from the position of about 1.0 to 1.5. When the vehicle reaches the desired position, it takes a second or two to stabilize, causing the small spikes after each acceleration.

Figure 10(b) shows the car X-axis position with inserted dead spots. The first command to move forward is issued around 25 seconds into the experiment. The first car attempts to move to the desired point but reaches a dead spot where it loses its positioning signal for a short time. This causes the "shaking" at the bottom of the figure as the controller tries to reestablish the car's position. The range detector is able to identify such a situation because those values are lower than the minimum in the dataset without anomalies. The loss detector recognizes the loss of communication while the car passes through a dead spot. Around 140 seconds into the experiment, we can see two lines that move up and down in a short period. This happens because the first car passes the dead spot by its side, corrects its trajectory by returning to the dead spot, and then returns to its position before the command to move was issued. This is another scenario where Palisade improves the anomaly detection by providing means to easily run two detectors using the same input data. We discuss how Palisade improves the anomaly detection in Section 7.

6.3. Comparison with Beep Beep 3

We evaluated Palisade against the stream-processing system Beep Beep 3. We developed a Beep Beep 3 processor to find all instances of the dead-spot anomaly described in Section 6.2. To provide an accurate comparison, we used the Beep Beep 3 HTTP palette to construct a distributed detector, with events sent and anomalies reported over a network connection. We found that Palisade and Beep Beep 3 attained comparable anomaly detection latency in this case, but that building such a distributed processor with Beep Beep 3 was more cumbersome than with Palisade. The data and configuration used for this comparison is available at [51].

Beep Beep 3 is a good choice for a comparison with Palisade because it is specifically designed for online, streaming data processing. Futhermore, the tool is highly flexible, supports arbitrary data types, and allows distributed processors to be created using official libraries. Like Palisade, Beep Beep 3 is more of an architecture and set of APIs than a standalone tool. However, unlike Palisade, Beep Beep 3 does not include out-of-the-box processors designed for anomaly detection, although such extensions exist [55].

We used Beep Beep 3 along with the Palisade RangeCheck and LossDetect processors to independently monitor events sent over a network. We ran all the detectors on a desktop computer with an Intel I5-6300U 2.4 GHz processor and 16 GB of memory running Linux 4.19.72. Palisade was run on Python 3.6.10 and Beep Beep 3 was executed using OpenJDK 1.8.0_252 (IcedTea 3.16.0). For Palisade, we streamed the data over Redis and for Beep Beep 3, as it did not support Redis, we used the Beep Beep 3 HTTP Palette and its serialization library. For the data, we used the same period of about 5 minutes from Figure 10(b) during which the car's position was reported 7,030 times. To simulate an online environment, we delayed publication between messages for the same period as the difference in their timestamps. For example, the difference between the timestamps of sequential messages A and B was 33 ms, we would publish message A and then delay 33 ms before publishing B.

To measure round-trip detection latency, we added the time of publication to each event and then copied those values to the output when a loss or range anomaly was reported. A second program monitored the results and recorded the timestamp when the anomaly report was received. We then subtracted the passed-through publish timestamp of the message that triggered the anomaly report (the most recent message) from the timestamp when the anomaly report was received.

To compare between Palisade and Beep Beep 3, we constructed a Beep Beep 3 stream processor that mimicked the behavior of both the RangeCheck and LossDetect Palisade processors. Figure 11 shows an outline of this processor, along with Beep Beep 3 programs for reading and printing events [56].

Figure 11 uses the official Beep Beep 3 drawing guide to show how events are read, transmitted, filtered, retransmitted, and printed. The top diagram in the figure shows the events being read from a file and transmitted using the HTTP palette to the central processor. The central diagram in the figure shows how events are filtered to only be included in the output if they are out-of-range or occur after a long delay. The events arrive via HTTP and are duplicated to follow two paths: in the lower path, they are tested to see if they are out-of-range or occur after a long delay (there is more logic here, not shown in the figure), in the upper path, they reach a filter (shown as a traffic light) which is gated based on the result of the lower path test. Events for which the test is true are transmitted via HTTP to the final program, shown as the third diagram. The third diagram shows how events that arrive are then printed to the console.

Beep Beep 3 is not designed for distributed processing, however. The only officially supported networking mechanism is direct HTTP connections, which necessitates tightly coupled components. That is, the program that reads the events from a file must know the address of the processor, which must, in turn, know the address of the program that prints the events. This is why the range and loss checks are performed in one processor; if they were separated into two processors, the file reader would need to send events directly to both end points. That Palisade components are loosely coupled is a fundamental advantage for distributed stream processing.

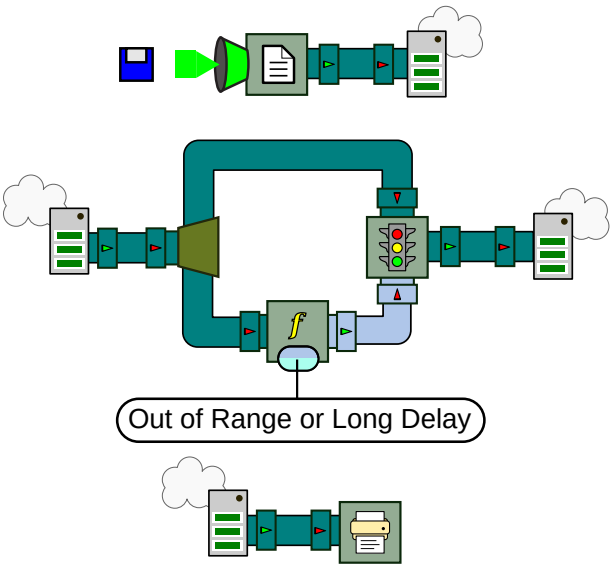


Figure 11: Graphical representation of the Beep Beep 3 processor omitting the details of how Range and Loss were computed.

Although it was not necessary to write new Palisade processors for the comparison, the existing processors are much simpler than the equivalent processor written with Beep Beep 3. The reason for this disparity is that the programming model for Beep Beep 3 is not aligned with the programming language in which it must be implemented. That is, Beep Beep 3 programs do not resemble Java programs. This can most clearly be seen in the code to apply a function to check if an event is outside a fixed range. In Palisade, this check is written in Python and requires a series of three conditional statements with inequality expressions. In Beep Beep 3, the check requires the creation of 14 objects, essentially requiring the author to create an abstract syntax tree by hand. Beep Beep 3 does support the creation of domain specific languages (DSLs) but intentionally avoids providing one [1].

The comparison in Table 5 shows that, in this case, Palisade and Beep Beep 3 had similar mean detection latency but the standard deviation of Palisade's latency was lower (lower is better). The higher standard deviation for Beep Beep 3's detection latency appears to be the result of the Java Virtual Machine's Just-In-Time (JIT) compiler as a few anomalies early in the experiment had many times longer detection latencies than the rest. These longer detection times could probably be avoided by implementing a boot-strapping process for the Beep Beep 3 processor, but this would require another component and added complexity not required by Palisade.

Table 5: Results of comparing Palisade with Beep Beep 3.

Round-trip latency	**Beep Beep 3**	**Palisade**
Mean (lower is better)	2.87 ms	2.85 ms
Standard deviation	3.2 ms	0.66 ms

7. Discussion

This section discusses the Palisade results and design choices. We divide the discussion in three parts: software architecture, performance, and anomaly detection.

7.1. Software Architecture Evaluation

Evaluating software architectures is not a straightforward task. There is no common language to describe different software architectures and no clear way to understand and compare different software concerns, such as maintainability, portability, modularity, and reusability [57]. Also, the effectiveness of the software architecture is related to the experience and knowledge of the development team, thus quality must be considered in this context.

We examined two software architecture evaluation methods, Software Architecture Analysis Method (SAAM) [57] and Architecture Tradeoff Analysis Method (ATAM) [58], to determine if we could objectively evaluate Palisade's architecture. We found that both methods are intended for evaluating monolithic software projects that serve specific business cases, and that they do not map well onto Palisade. However, both SAAM and ATAM argue that *modularity* and *extensibility* are important metrics for

[1] The API provided by Beep Beep 3 could arguably be described as a deep internal DSL

evaluating the quality of an architecture. While we cannot quantitatively measure these metrics, here we present an argument that they are well satisfied by our design.

One measurement of the extensibility of software is the number of lines of code that must be written to meaningfully extend it. The most common way to extend Palisade is to write a new processor node. The average number of lines in the current Palisade processor nodes (written in Python) is 144.25 lines (including comments). Autoencoder is the longest processor node with 601, while Clip detector is the shortest with 72 lines. The *Processor* abstract base class has 239 lines. The `nfer` detector, which uses the `nfer` Python API, has only 42 lines of Python code.

As discussed in Section 4.4, processor nodes are independent modules that share infrastructure from a base class. Editing a processor node has no effect on upstream or sibling processor nodes. Only nodes dependent on the output of the edited node may themselves require editing.

Constructing a new source node does not affect other source nodes in the system. Only processor nodes that will be subscribing to a new source may need adjustment, and then only if the new source differs from those that already exist. Adding a new processor node has a lesser impact than editing one, as no downstream nodes should be affected, typically. Instead a new processor node can be added to the system without a single modification to any other component.

For adding a new processor node, we consider the basic code to extend the base class. A new processor node requires at least 24 lines of code in Python. Obviously, the total number of lines depends on the complexity of the algorithm, but the processor abstract base class makes extending Palisade straightforward.

We compared the extensibility of Palisade with the CEP/Runtime Verification (RV) system Beep Beep 3 In Section 6.3. While it was possible to build processors in Beep Beep 3 that mimicked those in Palisade, they required tight coupling between components. In Beep Beep 3, constructing a new processor or sink node would require modifying the other nodes. Palisade's loose coupling between components means that these similar modifications are not required to support new or modified nodes.

Palisade can be used in any embedded system that provides a network interface. As the core Palisade functionality is built around the Redis publish-subscribe interface, any system that has a network interface can send data directly to Redis or to a server that then sends to Redis. Also, RESP is simple and would be easy to port to an embedded system without Linux support. Consequently, we believe that the integration of Palisade with any embedded system is a straightforward task.

7.2. Performance Evaluation

Palisade is built for low latency anomaly detection and this is evident from our comparisons with other frameworks. In the case study evaluation in Section 5.3, an `nfer` Palisade processor detected anomalies over 35 times faster than a comparable detector using the CEP system Siddhi. In the case study evaluation in Section 6.3, two Palisade processors detected anomalies with similar latency to a comparable detector using Beep Beep 3, which required tight coupling between components and a using a complicated API for performing simple data comparisons.

We looked for other appropriate frameworks to compare against Palisade detectors such as the Autoencoder processor, but we discovered that no such framework exists. It does not make sense to compare Palisade's performance against a framework which does not support many of the same core features or which is unusable in the same environments. Frameworks such as Extendible and Generic Anomaly Detection System (EGADS) [59] and Datastream.io [60] only support CSV input for offline detection, while Palisade operates online. Other frameworks like Esper [61] and TeSSLa [62] support online stream processing, but lack support for distributing processors over a network. Detectors like Hogzilla [63], StreamMill [64], and NiagaraCQ [65] are abandoned projects that cannot be installed. Others, like Thirdeye [66], can only be run in a cloud environment, making them ill-suited for latency comparisons. The lack of online, streaming, distributed, locally runnable anomaly detection frameworks shows the need for Palisade, and

we hope that our work motivates others to design comparable tools.

An important design decision in Palisade regards the copying of messages instead of passing message IDs. Once data arrives into a channel, Redis copies the messages to all nodes that subscribed to that specific channel. Another approach, found in Zero-Copy message protocols [67] for example, would be to pass just the message ID to all destination nodes. The ID would then be used to access a central database to retrieve the data. When most nodes require the data, however, the ID passing approach causes a performance bottleneck due to access serialization at the central database (increased latency). We assume that a node that subscribes to a channel needs the data on that channel, so the message copying approach reduces latency while not affecting the processing or memory requirements. This is a reasonable assumption given that it only requires different types of data to be assigned separate channels. The ID passing approach is usually used in micro-service architectures and is preferable when the target application needs all of the data (good for batch processing) [68].

7.3. Anomaly Detection

The multiple anomalies detected by different processors can be compared against each other to verify anomalies and thereby decrease the false positive rate of anomaly detection by Palisade (this could be done by a voter sink node, for instance). There are also cases where some anomalies are detected by only a subset of the detectors. Palisade covers these cases as a variety of detectors can be integrated with low-development effort (due to our design choices - command channel, abstract base class, and data formats).

The choice of parameters in the detector nodes plays a central role in the efficiency of such detectors. In our experiments, we varied the detector parameters until we found a configuration without false positives (Sections 5 and 6). This was possible because we could repeat the execution of the detectors several times. When the execution cannot be repeated, we suggest tuning the detector parameters using a system simulation.

8. Related Work

This section describes existing work related to Palisade. We divide the discussion by subject area: anomaly detection, Information Flow Processing (IFP) systems, anomaly detection with streaming frameworks, offline frameworks, and outdated or commercial frameworks. Few existing works combine the central features of Palisade: online, distributed anomaly detection for both time series and event streams. Our work is motivated by the lack of options in this niche area.

8.1. Anomaly Detection

Anomaly detection, sometimes called outlier detection, attempts to find unexpected or non-conforming patterns in data [5, 6, 7, 8, 9, 10]. Anomalies are distinct from noise, in that noise is not of interest and hinders analysis. The output of an anomaly detector may be either a *score* or a *label*, but the purpose is always to provide a verdict on whether an anomaly was detected at a given time.

Anomaly detection has appeared in statistics literature for many decades [69, 70], but more recently it has found application and been studied in other fields. In healthcare, anomaly detection is used to look for cardiac irregularities that might indicate heart failure or patterns of disease outbreak [71, 72]. In computer network security, anomaly detection is widely used in intrusion detection systems to look for suspicious activity [43, 73, 74]. Banks, insurance companies, and advertising firms, among others, employ anomaly detection to search for instances of fraud [75, 76, 77]. Heavy industry and safety-critical systems operators like airlines use anomaly detection for equipment damage detection [78, 79]. Recent work has shown how anomaly detection can be applied to detect events in a power grid [80].

Lightweight Online Detector of Anomalies (LODA) is a data streaming online anomaly detection system [81]. LODA uses a collection of one-dimensional histograms to improve the anomaly detection. The rationale behind the use of a collection of weak classifier is because together they can form a strong classifier [82]. LODA presented the same performance in

terms of precision of HS-Tress, but with better time to process a data stream.

Weber et al. proposed a two-stage anomaly detection framework for vehicle signals [83]. The first stage is based on static checkers (for CAN messages) and the second stage is based on machine learning algorithms (named learning checks). The learning checks stage implements a Recurrent Neural Network (RNN) and LODA. In a performance evaluation using CAN messages from a vehicle, RNN had a false positive rate of 0.065%. Palisade also supports both static and machine learning-based algorithms. However, Palisade supports the execution of all detectors in parallel or in any number of stages (not only two). In the same work, the authors defined seven types of anomalies that can occur in a sensor or Electronic Control Unit (ECU) [83]. Table 6 compares the seven types of anomalies propose in [83] with our nomenclature described in Section 3. We can note that the seven types of anomalies are a subset of ours. In this sense, we provide a more comprehensive overview of anomaly symptoms that can occur in embedded real-time systems.

Table 6: Anomaly symptoms defined in [83] compared to our proposed symptoms.

Symptoms in [83]	**Our Symptoms**
Sine anomaly	S-Wave
Plateau stuck anomaly	Loss
Peak anomaly	Spike
Negative peak anomaly	Spike
Noise	Noise
Plateau rise/fall anomaly	Clipping
Zero fall anomaly	Clipping/Loss

8.2. Information Flow Processing Systems

Palisade processes flows of information online, deriving high level events and alerts from the flow as data is received. This online flow processing has similarities to the definition by Cugola and Margara of a CEP system [84]. CEP systems are a kind of IFP system that supports continuous and timely processing of low-level event streams into high-level abstractions according to predefined rules. CEP systems are differentiated from Data Stream Management System (DSMS) systems in that DSMS systems work on generic data streams rather than event streams and their output is similarly unconstrained. Palisade supports both event and non-event stream information and includes processors, like `nfer`, that use predefined rules, and learning-based processors that use training data to construct behavioral models.

This section discusses some CEP systems in the literature that could be applied to some of the use cases for Palisade. However, all of these systems require predefined rules to construct their event abstractions, which most Palisade processors do not. As a result, these CEP systems should only be compared to Palisade processors with the same requirements.

Gigascope is a DSMS designed for network monitoring, intrusion detection, and traffic analysis [85]. Gigascope uses a Structured Query Language (SQL)-like query language called Gigascope Query Language (GSQL) that uses data streams as its input and output. Gigascope is explicitly aimed at intrusion detection in networked systems and is not a general solution for anomaly detection.

Triceps is an open source CEP system that does not define its own SQL variant, but rather has the user implement queries and operations directly in C++ or Perl [86]. Triceps is unique in that it is an embedded CEP. That is, Triceps is meant to be used as a library and to be embedded into other programs. This fills an interesting niche, but it is not a framework for distributed anomaly detection like Palisade. In the future it would be interesting to build a processor using Triceps, similar to the already existing `nfer` processor.

Esper is a CEP and DSMS for Java and .Net (Nesper) with a SQL variant called Event Processing Language (EPL) that Esper compiles to byte code [61]. Esper is designed for low latency and high throughput, as well as extensibility and low resource utilization. These traits make Esper a good candidate for online anomaly detection. Esper is designed to work well running inside a distributed stream processing framework and includes examples of integrations with Java networking libraries. However, Esper is not itself a distributed stream processing framework, and integrating Esper with Palisade would require building a variety of custom components to handle networking, serialization, and command.

Siddhi is an open source CEP system deployed by

companies such as Uber, eBay, and PayPal for use cases like fraud analysis and policy enforcement [52]. Siddhi uses a specification language called Streaming SQL and supports input from a variety of streaming sources such as Apache Kafka and NATS in diverse formats like JSON and Extensible Markup Language (XML). It supports streaming input and ouput, multiple end-points, and specification-based anomaly detection, and is one of the existing works closest to supporting Palisade's requirements (defined in Section 4). We compared the detection latency of Palisade with Siddhi, where detection latency is defined as the time difference between the instant data is generated by a source and the instant it is reported as anomalous by a detector. The results, reported in Section 5.3, show that Palisade responds over 35 times faster on average than Siddhi for our case study.

8.3. Anomaly Detection with Streaming Frameworks

Several other works have used data streaming frameworks for online detection of errors or anomalies. Lopez et al. discuss the characteristics and compare the throughput of three stream processing platforms (Apache Spark, Flink, and Storm) using a threat detection application [87]. Solaimani et al. used Apache Spark to detect anomalies for a multi-source VMware-based cloud data center [88]. Subramaniam et al. proposed a framework to detect anomalies online (outlier detection) in wireless sensor networks [89]. However, the authors only implemented the framework in a simulator. Du et al. built a streaming detector using Apache Storm that used k-Nearest Neighbors (k-NN) to detect anomalies in IP network traffic [90]. Shi et al. implemented an online fault diagnosis system based on Apache Spark for power grid equipment [91]. Song et al. proposed an integrated system for explainable anomaly detection using Apache Spark called EXAD [92].

Thirdeye is an anomaly detection framework based on Apache Spark [66]. It uses machine learning and artificial intelligence algorithms for cybersecurity, data analytics, and outlier detection. Thirdeye is designed for deployment on Amazon's AWS cloud computing platform. Palisade, by contrast, is designed to run on a curated local area network to reduce communication latency.

Beep Beep 3 is a stream-processing system that combines some aspects of CEP systems with ideas from RV [93]. Beep Beep 3 is primarily a set of Java APIs to build synchronous processors for arbitrary data types. The standard Beep Beep 3 APIs may be augmented with modules called *palettes* that implement interfaces such as network communication, temporal logic, and plots.

Beep Beep 3 has been studied for use in online, streaming anomaly detection [55]. However, Beep Beep 3 has different goals from Palisade that make it less well suited for distributed anomaly detection. We compared the detection latency of Palisade with Beep Beep 3, as well as the experience of building anomaly detectors using the two frameworks, in Section 6.3. The results show that the two tools have similar detection latency, but that Palisade is better suited than Beep Beep 3 for detecting anomalies in a distributed environment.

TeSSLa is a stream-based specification language and monitoring system designed for specifying and analyzing the behavior of systems where timing is important [62]. TeSSLa, like Beep Beep 3, combines aspects of CEP systems with RV. However, unlike Beep Beep 3, TeSSLa may only be used through an external DSL and its interpreter only accepts input via file arguments or standard in. While TeSSLa's language and theoretical foundation are exciting, its lack of network support means that it cannot currently operate as a distributed, online anomaly detection framework. Integrating TeSSLa with Palisade is also impossible because TeSSLa is only distributed as a compiled binary.

8.4. Offline Frameworks

Datastream.io is an open-source anomaly detection framework that allows users to integrate their custom detectors for testing and training [60]. The project plans to support online streaming but presently only supports the use of CSV files as input to perform offline detection.

EGADS is an open-source anomaly detection framework by Yahoo [59]. EGADS is a self-contained Java package developed for time-series anomaly detection, providing access to multiple detectors.

EGADS accepts input only in the form of CSV and standard-input and is no longer actively maintained.

Frankowski et al. used a variety of CEP systems, including Siddhi and their own SECOR CEP, to detect intrusions and anomalies [94]. Their work combined several CEP systems to periodically analyze log files and store the results in a database. They showed that it is possible to build an effective, signature-free anomaly detection framework using off-the-shelf components. However, they did not construct an online detector.

8.4.1. Outdated or Commercial Frameworks

Other frameworks have been proposed in the past but are unusable in practice because they are either expensive to license or unmaintained. NiagaraCQ was an early and influential continuous query system, but the software has not been available for at least a decade [65]. SASE was a stream processing system designed to support complex queries and high throughput, but it was last maintained in 2014 [95]. Cayuga was a stateful publish-subscribe system based on Non-deterministic Finite AutomatonNon-deterministic Finite Automata (NFAs) that was adapted as an event monitoring system, but it was last maintained in 2013 [96]. Stream Mill was a DSMS that combines predefined rules with statistical learning algorithms for mining queries [64], but it was last maintained in 2012. GEM is a commercial CEP vendor in the industrial space and, as such, their software is not freely available [97].

Hogzilla is an open-source anomaly-based Intrusion Detection System (IDS) targeted towards network communications [63]. Hogzilla purports to detect a wide range of network attacks including zero-day attacks. At the time of publishing, the software no longer runs and has not been maintained for some time. However, in October 2019 the project website was updated to report that the tool will be maintained and supported by a commercial partner, so Hogzilla may return to relevance in the area.

Many of the IFP systems Cugola and Margara review are either no longer maintained or locked up behind commercial licenses [84]. Other promising systems from their study that are unavailable or impractical include: the Borealis stream processor (abandoned 2008), StreamBase (commercial), SQLStream (commercial), Oracle CEP (commercial), Tribeca (disappeared), and TelegraphCQ (abandoned 2003).

Palisade is intentionally designed as a set of distributed micro-services built around a data streaming architecture. Palisade provides a balance between low latency anomaly detection and loosely coupled services.

9. Conclusion

In this article, we presented Palisade, a software framework for anomaly detection in embedded systems. We introduced a new taxonomy of anomaly symptoms, and we designed Palisade to support their detection using a variety of algorithms. Palisade is built around the Redis publish-subscribe interface, which allows running different anomaly detectors with the same input data across a distributed network. We demonstrated the viability of the proposed framework using two case studies, one using data from an autonomous vehicle and another one using data from an ADAS platform. We argued that Palisade is easy to operate and modify and that it detects anomalies with low latency.

As future work, we plan to integrate Palisade with the University of Waterloo's autonomous car and implement more learning-based anomaly detectors. The datasets used in the experiments is available online [51] and the Palisade source code is available upon request.

References

[1] X. Liu, H. Ding, K. Lee, L. Sha, M. Caccamo, Feedback fault tolerance of real-time embedded systems: Issues and possible solutions, SIGBED Rev. 3 (2) (2006) 23–28.

[2] A. Taylor, S. Leblanc, N. Japkowicz, Anomaly detection in automobile control network data with long short-term memory networks, in: 2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA), 2016, pp. 130–139.

[3] C. McCarthy, K. Harnett, A. Carter, Characterization of potential security threats in modern automobiles: A composite modeling approach, Tech. rep., National Highway Traffic Safety Administration, Washington (2014).

[4] E. Marin, D. Singelée, B. Yang, I. Verbauwhede, B. Preneel, On the feasibility of cryptography for a wireless insulin pump system, in: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, CODASPY '16, ACM, New York, NY, USA, 2016, pp. 113–120.

[5] S. Agrawal, J. Agrawal, Survey on anomaly detection using data mining techniques, Procedia Computer Science 60 (2015) 708–713, knowledge-Based and Intelligent Information & Engineering Systems 19th Annual Conference, KES-2015, Singapore, September 2015 Proceedings.

[6] V. Chandola, A. Banerjee, V. Kumar, Anomaly detection: A survey, ACM Comput. Surv. 41 (3) (2009) 15:1–15:58.

[7] A. Patcha, J.-M. Park, An overview of anomaly detection techniques: Existing solutions and latest technological trends, Computer Networks 51 (12) (2007) 3448–3470.

[8] Z. A. Bakar, R. Mohemad, A. Ahmad, M. M. Deris, A comparative study for outlier detection techniques in data mining, in: 2006 IEEE Conference on Cybernetics and Intelligent Systems, 2006, pp. 1–6.

[9] M. Agyemang, K. Barker, R. Alhajj, A comprehensive survey of numeric and symbolic outlier mining techniques, Intelligent Data Analysis 10 (6) (2006) 521–538.

[10] M. A. Pimentel, D. A. Clifton, L. Clifton, L. Tarassenko, A review of novelty detection, Signal Processing 99 (2014) 215 – 249. doi:10.1016/j.sigpro.2013.12.026.

[11] Redis, Website, `https://redis.io/`, accessed: May 2018 (2018).

[12] Mitre, Common attack pattern enumeration and classification, `https://capec.mitre.org/\\data/definitions/1000.html`, accessed: Sep 2018 (2018).

[13] Y. Bengio, Gradient-based optimization of hyperparameters, Neural Computation 12 (8) (2000) 1889–1900. doi:10.1162/089976600300015187.

[14] M. Feurer, F. Hutter, Hyperparameter Optimization, Springer International Publishing, Cham, 2019, Ch. 1, pp. 3–33. doi:10.1007/978-3-030-05318-5_1.

[15] C. Miller, C. Valasek, Remote exploitation of an unaltered passenger vehicle, in: Blackhat USA, IOActive, 2015, pp. 1–91.

[16] K. Rollick, A. Roczko, L. Mitchell, Combustible gas detector sensor drift: Catalytic vs. infrared, InTech Magazine (Aug 2010).

[17] M. Seiter, H. J. Mathony, P. Knoll, Parking assist, in: Handbook of Intelligent Vehicles, Springer, 2012, pp. 829–864.

[18] J. Petit, B. Stottelaar, M. Feiri, Remote attacks on automated vehicles sensors : Experiments on camera and lidar, in: Black Hat Europe, 2015, pp. 1–13.

[19] S. Mukherjee, H. Shirazi, I. Ray, J. Daily, R. Gamble, Practical DoS attacks on embedded networks in commercial vehicles, in: I. Ray, M. S. Gaur, M. Conti, D. Sanghi, V. Kamakoti (Eds.), Information Systems Security, Springer, Cham, 2016, pp. 23–42.

[20] Y. Mo, B. Sinopoli, Secure control against replay attacks, in: 2009 47th Annual Allerton Conference on Communication, Control, and Computing (Allerton), 2009, pp. 911–918.

[21] A. Bolshev, J. Larsen, M. Krotofil, R. Wightman, A rising tide: Design exploits in industrial control systems, in: 10th USENIX Workshop on Offensive Technologies (WOOT 16), USENIX Association, Austin, TX, 2016, pp. 1–11.

[22] C. Moreno, S. Fischmeister, M. A. Hasan, Non-intrusive program tracing and debugging of deployed embedded systems through side-channel analysis, in: Proc. of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES), ACM, New York, USA, 2013, pp. 77–88.

[23] C. Moreno, S. Kauffman, S. Fischmeister, Efficient program tracing and monitoring through power consumption – with a little help from the compiler, in: Design, Automation & Test in Europe Conference & Exhibition, DATE '16, IEEE, 2016, pp. 1556–1561. doi:10.3850/9783981537079_0829.

[24] S. Kauffman, C. Moreno, S. Fischmeister, Static transformation of power consumption for software attestation, in: 22nd International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA '16, IEEE, 2016, pp. 188–194. doi:10.1109/RTCSA.2016.45.

[25] T. T. Y. Lin, D. P. Siewiorek, Error log analysis: statistical modeling and heuristic trend analysis, IEEE Transactions on Reliability 39 (4) (1990) 419–432.

[26] S. M. Bellovin, Packets found on an internet, SIGCOMM Comput. Commun. Rev. 23 (3) (1993) 26–31.

[27] A. Haque, A. DeLucia, E. Baseman, Markov chain modeling for anomaly detection in high performance computing system logs, in: Proceedings of the Fourth International Workshop on HPC User Support Tools, HUST'17, ACM, New York, USA, 2017, pp. 3:1–3:8.

[28] H. Stark, Image recovery: theory and application, Elsevier, 1987.

[29] F. Marvasti, M. Analoui, M. Gamshadzahi, Recovery of signals from nonuniform samples using iterative methods, IEEE Transactions on Signal Processing 39 (4) (1991) 872–878.

[30] K. Sauer, J. Allebach, Iterative reconstruction of bandlimited images from nonuniformly spaced samples, IEEE Transactions on Circuits and Systems 34 (12) (1987) 1497–1506.

[31] H. Feichtinger, Discretization of convolutions and the generalized sampling principle, in: Progress in Approximation Theory, Academic Press, Boston; Toronto, 1991, pp. 333–345.

[32] K. Wang, S. J. Stolfo, Anomalous payload-based network intrusion detection, in: E. Jonsson, A. Valdes, M. Almgren (Eds.), Recent Advances in Intrusion Detection, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 203–222.

[33] J. Rumbaugh, I. Jacobson, G. Booch, Unified Modeling Language Reference Manual, The (2nd Edition), Pearson Higher Education, 2004.

[34] M. Dunne, G. Gracioli, S. Fischmeister, A comparison of data streaming frameworks for anomaly detection in embedded systems, in: Proceedings of the 1st International Workshop on Security and Privacy for the Internet-of-Things (IoTSec), Orlando, FL, USA, 2018, pp. 30–33.

[35] RabbitMQ, Website, `http://www.rabbitmq.com/`, accessed: Jan 2018 (2018).

[36] A. Kafka, Website, `https://kafka.apache.org/`, accessed: Jan 2018 (2018).

[37] NATS, Website, `https://nats.io/`, accessed: Jan 2018 (2018).

[38] ECMA, The JSON data interchange syntax 2nd edition, ECMA 404, European Computer Manufacturers Association, Geneva, Switzerland (12 2017).

[39] Json.org, http://json.org/, accessed: 2018-05-25.

[40] G. P. B. Documentation, Website, `https://developers.google.com/protocol-buffers/docs/overview#extensions`, accessed: Jun 2020 (2020).

[41] M. Kramer, Nonlinear principal component analysis using autoassociative neural networks, American Institute of Chemical Engineers 37 (2) (1991) 233–243.

[42] J. Lin, E. Keogh, S. Lonardi, B. Chiu, A symbolic representation of time series, with implications for streaming algorithms, in: Proceedings of the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, DMKD '03, ACM, New York, NY, USA, 2003, pp. 2–11.

[43] C. Warrender, S. Forrest, B. Pearlmutter, Detecting intrusions using system calls: alternative data models, in: Proceedings of the 1999 IEEE Symposium on Security and Privacy, 1999, pp. 133–145.

[44] S. Kauffman, K. Havelund, R. Joshi, nfer–a notation and system for inferring event stream abstractions, in: International Conference on Runtime Verification, Springer, 2016, pp. 235–250.

[45] S. Kauffman, K. Havelund, R. Joshi, S. Fischmeister, Inferring event stream abstractions, Formal Methods in System Design 53 (1) (2018) 54–82.

[46] J. F. Allen, Maintaining knowledge about temporal intervals, Communications of the ACM 26 (11) (1983) 832–843.

[47] S. Kauffman, S. Fischmeister, Mining temporal intervals from real-time system traces, in: Proceedings of the 6th International Workshop on Software Mining (SoftwareMining), Champaign, USA, 2017, pp. 1–8.

[48] M. M. Z. Zadeh, M. Salem, N. Kumar, G. Cutulenco, S. Fischmeister, SiPTA: Signal processing for trace-based anomaly detection, in: Proc. of the International Conference on Embedded Software (EMSOFT), New Dehli, India, 2014, pp. 1–6.

[49] O. Iegorov, S. Fischmeister, Mining task precedence graphs from real-time embedded system traces, in: 2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2018, pp. 251–260.

[50] A. Narayan, G. Cutulenco, Y. Joshi, S. Fischmeister, Mining timed regular specifications from system traces, ACM Trans. Embed. Comput. Syst. 17 (2) (2018) 46:1–46:21.

[51] S. K. M. D. G. G. W. K. N. B. S. Fischmeister, Palisade: A framework for anomaly detection in embedded systems dataset (2020). doi:10.21227/44z5-9k90.

[52] S. Suhothayan, K. Gajasinghe, I. Loku Narangoda, S. Chaturanga, S. Perera, V. Nanayakkara, Siddhi: A second look at complex event processing architectures, in: Proceedings of the 2011 ACM Workshop on Gateway Computing Environments, GCE '11, ACM, New York, NY, USA, 2011, pp. 43–50. doi:10.1145/2110486.2110493.

[53] Siddhi, Website, `https://siddhi.io/`, accessed: Jun 2020 (2019).

[54] D. Shin, A platform for generating anomalous traces under cooperative driving scenarios, Master's thesis, University of Waterloo (2018).
URL `http://hdl.handle.net/10012/13534`

[55] M. Roudjane, D. Rebaïne, R. Khoury, S. Hallé, Real-time data mining for event streams, in: 2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC), 2018, pp. 123–134. doi:10.1109/EDOC.2018.00025.

[56] S. Hallé, Event Stream Processing with BeepBeep 3: Log Crunching and Analysis Made Easy, Presses de l'Université du Québec, 2018.

[57] R. Kazman, L. Bass, M. Webb, G. Abowd, Saam: A method for analyzing the properties of software architectures, in: Proceedings of the 16th international conference on Software engineering, IEEE Computer Society Press, 1994, pp. 81–90.

[58] R. Kazman, M. Klein, P. Clements, ATAM: Method for architecture evaluation, Tech. rep., Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst (2000).

[59] Y. EGADS, Website, https://github.com/yahoo/egads, accessed: Sep 2019 (2019).

[60] Datastream.io, Website, https://blog.ment.at/datastream-io-open-source-anomaly-detection-64db282735e0, accessed: Sep 2019 (2019).

[61] E. Tech, Website, http://www.espertech.com/, accessed: Jun 2020 (2019).

[62] L. Convent, S. Hungerecker, M. Leucker, T. Scheffel, M. Schmitz, D. Thoma, TeSSLa: Temporal stream-based specification language, in: T. Massoni, M. R. Mousavi (Eds.), Formal Methods: Foundations and Applications, Springer International Publishing, Cham, 2018, pp. 144–162. doi:10.1007/978-3-030-03044-5_10.

[63] H. ids Data, Website, http://ids-hogzilla.org/, accessed: Jun 2020 (2020).

[64] H. Thakkar, B. Mozafari, C. Zaniolo, Designing an inductive data stream management system: the stream mill experience, in: Proceedings of the 2nd international workshop on Scalable stream processing system, ACM, 2008, pp. 79–88.

[65] J. Chen, D. J. DeWitt, F. Tian, Y. Wang, Niagaracq: A scalable continuous query system for internet databases, in: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, SIGMOD '00, Association for Computing Machinery, New York, NY, USA, 2000, p. 379–390. doi:10.1145/342009.335432.

[66] T. Data, Website, https://thirdeyedata.io/, accessed: Sep 2019 (2019).

[67] B. P. Swenson, G. F. Riley, A new approach to zero-copy message passing with reversible memory allocation in multi-core architectures, in: 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation, 2012, pp. 44–52.

[68] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina, Microservices: yesterday, today, and tomorrow, in: Present and Ulterior Software Engineering, Springer, 2017, pp. 195–216.

[69] F. Edgeworth, Xli. on discordant observations, The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 23 (143) (1887) 364–375.

[70] J. Pickands, Statistical inference using extreme order statistics, The Annals of Statistics 3 (1) (1975) 119–131.
URL http://www.jstor.org/stable/2958083

[71] A. L. Goldberger, L. A. Amaral, L. Glass, J. M. Hausdorff, P. C. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, H. E. Stanley, PhysioBank, PhysioToolkit, and PhysioNet: components of a new research resource for complex physiologic signals, Circulation 101 (23) (2000) e215–e220.

[72] W.-K. Wong, A. W. Moore, G. F. Cooper, M. M. Wagner, Bayesian network anomaly pattern detection for disease outbreaks, in: Proceedings of the 20th International Conference on Machine Learning (ICML-03), 2003, pp. 808–815.

[73] D. Snyder, On-line intrusion detection using sequences of system calls, Master's thesis, Florida State University (2001).

[74] B. Agarwal, N. Mittal, Hybrid approach for detection of anomaly network traffic using data mining techniques, Procedia Technology 6 (2012) 996 – 1003, 2nd International Conference on Communication, Computing & Security [ICCCS-2012]. doi:10.1016/j.protcy.2012.10.121.

[75] T. Fawcett, F. Provost, Activity monitoring: Noticing interesting changes in behavior, in: Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and

Data Mining, KDD '99, ACM, New York, NY, USA, 1999, pp. 53–62.

[76] S. Donoho, Early detection of insider trading in option markets, in: Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '04, ACM, New York, NY, USA, 2004, pp. 420–429.

[77] Ghosh, Reilly, Credit card fraud detection with a neural-network, in: 1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences, Vol. 3, 1994, pp. 621–630.

[78] E. Keogh, J. Lin, S.-H. Lee, H. V. Herle, Finding the most unusual time series subsequence: algorithms and applications, Knowledge and Information Systems 11 (1) (2007) 1–27.

[79] S. Basu, M. Meckesheimer, Automatic outlier detection for time series: an application to sensor data, Knowledge and Information Systems 11 (2) (2007) 137–154.

[80] Y. Zhou, H. Zou, R. Arghandeh, W. Gu, C. J. Spanos, Non-parametric outliers detection in multiple time series a case study: Power grid data analysis, in: Thirty-Second AAAI Conference on Artificial Intelligence, 2018, pp. 4605–4612.

[81] T. Pevný, Loda: Lightweight on-line detector of anomalies, Machine Learning 102 (2) (2016) 275–304.

[82] L. I. Kuncheva, Combining Pattern Classifiers: Methods and Algorithms, Wiley-Interscience, New York, NY, USA, 2004.

[83] M. Weber, G. Wolf, E. Sax, B. Zimmer, Online detection of anomalies in vehicle signals using replicator neural networks, in: 6th Embedded Security in Cars USA (escar USA), 2018, pp. 1–14.

[84] G. Cugola, A. Margara, Processing flows of information: From data stream to complex event processing, ACM Computing Surveys (CSUR) 44 (3) (2012) 15.

[85] C. Cranor, T. Johnson, O. Spataschek, V. Shkapenyuk, Gigascope: a stream database for network applications, in: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, ACM, 2003, pp. 647–651.

[86] Triceps: An innovative cep, http://triceps.sourceforge.net/, accessed: 2017-10-18.

[87] M. A. Lopez, A. G. P. Lobato, O. C. M. B. Duarte, A performance comparison of open-source stream processing platforms, in: 2016 IEEE GLOBECOM, 2016, pp. 1–6.

[88] M. Solaimani, M. Iftekhar, L. Khan, B. Thuraisingham, J. Ingram, S. E. Seker, Online anomaly detection for multi-source VMware using a distributed streaming framework, Softw. Pract. Exper. 46 (11) (2016) 1479–1497.

[89] S. Subramaniam, T. Palpanas, D. Papadopoulos, V. Kalogeraki, D. Gunopulos, Online outlier detection in sensor data using non-parametric models, in: Proc. of the 32Nd VLDB, 2006, pp. 187–198.

[90] Y. Du, J. Liu, F. Liu, L. Chen, A real-time anomalies detection system based on streaming technology, in: 2014 Sixth IHMSC, Vol. 2, 2014, pp. 275–279.

[91] W. Shi, Y. Zhu, T. Huang, G. Sheng, Y. Lian, G. Wang, Y. Chen, An integrated data preprocessing framework based on Apache Spark for fault diagnosis of power grid equipment, Journal of Signal Processing Systems 86 (2) (2017) 221–236.

[92] F. Song, Y. Diao, J. Read, A. Stiegler, A. Bifet, EXAD: A system for explainable anomaly detection on big data traces, in: 2018 IEEE International Conference on Data Mining Workshops (ICDMW), 2018, pp. 1435–1440.

[93] S. Hallé, When RV meets CEP, in: International Conference on Runtime Verification, Springer, 2016, pp. 68–91.

[94] G. Frankowski, M. Jerzak, M. Miłostan, T. Nowak, M. Pawłowski, Application of the complex event processing system for anomaly detection and network monitoring, Computer Science 16 (4) (2015) 351–371. doi:10.7494/csci.2015.16.4.351.

[95] H. Zhang, Y. Diao, N. Immerman, Recognizing patterns in streams with imprecise timestamps, Proc. VLDB Endow. 3 (1–2) (2010) 244–255. doi:10.14778/1920841.1920875.

[96] A. Demers, J. Gehrke, M. Hong, M. Riedewald, W. White, Towards expressive publish/-subscribe systems, in: Y. Ioannidis, M. H. Scholl, J. W. Schmidt, F. Matthes, M. Hatzopoulos, K. Boehm, A. Kemper, T. Grust, C. Boehm (Eds.), Advances in Database Technology - EDBT 2006, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 627–644. doi:10.1007/11687238_38.

[97] GEM, Website, `https://www.softwareag.com/gem/default.html`, accessed: Jun 2020 (2020).