Utilizing Genetic Programming to Identify Failures in ICS Networks

Jasenko Hosic, Jereme Lamps, Derek H. Hart
Sandia National Laboratories

Abstract

Researchers have previously attempted to apply machine learning techniques to network anomaly detection problems. Due to the staggering amount of variety that can occur in normal networks, the results have often been underwhelming. These challenges are far less pronounced when considering industrial control system (ICS) networks. The recurrent nature of these networks results in less noise and more consistent patterns for a machine learning algorithm to recognize. We propose a method of evolving decision trees through genetic programming (GP) in order to detect network anomalies, such as device outages.

Our approach extracts over a dozen features from network packet captures and netflows, normalizes them, and relates them in decision trees using fuzzy logic operators. We used the trees to detect three specific network events from three different points on the network across a statistically significant number of runs and achieved 100% accuracy on five of the nine experiments. When the trees attempted to detect more challenging events at points of presence further from the occurrence, the accuracy averaged to above 98%. Using our method, all of the evolutionary cycles of the GP algorithm are computed a-priori, allowing the best resultant trees to be deployed as semi-real-time sensors with little overhead.

1. Introduction

The ability to accurately identify intrusions or anomalies on a network has long been a goal of the security community [1-3]. Network administrators and security engineers alike want to know exactly what is occurring on their networks, but the amount of data flowing across all the nodes in large-scale networks is simply too great to manually examine. As a result, solutions have been created that analyze data in a number of automated ways. Network-based intrusion detection systems and host-based intrusion detection systems each have their merits, but they often depend on specifically crafted rules which can be circumvented. Signature-based approaches often suffer from being too specific to capture every kind of attack. Unique approaches involving machine learning or anomaly detection methods often achieve high accuracy rates, but still suffer from some imperfections. In a large-scale network scenario, the sheer volume of data generally renders these techniques unusable. Even with a 5% false positive rate, human experts would need to manually inspect an insurmountable number of false alarms.

Industrial control system (ICS) networks can be massive as well, but generally contain far less noise. The way in which programmable logic controllers (PLCs) and remote terminal units (RTUs) transmit data back to a front end processor (FEP) is cyclical. Usually data is transmitted at recurrent intervals, and this periodic nature is a much simpler pattern to define than the arbitrary chaos of a corporate network with endpoints controlled by users. We propose a machine learning approach that takes advantage of this fact in order to detect unwanted behavior on an ICS network.

Although using machine learning techniques on ICS network data was attempted in the past [4-7] the amount of complete solutions is sparse. Furthermore, the goals of prior work were fundamentally different. Rather than determining what type of undesired behavior occurred on the network, most of the research in this area focuses on identifying existing or new attacks. Our approach focuses on three key goals. First, we aim to detect the occurrence of specific effects or events, such as outages of devices on the network, not causes. Second, we want our solution to have a minimal presence on the network. Rather than considering all data flowing across every node in a network, we collect data flowing through one tap in the network, reducing the amount of processing power and space required to execute the solution. By extracting precise data about the traffic, it may be possible to glean key details about devices several hops away from the data collection tap. Third, we want to minimize the amount of information required to accurately determine if the event occurred. To show the validity of our approach, we tested our algorithm at different taps on the network with varying levels of access to the data pertinent to identifying each event.

Our proposed solution for meeting the aforementioned goals employs genetic programming (GP) to evolve decision trees. GP is an iterative, population-based meta-heuristic technique that follows an evolutionary cycle to produce solutions represented by trees or graphs. Multiple guesses at the solution are made, evaluated recombined to produce new solutions. After many such cycles emulating natural selection and evolution, a best solution is chosen [8]. In our approach, this

reinforcement technique evaluates evolved trees based on their ability to use normalized extracted features to accurately identify specific events in a labeled training set. This type of technique was used to great effect in [9], and avoids some of the problems with classic decision tree algorithms [10]. The purpose of the resultant decision trees is to act as sensors that take in buffered packet capture (PCAP) and netflow data, compute the feature extractions, and identify whether a specific event occurred. All of the machine learning iterations are performed *a-priori*, allowing the sensor trees to quickly flag buffers of data as soon as the feature extraction is complete.

In this paper, we focus on three experiments, each tested at three different locations in the network. Our test network is a standard 24-bus power network, virtualized on a server with virtual machines simulating ICS traffic such as modbus. We used a tool developed by Sandia National Laboratories to deploy the virtual network. The three events that we tried to detect are: a specific router failing, any router in the network failing, and a specific FEP failing. Our points of presence were at different locations of the 24-bus network, with some being close to the network outages while others were several hops away. The majority of our results achieved 100% accuracy in detecting specific events, and only dipped below 98% when too little data was provided to the machine learning algorithm.

This method can be used to answer a number of different questions about the state of ICS networks, given a rich feature set and the right amount of training data. Is the recurrent pattern of communication broken? Has a large exfiltration of data occurred? Has the process logic or firmware on a PLC been updated across the network? The remainder of this paper explores our methodology and test data in greater detail. The GP parameters are explained, and an analysis of the nine experiment results is shown. We end with a look at potential future work regarding this idea.

2. Evolutionary Computation

Evolutionary Computation (EC) is a computational intelligence concept that can be applied to a number of different problem domains. It is a population-based, iterative learning technique inspired by neo-Darwinian evolution theory and Mendelian genetics, commonly used as a black box search algorithm for optimization problems [11]. Although it is traditionally thought of as a reinforcement learning technique wherein the quality of potential solutions is evaluated throughout the iterative "learning" process using criteria and information about the problem domain, it can also be positioned to perform supervised learning tasks [12]. Supervised learning uses historical data to train a model and make future predictions based on past results or performance. This historical data is referred to as a training set.

The population-based nature of EC algorithms provides a two-fold advantage. First, the algorithm is able to make multiple attempts at finding an optimal solution in parallel, offering a faster convergence to the global optimum. Second, it provides a diverse pool of solution components from which the algorithm can derive better solutions, and it allows for a relative comparison of solution quality. Varying EC algorithm operators shifts the balance in the amount of exploration for new solutions and the amount of exploitation of previously explored areas within the search place [13].

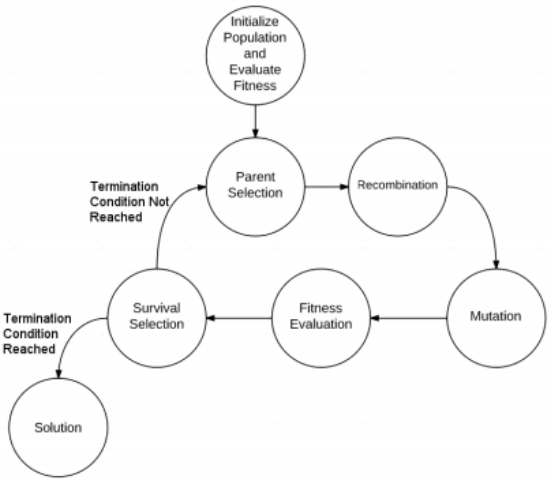


Figure 1. The General EC Cycle

2.1. The Evolutionary Computation Cycle

EC algorithms follow an iterative cycle illustrated in Figure 1 as well as described in greater detail in the proceeding subsections.

2.1.1. Representation. Prior to implementing an EC algorithm, a representation must be chosen for the particular problem. This representation is commonly binary, wherein each bit corresponds to a certain "gene" or component of the solution. There are other representations, such as an integer representation, but the most logical scheme will depend on the problem.

2.1.2. Initialization. An initial population (with represented as MU) is created as a starting point for the algorithms execution. The ways in which this initial population is established can vary, such as using a uniform random distribution or a percentage-based proportional allocation, but the ultimate goal is to make initial explorative attempts at finding an optimal solution.

2.1.3. Fitness Evaluation. This step is the quality determination and the reinforcement aspect of the algorithm. The fitness evaluation is a custom formula

for determining a numerical value for each solution relative to other possible solutions.

2.1.4. Parent Selection. Parent selection determines the two individuals in the population that will combine to produce new solutions. Many methods exist to perform this selection process, each introducing a different level of elitism for the evolutionary learning process. When only the best solutions are considered for the recombination phase, the algorithm is very elitist. Whether this is appropriate or not depends on the specific problem.

2.1.5. Recombination. The recombination operator develops new solutions based on the components of two parent solutions. Again, there are many way to accomplish this, such as cutting two parent binary strings in half and stitching halves of different parents together. λ offspring are generated after this phase.

2.1.6. Mutation. The mutation aspect is critical to a successful evolutionary algorithm. Mutation introduces more exploration to the algorithm, as solutions are mutated, with some probability, into other solutions by performing operations such as bit flips or bit swaps.

2.1.7. Survival Selection. Survival selection is similar to parent selection, and many of the techniques from that phase can be re-used here. Rather than selecting two parents to recombine into new offspring solutions, however, the purpose of this operator is to filter out solutions that should no longer continue to exist in the evolutionary process.

2.1.8. Termination. When a certain condition is reached, the algorithm should end, and the latest and best population is the resultant pool of optimal solutions. The termination condition could be based on a number of factors, such as the number of generations or evaluations that have iterated throughout the cycle, the amount of time that has passed, or the fitness level that has been reached.

2.2. Genetic Programming

GP is an iterative, population-based meta-heuristic technique that follows the evolutionary cycle shown in Figure 1 to produce solutions represented by trees or graphs. Like generic EC, multiple guesses at the solution are made, evaluated, and recombined to produce new solutions. After many such cycles emulating natural selection and evolution, a best solution is chosen [8]. The tree-based representation is encoded such that internal nodes take the form of some operators, such as Boolean logic or mathematical, and the leaf nodes are terminal values that act as inputs for the internal nodes when the trees are evaluated in order. Due to the varied representation, common EC operators such as bit flips will not be compatible. Therefore, different recombination and mutation operators must be implemented. Furthermore, because the tree structure is used as a meta-heuristic technique where the size of the heuristics commonly have no size constraints, the solution candidates can grow to an enormous size across many generations. In order to reduce heavy bloat, parsimony pressure can be applied such that smaller, more elegant solutions with equal fitness otherwise are favored over larger solutions. A hard tree depth limit can also be implemented.

3. Related Work

Using GP to develop new heuristics is not a unique concept. It has been explored with great success in prior works [14-17]. The novel aspect of our research lies in the application of the algorithm as well as the feature extractions. Others have proposed a similar approach [2][5][7], but there are some fundamental differences in the machine learning algorithms used and the goal they are trying to achieve. We used the effective portions of their work to enhance ours and create an accurate method for detecting anomalies.

Sommer et al [3] provided some warnings about using machine learning to detect network intrusions. Their work focused on less recurrent networks, but the conclusions are important to consider nonetheless. The major claim is that machine learning has had limited success in the intrusion and anomaly detection domain because that problem type is fundamentally different than the kinds of problems with which machine learning algorithms excel. Machine learning is better at determining similarities than it is at finding new, meaningful outliers. This is perhaps why machine learning techniques such as support vector machines (SVM) have the greatest success in classification problems where the possible groupings are known. Sommer et al. go on to suggest that the few successes these techniques have experienced in the security domain, such as spam filtering, are a result of detecting variations of known attacks rather than new attacks altogether. Lastly, there exists a high cost for misclassifying data. False negatives are generally unacceptable, and false positives must be resolved by subject-matter experts.

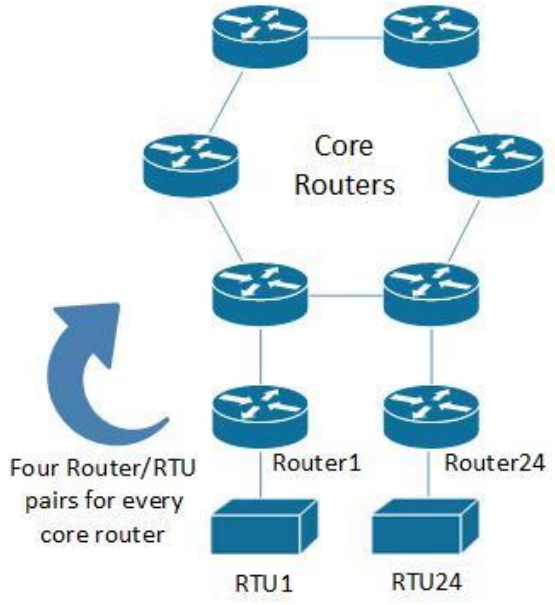
The implications of these challenges can be seen with the work of Yin et al. [18]. Their work was a great inspiration for our approach, but ultimately suffered from some of the problems described by Sommer et al. They used a GP approach to develop new rules for anomaly detection. The classic DARPA network data set was used to evolve the new rule set. This achieved great results, outperforming prior work by detecting 84 out of 148 events, but it is

not practical in a real scenario with live networks. During their testing phase, an unacceptable number of attacks went undetected. Their algorithm required two passes, which resulted in some undesirable inefficiency. Despite these few issues, their findings were a great starting point, and highlighted some of the problems we sought to overcome.

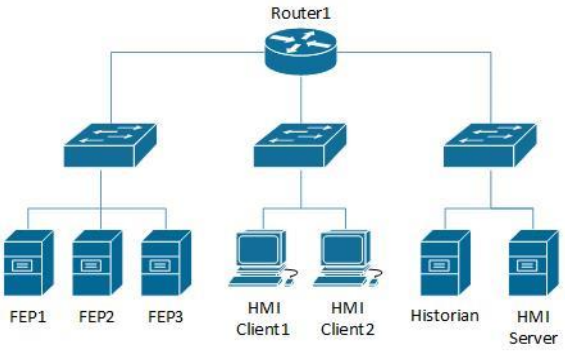
Lu et al. [19] used the same data set as [18], but recognized some of the issues involved with it. The DARPA data has long been the standard in network attack testing data, but it is outdated and incomplete. It also is not intended to represent periodic ICS networks, so we have opted not to use it. Lu et al. propose a slight alternate method as well. The result of their findings is also excellent, as their false positive and false negative rates are slightly above 5%, which is still too high for practical applications, but suggests that a similar approach would perform even better on a recurrent network. The technique attempts to find variations of known attacks, as opposed to new attacks, as is recommended by Sommer et al.

The recent work from [7] is one of two that closely resembles our goals. They use Supervisory Control and Data Acquisition (SCADA) network data and SVM to identify attacks. Over 1500 packets of data are considered in their approach, and their results are impressive. With certain splits of their data, they achieve nearly 100% accuracy. They extracted several features, such as packet rate and packet size. While the effects of an attack are our primary interest, Maglaras et al. attempted to identify the attack itself. Their promising results were nonetheless an inspiration, and we attempted to capitalize on their progress. Our data sets are much larger and consider many more protocols (both ICS communication and otherwise).

The other research that attempts to identify anomalies in ICS or SCADA networks is from Mantere et al [4-6]. Throughout their work, they discussed the challenges of using machine learning on recurrent networks, discovered valuable features to extract, and created a prototype that uses self-organizing maps (SOM) to relate their extracted features. To assist in the feature extraction process, Mantere et al. use Bro, a network security monitoring tool. Bro can aggregate data quickly and reduce some of the overhead associated with custom data extraction. While their results are mostly a proof of concept, initial testing demonstrates they can achieve as little as zero to three false positives per day. However, they did not perform any attacks throughout their training and testing data, so these results are still theoretical. They used packet captures from a real ICS network running for many days, which was an essential step in representing the problem practically. While we did not have the same kind of access to live data, our emulation techniques provide a realistic alternative. The RTU and FEP communications accurately model live systems, allowing the GP algorithm to evolve sensors that could be placed in real systems after testing.



(a) A subset of the 24-bus network



(b) A single bus connected to a router

Figure 2. The topology of the 24-bus network

4. Methodology

Developing decision trees to act as sensors in a live network must first begin with selecting appropriate features to extract from the available data. We deployed a virtual 24-bus power network with over 140 nodes, including routers, RTUs, FEPs, and other components. The RTUs and FEPs communicated with ICS traffic such as modbus, and the routers establish dynamic routing through the OSPF protocol. If FEPs were unable to communicate with their corresponding RTUs due to network or device failure, they continuously tried to reestablish connections, greatly increasing the amount of traffic they generated. Fig. 2(a) shows a sampling of our network structure. The same structure exists in multiple branches of the network. The rest was

omitted for clarity. Figure 2(b) shows how devices are connected to the routers. The network contains multiple human-machine interface (HMI) workstations, an HMI server housing the HMI applications, and a historian that aggregates all RTU status updates and FEP interactions. Not all branches in the network have as many devices connected to the routers.

The goal was to detect anomalies in semi-real-time. True real-time detection is difficult to achieve because a large amount of data needs to be aggregated before useful features can be extracted. We tested our method with many shorter buffer sizes, but found the best accuracy with twenty minute buffers. Even larger buffers may have produced better results for our more difficult tests, but we wanted to minimize the amount of storage and processing required implementing our solution. Table 1 describes our nine experiments and the location of the data buffering tap in each experiment.

Table 1. Experiment Descriptions

Exp.	Fail Event	Tap Location
1	Specific router	Next to target router
2	Specific router	Several hops, next to RTU
3	Specific router	Several hops, next to FEP
4	Any router	Next to router
5	Any router	Next to RTU
6	Any router	Next to FEP
7	Specific FEP	Several hops, next to router
8	Specific FEP	Several hops, next to RTU
9	Specific FEP	Several hops, next to diff. FEP

We performed data collection at each of the three tap locations listed in Table 1, consisting of ten total hours, or several gigabytes, of PCAP and netflow traffic. Throughout the data gathering, we forced multiple device failures. Our intent was to truly stress the versatility of our feature extraction and machine learning, so many of the data sets we collected captured numerous outages occurring at one time, with many devices coming back up after several minutes. Table 2 shows a description of the ten data sets that we used to comprise our training and testing data sets. At the start of each data collection, the network was fully operational and in a steady state. In our descriptions, we define t_x to represent the x^{th} minute for the data set. For example, t_5 defines the 5^{th} minute of data collection for the particular data set.

For these data sets to act as indicators in discerning the occurrence of the events, meaningful feature extraction algorithms need to be used. The feature extraction algorithms provide metrics that can be used by the GP to evolve the decision tree sensors.

Table 2. Data Collection Descriptions

Data No.	Data Description
1	No fail events for the 20 minute period
2	At t_{10} a router fails
3	At t_{10} a router fails At t_{17} it comes back online
4	At t_5 a router fails At t_{10} another router fails
5	At t_{10} a different router fails
6	At t_{10} a FEP fails
7	At t_{10} a FEP fails At t_{17} it comes back online
8	At t_7 a FEP and two routers fail At t_{14} one router comes back online
9	At t_5 a router fails At t_{10} it comes back online At t_{15} it fails again
10	At t_{10} a FEP and two routers fail

The following is a description of the features we extracted for the GP algorithm, followed by a comparison of their validity as discriminators of event occurrence.

4.1. Temporal Data

Five of the features we extracted were low-level temporal features. We computed the average number of packets sent by the target across the duration of the buffer. We also calculated the rate OSPF packets were being sent in cases where the target was a router. Another feature extracted the standard deviation of the number of packets sent at every minute of the buffer. We recorded the longest period of time a device would be silent. Lastly, we captured the average duration of each flow that the target initiated. The temporal data used in [4] and [7] was a great inspiration for these choices. If the device has a high packet rate early in the buffer and the rate drops, several of these features will be impacted. Some of our data sets attempt to circumvent these temporal features with short, periodic outages. If the device comes back up quickly after each outage, the average may not dip enough to be detected by the feature.

The importance of the flow duration feature is that it can detect differences in the types of communications used by the devices. It is possible for this feature to be helpful when discovering other, more nuanced network anomalies.

4.2. Communication Failure

We also wanted our sensors to capture obvious indicators of communication failure. The trees are given a count of the ``ICMP unreachable'' packets, incomplete handshakes (SYN, SYN/ACK, without

the last ACK), and communication pattern breaks. The communication pattern breaks are calculated by determining the pattern at which the target communicates. For instance, routers have a pattern of sending OSPF Hello packets every 10 seconds, any deviations in this pattern can be flagged as a potential router outage and counted. This is particularly helpful due to the recurrent behavior of these networks. The FEPs generally receive RTU information at fixed intervals with little discrepancy.

4.3. Communication Profile

Lastly, we included some high-level features to capture a profile of the kinds of communications in the network. We captured the length of each packet sent by the target, as well as the time-to-live (TTL) of every packet coming across the tap. The packet size would differ if the content of the communication changed, and the TTL would hint at a change in path, likely due to an outage or new device on the network.

We also made use of collocations within our features. Collocations are a measure of how two things are usually grouped. The classic example of collocation involves language. The words "running" and "water" are much more likely to occur together than the words "walking" and "water". Likewise, "tall" and "tree" make more sense together than "high" and "tree". These words are therefore more strongly connected. To numerically represent these strong connections, a mutual information equation can be used. The formula for mutual information compares the probability of two objects occurring together if they are independent and the probability of their actual occurrence together [20]. The formula is shown in Equation 1.

$$I = P(XY) \cdot \log \frac{P(XY)}{P(X)P(Y)} \qquad (1)$$

Within the netflows, we found strongly connected source-destination pairs occurring adjacent to each other and flagged any weak connections (based on an experimentally calculated threshold of 0.05). We used collocations for two separate PCAP features by first creating a label for each packet type, such as "ACK packet" or "OSPF Hello packet". The collocation feature then used these human-readable labels to determine what packet types often occur together. The feature was split in two, with the target as the source and as the destination. These features should indicate if packets that have previously not been seen, such as ICMP unreachable packets or new SYN packets attempting to establish connections, are captured.

4.4. Single Feature Tests

We ran these features individually on our data sets to validate the need for our GP approach. We chose thresholds that allowed each feature to perform optimally on the datasets. Table 3 shows the average accuracy of our best-performing features on one of our easier experiments, Experiment 4 from Table 1. The average was calculated by using *k*-fold cross validation and splitting our data sets into training and testing sets, then using the optimal threshold from training to categorize the remaining test data. We used a *k* value of five, and the training set size in each of the five folds was 80% of the total data set size. This meant that each data set was in the testing set exactly one time.

Table 3. Individual Feature Accuracy on Exp. 4

Feature	Accuracy (%)
Average packets/min	76.67
Packet type collocation	36.67
Communication break: destination	6.67
Communication break: source	60
OSPF packets/min	70
Standard deviation packets/min	50
Average packet length	51.67
Flow collocation	38.33
Longest silence	70
Average flow time	63.33
Average TTL	38.33

The highest average accuracy that any single feature achieved was 76.67%. Clearly, the features alone cannot produce acceptable results, justifying the need for machine learning to relate the extracted information.

4.5. Genetic Programming Algorithm

We developed a GP algorithm that evolves decision trees for each type of event. If multiple events need to be detected, a different tree can evolved for each event given the right features and training data. It is possible for one data buffer to contain many different events of interest, and this approach makes it possible to detect all of them individually with separate trees.

The terminals, or leaf nodes, in the GP trees are represented by the feature extraction algorithms presented in subsections 4.1 through 4.3. The features were all normalized to be between zero and one, inclusive, to prevent any one feature type from outweighing the others due to its numerical value. These normalized values within the terminal nodes of the trees are related by fuzzy logic operators AND, OR, NOT, NOR, NAND, and XOR. The rules of fuzzy logic operators dictate that an AND of two

floating point values is the minimum of the two, while OR is the maximum of the two values. A NOT is equal to one minus the normalized value. Using these rules, the other operators can be extrapolated. The reason for using fuzzy logic operators over setting a hard threshold (such as evaluating any value over 0.5 as true), is that reinforcement learning algorithms such as GP need to determine a solution's quality relative to other solutions. If every tree only returned a value of one or zero, there would only be two values upon which a solution's quality could be evaluated. By forcing floating point values to be returned by a tree, the spectrum of values between zero and one can be used by the algorithm to determine fitness. This allows for a hierarchy of solutions and makes the search space continuously multimodal.

The fitness function is key to evolving successful solutions. Our fitness function is simple. The training data is split into two sets, the PCAPs and netflows that captured the event occurring, and those that did not capture the event. Algorithm 1 shows how the fitness function uses these two sets to compute a numeric value to represent the quality of each solution.

Algorithm 1 Fitness function pseudocode

```
function get_fitness(tree)
    total ← 0
    for all data ∈ data_sets do
        val ← eval_tree(tree)
        if val ≥ accept_thresh && event then
            total ← total + num_not_event
        end if
        if val ≤ 1 − accept_thresh && not_event then
            total ← total + num_event
        end if
    end for
return total
end function
```

The purpose of adding the number of events in the training set to the total fitness if a true negative is detected (and vice versa for a true positive) is to remove any bias from unequal quantities of events and non-events.

5. Experimental Setup

For any GP algorithm to appropriately evolve a heuristic, the right parameters must be chosen. Otherwise, the selective pressure of the algorithm could be too elitist and not allow for enough exploration of the search space, or too relaxed and never converge to a global optimum. Most security practitioners are not machine learning or GP experts, and cannot be expected to tweak many parameters. As such, we used one set of default parameters in almost every experiment, and only modified a few inputs if our results were unacceptable. Even without complex parameter tuning or the use of meta-evolutionary techniques, these parameters perform well. Table 4 lists the parameters used in each experiment.

Table 4. GP Parameters

Parameter	Default	Exp 5.	Exp. 6
μ	100	200	200
λ	20	40	40
max depth	4	4	4
selection	*k*-tourn.	*k*-tourn.	*k*-tourn.
survival	*k*-tourn.	*k*-tourn.	*k*-tourn.
k	7	5	
crossover	single-point	single-point	single-point
mutation	sub-tree	sub-tree	sub-tree
mutation rate	.15	.8	.85
termination	5000 eval.	5000 eval.	5000 eval.
Acceptance Thresh.	.9	.75	.75

The use of a high acceptance threshold forces the algorithm to evolve trees that clearly separate the data into categories of events and non-events. The greater population and offspring sizes on tougher experiments encouraged exploration of the search space. To reduce the complexity of final solutions, we introduced parsimony pressure throughout such that smaller solutions were favored when the fitness was identical.

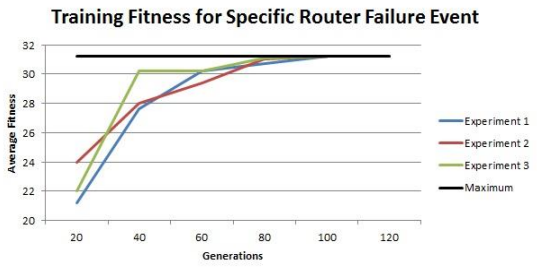
We tested the experiments with thirty runs for each of the cross validations. In total, we completed 150 runs of 5000 evaluations for each experiment. We used 80% of the data as training data and the rest as testing data. Just as with the single feature tests, the five-fold cross validation ensured that each of our ten data sets appeared in the testing set exactly once.

6. Results

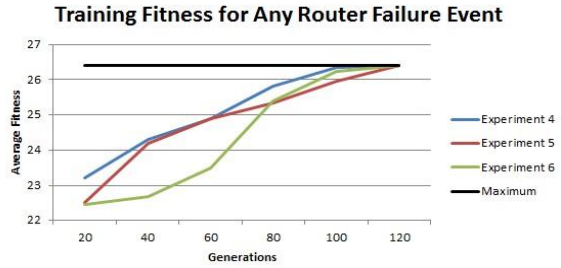
Fig. 3 shows the average maximum fitness values across generations during training.

At 5000 evaluations, the populations converge to maximum values in nearly all of the experiments. Determining the failure of a FEP from a tap near a different FEP was ultimately no better than a random search, and the training fitness does not converge to the possible maximum.

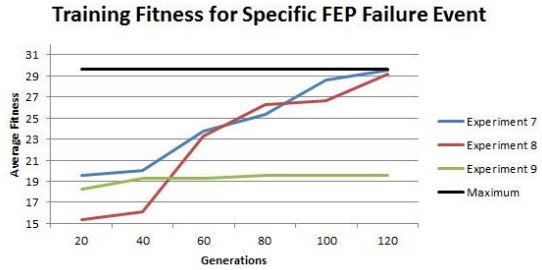
When using the best solution in to identify the testing data, the average accuracy was 100% for the majority of the experiments. Experiments 5 and 6 had an average accuracy of just over 98%. Fig. 4 shows a histogram of the average accuracies across all runs in all data splits.



(a) For a specific router failure event



(b) For any router failure event



(c) For a specific FEP failure event

Figure 3. Avg fitness vs generations for each event

Fig. 5(a) and Fig. 5(b) show sample overfitting plots for two data splits in experiments 5 and 6, respectively.

In supervised learning, when parameters are not appropriately tuned, there is always a danger of overfitting. In our case, the evolutionary process may bias the results heavily in favor of the training set, such that the resultant solutions are too specialized to correctly identify new data sets.

These plots were generated by verifying the accuracy of the best solution against the testing set at fixed generation intervals. If overfitting occurs, the average accuracy drops in later generations. Fig. 5(a) is an example of this. Note that the average accuracy shown is a representation of the accuracy in identifying the testing set only. Accuracy in identifying the training set was omitted. While overfitting is something that should be avoided, it does not have a large impact on the overall average accuracy of our solutions. The trade-off is avoiding parameter tuning for slightly worse results.

We believe not performing parameter tuning to be a more realistic approach, as practitioners likely will not have enough machine learning expertise to adjust the parameters effectively.

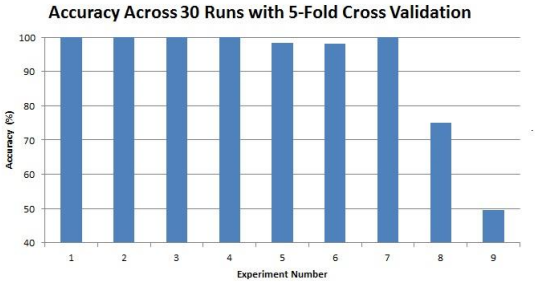
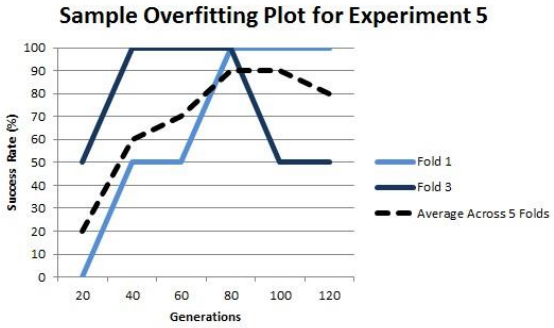
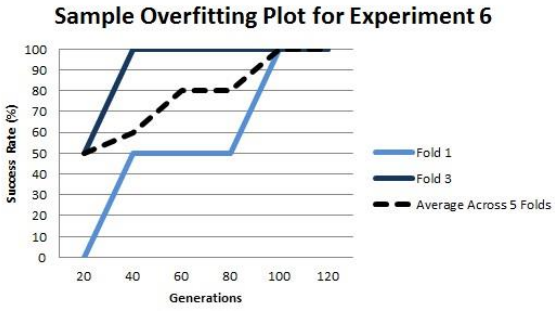


Figure 4. Accuracy per experiment, averages over all runs and cross validations



(a) Sample overfitting plot for experiment 5



(b) Sample overfitting plot for experiment 6

Figure 5. Representative overfitting plots of for experiments 5 and 6

7. Discussion

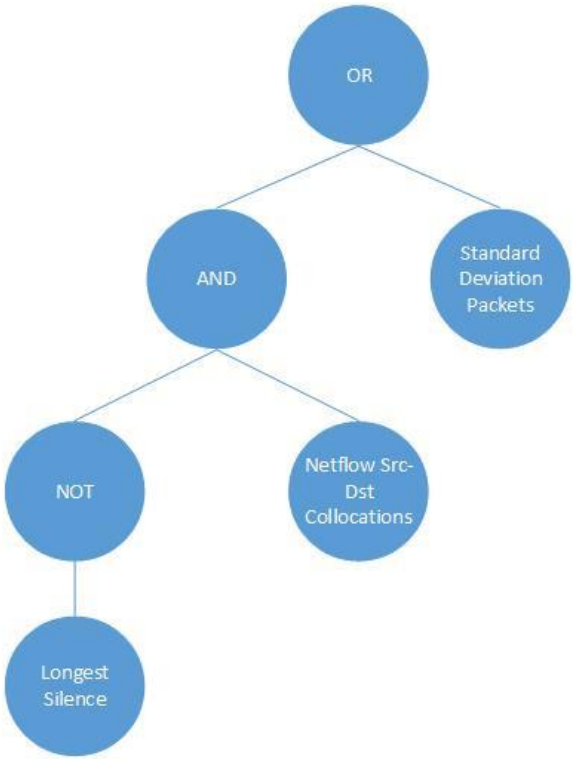
Most of the accuracies are perfect, highlighting the importance of using machine learning to identify known events over attempting to discover new useful outliers. The recurrent nature of ICS networks was a vital contributing factor in our solution's success. The poor accuracy on experiments 8 and 9 was a result of inadequate information. The taps in those particular locations in conjunction with our feature extraction algorithms did not provide enough information to discern events from non-events. It is obvious that collecting data near the source of the event allows for greater insight into what occurred. While it is still possible to glean some information

many hops away, it requires nuanced features. The types of data we collected is intended to be a worst-case scenario, with multiple devices going down, several coming back, or even a single router fluctuating between failing and operating normally. If a failure actually occurred, it would likely be easier to detect than the data that we used. The sensor trees had no problems identifying events in data sets where a lone failure occurred with no subsequent normal operations. The toughest data sets were those in which multiple routers failed because the FEPs increased their communications, and vital paths through the network were severed.

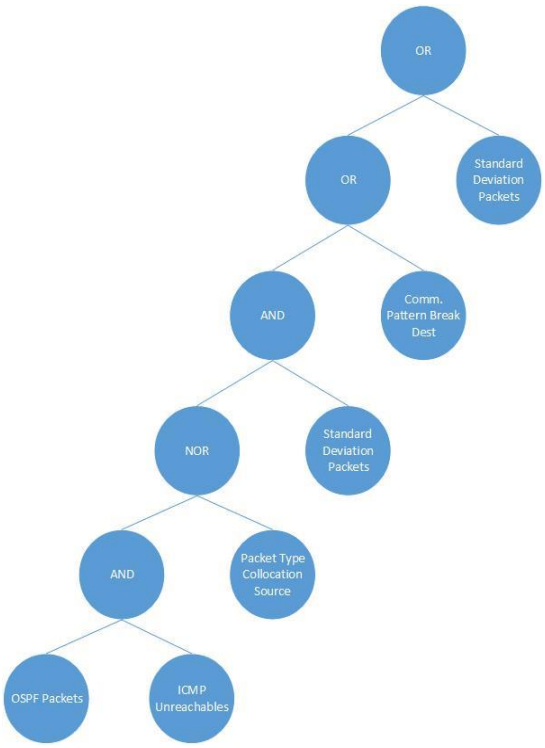
Throughout the testing process, we generated thousands of sensor trees. Fig. 6 shows two example trees for experiment 2. The trees illustrate the variety that can be achieved while maintaining a high accuracy. The tree in Fig. 6(a) will indicate that the router has failed if the device's longest period of silence is greater than the number of collocation anomalies, or if the standard deviation of packets increases immensely. This makes logical sense with the type of event the tree is detecting. Similarly, the tree in Fig. 6(b) makes use of a number our features to produce a more complicated, but likely more versatile tree. The great difference in tree structure is due to the solutions belonging to different data splits. The training set used to produce Fig. 6(b) was more difficult to optimize against.

In order to deploy this approach as a viable semi-real-time solution on a live system, it would be best to train on data capturing both the event and non-events, as we did for our experiments. Providing both types of data prevents over-specialization to one single data type and forces pattern recognition over outlier detection when deployed [3]. The sensor should also be placed as close to the event as possible, and a buffer size of twenty minutes should be used for optimal results. For more complete coverage and fewer false positives or false negatives, several different evolved decision trees should be placed at every location, and their consensus should be used to determine anomalies. If multiple varied trees agree that an event happened, it is more likely than if one tree returns a positive result. Because the trees are generated ahead of time, evaluation is nearly instant once the data buffer has been filled.

It is theoretically possible to detect more varied events than those explored in this paper, even if they are much more subtle. Rather than detecting strictly failures, the trees could be trained on data where devices react abnormally. If a firmware or logic update needs to be detected, this approach would likely perform very well given enough training data and the right feature selection algorithms. The results we achieved would not be nearly as positive if the features were not as diverse. The rich feature set is key to capturing the many possibilities that are associated with a network anomaly. If our feature set was less diverse, our accuracy would have been lower.



(a) A simple decision tree



(b) A more complex decision tree

Figure 6. Sample decision trees for experiment 2

8. Conclusion

Current capabilities in anomaly detection through the use of machine learning techniques have been limited. While some encouraging research has been done in the area, the fundamental goals of the previous work made the machine learning techniques not as effective as they could be. Rather than attempting to detect new attacks, our aim was to identify the occurrence of particular, known events. Specifically, we wanted to detect network failures in recurrent ICS networks. The periodicity of these networks allows for simple pattern recognition and meaningful feature extraction. We developed a GP approach to evolve decision trees for the purpose of relating the extracted features and identifying the network failures. The trees were tested in multiple locations on a large-scale virtualized network. Five of our nine experiments resulted in 100% average accuracy across a statistically significant number of runs. Two others achieved over 98% average accuracy. The experiments that had poor results suffered from a lack of available data.

Future work will attempt to build on our successes. Rather than virtualizing the network, we intend to include real ICS hardware in the loop. We will also add a diversity of ICS communication protocols. We would also like to detect more varied events, such as data exfiltration or PLC firmware updates. To succeed in these endeavors, we will need to include an updated feature set. The features used in the network failure experiments may not all be meaningful discriminators in future experiments.

9. References

[1] C. Y. Teo. “Machine learning and knowledge building for fault diagnosis in distribution network”. In *Electrical Power & Energy Systems*, pages 119–122, Apr. 1995.

[2] C.-F. Tsai, Y.-F. Hsu, C.-Y. Lin, and W.-Y. Lin. “Intrusion detection by machine learning: A review”. In *Expert Systems with Applications*, pages 11994–12000, 2009.

[3] R. Sommer and V. Paxson. Outside the Closed World: On “Using Machine Learning For Network Intrusion Detection”. In *IEEE Symposium on Security and Privacy*, SP 2010, pages 305–316, Mar. 2010.

[4] M. Mantere, M. Sailio, and S. Noponen. “Network Traffic Features for Anomaly Detection in Specific Industrial Control System Network”. In *Future Internet*, pages 460–473, Sept. 2013.

[5] M. Mantere, M. Sailio, and S. Noponen. “A Module for Anomaly Detection in ICS Networks”. In *Proceedings of the 3rd international conference on High confidence networked systems*, HiCoNS ’14, pages 49–56, Apr. 2014.

[6] M. Mantere, I. Uusitalo, M. Sailio, and S. Noponen. “Challenges of Machine Learning Based Monitoring for Industrial Control System Networks”. In *26th International Conference on Advanced Information Networking and Applications Workshops*, WAINA ’12, pages 968–972, Mar. 2012.

[7] L. A. Maglaras and J. Jiang. “Intrusion Detection in SCADA systems using Machine Learning Techniques”. In *Science and Information Conference*, SAI 2014, pages 626–631, 2014.

[8] J. R. Koza. “Overview of Genetic Programming”. In *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, pages 74–78. MIT PRESS, Cambridge, MA USA, 1992.

[9] J. Hosic, D. R. Tauritz, and S. A. Mulder. “Evolving Decision Trees for the Categorization of Software”. In *IEEE 38th International Computer Software and Applications Conference Workshops*, COMPSACW, pages 337–442, July 2014.

[10] J. Sun and X.-Z. Wang. “An initial comparison on noise resisting between crisp and fuzzy decision trees”. In *Proceedings of 2005 International Conference on Machine Learning and Cybernetics*, volume 4, pages 2545–2550, Aug. 2005.

[11] Jose L. Ribeiro Filho, Philip C. Treleaven, and Cesare Alippi. “Genetic-Algorithm Programming Environments”. *Computer*, pages 28-43, June 1994.

[12] Mario Bkassiny, Yang Li, and Sudharman K. Jayaweera. “A Survey on Machine-Learning Techniques in Cognitive Radios”. *IEEE Communications Surveys & Tutorials*, pages 1136-1159, July 2013.

[13] Jih-Yiing Lin and Ying-Ping Chen. “On the Effect of Population Size and Selection Mechanism from the Viewpoint of Collaboration between Exploration and Exploitation”. In *2013 IEEE Workshop on Memetic Computing*, MC, pages 16-23, April 2013.

[14] E. K. Burke, M. R. Hyde, G. Kendall, and J. R. Woodward. “A Genetic Programming Hyper-Heuristic Approach for Evolving 2-D Strip Packing Heuristics”. *IEEE Transactions on Evolutionary Computation*, 14(6):942–958, Dec. 2010.

[15] E. K. Burke, M. R. Hyde, G. Kendall, and J. R. Woodward. “Automatic Heuristic Generation with Genetic Programming: Evolving a Jack-of-all-Trades or a Master of One”. In *Proceedings of the 9th annual conference on Genetic and Evolutionary Computation*, GECCO ’07, pages 1559–1565, 2007.

[16] E. K. Burke, M. R. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and J. R. Woodward. “Exploring Hyper-heuristic Methodologies with Genetic Programming”. In *Computational Intelligence: Collaboration, Fusion and Emergence*, pages 177–201. Springer, Berlin-Heidelberg, Germany, Mar. 2009.

[17] M. Bader-El-Den, R. Poli, and S. Fatima. “Evolving timetabling heuristics using a grammar-based genetic programming hyper-heuristic framework”. *Memetic Computing*, 1(3):205–219, Oct. 2009.

[18] C. Yin, S. Tian, H. Huang, and J. He. “Applying Genetic Programming to Evolve Learned Rules for Network Anomaly Detection”. In *Advances in Natural Computation*, ICNC 2005, pages 323–331, Aug. 2005.

[19] W. Lu and I. Traore. “Detecting New Forms of Network Intrusion UsingGenetic Programming”. In *Computational Intelligence*, pages 475–494, Aug. 2004.

[20] J.-F. Lin, S. Li, and Y. Cai. “A new collocation extraction method combining multiple association measures”. In *Proceedings of the Seventh International Conference on Machine Learning and Cybernetics*, volume 1, pages 12–17, July 2008.

