# SECURE-GEGELATI Always-On Intrusion Detection through GEGELATI Lightweight Tangled Program Graphs

Nicolas Sourbier, Karol Desnos, Thomas Guyet, Frédéric Majorczyk, Olivier Gesny, Maxime Pelcat

**Noname manuscript No.**
(will be inserted by the editor)

# SECURE-GEGELATI

## Always-On Intrusion Detection through GEGELATI Lightweight Tangled Program Graphs

**Nicolas Sourbier · Karol Desnos · Thomas Guyet · Frédéric Majorczyk · Olivier Gesny · Maxime Pelcat.**

Received: date / Accepted: date

**Abstract** The fast improvement of Machine-Learning (ML) methods gives rise to new attacks in Information System (IS). Simultaneously, ML also creates new opportunities for network intrusion detection. Early network intrusion detection is a valuable asset for IS security, as it fosters early deployment of countermeasures and reduces the impact of attacks on system availability.

This paper proposes and studies an anomaly-based Network Intrusion Detection System (NIDS) based on Tangled Program Graph (TPG) ML and called *Secure*-Gegelati. Secure-Gegelati learns how to detect intrusions from IS-produced traces and is optimised to fit the requirements of intrusion detection. The study evaluates the capacity of *Secure*-Gegelati to act as a continuously learning, real-time, and low energy NIDS when executed in an embedded network probe. We show that a TPG is capable of switching between training and inference phases, new training phases enriching the probe knowledge with limited degradation of previous intrusion detection capabilities. The Secure-Gegelati software reaches 8× the energy efficiency of an optimised Random Forests (RF)-based Intrusion Detection System (IDS) on the same platform. It is capable of processing 13.2 $k$ connections/seconds with a peak power of less than 3.3 *Watts* on an embedded platform, and is processing in real-time the CIC-IDS 2017 dataset while detecting 84% of intrusions and raising less than 0.2% of false alarms.

**Keywords** Tangled Program Graphs Intelligence · Network Intrusion Detection · Cyber Security · Network Security · Real-time Processing

N. Sourbier
Univ Rennes, INSA Rennes, IETR, UMR CNRS 6164
20 Av. des buttes de Coësmes, France
Tel.: +332 23 23 90 22
E-mail: nicolas.sourbier@insa-rennes.fr

K. Desnos
Univ Rennes, INSA Rennes, IETR, UMR CNRS 6164
20 Av. des buttes de Coësmes, France
Tel.: +332 23 23 88 40
E-mail: karol.desnos@insa-rennes.fr

T. Guyet
IRISA - LACODAM
Unité pédagogique Informatique d'AGROCAMPUS-OUEST,
65 rue de Saint-Brieuc, Rennes, France
Tel.: +332 99 84 75 94
E-mail: thomas.guyet@irisa.fr

O. Gesny
SILICOM
3 E rue de Paris, 35510 Cesson-Sévigné, France
Tel.: +332 99 84 17 17
E-mail: ogesny@silicom.fr

F. Majorczyk
DGA-MI - CIDRE
136 La Roche Marguerite, 35170 Bruz, France
Tel.: +332 99 42 90 11
E-mail: frederic.majorczyk@supelec.fr

M. Pelcat
Univ Rennes, INSA Rennes, IETR, UMR CNRS 6164
20 Av. des buttes de Coësmes, France
Tel.: +332 23 23 86 49
E-mail: maxime.pelcat@insa-rennes.fr

## 1 Introduction

Intrusion detection consists of spotting the actions of attackers attempting to compromise the integrity, confidentiality, or availability of a computer resource [53]. The first Intrusion Detection System (IDS) was proposed by D. Denning in 1987 [7] as a way to early detect and prevent networking attacks and deviant behaviours. Intrusion detection is now used at a large scale

and is necessary to ensure confidentiality, integrity and availability of a resource as attackers proved their ability to bypass the protections of the resources [31,45, 35].

To enhance the security of a network, connections need to be analysed for countermeasures to be deployed. In a realistic information system context, the connection rate is high, making it impossible to be analysed by a human analyst in real-time. Thus, Network Intrusion Detection System (NIDS) have been created to facilitate the tasks of the analyst. A real-life network is dynamic: the traffic constantly evolves with the appearance of new features, services, operating systems, network topology etc. In the meantime, while the network is evolving, cyber attacks become more complex [31,35, 45]. NIDSs work under the assumption that attacks requests share a common basis and similarities. Anomaly-based Intrusion Detection Systems (AIDSs) aim to produce a model of the network normal using and to detect users' behaviours deviating from this model. In this context, AIDS are useful for their capacity to detect novelty [27,32,21]. Furthermore, AIDS can be used to create new attack *signatures*. Challenges for AIDS lie in the diminution of the false-positive alert rate and the production of ancillary information when an alert is raised. Indeed, false alarms are usually very time consuming and can easily cause a detection system to be rejected by analysts. Furthermore, AIDS are based on statistical inference and require an (often costly) initial training that hinders system adaptation. Finally, encrypted data packets can bypass the IDS and prevent an attacker from being detected.

From the difficulties that are faced by current AIDS, Reinforcement-Learning (RL) seems to address the detection problem in an interesting way. Indeed, the RL learning being conditioned by a reward function can prevent the RL agent from raising too many false-positive alerts. The reward system can be designed by an analyst and reward the teams that does not produce false positive alerts. The Tangled Program Graph (TPG) intelligence, as designed by Stephen Kelly [19], is a Multi-Agent Reinforcement Learning (MARL) algorithm based on Genetic Programming (GP) showing interesting properties of emergence. This emergence results in an interaction between a set of agents and the environment where the agents observe the environment and get more complex (tangled) in their response to observations. We also hypothesise that the properties of the TPG can also bypass some of the current difficulties of AIDS. Firstly, the TPG can train continuously and complex data to action behaviours have been demonstrated to emerge from its training, making it adapt over time to a dynamic environment. This online training also limits the offline training time required to obtain a correct model and the need of massive training data as it learns on incoming data without the necessity to store them. Finally, the TPG can be used as a classifier, raising alerts and giving classification information if needed. The current study focuses on the detection of attacks and thus, their classification is kept out of its scope. For making our study more realistic, we use as training data the network flow from the CIC-IDS 2017 data-set, and not the raw packet data (PCAP format) that would not be available and encrypted in a real setup.

Recent studies [46,52] propose to use Random Forests (RF) supervised classifiers to detect intrusions in the CIC-IDS 2017 IDS dataset [42]. RF can detect intrusions with an accuracy over 90%. However, RFs do not support continual learning, as they need to be retrained from scratch with a considerable amount of labelled data for incorporating a new attack class. Furthermore, the performances of the obtained models degrade in the first months after training and can drop by up to 23% within a year [48] due to the changes occurring on the network and to the novel attacks. K-Nearest Neighbours (k-NN) [38,2,9] have also been used for intrusion detection but face the same issue. A long time is required to make a model converge towards a good prediction rate.

The built and studied Secure-GEGELATI real-time monitoring system aims at progressively automating the detection of intrusions in an Information System (IS) by observing the incoming data. It constitutes a Security Information and Event Management (SIEM) device which helps a security analyst to early detect an attack. The monitoring system is required to have nearly perfect precision, potentially trading it off for a low recall. Indeed, false alarms are extremely time consuming, as they bring analysts to finally uncover that no attack was currently performed. Conversely, missing intrusions is less costly, as it brings the security analyst back to the situation where no live monitoring system was present. To be useful in practice, a live IDS monitoring system must:

- keep up with the pace of the IS incoming data,
- be embeddable in an always-on device, and thus energy efficient,
- raise extremely rare false alarms in a context where attacks are very rare events.

This paper is an extension of the conference paper [8] introducing the GEGELATI TPG as a customisable, scalable and deterministic TPG system. It proposes the following contributions:

- we study the GEGELATI TPG learning capacities and energy efficiency on a realistic and complex application,
- we demonstrate that the properties of TPG are well suited for building a learning-based NIDS. In particular, its energy efficiency and continual learning capacities on new incoming attack flows are studied,
- we introduce the open source Secure-Gegelati Tangled Program Graph (TPG) system for learning-based intrusion detection.

The TPG learning method has a lightweight structure, its training and inferring processes are known to be fast and adequate for online training [18]. This lightweight structure helps for online training on an embedded platform. The considered scenario is the following:

- Training phase: the analyst provides tagged data to train the real-time monitoring system, specifying whether an attack is ongoing or not.
- Monitoring phase: the live monitoring system is switched into a monitoring mode, continuously observes the IS and triggers an alarm when an attack occurs. With RL, the analyst can switch back the monitoring device into a training phase at any time and improve its sensitivity to new attacks.

Section 2 introduces state of the art methods for intrusion detection and situates Secure-GEGELATI among them. In Section 3, the TPG algorithm is defined and described. Section 4 presents the Generic Evolvable Graphs for Efficient Learning of Artificial Tangled Intelligence (Gegelati) TPG system and its deterministic and parallel execution properties. We explain in section 5 how the TPG can be adapted into a NIDS real-time probe. Finally, in section 6, we compare the performance of Secure-Gegelati with an optimised RF parallel NIDS in terms of precision, real-time processing and energy efficiency.

## 2 Related Work

Network security is one of the main cyber-security fields. We focus in this paper on the network intrusion detection problem that consists in detecting malicious usage of an IS network [47]. Several approaches to this problem exist. While network intrusion prevention systems take real-time countermeasures when a malicious behaviour is detected, NIDS aim to assist the security analyst by raising alerts. The three main types of IDS [6,21,27] are signature-based systems, anomaly detection systems and stateful protocol analysis.

Stateful protocol analysis systems detect deviations of protocol states and use predetermined universal profiles based on "accepted definitions of benign activity" developed by vendors and industry leaders [39]. It is mainly used for its ability to check reasonable thresholds for individual commands (min, max, length...) and makes it possible to identify as well unsuspected sequences of commands.

Signature based (or misuse) intrusion detection systems filter the network frames using human-defined signatures of the threats generally using regular expressions[11,14,25]. The main drawback of signature based detection is the high number of patterns that need to be handcrafted, stored and analysed, as each attack has a unique signature.

For their part, anomaly detection systems aim at producing a model of the normal behaviour and to detect behaviours deviating from this model. Nonetheless, anomaly detection is a difficult task as attackers often make changes to already known attacks to evade this particular detection technique. For instance, in the first quarter of 2017, more than 55.000 attack variations were discovered for only 15 attack families.

Stateful protocol analysis systems and anomaly detection systems identify deviations from a model of normal behaviour benign activity. Anomaly-based IDS use statistical inference with either unsupervised learning or supervised learning.

One of the main problems in NIDS is the scarcity of high quality labelled data-sets. This is why unsupervised learning is widely used on the problem. Amongst unsupervised traditional learning algorithms, the K-means algorithm usually produces the best results[2, 9,38]. As stated in [38], their exists a trade-off between a high accuracy using unsupervised learning and a time-efficient, low complexity model. While unsupervised learning is convenient due to its ability to train in real-time and to use unlabelled data, its performances are hindered by high false-positive rates. For instance, one can observe 21.8% of false-positives in [2] and more than 15% of false-positives for 20000 connections analysis in [9].

These solutions are impractical, as false positives are a consistent problem of NIDS. An analyst is able to study between 1 and 20 threats per day, making it important to focus on real threats and not on false positive alerts. Eventually, a NIDS that regularly generates false positive alerts will decrease the confidence of the analyst in the IDS and reducing recall to improve precision in this context makes a lot of sense.

Supervised learning methods have been used to create AIDS. These methods are trained offline based on labelled network packets or flow. A network packet is a

formatted unit of data containing control information and payload. Network flows sums up packet information such as the protocol or the type of service for given source and destination IP and ports. They have low robustness to network modifications such as any modification of the topology of the network, the appearance of new services, or novel threats. Indeed, experimental results in [48] show that an Machine-Learning (ML) based IDS trained at the beginning of a year can have an accuracy drop of up to 23% by the end of the year. These issues are discussed in [20, 22].

RL [15] is an alternative to both supervised and unsupervised learning. RL is the problem faced by an agent that learns behaviour through trial-and-errors interactions with a complex and dynamic environment. The actions of the RL agent have an impact on the environment. The actions learnt as reactions to a set of specific states of the environment is called the policy. One of the RL challenges is the exploration versus exploitation dilemma management where exploitation means that a learnt policy is used to infer the future reward in the observed environment. Using this feature, a RL method is able to keep including novelty in its policy and exploitation is particularly interesting for the design of an IDS constantly facing new threats.

We address the intrusion detection problem using an RL algorithm based on GP. RL for classification differs from the supervised learning methods in the amount of supervision required in the learning process. Supervised learning method require an explicit correction through class information while RL can learn from a batch of implicit corrections called "rewards".

GP has been used to assist the creation of both signature based IDS [10, 26, 34] and anomaly-based IDS [43]. Particularly, authors of [43] used their "GANIDS" architecture to decrease the amount of false positive alerts, making the IDS qualitative for the analysts. They showed that the use of GP helped them in reducing the amount of false positive while keeping high detection rates.

Furthermore, some RL frameworks exist that perform intrusion detection [5]. Several studies focus on the use of Deep reinforcement learning (Deep Q Networks) [23, 28, 41] and fewer consider using MARL to design their IDS [40]. The later method takes advantage of MARL scalability, to observe a distributed environment and solve the interactive problem of intrusion detection over DoS and DDoS attacks. A limitation of Deep Q Networks is that they remain heavily parameterized and energy costly methods.

As an IDS is an always-on application or device, it is interesting for the sustainability of the solution to set it up as a High-Performance Embedded Computing (HPEC) device [50] in the form of a network probe. Several studies pointed out this need [29, 30, 44, 49]. In particular, Viegas et al., [49] show how using an embedded device implementing Decision Trees, Naive-Bayes or K-NN algorithm allowed them to save up to 93% of energy and pointed out that the Decision tree is the most energy-efficient algorithm among the compared ones. As specified in [29], using an embedded IDS represents a trade-off between energy consumption and peak number of analysis per second. Indeed, the use of a neural network here dropped the load of the IDS to 6MBps.

In this paper, we study the capacity of TPG, a MARL framework proposed by S. Kelly [16] which shows interesting properties in terms of emergence and lightweight training. The TPG being fast and lightweight makes it a good candidate for a real-time embedded NIDS probe. TPG combines RL and GP into a multi-agent directed graph. RL relates the learning mechanisms of the policy strategy to action decided by the TPG agent. GP improves the learning mechanism with the emergence of new environmental observation rules. The TPG is composed of teams (vertices), programs (edges) and actions (vertices). The TPG programs observe the environment and select the best path in the graph until an action is reached.

Unlike Deep Reinforcement Learning methods which observe the environment in its entirety, the TPG agents select through trial and errors the relevant information for decision making.

Although it is based on the TPG intelligence, Secure-Gegelati differs from Reinforcement Learning in its conceptual learning approach for two reasons:

- Reinforcement Learning is normally based on sequential or stateful analysis were the current state of the environment depends on its previous states. This can hardly be simulated as the networks packets arrive from different sources.
- Reinforcement Learning states depend on the actions taken on the environment. Here, the TPG actions are classification actions and thus, do not modify the environment characteristics.

The RF algorithm, which is an extension of the Decision tree algorithm, is taken as a baseline. [49] stated that Decision trees are the most energy-efficient algorithm among the compared algorithms making Random Forests a relevant candidate. Furthermore, in [42], authors show that RF is one of the algorithms that perform best on the CICIDS-2017 dataset with a precision of 98%, a recall of 97% and F1-score of 97%.

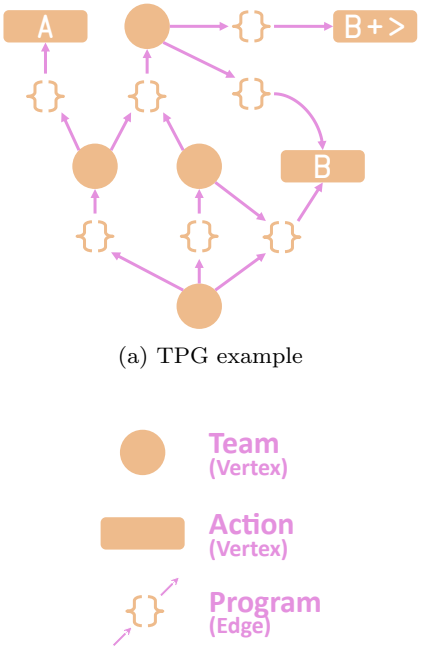
# 3 Tangled Program Graphs

The TPG model studied in this paper, which builds on technique from the genetic programming domain, was introduced by Kelly and Heywood [19] as a reinforcement learning technique. Principles of reinforcement learning and genetic programming are presented in Section 3.1, and the TPG model is detailed in Section 3.2.

## 3.1 Background: Reinforcement Learning and Genetic Programming

**Reinforcement learning** is a branch of machine learning techniques where artificial intelligence learns, through trial and error, how to interact with an environment. In reinforcement learning, artificial intelligence, called the learning agent, observes the current state of its learning environment, and interacts with it trough a finite set of actions. As a result of these actions, or because of external phenomena such as time or physics, the state of the learning environment evolves. By observing the constantly evolving state of the environment, the learning agent has the possibility to react and to build a meaningful sequence of actions. For the agent to learn which sequences of actions are useful, an additional reward mechanism is implemented. By rewarding useful behaviour of the learning agent, and penalising harmful or useless behaviour, this reward mechanism helps the learning agent select the most appropriate behaviour for each new experience. Although TPG have originally been developed for reinforcement learning purposes, the possibility to adapt them for other kinds of learning environments has already been demonstrated [18].

**Genetic programming** is a subset of machine learning techniques that mimics a natural selection evolution process to breed programs for a selected purpose. The iterative learning process of genetic programming can be summarised in four steps: 1/ Create an initial population of $n \in \mathbb{N}^*$ random programs. Then, iteratively: 2/ Evaluate the fitness of these programs against the learning environment. 3/ Discard the $m < n, m \in \mathbb{N}^*$ programs of the population with the worse fitness. 4/ Recreate $m$ new programs from remaining programs by using genetic operations, like mutations or crossovers. As detailed in [19,16], TPG add a compositional mechanism to this genetic learning process, which favours the emergence of stable clusters of useful programs by building a hierarchical decision structure.




(a) TPG example

(b) TPG semantics

Fig. 1: Semantics of the Tangled Program Graph (TPG)

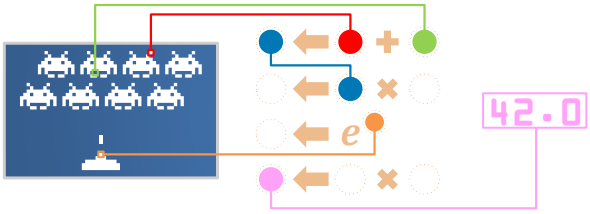



Fig. 2: *Program* from a TPG. On the left, the learning environment state fed to the *program*. In the middle, the sequence of instructions of the *program*. On the right, the result produced by the program.

## 3.2 TPG: Model and Learning Algorithm

**The semantics of the Tangled Program Graph (TPG) model,** depicted in figure 1, consists of three elements composing a direct graph: *programs*, *teams* and *actions*. The *teams* and the *actions* are the vertices of the graph, *teams* being internal vertices, and *actions* being the leaves of the graph. The *programs*, associated to the edges of the graph that each connects a source *team* to a destination *team* or *action* vertex. Self-loops, that is an edge connecting a *team* to itself, are not allowed in TPG.

From afar, a *program* can be seen as a black box that takes the current state of the learning environment as an input, processes it, and produces a real number,

called a *bid*, as a result. In more detail, a *program* is a sequence of simple arithmetic *instructions*, like additions or exponents. As depicted in figure 2, each *instruction* takes as an operand either data coming from the observed learning environment, or the value stored in a register by a previous *instruction*. The last value stored in a specific register, generally called `R0`, is the result produced by the *program*.

**The execution of a TPG** starts from its unique root *team*, when a new state of the environment becomes available. All *programs* associated to outgoing edges of the root *team* are executed with the current state of the environment as their input. Once all *programs* have completed their execution, the edge associated to the largest *bid* is identified, and the execution of the TPG continues following this edge. If another *team* is pointed by this edge, its outgoing *programs* are executed, still with the same input state, and the execution continues along the edge with the largest *bid*[1]. Eventually, the edge with the largest *bid* leads to an *action* vertex. In this case, the *action* is executed by the learning agent, a new resulting state of the environment is received, and the TPG execution restarts from its root *team*.

**The genetic evolution process of a TPG** relies on a graph with several root *teams*. The initial TPG created for the first generation only contains root *teams* whose outgoing edges each lead directly to an *action* vertex. At a given generation of the learning process, each root *team* of the TPG represents a different *policy* whose fitness is evaluated. Evaluating a root *team* consists of executing the TPG stemming from it a fixed number of times, or until a terminal state of the learning environment is reached, like a game-over in a video game. The rewards obtained after evaluating each root *team* of the TPG are used by the genetic evolution process. Worst-fitting root *teams*, which obtained the lowest rewards, are deleted from the TPG.

To create new root *teams* for the next generation of the evolution process, randomly selected remaining *teams* from the TPG are duplicated with all their outgoing edges. Then, these new edges undergo a random mutation process, possibly altering their destination vertex, and modifying their *programs* by adding, removing, swapping, and changing their instructions and operands. Surviving root *teams* from previous generations may become the destination of an edge added during the mutation process, thus becoming internal vertices of the TPG. This mutation mechanism favours the emergence of long-living valuable sub-graphs of connected *teams*. Indeed, useful *teams* contributing to higher rewards have a greater chance of becoming internal vertices of the TPG which can not be discarded unless they become root *teams* again. Hence, complexity is added to the TPG adaptively, only if this complexity leads to better rewards for the learning agent. A detailed description of this evolution process can be found in [16].

**The capabilities of TPG** have been extensively demonstrated [19,16] on the 55 video games from the Arcade Learning Environment (ALE) [4]. In this learning environment, the adaptive complexity leads to TPG with diverse sizes, depending on the complexity of the strategies developed to play each game. For example, there are two orders of magnitude between the smallest and largest networks built within these learning environments. On the performance side, TPG have been shown to reach a level of competency comparable with state-of-the-art deep-learning techniques on ALE games, for a fraction of their computational and storage cost. Compared to state-of-the-art techniques, TPG reach comparable competency with one to three orders of magnitude less computations, and two to ten orders of magnitude less memory needed to store their inference model. Recently, an extension of the TPG model supporting continuous action space was proposed in order to target new learning environments, like time-series predictions [18].

**Implementations of learning frameworks for TPG**, coded in C++, Java and Python, can be found in open-source repositories. The main motivations behind the creation of the Gegelati library is to have an efficient, embeddable, portable, parallel and deterministic library. Because of the efficiency and embeddability objectives, C++ was a natural choice for the development of Gegelati. Previous open-source C++ implementations, including the reference C++ code from Kelly [19], were neither parallel nor deterministic.

# 4 Gegelati: Parallel, Efficient and Embeddable framework for TPGs

Gegelati is an open-source framework for training and executing TPG. From its inception, the Gegelati library has been conceived to foster its adaptability to diverse learning environments, and its portability to various architectures, without sacrificing its performance. To this purpose, two original contributions have be integrated to the library: the parallelization of the deterministic learning process, presented in Section 4.1; and the support for customisable *instructions*, detailed in Section 4.2.

[1] If a team is visited several times, previously taken edges are ignored to avoid infinite loops.

### 4.1 Deterministic Parallelism and Portability

*What are the motivations?*
Portability of the GEGELATI library enables using it both on general-purpose and embedded architectures. Indeed, when training a learning agent intended to run on an embedded system, a common design process is to prototype the agent first on a general-purpose processor before embedding it on the embedded target. The portability also makes it possible to train a learning agent offline on a high-performance computing architecture, before deploying it on a less performing architecture for inference.

Parallelism of the learning process is an essential feature to accelerate the training of new learning agents, which fosters the adoption of new machine learning techniques. Indeed, the breakthrough of deep-learning models is largely due to the acceleration of their training process with Multi-Processor System On Chips (GPUs) [24]. Support for parallel computations is useful for general-purpose and high-performance computing architectures, but also for embedded systems which nowadays widely integrate heterogeneous Multi-Processor System-on-Chips (MPSoC).

Determinism of a learning process is the property that ensures that given a set of initial conditions, the learning process will always end with the same result. Determinism can only be obtained under the assumption that the state of the learning environment is itself changing deterministically, solely depending on the sequence of actions applied to it. Determinism is a key feature, especially for pseudo-stochastic learning process such as the training of TPG. Indeed, the result of training may partially depend on luck, which is exactly why being able to deterministically reproduce a result is crucial.

The determinism is antagonistic with the parallelism and portability objectives, and with the stochastic nature of the learning process, which makes all these objectives challenging to implement jointly. Indeed, parallelism is by nature a source of non-determinism as the simultaneity of computations accessing and modifying shared resources, often in an unknown order, tends to produce variable results.

*How does the deterministic and scalable parallelism work?*
During the learning process of TPG, the most compute-intensive parts are the fitness evaluation of the *policies*, and the mutations of the *programs* added during the evolution process. The fitness evaluation of individual *policies* can be deterministically executed in parallel, on the conditions that: 1/ the learning environment can be cloned to evaluate several policies concurrently, and 2/ any stochastic evolution of the learning environment state can be controlled deterministically. Under these conditions, the parallel evaluation of *policies* is possible, as the topology of the TPG, which is a shared resource for all *policies*, is fixed during this evaluation process. Similarly, the mutation of *programs* can be applied deterministically in parallel. Two kinds of mutations are applied to the TPG: mutations affecting the graph topology by inserting new root *teams* and edges; and mutations affecting *instructions* of the *programs* associated with the new edges. While mutating the graph topology cannot be done in parallel, the graph being a shared resource, individual *programs* are independent from each other and can be mutated in parallel.

To control a stochastic process, a Pseudo-Random Number Generator (PRNG) must be used each time a random number is needed. Given an initial seed, a PRNG produces a deterministic sequence of numbers. To ensure full determinism of the training of a TPG, a unique PRNG should be called in a fixed order during the whole training. Letting the parallel parts of the training process call the PRNG directly is not possible, as the absolute order in which parallel computations occur is itself stochastic. It is also not possible to give a pre-computed list of pseudo-random numbers to each parallel task, as the number of random numbers needed for each task is itself stochastic. For example, when mutating a *program*, mutations are applied iteratively until the program behaviour becomes "original" compared to pre-existing *programs* in the TPG. Hence, giving a fixed number of pre-computed random numbers for the *program* mutations is not feasible.

The parallelization strategy adopted in GEGELATI is based on the master/worker principle, with a distributed PRNG. The principle of the distributed PRNG is the use of two distinct PRNG instances: the $prng_{master}$ and the $prng_{worker}$. The $prng_{master}$ is exclusively used in the sequential parts of the learning process, which confers a deterministic nature to its usage, given an initial seed. Besides being used for stochastic tasks performed sequentially, like TPG topology mutations for example, the $prng_{master}$ is also used to generate a seed for each parallel worker task. In each worker task, a private $prng_{worker}$ is instantiated, and initialised with the seed provided by the $prng_{master}$. Since all calls to the PRNG from the worker tasks exclusively use their private $prng_{worker}$, the random number sequences generated in each parallel task are deterministic.

The pseudo-code of the master and worker tasks for the policy fitness evaluation are presented in Procedures 1 and 2, respectively. Communications between the tasks and load balancing of the computa-

**Procedure 1:** EvaluateAllPolicies

```
   Input:  TPG: G = ⟨Teams, Edges⟩
   Data:   PRNG: prng_master
           Job queue: JobQ
           Result Queue: ResultQ
 1 /* Prepare jobs */
 2 idx = 0
 3 for each root ∈ G.Teams do
 4 |   seed = prng_master.getNumber()
 5 |   job = { idx++, seed, root }
 6 |   jobQ.push(job)
 7 endfor
 8 /* Start parallel threads */
 9 for i = 1 to Num_PE − 1 do
10 |   Spawn thread: Worker(G, JobQ, ResultQ)
11 end
12 Call Worker(G, JobQ, resultQ)
13 Join all threads
14 /* Post-Process Results and Trace */
15 Sort ResultQ in result.jobId order
16 for each result ∈ resultQ do
17 |   Post-process result.trace // Archiving [19]
18 |   ...
19 endfor
```

**Procedure 2:** Worker

```
   Input:  TPG: G = ⟨Teams, Edges⟩
           Job queue: JobQ
           Result queue: ResultQ
   Data:   PRNG: prng_worker
           Learning environment twin: LE
 1 /* Poll for job */
 2 while JobQ.hasJob() do
 3 |   /* Setup for policy evaluation */
 4 |   job = jobQ.getNextJob()
 5 |   root = job.root
 6 |   prng_worker.reset(job.seed)
 7 |   LE.reset(prng_worker.getNumber())
 8 |   /* Evaluate policy fitness */
 9 |   trace = evaluate(G, root, LE, prng_worker)
10 |   result.jobId = job.id
11 |   result.trace = trace
12 |   resultQ.push(result)
13 end
```

tions are supported by a job queuing mechanism based on two queues: *JobQ* and *ResultQ*. Each policy evaluation job, prepared by the master procedure, encapsulates a unique job identifier `id`, a seed provided by the $prng_{master}$, and a root *team* from the TPG. All jobs are pushed in the *JobQ* queue before spawning as many worker threads as the number of secondary Processing Elements (PEs) in the target architecture. For each job it acquires from the *jobQ* queue, the worker procedure resets its $prng_{worker}$ using the seed contained in the job. Before evaluating the fitness of the root *team* contained in a job, the worker procedure resets its private copy of the learning environment, using a number given by the $prng_{worker}$. As a result of the policy fitness evaluation, described in details in [19], a result object encapsulating execution traces for the job is pushed in the *resultQ*. When all jobs have been processed, and all workers terminated, the master procedure is responsible for post-processing the traces stored in the *resultQ*. To ensure determinism of this post-processing, results stored in the *resultQ* are first sorted in ascending `job.id` order.

The master and worker procedures used for parallelising the mutations of *programs* are similar to the one used for policy fitness evaluation, with the difference that jobs encapsulate *programs* instead of root *teams*.

## 4.2 customisable Instruction Set

*What are the motivations?* In the seminal work on TPG [19], the *instructions* used in the *programs* are chosen exclusively among the following eight instructions: 4 binary operators $\{+, -, \times, \div\}$, 3 mathematical functions $\{cos, ln, exp\}$, and 1 conditional statement $res \leftarrow (a < b)? - a : a$. To further simplify the execution and mutation of *programs*, it was assumed that instructions only handle `double` operands.

As shown in related genetic programming works [3, 36], using a broader set of instructions with diverse data types can help improve the performance of learning agents, at the cost of longer training time. The extension of the instruction set used in the *programs* of the TPG has already been proposed in [17], where a set of instructions for 2D images operands is added, and in [12], with instructions accepting thirteen operands tailored for predicting properties of the learning environment. In Gegelati, both the number and types of operands, and the nature of instructions used in *programs* can be fully customised. Besides making the training more efficient for specific learning environments, this customisation feature may also be used to increase the efficiency of the TPG execution on specific hardware. Indeed, using an instruction set mirroring the instruction set of the architecture used for its execution may help increase the speed and the power efficiency of the TPG execution.

*How are customisable instructions supported?*

The support for customisable instructions within Gegelati is based on the three classes presented in Figure 3. When creating a new training environment, a developer may create her own set of instructions, by creating new classes inheriting from the `Instruction` class. With the `operandTypes` attribute, each instruction declares the number and type of operands it accepts when calling its `execute()` method. Currently, to keep the management of registers simple during program execution, only `double` results can be produced

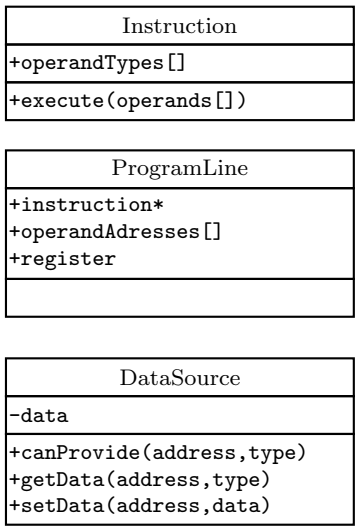



Fig. 3: Class diagrams of the data structures for customisable instructions.

**Procedure 3:** ExecuteProgram

```
   Input: Program: p
          Data sources: data
 1 for each line ∈ p do
 2 |   instruction = line.instruction
 3 |   operands[] = { ∅ }
 4 |   nbOperands = instruction.operandTypes.size()
 5 |   for i = 0 to nbOperands-1 do
 6 |   |   type = instruction.dataTypes[i]
 7 |   |   address = line.operandAdress[i]
 8 |   |   if data.canProvide(type, address) then
 9 |   |   |   operand = data.getData(type, address)
10 |   |   |   operands.insert(operand)
11 |   |   else
12 |   |   |   Exit with an error
13 |   |   end
14 |   end
15 |   result = instruction.execute(operands)
16 |   data.set(line.register,result)
17 endfor
```

by the `execute()` method. Each *line* of a *program* references an `instruction` from the set of available instructions, a destination `register` to store the result of its execution, and the addresses of operands to process, selected among all available data sources. The data sources accessible to the *lines* comprise both the registers used for storing instruction results, and the state of the learning environment. Data sources classes must inherit from the `DataSource` class which acts as a wrapper between the data and the *program* execution engine.

Procedure 3 presents the simplified pseudo-code for executing a *program* modelled with the classes from Figure 3. The core of the mechanism supporting customisable instructions lies between lines 5 and 14 of Procedure 3. For each operand of each *line*, the algorithm checks whether the data sources can provide the requested operand type at the requested address. If the data type can be provided by the data sources at the requested address, the data is fetched from the data sources, and later used for executing the `instruction` of the current `line` of the *program*. Otherwise, the program execution is terminated, which does not occur in practice, as the operand data types are taken into consideration when performing *program* mutations in GEGELATI. It is important to note that the `getData()` method may return data whose type differs from the native data type stored within the data source. For example, a data source storing screen pixels as `char` values can automatically return an equivalent `double` value, or even a neighbourhood of 3-by-3 pixels when an operand of type `char[3][3]` is requested.

```
auto myInstruction = LambdaInstruction<int,
    char[2]>(
  [](int a, char[2] b)->double {return a*(b
      [0] + b[1]);});
```

Listing 1: `LambdaInstruction` usage example

To ease the creation of new *instructions* for each training environment, a utility class `LambdaInstruction` is proposed in GEGELATI. The template class `LambdaInstruction` supports the creation of instructions for any number of operands, and for operands with primitive and non-primitive types as well as 1D and 2D C-Style arrays. A code snippet illustrating the creation of an instruction with the `LambdaInstruction` class is given in Listing 1. In this example, an instruction taking an `int` operand, and a 1D array of `char` is declared, using a simple C++ lambda function.

# 5 The Secure-Gegelati real-time prototype

Secure-GEGELATI is a TPG-based system that has been designed to perform intrusion detection on an IS.

In Section 5.1, we present the changes applied to TPGs to use them in an IDS. Section 5.2 details reinforcement learning in a NIDS context. Section 5.3 presents the embedded version of our system. Finally, we describe in Section 5.4 a practical use case example of the use of the probe.

## 5.1 An Anomaly-based Intrusion Detection System

Secure-Gegelati is a TPG system that has been adapted in the following ways :

1. It is tailored to perform an inference task. To build Secure-Gegelati, the TPG has been adapted to classification problems by increasing the probabilities of mutations of programs, and increasing the diversity of explored solutions. In particular, the mutations that cause the change of the outgoing edges to a different action are useful to discover samples from all the represented classes, including rare intrusion classes. After modification, Secure-Gegelati is able to create a model for rare events.
2. It is tailored to produce classification with low false-positive Rate (FPR) and continual learning. Originally, GEGELATI TPG is designed to solve RL challenges, i.e. to choose the actions to be applied to an environment so as to increase a reward. To formulate a classification problem as a RL challenge, the RL reward function must act as a loss function. A NIDS requires primarily a high precision to get a low false alarm rate, and secondarily a high recall to detect as many intrusions as possible. To obtain this behaviour, we chose to reward the agent with the F1-score function. This reward helped to minimise the FPR.

To help in the detection of intrusions, we added a few instructions to the original TPG instruction set ($\{+, -, *, /, max, log, exp\}$). The added instructions help in the identification of constant values such as 80 or 8080 for HTTP ports number. $max$ and $-$ are used to determine whether a state value is superior to another.

Secure-Gegelati faces three mains problems:

- learn in a highly imbalanced environment
- analyse network flow logs in real-time
- learn in a dynamic environment where new threats appear over time.

Intrusion detection is by nature a highly imbalanced problem because intrusions are rare. Secure-Gegelati is adapted for this problem of imbalanced learning by tuning the reward system. The rewards obtained when raising a correct alert is made artificially greater than the reward obtained when no alerts are rightfully raised. As a rule of thumb, we observe that the reward amount shall compensate for rare presence of the class in the dataset in a linear way. For example, the CICIDS-2017 dataset [42] is composed of 83% of benign network flow logs and 17% of attack network flow logs. In order to mitigate this problem, an alert raised on an attacks is rewarded 5 times ($83/17$) more than another action rightfully taken, making the class imbalance less problematic. As a rule of thumb, we observe that the reward amount shall compensate for rare presence of the class in the dataset in a linear way. Furthermore the ratio of deleted teams at the end of a generation has been set to 80%, less than in the original method, in order to keep knowledge of rarely occurring events by preserving enough root teams at the end of a generation.

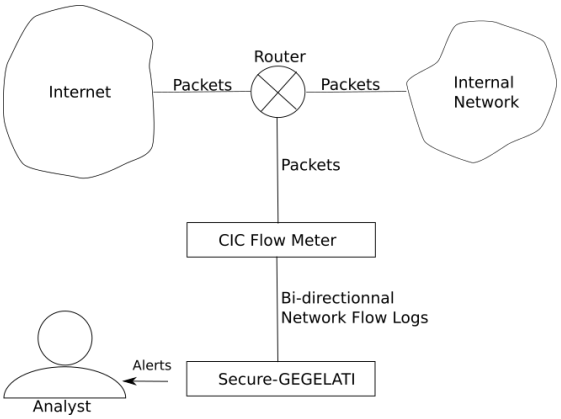



Fig. 4: In inferring Mode, Secure-Gegelati monitors Bi-directional Network flow logs (Network flows logs from both the request and the response) provided by the "CIC Flow meter" software from the raw packets logs. The analyst receives potential alerts.

To keep pace with the input data stream while training, we studied two training modes. The first one trains each root team with the same input stream of data and the second one stacks the network flow logs into a FIFO and different root teams unstack data through their learning. In the second training mode, each root team is trained with different data making the imbalanced data problem more important. The two methods will be compared in results sections on table 8.

Finally, in order to cope with changes on the network (see table 5) or with the appearance of novel threats (see table 6 and table 7), we altered the learning process in order to switch manually between training and inferring modes. This feature allows an analyst to update the probe when required. Figure 4 and 5 show the flow of data in both inferring and training mode.

## 5.2 A reinforcement learning-based probe

We designed the Secure-Gegelati system as a RL probe. RL is bound to three properties:

- taking actions on a dynamic environment

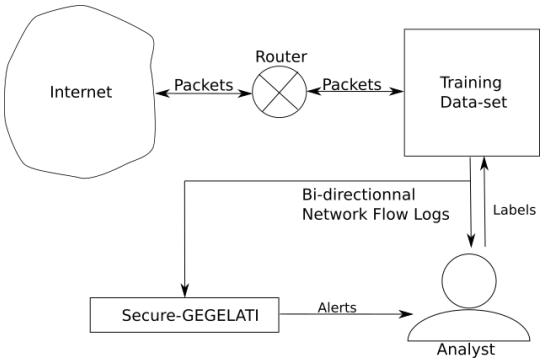



Fig. 5: When Secure-Gegelati is in training mode, it monitors network flows labelled by the analyst. The analyst labels this new training set of logs based on existing labels, expertise and other potential security mechanisms (such as signature-based ids) already set-up on the network. The new training logs are added to the previous training set. Secure-Gegelati itself continues to raise alerts while training.

- evaluating the internal model (inferring) and exploring of the action space (training)
- learning through rewards.

Even though Secure-Gegelati is trained from a labelled database, it cannot be considered a supervised learning system. Secure-Gegelati takes action on a dynamic environment, as we include the input data itself directly in the environment. When an action is taken (even though it is a classification action), the environment changes and a new line of net-flow log is made available, enabling the agent to make new observations and update its action policy. As the Secure-Gegelati system is a continual learning system, its evaluation of the internal model is kept low to the benefit of a high exploration allowing the agent to fit to the novelty appearing on the dynamic environment. New attacks and threats can thus be detected. Finally, Secure-Gegelati reduces the need for supervision of the learning environment by rewarding the agent through batches. There is no immediate correction of the learning as in supervised learning, but instead a delayed score that is given to each agent root team at the end of a batch, fostering the selection of the best root teams for the cloning and the mutations.

## 5.3 A real-time Embedded system

We use as a proof-of-concept target an octa-core Exynos 5410 SoC with four LITTLE ARM cores (A7) and four big ARM cores (A15), providing a set of 17 clock configurations from 200 MHz to 2.0 GHz for the A15 cores and from 200MHz to 1.4GHz for the A7 cores. The TPG graph being a highly dynamic and evolvable model, we choose to use this versatile platform and benefit from its high energy efficiency, optimises for running in handheld devices. This platform constitutes a portable baseline for future studies on hardware acceleration and accelerations of TPG programs on a SoC-FPGA is considered for future work.

Secure-Gegelati exploits the platform parallelism. As a low-power device, the number of cores used for the application can be adjusted at initialisation to keep pace with the incoming data flow while functioning at the lowest level of energy consumption. The clock frequency of the A7 cluster and the A15 cluster can be set at initialisation and updated at runtime to prevent the probe from being flooded.

The determinism of Gegelati helps in the validation of the embedded Secure-Gegelati probe on the octa-core Exynos 5410 SoC, ensuring that the training is occurring properly on the embedded platform.

To keep pace with the input stream of data in a training configuration, optimisations have been performed. At training, all root teams analyse the input data. The best root teams are selected at the end of the generation for replication and mutation. The least performing root teams are deleted. Online training in a real-time context forces us to analyse each network flow log only once. The best root team is not necessarily the one that will analyse a network flow log. The analysis during an online training of the probe is thus less performing than while inferring. The results shown in next sections include these modifications. Network flow logs arrive in a stack and each root team unstacks one log and analyses it. Each root team thus trains on different samples. The training step requires more memory, as several root team co-exist. In order to reduce the memory constraints on the embedded platform while training, the TPG parameters have been edited comparing with the state of the art parameters in [19]. The number of root teams have been decreased (from 360 to 200) and the size of the training batches have been increased (1000 to $50k$). The number of outgoing edges has been reduced (i.e. there are less programs in the graphs) and the program size have also been increased.

## 5.4 Using Secure-Gegelati as an IDS

At initialisation, the analyst trains the Secure-Gegelati probe offline providing labelled data or online, using results from other signature-based or anomaly-based IDSs to provide the labels. After setting the parameters of the embedded device such as

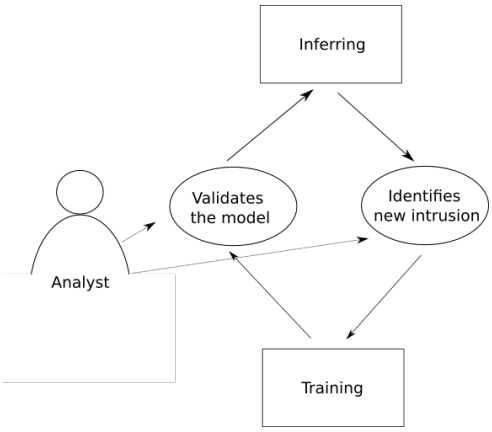



Fig. 6: After an initial training, the Secure-Gegelati IDS runs on the network. When a new intrusion is identified by the analyst, he updates the training set providing new labelled data and the probe is switched in training mode. New intrusion detection capability is checked on a validation set before turning the probe back in inferring mode.

started cores and frequency of the core clusters, the probe starts training. At any time, the analyst is able to freeze the training to use the probe in inferring mode. To update the device, he only has to switch back into training mode. This is illustrated in figure 6.

# 6 Experimental Results

This section sums up the experimental results of the proposed protocol and evaluates the utility of the TPG algorithm in the design of a NIDS. We chose to compare the experimental results with the RF algorithm due to their high detection rates on the CIC-IDS 2017 dataset 2 [46,52].

As stated in [32], the CIC-IDS 2017 data-set is relevant as it includes most of the features such as realistic network configuration, realistic network traffic, labelled observations and different attacks that makes it useful for a real-world scenario simulating a small IS [21,37]. We also use the CIC-IDS 2018 dataset as it is more realistic in terms of rareness of malicious events and as it sums up generated traffic on a network topology similar to a small-company. The data-sets are presented in more details in the section 6.1

We want to show here that the TPG is a relevant contribution to IDS design by measuring its performances on various experiments.

- Firstly, we measure its performances on the CIC-IDS 2017.
- Then, to demonstrate the adaptability of the Secure-Gegelati probe, we run the following experiments:
  1. we measure the evaluation performances of a TPG previously trained on CIC-IDS 2017 on the CIC-IDS 2018 dataset and show that it performs better than a supervised learning algorithm. We use the RF algorithm for comparison.
  2. We train a model without a category of attacks and study the reaction of Secure-Gegelati when we introduce this attack in the dataset. We show that Secure-Gegelati is able to evolve in order to discover the new attack and how it affects the learnt model.
- We show that TPG is a relevant solution in a real-time context and that it is able to keep pace with the net flow of the CIC-IDS data-sets.
- We show that the TPG-based IDS is useful on an embedded platform and measure its performance in terms of the number of analysis per seconds. Details on the embedded platform are available in section 5.3
- Finally, we measure the energy efficiency of the TPG-based NIDS on the embedded platform.

## 6.1 Data-sets

*6.1.1 The CIC-IDS 2017 dataset*

The CIC-IDS 2017 dataset is one of the intrusion detection data-sets with the most diverse and realistic range of cyber-attacks. It addresses recent attacks that are not available in other dataset using a range of different computers, operating systems and security features [33]. It has been generated using two networks. The first one, the victim Network, is a set of 5 servers and 10 computers using different operating systems (Windows, Linux and Macintosh) and necessary equipment such as routers, firewalls and switches. The attacker's network includes 4 computers using Windows 8.1 and Kali operating systems, one router and one switch. The CIC-IDS 2017 dataset is an IDS dataset that contains a week of generated traffic network frames. In this traffic can be found several anomalies labelled in 14 different categories. Most of the traffic is labelled as "normal traffic". The attacks are highly unbalanced but this disparity does not reflect a usual behaviour on an IS. Indeed, too many attacks are in it with respect to the amount of normal connections. Table 1 sums up the different classes and the amount of data per class.

Table 1: Distribution of classes in the CIC-IDS dataset in net-flow logs. Each net-flow log corresponds to 78 fields and 312 Bytes of raw data.

| Classes | Amount of net-flow logs |
|---|---|
| BENIGN | 2.359.087 |
| Dos Hulk | 231.072 |
| PortScan | 158.930 |
| DDoS | 41.835 |
| Dos Goldeneye | 10.293 |
| FTP-Patator | 7.938 |
| SSH-Patator | 5.897 |
| Dos Slowloris | 5.796 |
| Dos Slow-httptest | 5.499 |
| Bot | 1.966 |
| Brute force | 1.507 |
| XSS | 652 |
| Infiltration | 36 |
| SQL-Injection | 21 |
| Heartbleed | 11 |

The data used in the research is taken from the fully labelled CIC-IDS 2017 dataset. It sums up in a .csv file 78 network flow features from the captured network traffic (PCAP files). Information such as the destination port, the number of bytes per seconds or flags can be found in the dataset. Those information are represented on 32bits integers, floating-point numbers and Boolean. More information about the 78 extracted features have been defined and explained in the CICFlowMeter webpage [1].

The generated data stream reaches a peak of 170 connections/s during the Denial of Service attacks and the mean data stream is around 50 connections/sec. We assume that it is possible to generate the same network flow statistics as in CICIDS-2017 in real-time.

#### *6.1.2 The CIC-IDS 2018 dataset*

The CIC-IDS 2018 provides a similar dataset and is the result of a simulation of a much bigger IS. Indeed, the dataset generated comes from a set of 420 computers divided into 5 departments and thirty servers. The attacks are carried out from an "attacker" network composed of 50 machines using the Windows and Kali operating systems. The CIC-IDS 2018 dataset is composed of the same categories of attacks as the CIC-IDS 2017 dataset but with slight differences due to the modernisation of some attacks or the changes in operating system. The CIC-IDS 2018 dataset is more representative of a realistic small company-sized IS.

#### *6.1.3 Adjustments*

Some differences between the two data-sets required pre-processing to have consistent data for training and evaluation of the solution. In particular, the CIC-IDS 2018 dataset has two additional columns compared to the CIC-IDS 2017 dataset, and the CIC-IDS 2017 has information about header length whereas the other dataset does not.

#### *6.1.4 CIC-IDS 2017 Analysis*

The CIC-IDS 2017 paper [42] gives NIDS results in terms of Precision, Recall and F1 Score using seven ML algorithms. Those results have been reported in Table 2. Among those 7 supervised learning algorithms, 4 stand out. Indeed, k-nearest neighbours (KNN), RF, Iterative Dichotomiser 3 (ID3) and the Quadratic Discriminant Analyser (QDA) reach F1-scores above 90%. The Adaboost, Multi-Layer Perceptron (MLP) and Naive-Bayes algorithm reach lower detection detection rates.

Even though the ID3 algorithm produced the best results, we use RF to compare our results. RF have results close to the ID3 algorithm and a multi-threaded low level implementation is available in the Ranger framework [51]. This implementation is useful to try the RF on the embedded platform.

Table 2: Results on the CIC-IDS 2017 dataset using various ML Algorithms as reported in [42]

| Algorithm | Precision | Recall | F1-score |
|---|---|---|---|
| $KNN$ | 0.96 | 0.96 | 0.96 |
| $RF$ | 0.98 | 0.97 | 0.97 |
| $ID3$ | 0.98 | 0.98 | 0.98 |
| $Adaboost$ | 0.77 | 0.84 | 0.77 |
| $MLP$ | 0.77 | 0.83 | 0.76 |
| $Naive - Bayes$ | 0.88 | 0.04 | 0.04 |
| $QDA$ | 0.97 | 0.88 | 0.92 |

A comparison metric for IDS is given by the Intrusion Detection Capability $C_{ID}$ [13] based on information theory.

This $C_{ID}$ is the result of the computation of the mutual information ($I(\vec{X};\vec{Y})$ having $\vec{X}$ being the inputs log class of the IDS and $\vec{Y}$ being the classifications given by the IDS on the input logs $\vec{X}$) normalised by the entropy of $\vec{X}$: $H(\vec{X})$. An IDS has to determine whether a log is normal or represents a threat. SecureGegelati can be seen as a deterministic function that acts on the input stream ($\vec{X}$) and produces the output $\vec{Y}$ ideally being identical to $\vec{X}$ classes (i.e. "Benign" or "Attack"). The number of guesses represents $H(\vec{X})$

(i.e. the information content of $\vec{X}$) and the number of correct guess represents $I(\vec{X};\vec{Y})$. More details on the $C_{ID}$ can be found in [13].

The higher the $C_{ID}$ is, the better the IDS performs. In practice, having a perfect model ($FPR$ (False Positive Rate) and $FNR$ (False Negative Rate) both value 0) leads to a $C_{ID}$ of 1. This value helps to choose the best trade-off between Precision and Recall.

## 6.2 Performance of the RF and the TPG algorithms on the data-sets CIC-IDS 2017 and CIC-IDS 2018

We compare the algorithms in terms of accuracy, precision, recall (or sensitivity) and F1-score. It is more interesting in this study to maximise both Precision and Recall (high F1-score) as we want to detect as many attacks as possible (high recall) while generating as few false-positive alerts as possible (high precision).

### *6.2.1 RF implementation*

We use an open-source c++ implementation of RF for high dimensional data ("Ranger") [51] to compare with the TPG results. Table 3 sums up the precision, recall, F1-score and accuracy of the RF algorithm.

We can see that there are some differences with the results given in table 2 and in [52] using a state of the art RF algorithm. Those difference are due to a custom RF parameters tuning to reach a 100% precision. Such a precision is preferable because in the intrusion detection field, false positive alerts are costly time consuming.

Table 3: Machine learning statistics using RF on the CICIDS dataset for different training time.

| Time (s) | Accuracy | Precision | Recall | F1-score |
|---|---|---|---|---|
| 183 | 94.5 | 97.6 | 74.1 | 84.3 |
| 539 | 95.5 | 99.7 | 77.3 | 87.1 |
| 1867 | 94.8 | 99.8 | 73.9 | 84.9 |
| 3543 | 92.2 | 100.0 | 60.3 | 75.3 |

### *6.2.2 Using the TPG to analyse CICIDS*

Using the Gegelati framework gives good results in terms of accuracy, precision, recall and F1-score as shown on Table 4. The learning curves of the TPG highlight the discovery of new properties between the generations. As the reward function is based on the F1-score, precision (or recall) can drop if it leads to a final increase of the F1-score.

Table 4: Machine learning statistics using Secure-Gegelati on the CICIDS dataset depending on the total time (training + evaluation time).
Comparing with table 3, we can see that for a similar amount of time, the performances of both algorithms are close. In particular, we are not able to reach a 100% precision with Secure-Gegelati but the high recall makes it competitive.

| Time (s) | Accuracy | Precision | Recall | F1-score |
|---|---|---|---|---|
| 363 | 86.27 | 66.65 | 63.05 | 64.80 |
| 720 | 91.89 | 97.10 | 61.33 | 75.18 |
| 1701 | 94.74 | 96.72 | 76.32 | 85.31 |
| 3847 | 96.68 | 99.12 | 84.18 | 91.04 |

Comparing table 4 with table 3, we can see that for a similar amount of training time the results of Secure-Gegelati are slightly better than the results of the RF system. On one side, RF obtains no false positive alerts (really saving for an analyst). On the other side, the recall of the method is quite low. The RF implementation of the NIDS misses 40% of the attacks. Secure-Gegelati is not able to detect attacks with a 100% precision but it is incorrect less than one percent of the time. On the other side, Secure-Gegelati is able to detect over 84% of the attacks of the dataset. The learning of the RF being conditioned by statistics, a fast RF-based IDS will difficultly be able to detect the least represented classes. For example, Heartbleed or SQL-injections represent less than 0.001 % of the data available in the CICIDS-2017 dataset. A RF-based probe able to detect those attacks could be designed by changing learning parameters and thus increasing the learning time of the probe.

## 6.3 Adaptability of Gegelati

### *6.3.1 Inferring the previous models to the CIC-IDS 2018 data-set*

In real-world conditions, the network environment is dynamic. The network topology can change, new services or tools can be installed and new user-behaviour will be discovered. Even though switching from a tiny IS (represented by the CIC-IDS 2017 data-set) to a company-sized IS (CIC-IDS 2018), we want our algorithm to be robust and to keep detecting as many threats as possible (low false-negative rate / high recall) while not generating false-positive alarms.

We sum up in Table 5 the evaluation of trained RF algorithm and TPG algorithm for CIC-IDS 2018.

Table 5: Inferring models previously trained on the CIC-IDS 2017 data-set on the CIC-IDS 2018 data-set

| Algorithm | Accuracy | Precision | Recall | F1-score |
|---|---|---|---|---|
| *RF* | 70.58 | 100 | 0.08 | 0.16 |
| *TPG* | 91.0 | 95.3 | 24.5 | 39.0 |

The difference between the results presented in Table 5 comes from the learning mechanisms. In the first case, with RF, the learning is done by studying the whole dataset as well as selecting the discriminating variables and thresholds that permit a classification without false positives. When the available information changes, that threshold and variables become less accurate for the classification of the new net-flow logs and leads to more classification errors. We keep detecting 1 threat over 4 using a TPG as the programs activates by producing the highest return value. Even if the content of the information changes, the TPG selects the discriminating variable and applies a program on those. The sequence of programs leading to the raise of an alarm might still activate, even though the observed values changed.

Although the use of RF does not generate false-positive alerts, it raises a mean of an alert every 1250 attacks whereas the TPG is raising a positive alert every 4 attacks with 95% of precision. We prefer the use of a TPG-based IDS as $C_{ID}(TPG) = 0.15$ and $C_{ID}(RF) = 4 \times 10^{-3}$.

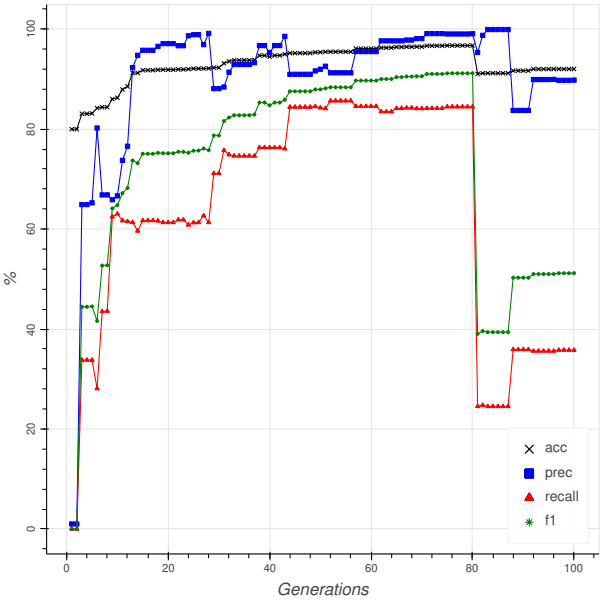



Fig. 7: Machine learning statistics using Secure-GEGELATI on the CICIDS dataset depending on the total time (training + evaluation time. At generation 80, the analysed connections are switched to CIC-IDS 2018 instead of CIC-IDS 2017).
Precision of the detection is kept high after the data-set change. Secure-GEGELATI trains with the new connection summaries and its recall goes up (between the 80th and 100th generation). Due to the massive changes and to the already existing knowledge base, it needs more time to fit the environment and perform an accurate detection.

### *6.3.2 Discovering new categories of attacks*

To demonstrate the detection of novel attacks, we carefully created two new data sets from the CIC-IDS 2017 dataset. The first one is the CIC-IDS dataset without all the attacks labelled as "Port Scan" and the traffic in between Port Scan attacks. The second is the CIC-IDS dataset without all DoS Slow Loris, Dos slow HTTP-test attacks and Port Scans. We trained offline for 50 generations on both dataset and added the port scan log to the environment while Secure-GEGELATI worked in inference mode. After a while, we decided to re-train the probe and wrote down the results after 1 generation and after 50 generations of training. Table 6 shows the results of the experiment were Port Scan attacks are the novelty whereas table 7 shows the results of the experiment where the DoS attacks (all categories) are added to the dataset.

We can see in Table 6 that most attacks are less detected after 50 re-train generations than after 1 re-train generation, marginally for some, significantly for others. This is partly due to the data imbalance described in table1 and also to the inner reward mechanism trying to prevent Secure-GEGELATI to raise false positive alerts Before the modification of the dataset, Dos Hulk is the predominant attack class in the dataset. When Port Scans are added, it becomes the second most represented class, making it important for Secure-GEGELATI to be detected. Port Scans are known to have a signature and thus it is more likely that Secure-GEGELATI will come out with a good set of observation to detect them efficiently. Dos attacks are typically discovered through their volume which is something Secure-GEGELATI is not doing. Secure-GEGELATI can eventually come out with correct observations to reach back 100% of detection on the Dos-Hulk attack through an extended training.

Table 6: This table sums up the per class true positives when adding Port Scan attacks to the training set after 50 generations. The inferring results are very different from the results of the offline training due to the change of the evaluation set (required to insert the port scan attacks in the dataset). Note that without knowing anything about Port-Scans, the TPG model is able to raise an alert for 40% of them. Retraining causes an instant drop of the True Negative Rate and an instant raise of the True Positive Rate (TPR). Secure-Gegelati tends to fit the most frequently occurring data of the dataset and thus becomes really good at detecting port-scan while keeping a low false-positive rate. × represents irrelevant data as they are not part of the evaluation set. Some attacks were not present in both evaluation sets or never detected.

| class | train (50 gen) | Inferring | re-train (1 gen) | re-train (50 gen) |
|---|---|---|---|---|
| BENIGN | 96.6 | 95.7 | 78.9 | 99.8 |
| XSS | 4.8 | 5.6 | 5.6 | 0 |
| DoS slow Loris | 29.1 | 42.6 | 38.0 | 36.4 |
| Dos slow HTTP-test | 58.9 | 66.0 | 63.8 | 59.6 |
| Dos hulk | 94.8 | 100 | 100 | 76.9 |
| Port Scan | × | 42.6 | 99.4 | 99.3 |
| TPR | 90.9 | 38.9 | 87.3 | 86 |

Table 7: This table sums up the per class true positives when adding Dos attacks to the training set after 50 generations of offline training. The inferring results are very different from the results of the offline training due to the change of the evaluation set (required to insert the DoS attacks in the dataset). This time the model is not able to detect any DoS attack in inferring mode. Retraining causes an instant drop of the True Negative Rate. Secure-Gegelati tends to fit the most present data of the dataset and thus fits to detect Dos Slow-Loris and DoS Slow-HTTP test while keeping a low false-positive rate. × represents irrelevant data as they are not part of the evaluation set. Some attacks were not present in both evaluation sets or never detected. The data imbalance is described in Table 1

| class | train (50 gen) | Inferring | re-train (1 gen) | re-train (50 gen) |
|---|---|---|---|---|
| BENIGN | 100 | 100 | 86.1 | 99.9 |
| Brute-force | 75.0 | 67.5 | 67.5 | 67.5 |
| XSS | 86.2 | 94.4 | 94.4 | 94.4 |
| DoS slow Loris | × | 0.0 | 20.9 | 20.9 |
| Dos slow HTTP-test | × | 0.0 | 59.6 | 59.6 |
| Port Scan | × | 0.0 | 99.7 | 99.4 |
| TPR | 48.2 | 10.3 | 96.3 | 96.1 |

Tables 6 and 7 sum up that the TPG is agile enough to detect new threats as they come and can be retrained online to update its knowledge base and keep performing a precise detection without generating too many false-positive alerts. Secure-Gegelati tends to maximise its rewards and thus, attacks that have only a few samples in its dataset are more likely not to be detected.

## 6.4 Real-time computation and energy efficiency of Secure-Gegelati

The training time $T$ of the TPG is conditioned by several factors:

- the number of samples or number of connections to analyse. A single policy takes a time $T = t$ to analyse a challenge and takes $T \approx n \times t$ to analyse n samples. The approximation is due to the depth of the chosen path. The more relay-teams descended while running the algorithm, the more time it takes to take an action.
- The number of root-teams will have a direct influence on the training time. A root-team takes $T \approx n \times t$ to train and $R$ concurrent root-teams take $T \approx n \times t \times R$.

To train the algorithm, we can choose to send the same sample to each root-team (M1). In this case, the analysis rate values: $A_{rate}(M1) \approx \frac{n}{T}$ Or we can train the algorithm sending samples to root-teams as they come (M2) which results in a boost of the analysis rate: $A_{rate}(M2) \approx \frac{R \times n}{T}$. We sum up in Table 8 the rates obtained at different stages of the training on the CIC-IDS 2017 dataset using a batch size of 50000 samples. When training a TPG, a batch corresponds to the amount of actions taken before receiving a reward.

Note that Table 4 was obtained using the second method where samples are analysed as they come and the root-teams do not learn on the same samples.

As seen in Section 6.1, the CIC-IDS 2017 dataset produces a mean of 50 connections per seconds and peaks to 170 connections per second. The Secure-

Table 8: Measuring the number of connections analysis per seconds using Secure-Gegelati. The first method (M1) trains a TPG by sending identical data to all teams whereas the second method (M2) stacks all the data in a buffer and teams unstack the data one at a time. In method M2, teams are training with different data through time.

| Gen. | Training time (s) | $A_{rate}(M1)$ | $A_{rate}(M2)$ |
|---|---|---|---|
| 1 | 27.29 | 1832 | 916052 |
| 20 | 38.53 | 1297 | 642861 |
| 40 | 52.89 | 945 | 472643 |
| 60 | 58.61 | 853 | 426577 |
| 80 | 66.03 | 757 | 378615 |

Gegelati algorithm is able to keep pace with the dataset using a X86 architecture. Through the generations, the graphs get more complex as relay teams are added. This is why the performance of the algorithm decreases with time.

As the peak number of connections per seconds on CIC-IDS values 170 connections per second, the embedded design must be efficient enough to perform online training at this rate. It is recalled that the four A7 cores of the Exynos 5410 platform can run at a maximal frequency of 1.4 GHz and the four A15 cores, at 2.0 GHz. We present in Table 9 the training times on the Exynos 5410.

Table 9: Reachable number of connection analysis per seconds using the Exynos 5410. We effectuate the training on TPG using 200 root-teams and training over a batch of 500 connection summaries. The frequencies $F_{A7}$ and $F_{A15}$ are in $GHz$.

| Cores | $F_{A7}$ | $F_{A15}$ | Train (s) | $A_{rate}$ |
|---|---|---|---|---|
| 4A7 + 4A15 | 1.4 | 2.0 | 7.52 | 13294 |
| 1A7 + 4A15 | 1.4 | 2.0 | 10.49 | 9533 |
| 4A7 | 1.4 | - | 22.88 | 4371 |
| 3A7 | 0.2 | - | 239.21 | 418 |
| 2A7 | 0.3 | - | 258.00 | 387 |
| 2A7 | 0.2 | - | 614.39 | 162 |
| 1A7 | 0.2 | - | 1286.58 | 77 |

The real-time constraint is satisfied on the embedded platform, even in the training phase. The table 9 shox that the rate reachable using Secure-Gegelati using 2 A7 cores at 300 MHz is superior to the connection rate on the CIC-IDS 2017 data-set.

#### *6.4.1 Energy efficiency of the IDS*

The x86 Intel Xeon W-2145 processor used for previous experiments in Section 6.3 has a 140 Watts peak thermal dissipation power (TDP). Using this architecture, Secure-Gegelati analyses up to $378k$ connections per seconds during training and $480k$ connections per second using the graph on inference.

The Energy efficiency ($E_{eff}$) of the x86 platform is thus : $E_{eff}(Training) = 2.7k\ connections/Watt$ and $E_{eff}(Inferring) = 3.4k\ connections/Watt$. As a comparison, the RF IDS has an energy efficiency of $E_{eff}(Inferring) = 400\ connections/Watt$ using a x86 architecture on inference. We used the framework RANGER [51], a parallel framework to train RF for a fair comparison with a parallel TPG-based IDS. The Secure-Gegelati software has thus 8× the energy efficiency of RF-based IDS.

On the Exynos, the chosen solution (using 2 A7 cores at a 300MHz frequency) consumes 0.05 W. It results in an energy efficiency $E_{eff}(Inferring) = 200\ kconnections.W^{-1}$.

# 7 Discussion

## 7.1 Learning algorithms

TPG and RF strongly differ in their learning mechanisms. While RF ingests all the training data at once to build a model, TPG progressively incorporate it and can recover from badly labelled data by feeding the system with new correctly labelled data. The training time and performance of RF is strongly impacted by the amount of data and the chosen learning parameters (tree depth, number of trees, etc) that need to be tuned. With a fine tuning of these parameters and large computation time, the models of predictions can be very accurate. The drawback of RF is however that they need to be trained and fine tuned from scratch to incorporate a new intrusion type.

TPG take a longer time than RF to converge to an accurate model, as their model results from trials and errors. Parameterization of the TPG agent is rather easy, the state of the art parameters used by Stephen Kelly [18] suit most learning application and the sensitivity of the parameters is low. Light modifications of the parameters do not affect much the learning process but can bring interesting properties such as light graph structure or ability to detect rare events. However, the positive point is that TPG are able to detect attacks while training . Online training makes TPG more fit to detect novelty. Even if TPG take time to converge, they

finally adapt and are able to detect new threats as they arrive.

### 7.2 Limitations and Future Work

The Secure-Gegelati TPG algorithm in its current form is still limited for the observation of rare events. Indeed, rarely activated teams and programs may be deleted, which can cause issues in real world conditions for very rare intrusion detection. In the CICIDS 2017 dataset, there is 1 attack out of 5 connections so the test conditions overrate intrusions and the problem does not appear.

A false alarm rate of 0.2% is low but results in too many false alarms in the practical context of an IDS. The current detection system needs to be complemented with temporal filtering, exploiting the multi-connection nature of most intrusions, to reach the extremely low false alarm rates required in practice. This objective is kept as future work [17].

The study of Secure-Gegelati has proven that TPG are well suited for adaptive network intrusion detection. However, the Secure-Gegelati TPG model currently requires a large amount of labelled data to converge. As a future work, we intend to reduce the need of supervision by introducing semi-supervised learning into the TPG framework.

## 8 Conclusion

This paper has introduced the Secure-Gegelati learning-based real-time NIDS and has demonstrated the agility and energy efficiency reached by the resulting network probe. Secure-Gegelati combines several capabilities required in a NIDS: rare events detection with very few false alarms, as Secure-Gegelati detects more than 80% of the intrusions with a precision over 99%; high energy efficiency, as Secure-Gegelati is 8× more energy efficient than RF in the same inference conditions; high scalability as the speedup over 4 embedded cores reaches 96.9% of the optimum. Furthermore, thanks to the TPG intelligence, Secure-Gegelati is a flexible tool that adapts to novel threats. The system can anytime be switched into a training mode and be fed with new labelled benign and intrusion data to improve its capabilities. In order to advance on agile NIDS, our short term future work will concentrate on evaluating Secure Gegelati in a more realistic context than the one offered by CIC-IDS 2017 dataset. To this end, a sand-boxing infrastructure will be setup with normal traffic and attacks simulations.

## References


1. Intrusion Detection Evaluation Dataset (CIC-IDS2017). [Online; accessed 22-September-2021].
2. Simon D. Duque Anton, Sapna Sinha, and Hans Dieter Schotten. Anomaly-based Intrusion Detection in Industrial Data with SVM and Random Forests. *arXiv:1907.10374 [cs]*, July 2019. arXiv: 1907.10374.
3. Daniel Atkins, Kourosh Neshatian, and Mengjie Zhang. A domain independent genetic programming approach to automatic feature extraction for image classification. In *2011 IEEE Congress of Evolutionary Computation (CEC)*, pages 238–245. IEEE, 2011.
4. M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, jun 2013.
5. James Cannady. Next Generation Intrusion Detection: Autonomous Reinforcement Learning of Network Attacks. page 12, 2000.
6. Hervé Debar, Marc Dacier, and Andreas Wespi. A revised taxonomy for intrusion-detection systems. *Annales Des Télécommunications*, 55(7):361–378, July 2000.
7. D.E. Denning. An Intrusion-Detection Model. *IEEE Transactions on Software Engineering*, SE-13(2):222–232, February 1987. Conference Name: IEEE Transactions on Software Engineering.
8. Karol Desnos, Nicolas Sourbier, Pierre-Yves Raumer, Olivier Gesny, and Maxime Pelcat. Gegelati: Lightweight artificial intelligence through generic and evolvable tangled program graphs. In *Workshop on Design and Architectures for Signal and Image Processing (14th edition)*, pages 35–43, 2021.
9. Gangsong Dong, Yi Jin, Shiwen Wang, Wencui Li, Zhuo Tao, and Shaoyong Guo. DB-Kmeans:An Intrusion Detection Algorithm Based on DBSCAN and K-means. *2019 20th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, 2019.
10. Salma Elhag, Alberto Fernández, Abdullah Bawakid, Saleh Alshomrani, and Francisco Herrera. On the combination of genetic fuzzy systems and pairwise learning for improving detection rates on Intrusion Detection Systems. *Expert Systems with Applications: An International Journal*, 42(1):193–202, January 2015.
11. Meera Gandhi and SK Srivasta. Detecting and preventing attacks using network intrusion detection systems, 2008.
12. Olivier Gesny, Pierre-Marie Satre, and Julien Roussel. Cbwar: Classification de binaires windows via apprentissage par renforcement. In *Computer & Electronics Security Applications Rendez-vous (C&ESAR)*, 2018.
13. Guofei Gu, Prahlad Fogla, David Dagon, Wenke Lee, and Boris Skoric. Measuring intrusion detection capability: an information-theoretic approach. pages 90–101, January 2006.
14. Neminath Hubballi and Vinoth Suryanarayanan. False alarm minimization techniques in signature-based intrusion detection systems: A survey. *Computer Communications*, 49:1–17, August 2014.
15. L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement Learning: A Survey. *arXiv:cs/9605103*, April 1996. arXiv: cs/9605103.
16. Stephen Kelly. Scaling Genetic Programming to Challenging Reinforcement Tasks through Emergent Modularity. June 2018. Accepted: 2018-06-21T16:04:28Z.
17. Stephen Kelly and Wolfgang Banzhaf. *Temporal Memory Sharing in Visual Reinforcement Learning*, pages 101–119. Springer International Publishing, Cham, 2020.

18. Stephen Kelly and Malcolm I. Heywood. Multi-task learning in Atari video games with emergent tangled program graphs | Proceedings of the Genetic and Evolutionary Computation Conference, 2017.
19. Stephen Kelly, Robert J. Smith, and Malcolm I. Heywood. Emergent Policy Discovery for Visual Reinforcement Learning Through Tangled Program Graphs: A Tutorial. In Wolfgang Banzhaf, Lee Spector, and Leigh Sheneman, editors, *Genetic Programming Theory and Practice XVI*, Genetic and Evolutionary Computation, pages 37–57. Springer International Publishing, Cham, 2019.
20. Richard A Kemmerer and Giovanni Vigna. Intrusion detection: a brief history and overview. *Computer*, 35(4):supl27–supl30, 2002.
21. Ansam Khraisat, Iqbal Gondal, Peter Vamplew, and Joarder Kamruzzaman. Survey of intrusion detection systems: techniques, datasets and challenges. *Cybersecurity*, 2(1):20, July 2019.
22. Kwangjo Kim and Muhamad Erza Aminanto. Deep learning in intrusion detection perspective: Overview and further challenges. In *2017 International Workshop on Big Data and Information Security (IWBIS)*, pages 5–10. IEEE, 2017.
23. K. C. Krishnachalitha and C. Priya. Wireless Sensor Network-Based Hybrid Intrusion Detection System on Feature Extraction Deep Learning and Reinforcement Learning Techniques. In Sheng-Lung Peng, Le Hoang Son, G. Suseendran, and D. Balaganesh, editors, *Intelligent Computing and Innovation on Data Science*, Lecture Notes in Networks and Systems, pages 335–341, Singapore, 2020. Springer.
24. Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.
25. Christopher Kruegel and Thomas Toth. Using Decision Trees to Improve Signature-Based Intrusion Detection. In Giovanni Vigna, Christopher Kruegel, and Erland Jonsson, editors, *Recent Advances in Intrusion Detection*, Lecture Notes in Computer Science, pages 173–191, Berlin, Heidelberg, 2003. Springer.
26. Wei Li. Using genetic algorithm for network intrusion detection. January 2004.
27. Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1):16–24, January 2013.
28. Manuel Lopez-Martin, Belen Carro, and Antonio Sanchez-Esguevillas. Application of deep reinforcement learning to intrusion detection for supervised problems. *Expert Systems with Applications*, 141:112963, March 2020.
29. Francisco Maciá-Pérez, Francisco J. Mora-Gimeno, Diego Marcos-Jorquera, Juan Antonio Gil-Martínez-Abarca, Héctor Ramos-Morillo, and Iren Lorenzo-Fonseca. Network Intrusion Detection System Embedded on a Smart Sensor. *IEEE Transactions on Industrial Electronics*, 58(3):722–732, March 2011. Conference Name: IEEE Transactions on Industrial Electronics.
30. Deepak Mehta, Alie El-Din Mady, Menouer Boubekeur, and Devu Manikantan Shila. Anomaly-Based Intrusion Detection System for Embedded Devices on Internet. page 5, 2020.
31. Benoit Morel. Artificial intelligence and the future of cybersecurity. *Proceedings of the ACM Conference on Computer and Communications Security*, October 2011.
32. Nour Moustafa, Jiankun Hu, and Jill Slay. A holistic review of Network Anomaly Detection Systems: A comprehensive survey. *Journal of Network and Computer Applications*, 128:33–55, February 2019.
33. Ranjit Panigrahi and Samarjeet Borah. A detailed analysis of CICIDS2017 dataset for designing Intrusion Detection Systems | Panigrahi | International Journal of Engineering & Technology, 2018.
34. Gowher Parry and S. Kumar. Genetic algorithms in intrusion detection systems: A survey. *International Journal of Innovation and Applied Studies*, 5:233–240, January 2014.
35. Costin Raiu. Cyber-threat evolution: the past year. *Computer Fraud & Security*, 2012(3):5–8, March 2012.
36. Esteban Real, Chen Liang, David R. So, and Quoc V. Le. Automl-zero: Evolving machine learning algorithms from scratch. In *Proceedings of the 37th International Conference on Intelligent User Interfaces*, 2020.
37. Markus Ring, Sarah Wunderlich, Deniz Scheuring, Dieter Landes, and Andreas Hotho. A Survey of Network-based Intrusion Detection Data Sets. *Computers & Security*, 86:147–167, September 2019. arXiv: 1903.02460.
38. S. Sandosh, V. Govindasamy, and G. Akila. Enhanced intrusion detection system via agent clustering and classification based on outlier detection. *Peer-to-Peer Networking and Applications*, 13(3):1038–1045, May 2020.
39. Karen Scarfone and Peter Mell. Guide to Intrusion Detection and Prevention Systems (IDPS). Technical Report NIST Special Publication (SP) 800-94, National Institute of Standards and Technology, February 2007.
40. Arturo Servin and Daniel Kudenko. Multi-agent Reinforcement Learning for Intrusion Detection. In Karl Tuyls, Ann Nowe, Zahia Guessoum, and Daniel Kudenko, editors, *Adaptive Agents and Multi-Agent Systems III. Adaptation and Multi-Agent Learning*, Lecture Notes in Computer Science, pages 211–223, Berlin, Heidelberg, 2008. Springer.
41. Kamalakanta Sethi, Rahul Kumar, Nishant Prajapati, and Padmalochan Bera. Deep Reinforcement Learning based Intrusion Detection System for Cloud Infrastructure. In *2020 International Conference on COMmunication Systems NETworkS (COMSNETS)*, pages 1–6, January 2020. ISSN: 2155-2509.
42. Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorbani. Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization. In *ICISSP*, 2018.
43. K. G. Srinivasa. Application of Genetic Algorithms for Detecting Anomaly in Network Intrusion Detection Systems. In Natarajan Meghanathan, Nabendu Chaki, and Dhinaharan Nagamalai, editors, *Advances in Computer Science and Information Technology. Networks and Communications*, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pages 582–591, Berlin, Heidelberg, 2012. Springer.
44. B. Suresh and M. Ventachalam. IJSRET Volume 5 Issue 6, Nov-Dec-2019, November 2019. Library Catalog: ijsret.com.
45. Symantec. *ISTR Volume 22| Symantec*. 2017.
46. Serpil Ustebay, Zeynep Turgut, and Muhammed Ali Aydin. Intrusion Detection System with Recursive Feature Elimination by Using Random Forest and Deep Learning Classifier - IEEE Conference Publication, January 2019.

47. PAUL C VAN OORSCHOT. *COMPUTER SECURITY AND THE INTERNET: tools and jewels*. SPRINGER NATURE, S.l., 2020. OCLC: 1120697311.
48. Eduardo Viegas, Altair Santin, Alysson Bessani, and Nuno Neves. BigFlow: Real-time and reliable anomaly-based intrusion detection for high-speed networks. *Future Generation Computer Systems*, 93:473–485, April 2019.
49. Eduardo Viegas, Altair O. Santin, André França, Ricardo Jasinski, Volnei A. Pedroni, and Luiz S. Oliveira. Towards an Energy-Efficient Anomaly-Based Intrusion Detection Engine for Embedded Systems. *IEEE Transactions on Computers*, 66(1):163–177, January 2017. Conference Name: IEEE Transactions on Computers.
50. Marilyn Wolf. *High-Performance Embedded Computing*. Elsevier, 2014.
51. Marvin N. Wright and Andreas Ziegler. ranger: A Fast Implementation of Random Forests for High Dimensional Data in C++ and R. *Journal of Statistical Software*, 77(1):1–17, March 2017. Number: 1.
52. Arif Yulanto, Parman Sukarno, and Anggis Suwastika. Improving AdaBoost-based Intrusion Detection System (IDS) Performance on CIC IDS 2017 Dataset - IOPscience, 2019.
53. Diego Zamboni. Using Internal Sensors For Computer Intrusion Detection. August 2001.