DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Real-Time Remote Intrusion Detection

Valentin Zieglmeier

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Real-Time Remote Intrusion Detection

Echtzeit-Fernerkennung von Eindringlingen

Author:	Valentin Zieglmeier
Supervisor:	Prof. Dr. Alexander Pretschner
Advisors:	Thomas Hutzelmann
	Severin Kacianka
Submission Date:	15.05.2018

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 14.05.2018 Valentin Zieglmeier

Danksagung

Ich danke dem Lehrstuhl von Prof. Dr. Alexander Pretschner für die Möglichkeit, dort meine Masterarbeit zu verfassen. Mein besonderer Dank gilt meinen Betreuern Thomas Hutzelmann und Severin Kacianka.

Thomas Hutzelmann hat mir immer das Gefühl gegeben, dass wir gemeinsam an diesem Projekt arbeiten. Unsere Diskussionen und Gespräche waren spannend und hilfreich. Ich danke ihm, dass er sich so viel Zeit für diese Arbeit genommen hat.

Severin Kaciankas Pragmatismus und Weitblick waren ein guter Ausgleich für die unendlichen Ideen, die Thomas und ich oft hatten. Seine langjährige Erfahrung mit wissenschaftlichen Arbeiten war mehrfach der ausschlaggebende Punkt, ein Kapitel oder einen Abschnitt in die richtige Richtung zu lenken. Ich danke ihm, dass er diese Erfahrung eingebracht hat.

Ich will diese Gelegenheit auch nutzen, denen zu danken, die mich sonst bei der Arbeit direkt oder indirekt unterstützt haben.

Großer Dank gilt Sebastian Kreisel und Angelika Stefan. Ihre Unterstützung war inhaltlich, aber auch moralisch, sehr wichtig für mich.

Abschließend danke ich Amar Šaljić, Mikayla Webster, Frank Lu, Philipp Bock, Klaus Ondrag, Lisa Maier, Carina Schmidt, Stefan Dittrich und meinen Eltern für ihre Tipps, ihre Hinweise und Hilfe jeglicher Art.

Abstract

This thesis analyses security measures for connected vehicles. It is based on the real-world use case of providing server-side security for passenger cars.

New features and services such as automatic software updates and subscriptions to premium functions require a constant connection between vehicles and a central server. This added connectivity raises the likelihood of exposure to attackers and risks unauthorised access.

A possible answer to this issue are intrusion detection systems (IDS), which aim at detecting these intrusions during or after their occurrence. The problem with IDS is the large variety of possible systems with no sensible option for finding or comparing them.

Our contribution to this problem comprises two parts: First, a comprehensive literature survey and a taxonomy of IDS. Second, the conceptualisation and implementation of a testbed for the automotive real-world scenario.

The literature survey is conducted to give an overview of approaches to IDS in general. It is aimed at discovering solutions that have been shown to work in practice. Based on this survey, a taxonomy for differentiating IDS is presented.

The results of the literature survey indicate that no appropriate method for comparing different IDS for the automotive real-world scenario exists. In response to this, we develop an artificial testing environment for this use case, a so-called testbed. To verify the validity of our approach, we evaluate the testbed from multiple perspectives, regarding its performance and scalability with additional components, its fitness for purpose, and the quality of the data it generates.

Our evaluation shows that the testbed makes the effective assessment of various IDS possible. It solves multiple problems of existing approaches, including class imbalance. Additionally, it enables reproducibility and generating data of varying detection difficulty. Tests of a naïve IDS with testbed data confirm that detecting intrusions in the generated datasets is a challenge.

Contents

1 Introduction

Connected vehicles are becoming commonplace. In 2015, 35 % of new cars sold were already connected to the internet [2]. Accenture estimates that by 2020 that number will rise to 98 %, reaching 100 % in 2025 [2].

Manufacturers push for higher connectivity, as it offers advantages such as the ability to offer automated software updates which do not require customers to perform a manual update process in a repair shop [9, p. 61]. Additionally, new features enabled by internet connections are used as selling points. For example, vehicles may be monitored and controlled with a smartphone [9, p. 61]. Finally, vehicle-to-vehicle communication allows for advanced autonomous features such as cooperative collision warnings [126].

However, this connectivity also raises the likelihood of exposing vulnerabilities, which increases the risk of attacks [81, p. 2898]. Checkoway et al. [17] identify the vehicle remote telematic systems providing “continuous connectivity via cellular [...] networks” as “perhaps the most important part of the long-range wireless attack surface” [17, p. 5].

For passenger cars, traditional defence measures may not always be feasible. The vehicles are expected to subsist for long periods of time, with the average age of light vehicles in the United States rising from 10.6 years in 2010 to 11.6 years in 2016 [45]. It is reasonable to expect that it will rise even further in the future. Zhang et al. [129] argue that this long lifespan makes it hard for manufacturers to predict the necessary hardware for on-board protection [129, p. 15]. Because of strict limits in production budgets, manufacturers are likely to minimise the cost for each vehicle and thus only include the minimum required security hardware.

Therefore, off-board protection may be necessary for sufficient security over the lifespan of the vehicle. An important challenge with this approach is balancing on-board processing load and the communication of the vehicle to the manufacturer server [129, p. 15]. Ideally, the existing vehicle communication is utilised for this purpose, as it would neither require additional processing nor communication.

In the following, we briefly describe the application of intrusion detection in this scenario.

Intrusion detection

We define intrusions as modifications to the behaviour of a system without the manufacturer’s knowledge. That includes unauthorised access or fraud, but also software bugs (see Chapter 2). It is challenging to differentiate various types of intrusions, as their effects may be identical. For example, software bugs can result in disallowed system behaviour without any malicious intent.

To defend against intrusions, one can employ intrusion detection systems (IDS). They can be used as a second line of defence after preventive protection mechanisms such as software integrity verification [78]. IDS are built to detect intrusions that have already taken place, aiming to mitigate their consequences.

A considerable advantage of IDS in the scenario of connected vehicles is that they can be employed non-invasively. That means that a complete off-board protection mechanism can be implemented with their help, solving some of the problems such as production budget limits that we discussed before. Additionally, IDS may be combined with a manual review of potential intrusions. The system can preselect probable threats and bring them to the attention of security personnel, so that they can defend customers' vehicles more effectively against threats.

Structure of this thesis

First, we define terms and abbreviations used throughout this thesis in Chapter 2. That is followed by a literature survey in Chapter 3, in which we describe the taxonomy of IDS that we developed and discuss examples for individual approaches to IDS. We focus on those types of systems that have been shown to work in practice and omit theoretical concepts with no proven performance.

The results of the literature survey indicate that there is no appropriate solution for comparing different IDS for the automotive real-world scenario. Hence, Chapter 4 is discussing the concept as well as the implementation of our testbed (defined in Chapter 2), which aims at allowing the comparison of various types of IDS. That is followed by the evaluation of our testbed in Chapter 5. We concentrated on evaluating its performance, fitness for purpose and the quality of the data it generates.

Chapter 6 describes the limitations of our approach. Following that, Chapter 7 presents areas of interest for future work we have created with our work. Finally, we conclude this thesis in Chapter 8.

2 Terms and definitions

We use some terms and abbreviations throughout this thesis without further explanation. To enable the reader to understand their meaning, we briefly list and define them in the following.

2.1 Real-time remote intrusion detection

To describe real-time remote intrusion detection, we define each individual component of that term.

Intrusion: We define intrusions as modifications to the behaviour of a system without the manufacturer's knowledge. On the one hand, this comprises malware, unauthorised access and anomalies, including software bugs, that mostly occur without the user's knowledge. On the other hand, it encompasses misuse and fraud that can take place with or without the user's knowledge. Different types of intrusions are often difficult or impossible to differentiate from each other and do not always correspond to illegitimate actions. This definition corresponds to that of NIST (National Institute of Standards and Technology) publication 800-94 [92]. In this document, Scarfone et al. define intrusions as "incidents, which are violations or imminent threats of violation of computer security policies, acceptable use policies, or standard security practices" [92, p. 2-1]. They list malicious actions such as unauthorised access and misuse, but also legitimate use of a system as possible causes for detectable incidents [92, p. 2-1].

Intrusion detection: An intrusion detection system (IDS) aims at detecting intrusions during or after their occurrence and hence mitigating their consequences. For this purpose, they monitor events for possible violations of some defined policy or expected behaviour [92, p. 2-1]. These systems are the second line of defence after those that actively prevent attacks, such as firewalls. Contrary to those, IDS are designed to accommodate the fact that not all intrusions can be prevented [92, p. 2-2].

Remote intrusion detection: Commonly, IDS are employed within the system they intend to protect. Some architectures or system designs do not allow this type of defence measure to be deployed. The performance impact may be too high or the deployment cost unaffordable. Remote intrusion detection refers to an IDS deployed on a remote server with which the system under protection communicates, for example through web requests. The IDS tries to detect intrusions based on the communication of the protected system with the remote server [68, p. 20].

Real-time intrusion detection: IDS can be built for off-line or real-time detection. Off-line detection comprises working with long-term data, aggregating information about individual clients or behaviour patterns over time. In this scenario, the IDS can perform calculations or aggregate data before its detection process [62, p. 59]. This mechanism can be too time-consuming for some system designs. Real-time intrusion detection aims at detecting intrusions as they happen or soon thereafter, keeping up with the speed of continuously incoming data [62, p. 59].

2.2 Other terms

Apart from the terms explained before, we use the following concepts.

Microservices: Instead of employing a monolithic application for handling all requests, a server can make use of the microservice architectural style. This means splitting up functionality to isolated services, so-called microservices, that communicate via an API (see Section 2.3). This allows for independent deployment and increased flexibility and scalability [65].

Testbed: A testbed is an artificial testing environment for interactive tests of complex systems. The goal of a testbed is to simulate the environment in which the system under test will operate, similar to the practice of system testing. Additionally, it can be used for generating test data [10, p. 553].

2.3 Abbreviations

Finally, an alphabetical list of abbreviations used in this thesis.

API: An application programming interface enables computer programs to interact with each other through defined functions. These allow them to access underlying functionality of the respective program. A more detailed explanation can be found in [10, p. 21].

JSON: The “JavaScript Object Notation” is a data-interchange format with a minimal syntax that can be quickly parsed and generated. The format is defined in RFC 8259 [7].

ROS: The “Robot Operating System” is a robotics middleware aimed at, among other goals, being accessible, multi-lingual and thin. Additionally, it enables peer-to-peer communication, which means no central master server is necessary. This enables the effortless use of ROS in highly distributed systems [86].

XML: The “Extensible Markup Language” is a data-interchange format with high flexibility and generalisability. The format is defined in [8].

3 Survey of approaches to intrusion detection

We propose two aspects to differentiate IDS, the *detection method* and the *detection technique* (see Section 3.2). Based on these two aspects and our literature survey we arrive at a taxonomy of IDS in Section 3.3. Following that, Section 3.4 discusses different approaches to IDS. Finally, we give a gap analysis of existing research in Section 3.5.

3.1 Method

Following Wohlin [125], we employed the techniques referred to as "forward" and "backward snowballing" [125, p. 3] to search available literature. After starting off with central survey papers, we followed citations from and to these sources to identify other surveys and approaches. We then searched examples for each of the approaches we identified and traced those back to survey papers discussing them.

To verify the relevance of individual sources, we took their ranking in search engines and databases, and the number of other literature citing them as a basis. Approaches mentioned in multiple survey papers were given more weight.

Developing the taxonomy

We started the taxonomy by considering prior work. Liao et al. [68] and Chandola et al. [13] offer the most comprehensive taxonomies we could find. These were our starting point.

Refining the taxonomy

To find a more concise and fitting representation and complete the taxonomy, further sources were reviewed. Firstly, other surveys and overviews were used to expand, merge or purge categories. For the statistical category, Hodge et al. [43] offer a good overview of different techniques. For better delimitation of the clustering category, Leung et al. [64] offer a helpful categorisation. Additional subcategories were found in [32, 52, 62, 82].

Verifying the validity of categories

To make sure that we only discuss relevant categories, each of them was verified. We did a literature search for techniques in the respective areas. With the results we confirmed that these are not just theoretical solutions but can be employed in practice.

3.2 Differentiating intrusion detection systems

There are numerous criteria for differentiating IDS. Lazarevic et al. [62] propose a taxonomy covering the categories "Information source", "Analysis strategy", "Time Aspects", "Architecture" and "Response" (see Figure 3.1).

Another more extensive taxonomy is described by Liao et al. [68]. It includes the categories “Timeliness”, “System Deployment”, “Detection Strategy” and “Data Source” with further subcategories (see Figure 3.2).

While both cover numerous aspects, we can see that most of those are independent of the actual implementation of the system. Either they are fixed because of technical restrictions (some systems cannot be used in real-time scenarios for example) or they are interchangeable for individual solutions (like the response type).

Hence, we focus our survey on the two aspects that we consider most important when differentiating between IDS, the *detection method* and the *detection technique*.

3.2.1 Detection method

The *detection method* is one of the most fundamental aspects for differentiating between IDS. We define two possible methods, *signature-based* and *anomaly-based* detection. Other authors often use the same terms, as seen in Axelsson [3, p. 2] and Kabiri et al. [52, p. 84], or similar ones, like “misuse detection and anomaly detection” [62, p. 45] or “knowledge-based” and “behavior-based” detection [22, p. 363].

These terms all refer to the same two principles: *Signature-based* detection uses existing knowledge about possible intrusions to detect them, while *anomaly-based* detection analyses the normal behaviour of the system and flags anomalous actions as possible intrusions.

We want to note that Liao et al. [68] argue for a third variant, namely “Specification-based” detection. The information gain when differentiating further from *signature-based* detection as to where the data stems from seems marginal to us, so we forego this category.

Hodge et al. [43] also argue for three types of systems, differentiating *signature-based* detection further as “Type 2” detection, which models “both normality and abnormality” [43, p. 89] as well as “Type 3”, which models “only normality or in very few cases [...] abnormality” [43, p. 90]. This split does not influence the actual detection process. No matter if normality or abnormality was modelled, the system looks for certain predefined patterns. The only difference is what behaviour it identifies as an intrusion; that which has been modelled (when modelling abnormality) or that which has not been (when modelling normality). Accordingly, we omit separating these subcategories.

3.2.2 Detection technique

The second aspect when differentiating IDS is the *detection technique*. This refers to the actual algorithm that is being used when analysing incoming data. We define six categories of techniques, namely *rule-based*, *machine learning*, *statistical*, *clustering-based*, *state-based* and *pattern-based* techniques. Each of these categories groups different techniques that we elaborate on in Section 3.4.

Chandola et al. [13] describe six techniques, differentiating “Classification Based”, “Clustering Based”, “Nearest Neighbor Based”, “Statistical”, “Information Theoretic” and “Spectral” [13, p. 5]. We see “Nearest Neighbor Based” techniques as a part of our *statistical* category, “Spectral” techniques can be classified as *machine learning* and “Information Theoretic” techniques are merely one type of *clustering-based* techniques.

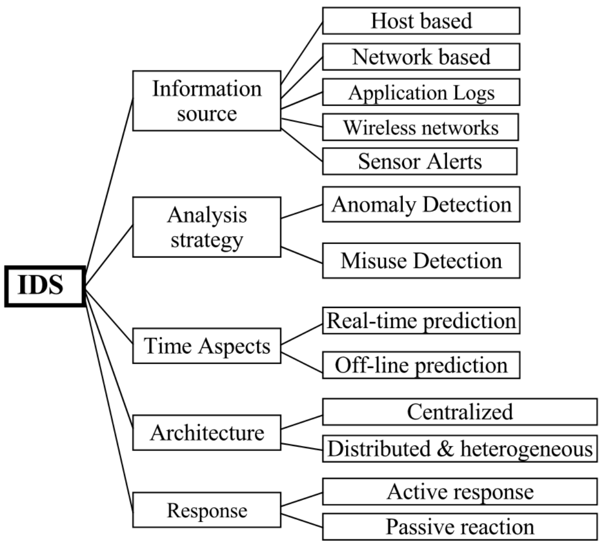


Figure 3.1: *A taxonomy of intrusion detection systems, taken from [62, p. 34].*

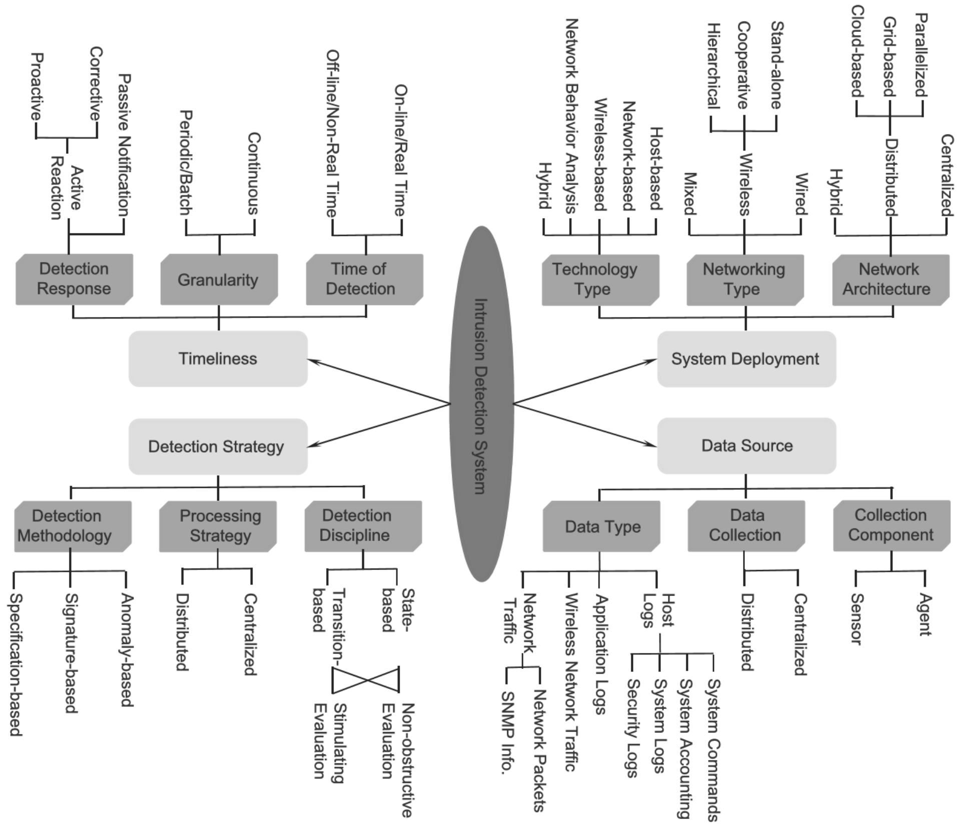


Figure 3.2: *A taxonomy of intrusion detection systems, taken from [68, p. 21].*

Liao et al. [68] argue for five different categories, namely “Statistics-based”, “Pattern-based”, “Rule-based”, “State-based” and “Heuristics-based” techniques [68, p. 17]. While the techniques they group into these categories do not always correspond to how we group them, we agree with most of their overall categories. Still, we do consider the category of *clustering-based* techniques important enough to add to this list. Additionally, their differentiation between “Rule-based” and “Heuristics-based” techniques is not entirely clear. We group some of those into the *rule-based*, others into the *machine learning* category.

In general, the distinction is not absolute and different ways of classifying systems may make sense. Considering this, our categories allow a concise and coherent grouping of the types of techniques that we found.

3.3 A taxonomy of intrusion detection systems

Figure 3.3 shows our taxonomy based on the aspects *detection method* and *detection technique*, used to classify different IDS. On the following page, Table 3.1 gives an overview of all subcategories and a corresponding example from literature. These approaches are discussed in Section 3.4.

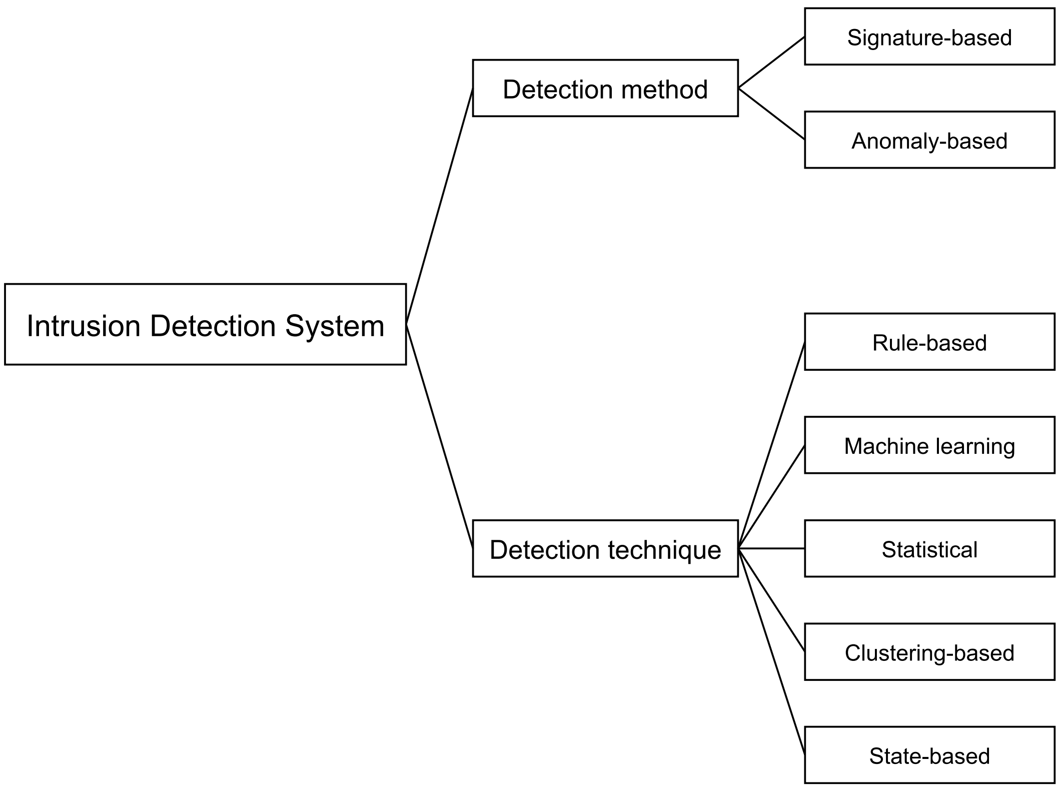


Figure 3.3: *Our taxonomy of intrusion detection systems.*

Detection technique		**Detection method**[a]		**Exam-ple**	**Sec-tion**
		Sig.	**Ano.**		
Rule-based	Simple rule-based	✓	–	[69]	3.4.1
	Fuzzy logic	✓	–	[23]	
	Pattern matching	✓	–	[99]	
	Model-based reasoning	✓	–	[33]	
Machine learning	Artificial Neural Networks	✓	✓	[11]	3.4.2
	Bayesian Networks	✓	✓	[60]	
	Support Vector Machines	✓	✓	[75]	
	Genetic Algorithms	✓	✓	[66]	
	Artificial Immune Systems	–	✓	[31]	
	Swarm Intelligence	✓	✓	[18][h]	
Statistical	Distribution-based	✓	✓	[42]	3.4.3
	Kernel Density Estimation	✓	✓	[128]	
	Regression analysis	✓	✓	[112]	
Clustering-based	Distance-based	–	✓	[48][h]	3.4.4
	Density-based	–	✓	[79]	
	Grid-based	–	✓	[64][h]	
	Hierarchical	–	✓	[71]	
State-based	Automata	✓	✓	[46]	3.4.5
	Markov chains	✓	✓	[127]	
	Coloured Petri nets	✓	✓	[61]	

[a] Detection method: Signature-based (Sig.), Anomaly-based (Ano.)

[h] Hybrid system, utilising multiple techniques

Table 3.1: *An overview of intrusion detection systems with corresponding examples.*

3.4 Intrusion detection techniques

In the following sections we describe examples for each of the intrusion detection techniques that we define in our taxonomy.

3.4.1 Rule-based techniques

The most basic approach to intrusion detection are *rule-based* techniques. They employ *signature-based* detection, as static rules need to be defined in advance based on domain knowledge.

Simple rule-based

These systems are based on simple “if-then-else” rules that are written to detect intrusions based on existing knowledge.

P-BEST [69] is a tool set that can be used for this scenario. To create rules for it, they must be written in the “P-BEST production rule specification language” and are then translated and compiled [69, p. 3]. Such rules can trigger actions or activate other rules to be checked.

Fuzzy logic

Fuzzy logic systems are based on the premise that not all decisions are a simple “yes” or “no”, instead the answer may only be partially right or wrong.

FIRE [23] is one such system that is based on a “fuzzy analysis engine”. Fuzzy rules are used to “determine the likelihood of specific or general network attacks” [23, p. 301]. The process consists of data collection, data processing and fuzzy threat analysis. Different data sources can be combined and given individual weights, depending on the relevance of the data they collect. Additionally, some historical data is preserved to allow for statistical comparisons.

Generating rules for *rule-based* IDS is well suited for a hybrid approach, as shown by Gomez et al. [34]. Their work describes the use of genetic algorithms (see page 13) for rule generation for a fuzzy logic system. Starting with a random population, the chromosomes are iteratively selected, crossed over, and mutated to arrive at an end condition. When that is satisfied, the best solution is returned and used to create a rule [34, pp. 3 sq.].

Pattern matching

Detecting attacks or legitimate behaviour is also possible with *pattern matching* systems. These are used to model more complex patterns that can be matched against the actual user behaviour.

A possible application of this principle is *file system checking*. Kim et al. [55] describe Tripwire, a tool designed for verifying the file system integrity. Its goal is to ensure that file changes are detected and alerts are raised for administrators. To efficiently detect changes, they propose storing fixed-size hashes for each file [55, p. 20]. The program regularly computes these signatures again and compares them to the stored ones. This can happen hourly or daily, depending on the use case [55, p. 26]. The tool can then either report its findings directly or write the reports to a file [55, pp. 24–25].

Another interesting application is described by Joyce et al. [51]. They propose *keystroke monitoring* for user identity authentication. Users must type in a specific string multiple times during registration. The latency signature for their typing is saved and mean and standard deviation are calculated. Outliers are discarded based on their latency. Finally, curves corresponding to the latency values for each letter are computed [51, p. 171]. To verify the user's identity, their typing pattern is compared to the stored signature. If more than 60 % of the tests are similar enough, the process completes [51, p. 172].

Model-based reasoning

A different approach is taken by systems utilizing *model-based reasoning* [1]. Models of attacks are built in advance and the system assumes that any of these may be happening at any moment. The likelihood for each is monitored based on the user behaviour and an alarm is raised in case an attack becomes highly likely.

Garvey et al. [33] describe the concept of a model-based reasoning system for the real-time intrusion detection expert system IDES [72]. Sequences of user behaviour exemplifying intrusion models are stored in a database, represented by the user actions. A separate store of currently possible attacks is used by the "anticipator" component to guess the next possible steps, which are then translated to actual audit data by the "planner", and finally handed over to the "interpreter" that compares them to the actual audit data from the monitored system [33, p. 4]. The limiting factor is that the specific behaviour needs to be recognisable and clearly distinguishable from normal, legitimate behaviour [33, p. 5].

3.4.2 Machine learning techniques

Machine learning techniques are widely researched in the context of intrusion detection. They can make use of *signature-based* detection (with supervised learning [38]) as well as the more traditional *anomaly-based* detection (with unsupervised learning [39]), depending on the availability of labelled data. In some scenarios, such as when analysing real-world usage, labelled data in sufficient quantities may not be available nor feasible to produce. This makes either automatic labelling measures or the use of unsupervised or semi-supervised learning necessary.

Artificial Neural Networks

The approach of *Artificial Neural Networks* (ANN) is to train a network of nodes for problem-solving. By supplying an ANN with training data, it slowly "learns" either based on provided labels or by adapting to the "normal" characteristics of the data. Instead of offering a binary answer, the network outputs probabilities for certain classifications.

Cannady [11] describes the application of an ANN to *signature-based* detection for supplanting a simple *rule-based* system. For training the network they used simulated attacks based on specific attack patterns. The generated data was heavily pre-processed in a multi-step process to ensure its suitability for training [11, pp. 449–450]. Then, the ANN was trained using this data, which took a total of 26 hours to complete. In their evaluation they achieve a test data correlation of 97.55 % [11, p. 451].

Debar et al. [20] developed a different approach. They use the ANN to learn time series data, utilising *anomaly-based* detection to try to predict legitimate user behaviour and raise alarms if the assumptions did not hold [20, p. 243]. In their model, it is necessary to set a fixed parameter N denoting the number of past events used by the ANN for learning. This parameter cannot be changed without retraining the model [20, pp. 243–244]. Additionally, they formulate three hypotheses that need to hold for their approach to work well. Their first hypothesis is that user behaviour follows certain patterns with fixed start and end points that repeat regularly. Secondly, they assume that the user follows a certain logic when interacting with the system, which can be understood by an algorithm. Their last hypothesis is that there are correlations between audit data dimensions, allowing the neural network to take advantage of them when learning [20, pp. 241–242]. In their experiments, the ANN achieves prediction accuracies of 78.33 %–92.25 % for the first command submitted by a user, which rapidly decrease with following commands, reaching as low as 0.98 % for the third command [20, pp. 247–248].

Bayesian Networks

A simpler approach to network learning are *Bayesian Networks* (BN). Contrary to ANN, which train a set of nodes with no specific meaning on their own in a black-box way, a BN consists of nodes with defined meaning. We use the term *hypotheses nodes* for nodes that represent a hypothesis, like a specific intrusion into the system. With the term *observables nodes*, we refer to nodes representing observable behaviour of the system. The nodes of a BN are connected in a directed acyclic graph (DAG). Additionally, the probability of each node is saved. Each node is dependent on its own probability as well as that of its parents. With such a network, one can model certain states and their relationship to each other in a human-understandable way.

Kruegel et al. [60] describe the use of a BN for intrusion detection. They define one root *hypothesis node* that represents “anomalous” or “not anomalous”. Additionally, each possible dimension of the observed data is modelled as an *observable node*. The root node is connected to each of those child nodes. Edges between the child nodes represent the possible causal dependencies between them. Finally, additional variables are introduced as *observables nodes* with relationships to other *observables nodes* they may influence [60, pp. 17–18]. To increase the accuracy of the classification, different models are built for features of the observed events. An additional confidence node in each model is used to weigh the results of the models against each other. This relationship is modelled in a higher-level BN that connects the inputs and the different models, as well as their respective confidence level. In their evaluation, they observe an improvement of 20 percentage points in accuracy when comparing their BN to simple *rule-based* detection. They do note, however, that legitimate behaviour deviating from the training data was falsely flagged by the network [60, p. 22].

Support Vector Machines

Support Vector Machines (SVM) can be used for binary classification of data. They are based on the principle of mapping the training data to a feature space of higher dimension. In that space a hyperplane is created that is supposed to separate the instances with as much margin as possible. All future instances that need to be classified are then mapped to the same feature space and classified based on their position [40, pp. 19–20].

Mukkamala et al. [75] compare the performance of an ANN with SVM. They work with the KDD Cup 1999 dataset [53] which consists of labelled training data as well as test data. This data is pre-processed into machine-readable form, used to train the SVM, which is then tested [75, p. 1704]. The SVM show a comparable performance when detecting attacks, while requiring $\frac{1}{50}$ of the training time compared to ANN [75, p. 1706]. An important drawback they note is that SVM can only be used for binary classifications (attack or no attack), meaning there is no option to differentiate between attack categories or severities [75, p. 1707].

SVM work most efficiently when the training data is small, diverse and of high quality [44, p. 310]. To achieve this goal, Horng et al. [44] use the hierarchical clustering algorithm BIRCH [130]. With it they reduce the number of instances used for training the SVM [44, p. 307]. To be able to differentiate between different types of attacks, they train SVM classifiers for each type of attack separately. The finished IDS consists of a combination of all classifiers. In their evaluation, they found that for normal behaviour and attacks that had a sufficient number of training instances, detection rates were very high, ranging from 97.55 % to 99.29 %. For data with only few training instances, detection rates were significantly lower, resulting in a detection rate of only 19.73 % for U2R (unauthorized access to root privileges) attacks [44, p. 311].

Genetic Algorithms

The principle of *Genetic Algorithms* (GA) is based on evolution. First, a random group of so called *chromosomes* is created that encode possible solutions to the problem. Each of these is evaluated to determine its fitness based on the quality of its solution. Then, selection (duplication), recombination (crossover) and mutation operations are performed to arrive at a new generation. Chromosomes with higher fitness have a higher chance of reproducing, resulting in a more and more refined population [123, pp. 67–68].

Li [66] describes a classical application of a GA in the context of IDS. They encode rules classifying a certain network traffic pattern as anomalous or normal in chromosomes, evolve them, and evaluate them based on historical data [66, p. 4]. The goal of this process is to automatically create the best rules for detection based on known intrusions. This requires manual classification of the training data by experts before running the algorithm, meaning they employ *signature-based* detection.

A different application of GA is discussed by Sindhu et al. [101]. They try to solve the problem of data redundancy by employing a GA. Their approach is more indirect, utilising an ANN resembling a tree structure, referred to as a “neurotree”, that is trained with instances selected by the GA. To allow for easy evaluation, they encode the features of the KDD Cup 1999 dataset [53] to a simple binary form, encoding “feature is included” or “not included". Each chromosome represents a data instance that may be used for training the "neurotree". The fitness of the chromosomes is then evaluated by training the "neurotree" with the new population and calculating the precision and recall when classifying new instances. Additionally, data instances with fewer features are given favourable fitness [101, p. 131].

Artificial Immune Systems

Artificial Immune Systems (AIS) are based on the concept of how biological immune systems know to differentiate the "self" from the "other", thus utilising *anomaly-based* detection [19].

Forrest et al. [31] were among the first to describe the use of *negative selection* algorithms inspired by immune systems for anomaly detection. They discuss a basic concept of defining "self" for Unix processes. This definition is based on short sequences of system calls which are learned and stored over a period. These are system-specific and not generalisable, as every system has unique characteristics [31, p. 121]. After a database of normal behaviour has been built up, it can then be used for detecting anomalous behaviour. New incoming traces are matched against patterns in the database. Finally, the degree of mismatch is calculated and used as a basis for an abnormality score [31, p. 122].

This approach is utilised in a supervised learning IDS described by Hang et al. [37]. They detail a hybrid system based on a GA to learn data patterns and on an AIS for selecting new synthetically generated samples. This process is aimed at solving the problem of imbalanced data sets used for training a *machine learning* algorithm. It can be combined with various supervised learning techniques. The GA is based on co-evolution of non-interbreeding sub-populations that each evolve to a specific pattern of the normal data [37, p. 348]. Unlabelled examples are then assigned to the anomaly class by an AIS algorithm if they do not fit any of the normal class patterns. This process is repeated until the data sets are balanced [37, p. 349].

Swarm Intelligence

Another biologically-inspired class of algorithms are *Swarm Intelligence* (SI) systems. They take inspiration from the concept of self-organised, social insects such as ants. These do not follow a centralised authority, instead they work independently while collectively handling complex tasks.

Chung et al. [18] detail a hybrid system making use of different SI concepts. Additionally, to prepare the training data, they use *k-means* clustering (see page 16) to convert continuous variables to discrete ones. The cluster centroids are used as discrete values for the variables. Their feature selection algorithm uses these to try to find optimal training instances with the smallest number of features [18, p. 3017]. After the features have been selected, a variant of *particle swarm optimisation* with a *local search* strategy is employed to find an optimal solution. This extra step is added to prevent the algorithm from prematurely terminating with only a suboptimal solution [18, pp. 3017–3018]. After evaluating the developed rules according to their specificity and sensitivity in solving the problem, they are pruned, making them easier to understand and less likely to be overfitted [18, p. 3018]. In their evaluation, they show that their algorithm performs better than *simplified swarm optimisation* and *particle swarm optimisation* algorithms, with accuracies of 85.6 %–93.6 % [18, p. 3020].

3.4.3 Statistical techniques

Statistical intrusion detection assumes that the observable events can be mapped to a statistical model, with intrusions either occurring in low-probability areas of the model or being matched by a different model. *Statistical* models can be based on previous knowledge about the data, employing *signature-based* detection, or on learnings from historical data, employing *anomaly-based* detection.

Distribution-based

A basic *statistical* approach to intrusion detection is *distribution-based* detection. For this, a certain distribution of the data is assumed, either by applying assumed distributions or by evaluating historical data.

One of the most basic distributions is the *Gaussian distribution* (normal distribution). If the data can be assumed to be normally distributed, there are several methods to find outliers that may represent an intrusion. Grubbs [35] describes the *Grubb's test* for finding outliers in normally distributed data. They use the test criterion T_{n} that is defined as $T_{\mathrm{n}} = (x_{\mathrm{n}} - \bar{x})/s$, where $\bar{x}$ is the average of all n values and s is the estimated standard deviation [35, pp. 4–5].

To account for more complex distributions of the data, a combination of distributions can be assumed. Eskin [27] train such a *mixture model* with a *machine learning* algorithm. This hybrid approach assumes two distributions, the "majority" and the "anomalous" distribution [27, p. 256]. To detect intrusions, they calculate the likelihood of an element belonging to the respective distribution. For this, they compare the "change in log likelihood of the distribution if the element is removed from the majority distribution $M_{\mathrm{t\text{-}1}}$ and included with the anomalous distribution $A_{\mathrm{t\text{-}1}}$" [27, p. 257]. Their approach is limited by the need to retrain the *machine learning* model after every step. This is necessary because the sets of points used for calculating the distribution are changed by the system [27, p. 257].

Helman et al. [42] describe a basic form of *random process*-based detection. They formulate two processes, a process N for normal behaviour and a process M for intrusions. For each data instance, the probability of it being generated by each process is calculated and taken as a basis for classifying the instance as legitimate or not [42, p. 888]. The authors note that prior knowledge about the distribution of the input data is necessary for optimal detectors [42, p. 890]. Additionally, optimising the algorithm may not be computationally feasible for most scenarios [42, p. 894].

Kernel Density Estimation

Kernel Density Estimation (KDE), also known as parzen-window estimation, assumes that a probability density function exists that can explain the data. KDE is the process of trying to estimate that function [96].

Yeung et al. [128] use this approach in their IDS. They assume that functions such as the *Gaussian distribution* are too limited to explain real data. To overcome this limitation, they employ a KDE based on *Gaussian kernel functions*. They choose a non-parametric approach that allows them to model arbitrary distributions [128, p. 386]. To test if an instance is anomalous, they assume that it was generated from the model of normal data. In case that this probability is not greater than a threshold ψ, the instance is assumed to be anomalous [128, p. 386].

Regression analysis

Regression analysis studies the influence of one or more independent variables on a dependent variable. It can be used to estimate the expected value of a dependent variable for given independent variables [84].

One approach that is well suited for the problem of intrusion detection is *multinomial logistic regression*. It allows the dependent variable to have more than two possible categories compared to *binary logistic regression*. Instead of just classifying an instance as attack or no attack, this allows for differentiation between different attack types. Wang [112] use five categories, one for legitimate behaviour and four for various attack types. Additionally, they model the features of the input data as independent variables. All of them are converted to binary variables [112, p. 664]. Potential risk factors are preselected with thousands of simulations drawing random observations and calculating the significance level for the variables. After this, the final multinomial model is developed. To allow for a classification as legitimate or attack, thresholds for the outcomes are calculated for each possible attack based on the predicted means for each type [112, pp. 664–665].

3.4.4 Clustering-based techniques

While many algorithms focus on classifying individual instances based on their properties or relationships to all other instances, *clustering-based* techniques analyse them regarding the cluster they belong to. Some clustering algorithms exist that do not cluster all instances, allowing the selection of unclustered items as outliers. Most algorithms cluster all instances, with outliers instead being defined by their cluster (small or large, dense or sparse) or their relationship to their cluster (close to or far from the centroid) [13, pp. 27–28].

Distance-based

The most common form of *clustering* are *distance-based* algorithms. These are based on some distance metric for individual instances and cluster each element based on the cluster that fits it best.

A simple and well-known *distance-based clustering* algorithm is the *k-means* clustering. A set of k points is chosen as centroids for k clusters and data instances are assigned to the cluster whose centroid is closest to them, based on some distance metric. Jiang et al. extend this algorithm. Their adaptation allows instances that are far enough away from the nearest cluster centroid to form a new cluster. This enables the use of *k-means* clustering in large datasets. If the number of clusters exceeds a threshold, the two closest clusters are merged (single linkage merging) [48, pp. 694–694]. Then, a minimum spanning tree is constructed with the before-calculated cluster centroids. Its longest edge is removed, resulting in two subtrees. The members of the tree with less nodes are then regarded outliers [48, p. 696].

Selecting arbitrary cluster centres can result in suboptimal solutions [43, p. 100]. To overcome this, *k-medoids* clustering chooses instances from the data set as cluster centroids. Bolton et al. [6] describe an application of *k-medoids* clustering for fraud detection. First, each instance is assigned to a cluster, which they refer to as a "peer group". All members of the group are tracked over time and their behaviour is analysed. In case one instance behaves different from the rest of the peer group, it is selected for review [6, p. 9].

Density-based

Density-based algorithms assume that clusters of elements lie in dense regions. These algorithms do not assign every element to a cluster, instead they expand the cluster regions until a density threshold is reached [90].

One application of *density-based* clustering for *anomaly-based* detection is described by Oh et al. [79]. First, the audit data is clustered using DBSCAN [29]. The instance data as well as inter-transactional and intra-transactional properties are used as a basis for the algorithm. Instead of clustering whole instances, each group of features that forms a well-defined cluster is clustered separately. This results in one instance belonging to multiple clusters [79, pp. 599–601]. The result of the clustering is stored in a profile for each data dimension of the instances. Existing profiles can then be compared with new transactions of the user. Both the internal distance of the new instance compared to its cluster, as well as the ratio difference compared to other clusters are used to calculate the anomaly probability. The thresholds for each measure can be decided individually based on historical data [79, pp. 602–605].

Grid-based

Instead of starting with individual instances, *grid-based* clustering starts with dividing the input data into a grid of cells of even size. Cells that contain most entries relative to the rest are split. Those that lie under a minimum threshold of entries are discarded. Groups of adjacent and dense cells are then defined as clusters. This clustering process is faster than traditional clustering approaches [80].

Leung et al. [64] developed a hybrid clustering method for *anomaly-based* detection. First, they cluster the training instances with their algorithm. This happens by first forming a grid and finding frequent item sets within it [64, pp. 335–337]. Then, in a separate step, the individual records are assigned several frequent item sets, depending on their attributes. This data is then mined to cluster the elements [64, p. 337]. To prevent overfitting, a count back method is used that prunes all sets that do not receive enough support [64, p. 338]. Finally, duplicate clusters are removed [64, p. 339]. The remaining clusters are assumed to represent normal behaviour. For incoming instances, their membership in these clusters is tested. Those instances that lie outside all clusters are defined as anomalies [64, p. 340].

Hierarchical

In general, clustering algorithms find a set of clusters that are all assumed to be equal. *Hierarchical* clustering expands this with a hierarchy of clusterings, from a 1-cluster encompassing all elements up to the single element level.

Loureiro et al. [71] describe the use of *hierarchical* clustering for outlier detection. After the dataset has been clustered with a *hierarchical* clustering algorithm, the resulting hierarchy is split at a specific level. This results in a predefined number of clusters. Outliers are assumed to be those instances that belong to the smallest clusters [71, pp. 4–5]. In their work, they also compare different distance metrics and clustering algorithms. When comparing the *Euclidean* and *Canberra* distance functions, they find that using the *Canberra* function significantly improves results [71, p. 8].

3.4.5 State-based techniques

State-based techniques are based on the idea of modelling and tracking either the state of the observed system, or of the intrusion detection. These IDS can either utilise *anomaly-based* detection, by converting historical data into models, or *signature-based* detection, by modelling attacks or legitimate behaviour directly.

Automata

Techniques based on *automata* allow for basic modelling of the system state or tracking of multi-step attacks.

One well researched *automata*-based system is STAT [46, 109, 110], short for "state transition analysis tool". It is based on two assumptions. First, that attackers have some form of access to the system, and second, that they gain new capabilities through their attack [46, p. 184]. Intrusions are transitions from an initial state to a compromised state. The differentiating steps required to complete the attack, referred to as *signature actions*, are identified and modelled [46, pp. 184–185]. This limits their approach to intrusions that can be identified by the system state [46, p. 187]. STAT follows a simple procedure. After collecting the audit data, it is pre-processed, analysed by the intrusion detection components and presented to the security officer [46, p. 188].

Sekar et al. [97] describe a hybrid system that combines the advantages of *automata* with a *statistical*, *anomaly-based* detection. Their approach focuses on network intrusions. First, automata are manually built from specifications such as the internet protocol (IP). For every received packet, a new automaton instance is created based on the respective protocol. All existing machines may transition based on the packet [97, p. 266]. The statistical component is used in parallel to track transition frequencies, common values for certain states, as well as variable state distributions. This can be done either for all instances, or for a subset [97, p. 266].

To generalize the *automata*-based approach to systems where specifications do not exist or are incomplete, Wagner et al. [111] propose a system that automatically derives models with static analysis of software. They focus on interactions of the program with the system, observable in its system call trace. This simplifies monitoring the program. Based on the application source code, they build a static model and reduce it to its most important parts [111, pp. 157–158]. To model the program behaviour, they discuss multiple possible models. Their trivial model checks whether a system call could have been made by the program or not. Their other approaches range from the basic "callgraph model" that consists of a control-flow graph, to the "abstract stack model" that additionally monitors the call stack to prevent impossible steps [111, pp. 158–161].

Markov chains

Like automata, *Markov chains* model states and transitions. But instead of input symbols being consumed when transitioning states, the transitions correspond to probabilities for neighbouring states based only on the current state [76].

Ye et al. [127] use *Markov chains* to model the system behaviour, similar to automaton-based systems. While those can only allow or disallow certain patterns of inputs to appear in sequence, this system can differentiate further. First, possible events are modelled as states for the Markov chain model. Observing normal system behaviour, the transitions and their probabilities are learned [127, pp. 117–118]. This model is then used while monitoring the system to detect intrusions. Sequences of states are tracked, and their probability of occurrence is calculated as $\log(P(X_{\text{t-N+1}}, \ldots, X_{\text{N}}))$ [127, p. 119]. The resulting score corresponds to the probability that the sequence was legitimate. Ye et al. note that previously unseen patterns result in probabilities of zero. While this can be problematic for intrusion detection, they expect legitimate behaviour to follow previously seen patterns, and attacks to be more likely to be previously unseen [127, p. 119].

Coloured Petri nets

An extension of Petri nets, *Coloured Petri nets* (CP-nets) allow for a more powerful description of concurrent systems. Instead of requiring duplicates of entire subnets to differentiate processes, explicit node information can be encoded by functions attached to the arcs of the nets [47, p. 249].

Kumar et al. [61] expand on the state transition analysis approach [46] (see Section 3.4.5 on page 18). In their model, guards represent the signature context and vertices the system states. Like in the state transition analysis approach, the graph models transitions from an initial to a compromised state [61, p. 2]. Instead of just looking at directly subsequent events, their approach can handle more complex, parallel interactions with the system, like multiple simultaneous logins by the same user [61, p. 6]. Additionally, they can model partial orders as well as token duplication and merging without losing any information [61, p. 8]. While their approach can cover advanced, long-term, multi-step attacks, this leads to a big computation time and space overhead in normal use. That is because partial matches of signatures can happen in numerous scenarios and are tracked until the possibility of an intrusion has been refuted [61, p. 10].

3.4.6 Hybrid systems

Most systems cannot be clearly defined as one specific type of IDS. Ideas from multiple areas are often combined to create better solutions.

We define hybrid systems as conscious combinations of multiple techniques. Theoretically, almost all discussed forms may be combined in some form. But this may not always result in a better IDS. The most interesting hybrid systems combine those techniques that could ideally mitigate the shortcomings of the respective other technique to improve results or speed up the computation.

We have already discussed some examples of hybrid systems before. They were grouped in those categories that represent their core capability, with additional functionality added to it in form of a different technique. In the following sections, we examine what we see as some of the most interesting uses of hybrid systems.

Rule generation

Manual rule creation for *rule-based* systems (see Section 3.4.1 on page 10) can be time intensive and may not lead to adequate results. The rule generation process for these systems can be automated with algorithms.

Gomez et al. [34] describe the use of a GA for rule generation. This allows immediate verification of the quality of the rule and automates the explorative testing that is necessary for manually finding new rules.

Data labelling for supervised learning

In many scenarios, labelled training data for *machine learning* algorithms is not available. As unsupervised learning may not be sufficient, some form of automatic labelling can be used to allow for semi-supervised or supervised learning.

Portnoy et al. [85] describe the use of *hierarchical* single-linkage clustering for automated labelling of training data. To allow for this, they base their work on two assumptions. First, they assume that an overwhelming number of instances in the data is normal (> 98 %) [85, p. 6]. Second, they assume that similar instances are clustered together. If these assumptions hold, their approach may offer a promising solution to training with unlabelled data.

Imbalanced data sets for machine learning

Most supervised *machine learning* techniques require a data set that is balanced between data points of different classes. Real-world datasets are naturally imbalanced though, resulting in difficulties when trying to train a ML algorithm on them.

Hang et al. [37] describe how GA and AIS can be combined to generate more anomalous samples (see Section 3.4.2). The GA learns patterns in the data and new instances are selected by the AIS until the dataset is balanced.

3.5 Gap analysis of existing research

When analysing prior work on intrusion detection, we found that the field is widely researched. Interestingly, we could not find a comprehensive overview work and up-to-date taxonomy of IDS. Additionally, we noticed that existing research often follows an isolated approach. To the best of our knowledge, there exists no satisfactory option for comparing IDS in terms of effectiveness or detection accuracy.

We identified these issues as important to work on. Accordingly, we define the development of a practical solution for evaluating various IDS as our goal.

4 Creating the testbed

Our objective is to assess the adequacy of various types of IDS for the automotive real-world use case provided by our industry partner. As we cannot use actual data from our industry partner due to privacy concerns, we require an artificial dataset that fits our use case.

First, we describe the use case in Section 4.1. Following that, we discuss options for obtaining test data in Section 4.2 and conclude that a testbed is required. In Section 4.3, we elaborate on the concept of that testbed and Section 4.4 gives an overview of the implementation of our testbed.

We make three assumptions in this chapter, prefixed by *Assumption.* In Section 4.5, we give an overview and discuss them.

4.1 Use case: Server-side detection of compromised clients

Our use case is based on a client-server architecture utilising microservices [65] as used by our industry partner in the automotive industry. Multiple clients in the hands of customers communicate with a central, manufacturer-controlled server, consisting of multiple microservices and a logging component. Each client request is handled by the responsible microservice, which in turn communicates to a logging component that stores information about the request in a log store (see Figure 4.1).

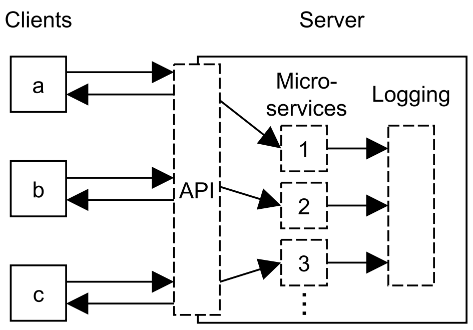


Figure 4.1: *Our use case: The clients a, b and c communicate with the server and its microservices. Those in turn log the requests in a logging component. Arrows represent communication. Components are dashed.*

Clients

The server is our frame of reference, and we view the clients from an outside perspective as data generators that regularly send requests to different server microservices.

Assumption 1. Normal and compromised clients can be differentiated based on differences in their communication with the server.

This assumption is discussed in Section 4.5.1 on page 40.

Microservices

This thesis focuses on detecting intrusions from log data generated by the logging component. Much like the clients, we view the microservices as black boxes. They receive request data, process it and send data based on those requests to the logging component.

Assumption 2. The microservices in our use case do not alter the data they receive, for example by condensing or extending it.

This assumption is discussed in Section 4.5.2 on page 41.

Logging component

The data that is available for intrusion detection is recorded by the logging component. In our industry partner's implementation, it is server- and implementation-centric. Hence, the information stored consists of details only known to the server and specific to the implementation of each microservice. We cannot discuss the actual log format used by our partner, so we have reduced it to features contained in most log entries that we deem sensible for the use case. These are as follows:

Attribute	**Range of values**	**Description**
Request time	1–*Int Max*[*]	UNIX Epoch time[†]
Client identifier	$[A-Z1-9]\{7\}$	Alphanumeric
Microservice name	$[A-Z]\{1,10\}$	Alphabetical
Client position	$[0-9]+.[0-9]+, [0-9]+.[0-9]+$	Coordinates
Data sent or received	*No constraints*	Microservice dependent

[*] See Trybulec [107]

[†] Seconds since 00:00:00 UTC, 1 January 1970

Table 4.1: *Log features similar to our industry partner's implementation, defined using regular expressions.*

There are numerous other features that we consciously left out. Most of these were omitted because they appear very seldom in the log data and often contain only unstructured or undefined data. These application-specific features go beyond the scope of our work. Some features were skipped because they redundantly mirror others in content or because their values are constants. Additionally, we left out a small number of features because we consider them non-essential. Those include microservice exceptions, various version numbers and data regarding the logging component.

4.2 Obtaining test data for evaluation

We want to evaluate different IDS for our use case, for which we need realistic test data. Due to the privacy-sensitive nature of the data being collected, it is impossible to use actual data from our industry partner. We therefore evaluate possible alternatives. Either we can use existing and freely available datasets, or we have to artificially generate data.

4.2.1 Using existing data

Intuitively, the easier option would be to use an existing dataset. The requirements for such a dataset are as follows:

1. Fitting to our use case
   a) Containing intrusions: Intrusion detection, contrary to simple anomaly detection, implies some form of intrusion that can be detected.
   b) No pre-processed or aggregate features: Real-time detection entails the inability to pre-process or aggregate data with information from future connections before detection.
2. Up-to-date (after 2014): Outdated data is harder to map to the microservice architecture we are working with. We consider 2014 the year that microservices started gaining traction [65].
3. Well-formed (machine-readable, labelled): To allow for sensible evaluation, the data needs to be labelled.

In the following, we discuss possible datasets that we found during our literature survey.

The KDD Cup 1999 dataset

An often-used dataset for the evaluation of IDS is the KDD Cup 1999 dataset [53]. It is based on the DARPA 1999 dataset [70] and there exist updated versions of the same data, like the NSL-KDD dataset [77]. Among the papers that we surveyed, the following make use of this data: [18, 27, 28, 34, 44, 59, 60, 64, 66, 67, 69, 75, 79, 85, 97, 98, 101, 108, 110, 112, 127, 128]

These datasets do not fulfil multiple of our requirements, making them unusable in our case. The most obvious is that the data is severely outdated (req. 2), being over 18 years old as of now. Also, while the original set consists of multiple gigabytes of data, prior analyses have found that most of the data is redundant [105]. The NSL-KDD set resolves this, but that means it only consists of less than 22 % of the entries of the original set [77]. Finally, most features in these datasets are pre-processed or aggregated (req. 1b). In the group of features “suggested by domain knowledge” [54], many aggregate data from all connections by the client. Additionally, all features that have been “computed using a two-second time window” [54] are, as the name suggests, pre-processed and aggregated. Such features hide raw information that would be available for real-time detection.

These factors make any dataset based on the DARPA 1999 dataset unsuitable for our evaluation.

Other datasets used in related work

All authors discussing intrusion detection specifically did not make their data available. Either they skipped evaluation, meaning they needed no data, or they used data that they could not publish.

In research discussing *anomaly-based* detection, a field which allows more general-purpose data to be used for evaluation, we found some researchers using publicly available datasets. Those are the *Iris flower set* [24] and a sugar-cane breeding database [49] (both used in [48]), *NHL* player statistics (used in [56, 57]), *NBA* player statistics (used in [88]) or *INTRASTAT* trade data (used in [71, 102]).

These datasets do not fit our use case, making them unsuitable for our purpose. Additionally, as our work focuses not just on *anomaly-based* detection but includes *signature-based* detection, we require the dataset to include some form of well-defined intrusion that can be detected (req. 1a).

Freely available data

In addition to these academic datasets, we also evaluated freely available data from online sources.

Among the datasets we considered are the *Wikimedia data dumps* [124]. Their contents are unrelated to our use case and the data does not conform to our requirement of being well-formed (req. 3). For each language, it consists of a single-file XML dump of all pages in the corresponding wiki. Using this data would require extensive pre-processing and not guarantee sensible results, as labels are missing.

We also considered the *Tor Metrics data* [106]. These datasets consist of pre-aggregated data that hides raw information, which makes them unfit for the intended real-time use of our system (req. 1b).

4.2.2 Generating test data

As we found no appropriate dataset for our use case, we conclude that it is necessary to generate test data ourselves.

Naïve approach

The easiest way to generate data is to randomly generate it based on certain parameters. Those may include the type of data being generated, or content patterns. Our system aims for real-time detection, but that could be emulated by feeding single entries one by one into the IDS.

We aim for emulating a realistic, complex system. Within clients, multiple components interact and depend on each other for their computations. Additionally, the clients' communication with the server can, to a certain extent, be unpredictable in content and sequence. It depends on implementation, connection speed, server priority and system performance.

Assumption 3. The unpredictability of the interoperation of components, clients and a server cannot be adequately emulated with basic data generation.

This assumption is discussed in Section 4.5.3 on page 41. With Assumption 3 we can conclude that naïve generation of random data is not sufficient in our case.

Testbed approach

A complete testbed is used to generate realistic historical data and for live testing of the IDS. Such a testbed is based on sophisticated simulations of clients from our use case that we try to model. It is built to comply to our requirements of generating up-to-date, well-formed and sufficiently large datasets. We conclude that this approach is most promising for our scenario.

Accordingly, we have created a testbed that simulates the real-world use case. In the following sections, we describe its concept and implementation.

4.3 Testbed concept

For comprehensive evaluation of IDS for the real-world use case (described in Section 4.1), we design a testbed. It is aimed at modelling that use case as closely as possible. Hence, it is simulating a client-server architecture with individual clients made up of a unique layout of different components. Each client makes use of inter-component communication and sends requests to the server. The server stub only consists of a simulated logging component that generates the log data used for intrusion detection based on incoming requests from the clients. Additionally, the server allows adding on the IDS for live detection (see Figure 4.2). Because the microservices in our use case are assumed to not alter the data they receive (see Assumption 2 on page 24), they are omitted.

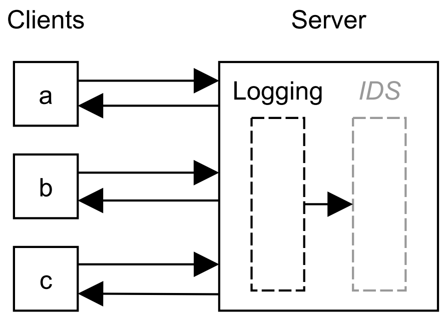


Figure 4.2: *The testbed concept: The simulated clients a, b and c communicate with a server stub that only consists of a simulated logging component. Denoted in grey is the potential for adding an IDS. Arrows represent communication. Components are dashed.*

4.3.1 Clients

As described before, each client consists of multiple components. These produce and consume data and are linked to a central communication unit that handles sending requests to the server (see Figure 4.3).

There are two types of components we can derive from real-world clients.

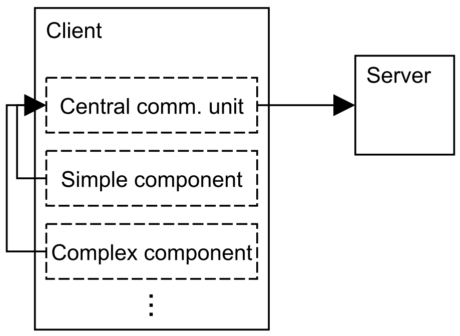


Figure 4.3: *The basic architecture of each client. Multiple simple and complex components are combined and send data to one central communication unit. That in turn sends request to the server. Arrows represent communication. Components are dashed.*

Simple components

Simple components generate various types of data periodically. One example is a temperature sensor. This data is then used as a basis for requests to the server.

In our model, we represent each such component with a random number generator based on a probability distribution, such as the *normal distribution*. Periodically, it generates new data and sends that to the central communication unit. We refer to this component type as “Data generator”.

Complex components

More complex components in our scenario use information about the state of the client, like its position, sensor data or information about its surroundings, to allow the central communication unit to make more sophisticated requests to the server. One such example may be a request about the nearest point of interest (POI) based on the current position.

Our solution to modelling such a component is what we refer to as “2D simulator” (see Figure 4.4). It consists of a simulated environment in which an independent unit representing the client can move around (denoted in the Figure as a black circle with a white “C” on it). This environment can have any defined size and different background colours. The colours represent the interpretation of certain positions by the unit moving around in the environment. A movement generator creates random movement commands for the unit. Periodically, the unit publishes information about its environment and position. This data is used both by the movement generator to adapt its movement commands to the environment as well as by the central communication unit to send requests to the server.

Central communication unit

Finally, the central unit responsible for collecting all generated data as well as sending requests to the server can be thought of as a simple pipe [74]. It does only basic data transformations and then sends the result as a request to the server.

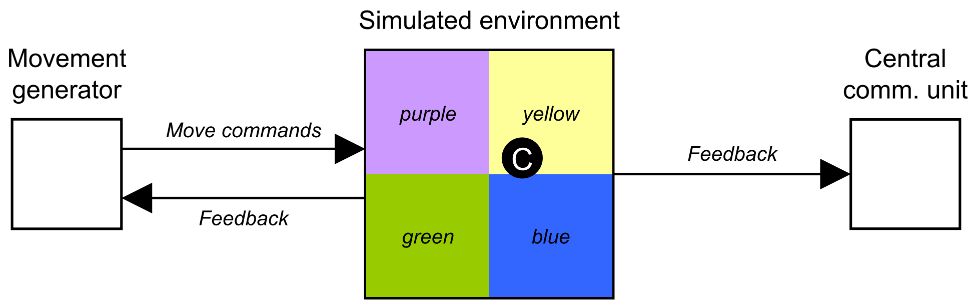


Figure 4.4: *A 2D simulator component. The movement generator sends random movement commands to the simulated environment and its contained simulated unit, represented by a black circle with a white "C" on it. The simulation sends feedback to both the generator and the communication unit. Arrows represent communication.*

All these are linked together as seen in Figure 4.3, with data generating components sending data to the communication unit, which in turn sends that data off to the server as requests.

4.3.2 Intrusions

We define intrusions as modifications to the behaviour of a system without the manufacturer's knowledge. That includes unauthorised access or fraud, but also software bugs (see Chapter 2 on page 3).

In our testbed, we conceptualise intrusions as modifications to individual components that lead to differences in their communication. That can mean erroneous values being generated by data generator components, or intentional bugs in the colour readings of the 2D simulator.

These intrusions can be combined in various ways to form intrusion scenarios. One possible scenario is a client consisting of mostly normal components, except for one compromised data generator and a 2D simulator with modified reaction patterns. This leads to the unchanged communication unit to send off different requests to the server compared to a normal client (see Figure 4.5).

4.3.3 Server

The server we model consists of multiple microservices that are being called by clients and can respond individually. Each of the requests is stored in a log for later inspection. As the server functionality is not part of our work, it is omitted. That means the testbed server consists only of the logging component (see also Figure 4.2 on page 27).

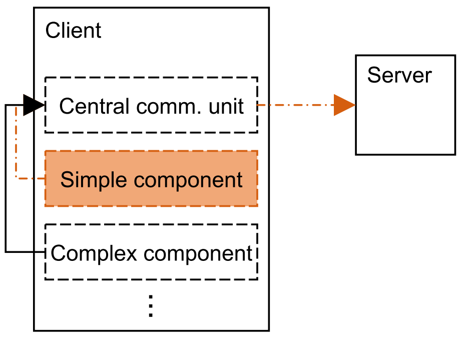
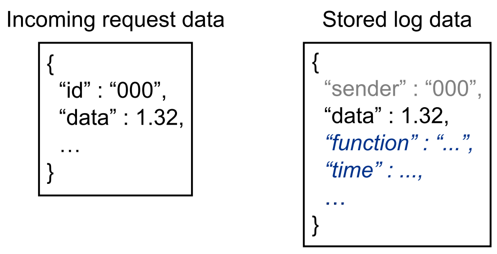


Figure 4.5: *A compromised client. Denoted in orange (shaded) is a compromised component. Its communication differs from normal components, denoted in orange (dash-dotted). Arrows represent communication. Components are dashed.*

Logging component

This unit adds some information such as the current time and an identifier to the request data, transforms it to fit our expected format and stores the result in a log, as seen exemplary in Figure 4.6. Denoted in grey is data that may be transformed, for example by renaming fields. Denoted in blue (italics) is data that may be added, like a "time" field for storing when the request reached the server.



```
{
  "id" : "000",
  "data" : 1.32,
  …
}
```

```
{
  "sender" : "000",
  "data" : 1.32,
  "function" : "…",
  "time" : …,
  …
}
```

Figure 4.6: *An exemplary conversion of a client request to a stored log entry by the simulated logging component. Denoted in grey is transformed data, in blue (italics) added data.*

Server modes

We aim for evaluating real-time IDS. Still, our testbed also allows off-line IDS to be evaluated. The server can be started in different modes, depending on what kind of detection is executed.

The server can run in *store mode*, meaning it stores all data that has been processed by the logging component on disk. Additionally or exclusively, it can run in *detection mode*, feeding an IDS every incoming request directly, after it has been processed, for real-time detection.

4.4 Testbed implementation

The testbed described before was implemented in form of multiple programs, scripts and tools. In following, we describe how they are built and how to use them.

4.4.1 Client orchestration

We want to orchestrate an arbitrary number of clients dynamically. That requires the following steps:

1. Definition of clients
2. Dynamically starting clients
3. Relaying of communication

This is achieved using the "Robot Operating System" (ROS) [86] (see Chapter 2 on page 3). In the following, we describe how ROS allows us to complete these steps.

Definition of clients

Each client should have an individual identifier, behaviour and layout, which refers to the components it contains (see Section 4.3.1).

We use a ROS *launch file* to define the layout of our clients. This is a special XML file containing elements expected by ROS (see Listing 4.1). A client in our use case is represented by a *group* in the *launch file*, with its *namespace* representing the identifier of the client. Each of its components is an individual Python program, represented by a *node*. We package those programs in a client by subordinating them to its *group*.

```
<launch>
  <group ns="CLIENT1">
    <node name="gauss" ... type="data_generator.py"
      args="generate gaussian" />
    ...
  </group>
  ...
</launch>
```

Listing 4.1: *An exemplary ROS* launch file.

In our example (see Listing 4.1), we find all elements of the *group* of a client. Its *namespace* ("ns"), which serves as its identifier, is "CLIENT1". The group contains a *node*, which references the Python program `data_generator.py`. This program is given the arguments ("args") "generate gaussian".

This enables all our requirements: We can define an individual layout of components by simply combining *nodes* in the *group*. Their behaviour can be individually defined through arguments and the client has a unique identifier.

Automatic launch file creation: We have built a command line tool to aid in the creation of large, randomized *launch files*. This tool automatically creates a file in XML and *launch file* syntax. It allows the user to specify multiple command line options, including: The number of clients to create, if and how many clients should be intruded, and the difficulty level.

Dynamically starting clients

We make use of the ROS tool `roslaunch` to start all programs described in the *launch file*. This tool takes care of all required steps to set up the ROS infrastructure and passes command line arguments to the respective programs.

Relaying of communication

Finally, relaying communication is also made easy with ROS. It makes use of a publish-subscribe architecture and handles connecting publishers to subscribers through the ROS *master*.

In the following, we describe the individual components of the clients and give an overview of how they publish and subscribe to ROS topics.

4.4.2 Clients

The clients consist of multiple components, as we described before. Each of those is implemented as a separate program.

Data generators

The data generators representing simple components are, at their most basic, two classes and one program, written in Python (see Figure 4.7).

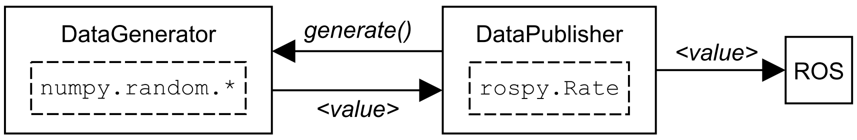


Figure 4.7: *The basic implementation of a data generator. The* DataGenerator *generates values when prompted by the* DataPublisher. *Arrows represent communication. Components are dashed.*

The *DataGenerator* class is a container for a *NumPy* [50] random number generator. We are using ten generators based on continuous probability distributions from that library. That includes more common choices like the *normal* and the *uniform* distribution, and less common options that offered the same API in *NumPy*, such as the *Rayleigh* distribution (see Table 4.2).

For the *Beta Prime*, *von Mises*, *Wald* and *Weibull* distributions, no default parameters are provided by *NumPy*. We chose their parameters to be similar to those of the other chosen distributions, aiming for comparable means and distribution interval spans.

Distribution	***NumPy* function***	**Parameters**	**Distribution interval†**	**Mean**
Normal [118]	`normal`	$\mu = 0, \sigma = 1$‡	$[-3.09, 3.09]$	0
Gumbel [114]	`gumbel`	$\alpha = 0, \beta = 1$‡	$[-6.91, 1.93]$	−0.58
Laplace [116]	`laplace`	$\mu = 0, \beta = 1$‡	$[-6.21, 6.21]$	0
Logistic [117]	`logistic`	$\mu = 0, \beta = 1$‡	$[-6.91, 6.91]$	0
Beta Prime [113]	`pareto`	$p = 1, q = 2$	$[0.00, 30.62]$	1
Rayleigh [119]	`rayleigh`	$\sigma = 1$‡	$[0.04, 3.72]$	1.25
Uniform [120]	`uniform`	$min = 0, max = 1$‡	$[0, 1)$	0.5
von Mises [121]	`vonmises`	$\mu = 0, \kappa = 1$	$[-3.12, 3.12]$	0.00
Wald [115]	`wald`	$\mu = 1, \lambda = 1$	$[0.08, 8.35]$	1
Weibull [122]	`weibull`	$\alpha = 5, \beta = 1$	$[0.25, 1.47]$	0.92

* Function in the `numpy.random` module.

† Interval containing approximately 99.8 % of samples.

‡ Default parameters.

Table 4.2: *The random number generators we use with the* DataGenerator*. Values are rounded to two decimal places.*

The *DataPublisher* class handles publishing new values to ROS. It polls the *generate()* method of the *DataGenerator* every 0.1 seconds per default to retrieve a new value, utilising a `rospy.Rate` object for rate limiting. Finally, it publishes the generated value to a unique ROS topic subordinated to the *namespace* of its client.

```
<node name="gauss" pkg="client"
  type="data_generator.py" args="generate gaussian" />
```

Listing 4.2: *A full* node *for a data generator.*

2D simulator

The 2D simulator representing complex components is based on the ROS *turtlesim* [30]. We reimplemented it in Python without a graphical user interface. We refer to its components as *Simulator*, *Frame* and *Unit* (see Figure 4.8).

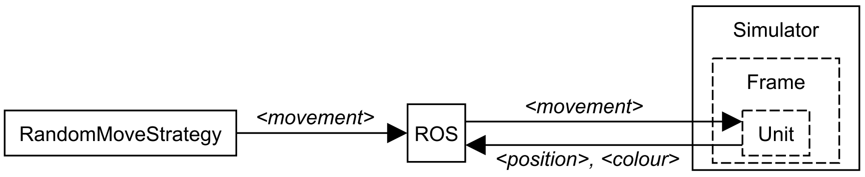


Figure 4.8: *The basic implementation of the 2D simulator. The* RandomMoveStrategy *generates random movement commands for the* Unit*, which publishes its current position and colour reading to ROS. Arrows represent communication. Components are dashed.*

The main class, *Simulator*, is started from the *launch file*. It initialises a *Frame*, which represents the simulated environment (see Figure 4.4 on page 29). The *Frame* initialises a *Unit* and spawns it in the environment.

A *RandomMoveStrategy* is launched from the *launch file* in the same *group* as the *Simulator*. It publishes movement commands to ROS which are read by the *Unit*. That moves according to the commands and publishes its position, speed and the current colour reading (see Figure 4.9) to ROS every 16 milliseconds. The colour reading consists of an RGB (red, green and blue) colour, with each value $r, g, b \in [0, 255]$.

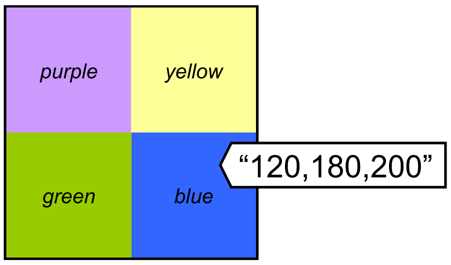


Figure 4.9: *An exemplary colour reading by the unit. The label contains the message that the unit sends at that location.*

The colours represent the interpretation of certain positions by the unit moving around in the environment. For simplicity, we have only defined four arbitrary colour options for the simulator background (see Table 4.3).

Colour	**RGB**
Purple	150, 140, 200
Yellow	170, 250, 140
Green	120, 180, 130
Blue	120, 180, 200

Table 4.3: *The colour options for the simulator background.*

Secondly, requests are also regularly sent based on the position of the unit. They are inspired by the real-world use case of our industry partner and resemble some of the requests that clients in their scenario make. There are three request types currently implemented (see Table 4.4).

Request type	**Data sent**
Country code	(x, y) : The current position of the unit
Point of interest	(x, y, t) with $t \in T : T$ is the set of all point of interest types
Route	$(x, y, x_t, y_t) : x_t \neq x \wedge y_t \neq y : (x_t, y_t)$ is the target position

Table 4.4: *The positional requests sent from the 2D simulator.*

A “Country code” request is a simple inquiry for the associated country code for the current position. Similarly, a “Point of interest” (POI) request is an inquiry for a specific POI type at the current position. Finally, the “Route” request is an inquiry for routing to a specified target position (x_t, y_t).

Finally, the unit has basic movement intelligence. Certain zones can be marked as “illegal” and the unit will try to avoid those.

To allow for reproducibility, the *RandomMoveStrategy* can be seeded, and its seed as well as its movement intelligence are specified in the *launch file* (see Listing 4.3).

```
<group ns="sim">
  <node name="simulator" type="sim.py" pkg="client"/>
  <node name="mover" type="random_mover.py"
    args="--seed 210 --intelligence return" pkg="client"/>
</group>
```

Listing 4.3: *A full* group *for a 2D simulator.*

Central communication unit

The central communication unit is responsible for collecting all data generated by a client and sending requests to the server accordingly (see Figure 4.3 on page 28). In practice that means subscribing to all topics of its client through ROS to ensure it receives the data that the components send (see Figure 4.10).

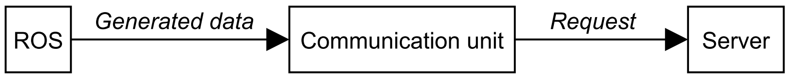


Figure 4.10: *The basic implementation of the communication unit. It subscribes to all topics of its client and receives the corresponding messages from ROS. Those are then converted to requests for the server. Arrows represent communication.*

To ensure the unit knows about all components its client contains, the corresponding ROS topics are passed as arguments (see Listing 4.4). Additionally, it receives the identifier of its client (here “CLIENT1”).

```
<node name="com" type="communication_unit.py" pkg="client"
  args="CLIENT1 --topics generator_1 generator_2" />
```

Listing 4.4: *A full* node *for a communication unit. It receives the client identifier (“CLIENT1”) as well as the ROS topics it subscribes to (e.g. “generator_1”) as arguments.*

Labelling: Each communication unit is informed by the components of its client about the nature of the data they send. That means it knows which requests are intruded and the type of intrusion that applies to the data.

To enable sensible evaluation, the communication unit can add this information to the requests it sends to the server, which in turn can then label the request data.

4.4.3 Intrusions

The reason why we simulate this environment is that we intend to evaluate a real-time remote IDS. We try to model intrusions for this purpose.

Intrusions into components can have varying difficulty levels. The rationale for these is that we aim for increasing the identifiability and with it the information entropy of the generated data compared to the expected data for easier intrusion levels. With this, we aim for adjusting the difficulty of detecting a compromised client. We describe this in more detail in the following.

Data generators

We currently model two types of intrusions for data generators in our system. The first is the *off-value generation*.

Given the distribution interval $R := [r_{min}, r_{max}]$ containing 99.8 % of samples of the underlying distribution function (see Table 4.2 on page 33), the mean m of the distribution, and its spans $s_{left} := m - r_{min}$, $s_{right} := r_{max} - m$ (see Figure 4.11). Further given a factor f that we define (see Table 4.5).

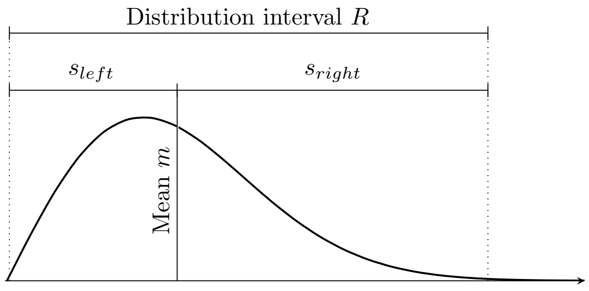


Figure 4.11: *Explanation of the distribution interval R and the spans s_{left} and s_{right}.*

Instead of the normal value $v \in R$, a value v_c is broadcast:

$$v_c \in \{m - s_{left} \cdot f, m + s_{right} \cdot f\}$$

Depending on the intrusion level, the factor f differs (see Table 4.5). The smaller this factor, the higher the probability that the value could have been generated by the distribution function.

Intrusion level	**Factor f**
Easy	5
Medium	1.5
Hard	1.001

Table 4.5: *Factor f per intrusion level.*

The second intrusion type, the *significant error generation*, is based on a generated value from the underlying number generator.

Given $m, s_{left}, s_{right}, v$ (see above) and a factor f (see Table 4.5), an erroneous value v_e is broadcast:

$$v_e = \begin{cases} m + s_{right} \cdot f + v^2 & v \geq m \\ m - (s_{left} \cdot f + v^2) & otherwise \end{cases}$$

2D simulator

The simulator allows for multiple types of intrusions. Firstly, the environment background has different colours. This can be changed to represent erroneous readings (see Figure 4.12). In both situations the unit sends information about its position. Because the environment is modified, the colour it reads differs, representing an intrusion.

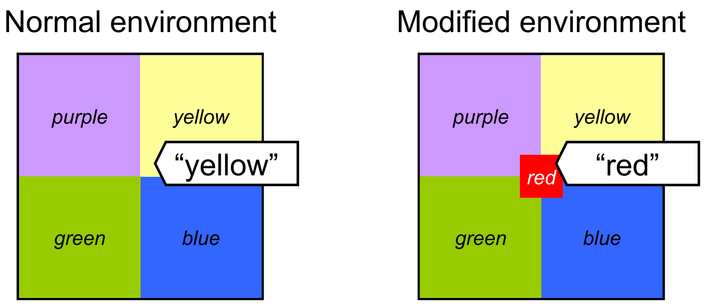


Figure 4.12: *A normal and a modified environment. The labels contain the message corresponding to the colour that the unit sends at the location both labels point to.*

Depending on the intrusion level, this area can be increased in size (see Table 4.6) and its colour can be modified (see Table 4.7). For different intrusion levels, we aim for varying the detectability of the erroneous colour compared to the legal colours. Hence, we need a similarity measure to derive a similarity relationship between different colours.

We define our colours in code as RGB (red, green, blue) with $r, g, b \in [0, 255]$. The background has four different legal colour options. We chose *purple* (150, 140, 200), *yellow* (170, 250, 140), *green* (120, 180, 130) and *blue* (120, 180, 200). The reasoning for our choices is following.

If we imagine the colours as points in a three-dimensional space, we can calculate the distance between two points p and q using the *Euclidean distance*:

$$d_{Euclid}(p, q) = \sqrt{(p_r - q_r)^2 + (p_g - q_g)^2 + (p_b - q_b)^2}$$

For our following calculations, we scale the values of the colour dimensions down to $r_s, g_s, b_s \in [0, 1]$. With the distance formula, we can derive the minimum and maximum distance of two points in our space: $d_{Euclid} \in [0, \sqrt{3}]$.

The goal when choosing our legal colours was to have moderately similar colours that can still be differentiated from each other. We define that as having a distance of $d \in [0.15, 0.5]$.

This holds true for our four legal colours. Their distances all lie between $d_{min} \approx 0.196$ and $d_{max} \approx 0.498$. We also calculate their average distance and arrive at $d_{avg} \approx 0.256$.

We aim for increasing the distance of the erroneous colour to all legal colours for easier intrusion levels compared to the average distance d_{avg}. Simultaneously, it should never fall below the maximum distance d_{max} to ensure that the values are anomalous. With this requirement, we have defined a colour for each intrusion level (see Table 4.7).

Intrusion level	**Area size**
Easy	40 %
Medium	20 %
Hard	5 %

Table 4.6: *Size of erroneous area per level, relative to the total environment size.*

Intrusion level	**Erroneous colour (RGB)**	**Average distance**
Easy	255, 0, 0	1.103
Medium	200, 50, 50	0.774
Hard	170, 80, 80	0.590

Table 4.7: *Erroneous colour per level. The "average distance" column lists the average* Euclidean distance *of the colour to all legal colours (rounded).*

Secondly, the unit employs basic movement intelligence. It reacts to its surroundings and can alter its movement based on it. Certain zones can be marked as "illegal" and the unit can react to reaching those. This reaction can be modified: Instead of trying to avoid the illegal zone, the unit stays in it and continuously sends its colour (see also Figure 4.12 on page 37). The intrusion levels for the simulated environment apply here as well. The erroneous colour marks the illegal zone, and, depending on the intrusion level, its colour is closer or further from the legal colours.

Lastly, we have defined possible intrusions for each of the three types of requests that are sent based on the position of the unit (see Table 4.8).

Request type	**Intrusion**	**Request sent**	
Country code	Wrong position	Normal:	x, y : current position
		Compromised:	$x_c, y_c : n_x, n_y \geq 10 \wedge x_c = x \pm n_x \wedge y_c = y \pm n_y$
Point of interest	Illegal type	Normal:	$t \in T$: One of the legal types in T
		Compromised:	$t_c \notin T$
Route	Route to self	Normal:	$x_t, y_t : x_t \neq x \wedge y_t \neq y \wedge$ x, y : current position
		Compromised:	$x_{tc}, y_{tc} : x_{tc} = x \wedge y_{tc} = y$

Table 4.8: *Intrusions for positional requests.*

These types of intrusions may be detectable with domain knowledge-based rules. Because of their nature, we have implemented a different type of intrusion levels for these requests. Depending on the level, a compromised unit sends one of these compromised requests with varying probability (see Table 4.9), making such a unit easier or harder to detect.

Intrusion level	Likelihood
Easy	40 %
Medium	20 %
Hard	5 %

Table 4.9: *Intrusion likelihood per intrusion level.*

Summary

We have discussed different intrusion types and how we try to represent them with the components of the testbed. For each intrusion type, we have defined the intrusions and described different intrusion levels.

Finally, Table 4.10 gives an overview over the components and the effects that the intrusions defined before have.

Component	Expected behaviour	Intrusion effect
Data generator	Generates values based on one of our distributions	Values deviate from distribution
2D simulator	Reads the colour background	Erroneous colour reading
	Avoids illegal zone	Stays in zone
	Sends legal positional request	Invalid or illegal request

Table 4.10: *Overview of possible intrusions for each component.*

4.4.4 Server

The testbed server is implemented in *Bottle* [41], a lightweight Python web framework. For its networking component, we make use of the *gevent* [5] library.

The server can run in *detection mode*, feeding the IDS component incoming requests directly, and, additionally or exclusively, in *store mode*, saving incoming requests to disk.

Architecture

The API endpoints are mapped to Python routines in the logging component. It transforms the request data to a logging format (see Figure 4.6 on page 30) and, depending on the selected server mode, stores the result utilising a data access object (DAO), or sends it to the IDS component directly (see Figure 4.13).

Stored data

When in *store mode*, the server sends one log line for each incoming request to the DAO to store. If the requests contain information about possible intrusions (see *Central communication unit* on page 35), this data is used to label the stored log entry. The labels can then be used for evaluation.

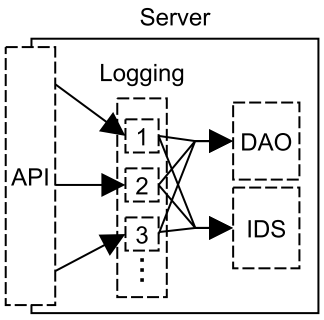


Figure 4.13: *The testbed server. Incoming data is forwarded to routines in the logging component. These in turn send the transformed log data to a DAO or to the IDS. Arrows represent communication. Components are dashed.*

Each request is stored by the DAO as one JSON string per line. The label is added after the log line, separated by a comma (see Listing 4.5).

```
{"client_id": "S953564", "microservice": "GAUSSIAN_1",
  "level": "DEBUG", "data_received": "3.1875026226",
  "time_unix": 1523043651, "position": "150,210",
  "log_id": "b7def0bd-a097-4615-a09e-73bd894c751f" },
  normal
```

Listing 4.5: *An exemplary log line.*

The Listing shows all log features that we discussed before (see Table 4.1 on page 24). The identifier of the client (“client_id”), the microservice name (“microservice”), the position of the client (“position”), the request time (“time_unix”) and the received data (“data_received”). Additionally, the logging component adds a log level (“level”) and a unique log identifier (“log_id”).

4.5 Assumptions

We base the concept and implementation of our testbed on three assumptions. Each of those, when unfounded, may have an impact on the validity of our solution.

4.5.1 Assumption 1: Compromised clients can be identified from their communication with the server

Our first assumption is most significant, as it is the prerequisite for our approach. We assume that normal and compromised clients can be differentiated based on differences in their communication with the server. If this does not hold true, remote intrusion detection as we envision it would be impossible.

Before making this assumption, we considered existing research on cloud-based security systems for vehicles. Zhang et al. [129] describe a cloud-based malware scanner that is based on offloading malware detection to the security cloud when necessary [129, p. 17]. We also found research on remote anti-theft systems. Ramadan et al. [87] describe a SMS-based anti-theft system that is based on interaction with the vehicle owner. When their car is started, they are alerted and can activate a tracking service [87, p. 89]. Finally, Eiza et al. [25] discuss multiple cloud-based security solutions, including the Ericsson Connected Vehicle Cloud [26]. They find the biggest risk for these systems to be high cost of connections and regions with missing connectivity [25, p. 5].

Accepting this premise as given, we find a second threat to our approach. If an intruder knows or can suspect that the system they intrude into is being remotely protected, they can try to prevent the communication of the system with the server completely, either through the software or by physically blocking the signals sent. To prevent this, the system itself could monitor its connection to the server and increase the protection level in case that connection is lost.

Considering the existing research on cloud-based security systems for vehicles and the mitigation for connection loss that we suggest, we argue that our assumption is suitable.

4.5.2 Assumption 2: Microservices in our use case do not alter data

We assume that the microservices in our use case do not alter the data they receive, for example by condensing or extending it. We need to make this assumption for our system design to cohere. This is hard to verify, as each microservice may be designed by a different author. A software bug in a microservice or carelessness on the side of the developer can be a risk for the detection of intrusions. Still, this is an issue for any IDS, independently of which data it uses for detection.

4.5.3 Assumption 3: Unpredictable interoperation of components and systems cannot be emulated with basic data generation

Our final assumption is that the unpredictability of the interoperation of components, clients and a server cannot be adequately emulated with basic data generation. This is the premise for our work on the testbed. Considering the large variety of existing testbed approaches to intrusion detection evaluation [21, 73, 89, 100], we conclude that this assumption is fair to make. Some effort has been made to alleviate the need for a complex test environment by generating more realistic test data [36], but we consider this an open problem.

5 Evaluation

Our evaluation comprises multiple steps. We intend to assess different aspects of our implemented testbed to show that it fulfils its purpose, formulated as research questions (RQ). First, we focus on basic performance metrics such as parallelisability and scalability in Section 5.1. Then, we discuss the fitness for purpose of the testbed in Section 5.2. Finally, Section 5.3 evaluates the quality of the data generated by the testbed.

5.1 Testbed performance

We want to evaluate how well the testbed runs when the number of components is scaled up. This is aimed at assessing the suitability of the testbed for various scenarios.

5.1.1 Prerequisites

First, we introduce prerequisites for the following research questions.

Utilised machine

For our experiments, we used a machine with a 12-core 2.5 GHz *QEMU* [4] virtual CPU and 32 GB of main memory. The operating system used was *Ubuntu* 16.04.4 LTS [12]. The machine was provided by the Leibniz Supercomputing Centre [63].

Metrics

In the following, we will use the *mean squared error* MSE and *coefficient of determination* R^2, two metrics often used to assess the quality of estimations. They are defined as follows. Given a set of n values $y = [y_1, \dots, y_n]$ with mean $\bar{y}$, and a set of estimations $\hat{y} = [\hat{y}_1, \dots, \hat{y}_n]$. The MSE is defined as:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

For better estimations, the MSE tends towards 0, worse estimations increase it. R^2 is defined as:

$$R^2 = 1 - \frac{\sum_{i=1}(y_i - \hat{y}_i)^2}{\sum_{i=1}(y_i - \bar{y})^2}$$

For better estimations, it tends towards 1, worse estimations lower it.

5.1.2 RQ 1: How fast can the testbed be started?

ROS is based on a peer-to-peer architecture. That means individual components only need to register with the ROS *master* once, then they are directly linked with their respective peers.

We can derive a basic performance metric for the testbed from that, namely the start-up time based on the number of components. That refers to the time it takes the ROS *master* to link up all the components in the testbed and start them.

Experimental set-up

This experiment was split up in cycles and rounds. Each cycle is identified by the number of components that are started. Each round represents one measurement.

For each round, we started the server and let it initialise. Then we launched an adapted ROS start-up script, which is configured to store the system time just before the ROS *launch* command is triggered. On the server side, we adapted the log routine to stop the server directly after the first request comes in. Just before that, the system time is stored again. Both components run on the same machine, resulting in their time measurements to be identical.

There is always a minimum of two components in our testbed, namely a data generating component and a central communication unit that sends the generated data to the server (see Figure 4.3 on page 28). Accordingly, we chose these two components as our baseline. All experimental *launch files* contain one communication unit as well as various numbers of data generators (see Listing 5.1).

```
<launch>
  <group ns="CLIENT1">
    <node name="gauss_1" type="data_generator.py" ... />
    <node name="com" type="communication_unit.py" ... />
  </group>
</launch>
```

Listing 5.1: *The simplified ROS* launch file *for the cycle with two components (nodes).*

For each cycle, we ran ten rounds of the same experiment. The results below are averages from those rounds. The maximum standard deviation of all measurements in one cycle was 0.26 seconds, or 1.3 %, for the cycle with 400 components, which took 20.4 seconds total. On average, the standard deviation of measurements in all cycles was 0.07 seconds. Hence, we consider the results sufficiently reproducible.

Results

Our results in Figure 5.1 are plotted in two dimensions. Denoted as t_a is the absolute start-up time in seconds. The relative start-up time per component is denoted as t_r.

We can see two properties of the data in our plot. The absolute start-up time increases linearly, as expected. Contrary to that, the relative start-up time per component shows exponential decay, converging towards 50 milliseconds.

This effect is most likely related to the cold-start time of the system which seems to be constant, independent of the number of components. Based on our results, we can approximate it as less than 1.5 seconds.

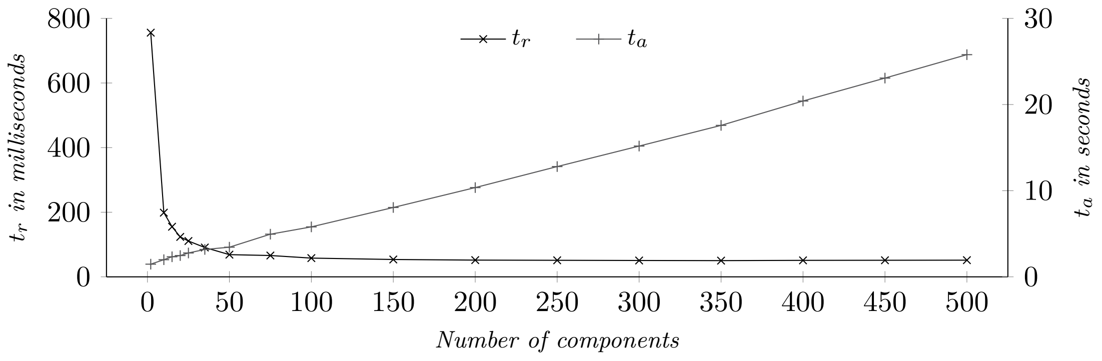


Figure 5.1: *The start-up time of the testbed relative to the number of components started. t_a denotes the absolute time in seconds. t_r denotes the relative time per component in milliseconds.*

5.1.3 RQ 2: How many components can be run in parallel?

An important quality of the testbed is its performance impact on the host machine. It determines the necessary hardware and estimated cost of conducting scenarios that depend on a large number of components running in parallel.

Experimental set-up

Evaluating the number of components running in parallel is delicate. The results highly depend on various parameters that we can modify. To ensure that our experiments are meaningful, we list these parameters first.

Based on our architecture, we have a minimum of two components in our testbed (see also Listing 5.1 on page 44). For data generation, we used our simple data generator components. They were configured with a publish rate of 2 Hz. This corresponds to sending a new data point every 0.5 seconds.

To evaluate the performance impact of the testbed, we started profiling the system 60 seconds before launching the testbed. The start-up procedure (see Section 5.1.2) was ignored. Following that, we continued profiling for an additional 60 seconds.

We collected five performance profiles for each cycle. The results below are averages from these rounds. The maximum standard deviation of main memory usage measurements in one cycle was 0.59 percentage points, or 1.58 %, for the cycle with 250 components, which had a total usage of 37.6 %. For all cycles, the average standard deviation was 0.08 points. For the CPU load, we calculated a maximum standard deviation of 0.18 points, or 1.2 %, for the cycle with 500 components, which had a total load of 16.0 %. For all cycles, the average standard deviation was 0.4 points.

Results

In Figure 5.2, we plot our results with the CPU usage denoted as *cpu* and the main memory usage denoted as *mem*. The values are given in percent and were calculated by subtracting the average idle load before starting the testbed from the average full load while it was running.

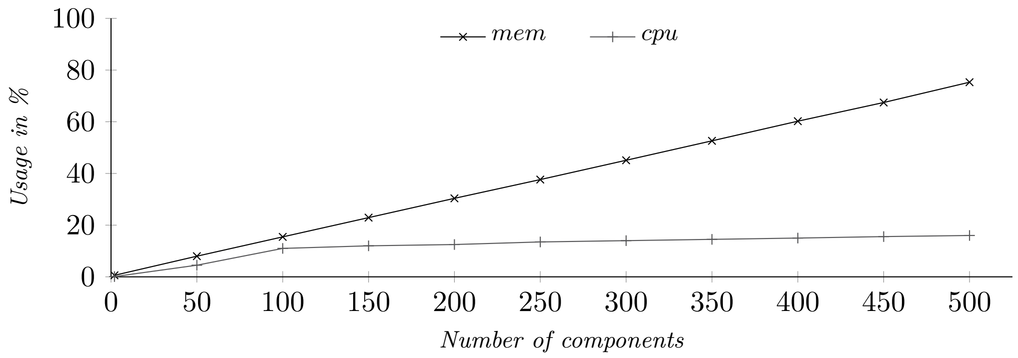


Figure 5.2: *The resource usage of the testbed relative to the number of components running. cpu denotes the CPU usage. mem denotes the main memory usage.*

On our system, the main memory is filled when running more than 650 components. As we can see in our result, the CPU usage is moderate, even for 500 components, and would allow for significantly more to be run.

Interestingly, the CPU usage rises rather sharply when increasing the number of components from 2 to 50 and 100, but then shows only slight increase for higher values. This behaviour was reproducible. We assume that there is an overhead connected to running these components that stays constant for higher component counts. As expected, the memory usage rises linearly.

5.1.4 RQ 3: How well does the testbed scale with additional components?

We are interested in the scalability of our testbed. Ideally, it scales almost linearly, as this allows arbitrarily increasing the number of components if necessary.

To evaluate this research question, we can make use of our existing evaluation results. We regard the following metrics:

- Start-up time
- CPU load
- Main memory usage

We perform a *regression analysis* to approximate the time complexity based on our measurements. For each analysis, we give the MSE and R^2 to assess the quality of the estimation compared to the measurements.

Because we could only measure reliably for component counts of up to 500, we do not generalise further than for $n \to 5000$. For component counts larger than 5000, we would need more measurements to be able to sufficiently predict the limiting behaviour.

For an elaboration on our measurement process and precision, consult the respective research question (see Section 5.1.2 on page 43 and Section 5.1.3 on page 45).

Start-up time

The start-up time grows linearly (see Figure 5.1 on page 45), so we estimate it with linear regression. After multiple iterations with subsets of our measurements, we obtain: $g_s(n) \approx 0.0476n + 1.141$ with MSE < 0.143 and $R^2 > 0.998$ for the complete set of measurements (see Figure 5.3). We assume that the constant 1.141 represents the cold-start time.

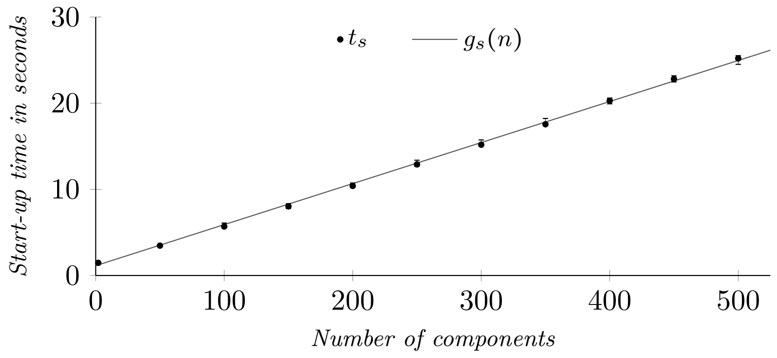


Figure 5.3: *Measurements of the start-up time t_s for different numbers of components and the approximation $g_s(n)$. Each dot is the mean of our measurements for that cycle. The bars above and below each dot indicate the maximum and minimum measurement, respectively.*

From our approximation, we can derive the limiting behaviour:

$$\begin{aligned} f_s(n) &= \mathcal{O}(0.0476n + 1.141) \\ &= \mathcal{O}(n), n \to 5000 \end{aligned}$$

CPU load

If we only regard the CPU load for component counts of 100 or more, it grows linearly (see Figure 5.2 on page 46). We estimate it utilising linear regression and obtain: $g_c(n) \approx 0.0127n + 9.756$ with MSE < 0.031 and $R^2 > 0.988$ (see Figure 5.4).

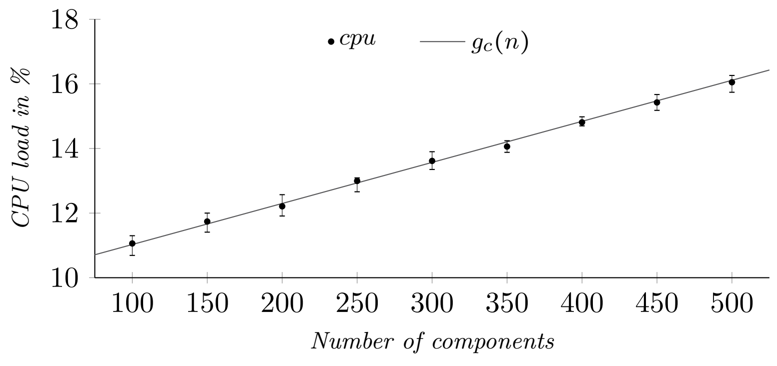


Figure 5.4: *Measurements of the CPU load for different numbers of components, denoted as cpu, and the approximation $g_c(n)$. Each dot is the mean of our measurements for that cycle. The bars above and below each dot indicate the maximum and minimum measurement, respectively.*

With this approximation, we conclude:

$$\begin{aligned} f_c(n) &= \mathcal{O}(0.0127n + 9.756) \\ &= \mathcal{O}(n), n \to 5000 \end{aligned}$$

Main memory usage

The main memory usage grows linearly (see Figure 5.2 on page 46). Accordingly, we try to estimate it utilising linear regression and obtain: $g_m(n) \approx 0.1491n + 0.447$ with MSE < 1.262 and $R^2 > 0.997$ (see Figure 5.5).

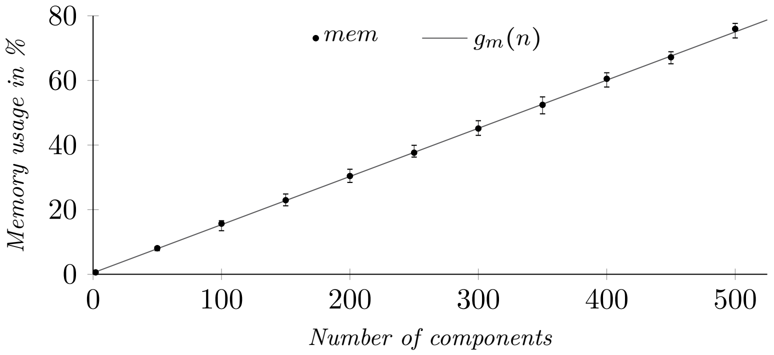


Figure 5.5: *Measurements of the main memory usage for different numbers of components, denoted as mem, and the approximation $g_m(n)$. Each dot is the mean of our measurements for that cycle. The bars above and below each dot indicate the maximum and minimum measurement, respectively.*

We conclude:

$$\begin{aligned} f_m(n) &= \mathcal{O}(0.1491n + 0.447) \\ &= \mathcal{O}(n), n \to 5000 \end{aligned}$$

Conclusion

We can show for all our measures that a linear approximation can sufficiently explain our measurements. We derive a limiting behaviour of $\mathcal{O}(n)$ for component counts of up to 5000, meaning our testbed scales linearly. For a valid estimation for higher n, we would need additional measurements.

5.1.5 RQ 4: How many dataset lines can be generated in a certain time?

When generating test data, an important performance metric is the time it takes to generate a new dataset. When parameters are changed, large quantities of data need to be quickly generated, a process which may have to be repeated often.

Accordingly, we tuned our testbed with the available resources to optimise the data generation speed.

Experimental set-up

The web server was configured to count its internal list of log entries every thirty seconds. From the number of newly added entries, it calculated a velocity of new entries per second. This velocity was stored.

We aggregated the peak performance metrics over two hours to ensure meaningful results.

Results

With our measurements over two hours and a total of $1,587,607$ entries generated, we arrived at the data shown in Figure 5.6. The box plot shows the minimum, maximum and average number of entries generated per second, as well as the 2nd and 98th percentile of the collected data.

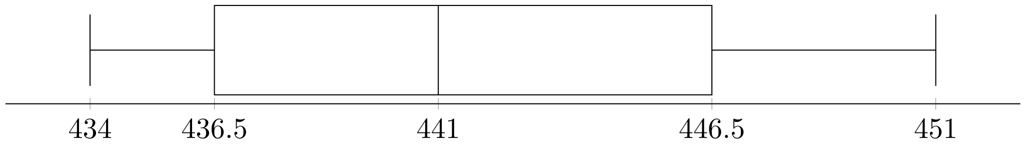


Figure 5.6: *The average number of entries generated by the testbed per second under maximum load in our set-up. The whiskers are set to the 2nd and the 98th percentile.*

The speed of data generation was limited by the testbed server. This was expected, as the other components can publish data independently, but the server needs to collect all incoming data in one place. Hence, the data generation can be accelerated further by improving the performance of the server.

5.2 Fitness for purpose

We created the testbed to serve the purpose of evaluating various IDS for the real-world use case of our industry partner. We saw the need for this complex solution due to limitations of existing options. In the following, we evaluate if our testbed succeeds in overcoming these.

5.2.1 RQ 5: Do we solve the problems of existing datasets?

In our research of IDS (see Chapter 3), we identified three problems in the datasets used for evaluation.

Too few entries: Many IDS techniques require a significant amount of data for sufficient evaluation [11, 57]. Knorr et al. [57] used a dataset with only 855 entries. To obtain a sensible amount of data, they artificially created data of similar distribution [57, p. 245]. Cannady [11] makes use of *Artificial Neural Networks* (ANN). They note that these require large amounts of data that are difficult to obtain [11, p. 5].

Due to our testbed being able to arbitrarily generate new entries when needed, we consider this problem solved.

Class imbalance: Training data showing class imbalance is an important problem if it is used for *machine learning* systems, as it leads to worse classifier performance [16]. A common scenario for IDS, especially those making use of *anomaly-based* detection, is to have data that consists of mostly normal instances. Efforts are made to alleviate this issue, for example by over-sampling the minority classes [15].

We can tune the testbed to generate balanced data, but we cannot precisely predict the resulting class imbalance in the data, so we have to evaluate it in the following.

Redundancy: Data redundancy leads to a bias in *machine-learning* systems [67, 101]. This is a problem of the original KDD Cup 1999 dataset [53] identified by [67, 101, 105]. Mitigating this by deleting the redundant records leads to fewer entries in the dataset, an issue in itself.

The redundancy in our generated data can also not be predicted for every scenario, so we evaluate it in the following.

From these problems, we can derive quality metrics. As we consider the problem of too few entries solved with the testbed, we evaluate the class imbalance and redundancy of the generated data.

Measuring class imbalance

As a quality measure for the number of elements per class in a dataset we use the *dispersion index*, defined as follows: Given the variance σ^2 and the mean μ of the data, the dispersion index is calculated as $\frac{\sigma^2}{\mu}$. Ideally, this measure is 0 for an exactly balanced dataset.

Measuring redundancy

We count the duplicates in our datasets as follows: Consider a log line l_i in line i of a log file. We define this log line as a duplicate if the following holds: $i > 1 \land \exists\ l_{i-1}$ where the *client_id*, *level*, *position*, *data_received* and *label* in l_i and l_{i-1} are identical, then l_i is a duplicate.

Experimental set-up

We used an adapted launch file, tuned to generate balanced data. With it, we generated a dataset containing 10 million data points. Then, we ran our analysis over the file.

The dataset contains data points for all 14 categories. Each data category has approximately $715,000$ data points. Regarding the aspect of class imbalance, we consider two classes, the normal and the intrusion class. For each data category, we calculate the relative class imbalance. We consider relative deviations of less than 1 % ($7,150$ data points) as ideal, and of less than 5 % ($35,750$ data points) as good. That corresponds to a dispersion index of 18 or less for ideal, and of 450 or less for good results.

Regarding duplicates in the data, we consider 0 % to be ideal and anything less than 1 % to be good.

Results

We split up the results based on the underlying component. All data generators behave identically regarding intrusions, so their results are grouped as *Generators*. The same is true for positional requests, grouped as *Positional*. The results for colour requests are listed separately (see Table 5.1).

Measure	**Generators**	**Positional**	**Colour**
Number of elements	$7,021,192$	$2,160,908$	$817,900$
Class dispersion index	250.8	7.8	32.1
Duplicates	0 %	0.016 %	0.009 %

Table 5.1: *The results for different categories of data, grouped. The complete dataset contains* 10 *million data points. Values are rounded.*

All data categories succeed in meeting our strict duplicate targets. The class balance is not perfect, but for colour requests we observe good and for positional requests ideal results. The *Generators* show an acceptable dispersion index below our threshold of 450.

5.2.2 RQ 6: Can we reproduce data of similar distribution?

Data generated by a testbed is often not considered static but needs to be reproduced multiple times. For comparable behaviour over multiple runs of the same set-up, it is essential that the generated data is similarly distributed.

To derive sensible quality metrics, we consider synthetic over-sampling techniques. When creating synthetic data, the objective often is to recreate data of similar distribution, for example by regarding the distribution of attributes [57, p. 245]. Alternatively, SMOTE [15] defines samples in a “feature space” [15, p. 328]. It uses neighbouring samples from the minority class and creates synthetic samples that lie between them in this space [15, p. 328].

For our data generators, we consider this research question to be fulfilled. The underlying number generators are based on the principle of generating data based on a statistical distribution, resulting in equally distributed data over multiple runs. Exemplary, we show this for our *normal distribution*-based generator.

The more interesting component is the 2D simulator. For each of the request types, we want to assess if the distribution of their possible forms stays similar over multiple runs.

All components generating random data are seeded, so that multiple runs ideally result in identical behaviour. Timing differences and other influences can still lead to discrepancies in the results, so we calculate the actual divergence.

For evaluation, we generate three datasets. To avert sampling errors influencing the result, each set is generated with $250,000$ data points. We consider the first set the baseline, and the following sets are compared to it. We give the coefficient of determination R^2 (see Section 5.1.1 on page 43) for both sets compared to the baseline. If the baseline distribution is perfectly reproduced by the following set, this measure is 1. We use the R^2 as it is commonly used to show how well data points correspond to a given baseline.

Data generators

First, we want to show the relative distribution of data generated by a *normal distribution* generator, exemplary for our data generators. We generate three datasets with $250,000$ data points each, containing only data generated by that component. The intrusion level is set to *hard* (see Section 4.4.3 on page 36). In accordance with our implementation of intrusion levels, changing the level only scales the possible values with a constant. Hence, it does not influence the reproducibility of the data. We select level *hard* because the generated values are closer to zero and as such easier to visualise.

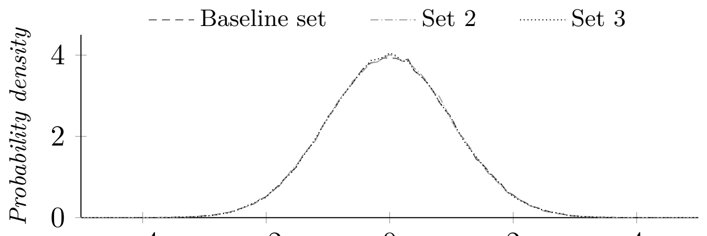


Figure 5.7: *Relative frequency of values of the normal class generated with* normal distribution *data generator components. Each set contains approximately* $178,000$ *data points.*

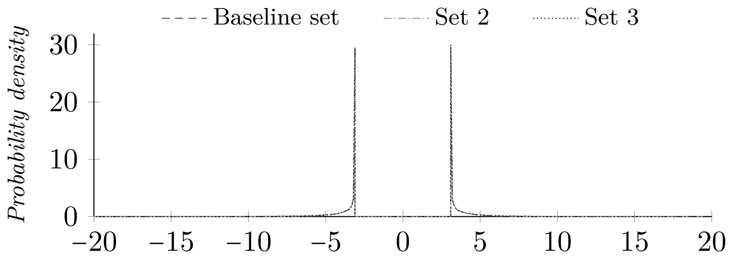


Figure 5.8: *Relative frequency of values of the intrusion class generated with* normal distribution *data generator components. Each set contains approximately* $71,000$ *data points.*

For visualisation, we split the generated data up in members of the normal and the intrusion class. In Figure 5.7, the distribution of normal data points of all three sets is compared. As expected, it follows a *normal distribution.* To arrive at this plot, we split the data up in bins of size 0.1. Then, we calculated the relative frequency of data points in each bin compared to the total number of data points generated. The R^2 value of Set 2 compared to the baseline set is approximately 0.9995, for Set 3 it is approximately 0.9996.

In Figure 5.8, the distribution of data points of the intrusion class of all three sets is compared. Following our definition, no values are generated within the distribution interval $[-3.09, 3.09]$ (see Table 4.2 on page 33). Again, we calculated the relative frequency of data points in each bin of size 0.1 compared to the total number of data points generated. The R^2 value of Set 2 compared to the baseline set is approximately 0.9998, for Set 3 it is approximately 0.9997.

As expected, this component generates sufficiently reproducible data.

2D simulator requests

Next, we consider the requests made by the 2D simulator component, which consist of categorical data. For each request type, we generate three datasets with $250,000$ data points each, containing only those data points corresponding to the request that is investigated.

Colour values: We have defined four legal and one illegal colour option: purple, yellow, green, blue and red (intrusion; see Section 4.4.3 on page 36). In Figure 5.9, the relative frequency of each colour type is compared for the three sets. They may seem identical, and they almost are, but there is a slight deviation in relative frequency between the sets. The R^2 value of Set 2 compared to the baseline set is approximately 0.9999, for Set 3 it is approximately 0.9997.

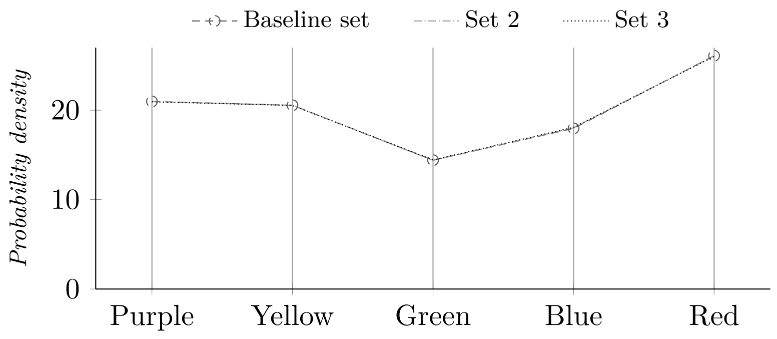


Figure 5.9: *Relative frequency of colour values generated with 2D simulator components. Each set contains* $250,000$ *data points.*

Country codes: We have defined five country codes that correspond to different areas of the simulated environment, namely “AT”, “CH”, “DE”, “FR” and “IT”. In Figure 5.10, their relative frequency in the three sets is compared. Again, we find an almost identical distribution with only slight deviations in relative frequency. The R^2 value of both Set 2 and Set 3 compared to the baseline set is approximately 0.9999.

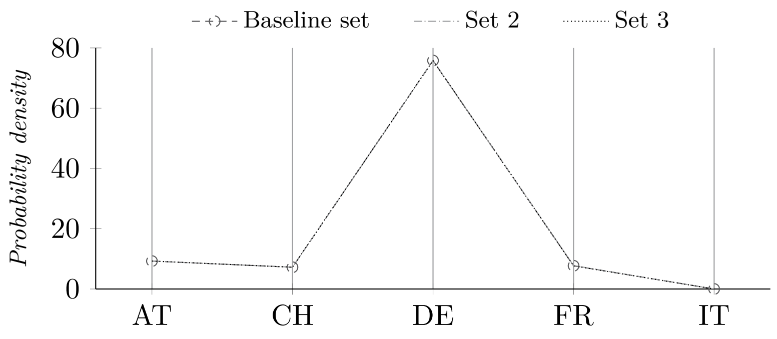


Figure 5.10: *Relative frequency of country codes generated based on 2D simulator request data. Each set contains* $250,000$ *data points.*

Routes: Route requests consist of two positions, the start and the target position, and are accordingly not categorical. Instead, we consider the relative frequency of normal to intruded requests based on our intrusion definition. Accordingly, we compare the relative frequency of the two classes "normal" and "intruded" in Figure 5.11. As expected, the distributions are almost identical. The R^2 value of both Set 2 and Set 3 compared to the baseline set is approximately 0.9999.

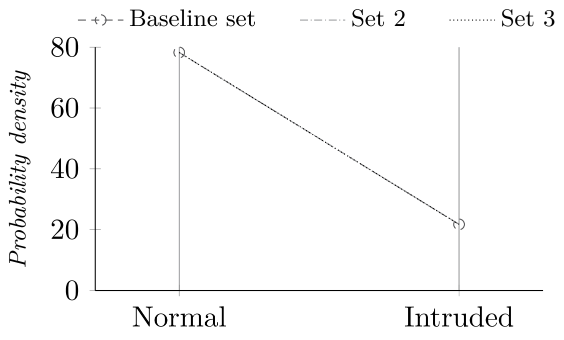


Figure 5.11: *Relative frequency of the classes of route requests generated based on 2D simulator request data. Each set contains* 250, 000 *data points.*

POI pairs: Point of interest (POI) pairs consist of a POI type and a POI result. Clients request a POI of a specific type for their location. The server retrieves the POI result and stores the requested type and the result as a log entry.

There are two legal POI types with three possible results each, and two illegal types that both map to the same result, namely "Invalid". Each allowed combination of a POI type and result is one possible form of a POI request. In Figure 5.12 we compare the relative frequency of possible forms of POI pairs in the three sets. This data category also shows almost identical distributions for different sets. The R^2 value of Set 2 compared to the baseline set is approximately 0.9995, for Set 3 it is approximately 0.9998.

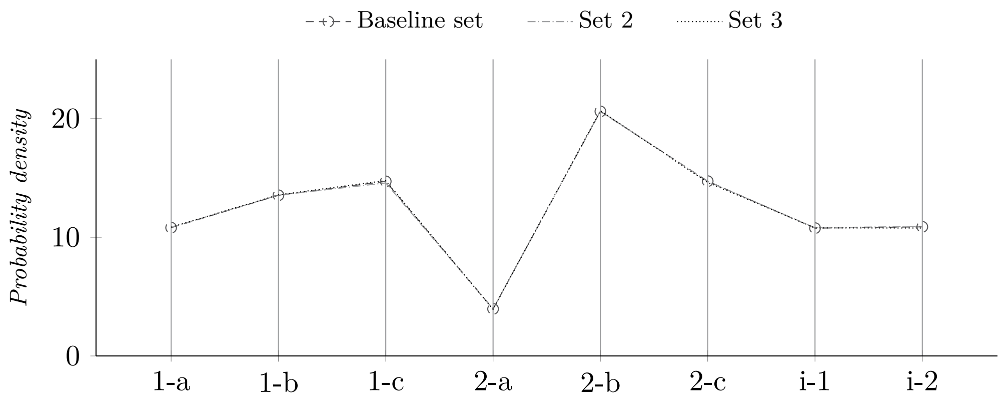


Figure 5.12: *Relative frequency of POI pairs generated based on 2D simulator request data. Each set contains* 250, 000 *data points.*

2D simulator positions

Finally, we evaluate the relative frequency of the simulated unit being located at various positions of the simulation. This is trickier to visualise, as we have significantly more points to compare. We approach this problem with heat maps. For easier visualisation, we divide the coordinate space into bins of 20×20 pixels in size. The relative frequency of each bin is signified by the shade of the cell that represents it (see Figure 5.13). For higher frequencies, the box shifts to a darker shade. The highest frequency is visualised in black.

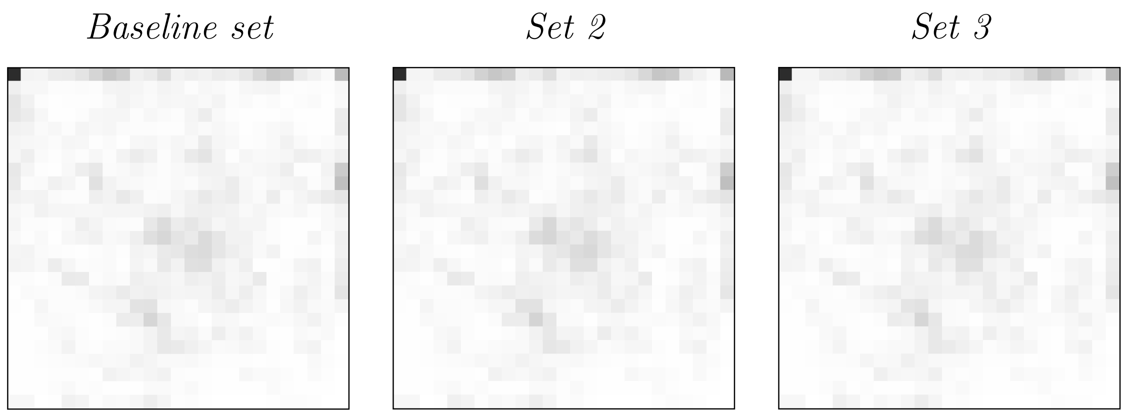


Figure 5.13: *Relative frequency of a coordinate in the respective bin. The highest frequency is denoted in black, lighter shades represent lower frequencies. Each set contains* $250,000$ *data points.*

The heat maps all show a similar pattern. Most noticeable is the dark cell in the top left corner. The simulated unit seems to disproportionally often remain in that area. Along the edges and around the centre we see more areas of higher relative frequency, all of which are mirrored in the other sets.

This intuition is confirmed when calculating the R^2 value. For Set 2 compared to the baseline set it is approximately 0.9965, for Set 3 it is 0.9962.

Conclusion

When generating data multiple times, the resulting distributions of values are ideally the same or very similar. This objective is fulfilled for all testbed components, including the more complex movement patterns of the 2D simulator.

5.2.3 RQ 7: Can we evaluate various types of IDS?

In our literature survey of existing approaches to IDS we found various, heterogeneous systems. Existing evaluation options were often limited to specific types of systems. For example, authors utilising *anomaly-based* detection can use general-purpose datasets such as the *Iris flower set* [24] for evaluation. For *signature-based* systems, such datasets offer no sensible evaluation option as they are missing intrusions that could be detected.

For this research question, we want to show that our testbed theoretically allows the evaluation of various types of IDS.

Anomaly-based detection

IDS utilising *anomaly-based* detection require data with some form of anomalous behaviour that they can detect. This use case is covered by our data generators (see Section 5.2.2 on page 52). They are based on probability distributions, which allow us to define normal behaviour as that which is most likely to happen based on the underlying distribution. Accordingly, we introduced intrusions that aim at being discernible from normal data (see Section 4.4.3 on page 36).

Signature-based detection

To evaluate *signature-based* detection systems, our data requires well-defined signatures that can be coded into these systems. Our 2D simulator component aims at providing these. The data it generates is used to create requests aimed at emulating domain knowledge-based patterns. Similarly, we introduced intrusions that are based on breaking some defined rule. Currently, they are built to also be detectable by *anomaly-based* systems.

Advanced detection systems

Advanced systems such as *Artificial Neural Networks* (ANN) or *model-based reasoning* approaches are currently impossible to evaluate with our testbed. We have not defined long-term behavioural patterns or specific user profiles that would allow for that. These systems may best be evaluated with real-world data.

Conclusion

We consider the testbed to be sufficient for evaluating *anomaly-based* as well as *signature-based* systems on a theoretical basis. Due to the automatic labelling of testbed data, the detection accuracy can be measured effectively. For more advanced detection systems, our testbed does not offer adequate options for evaluation.

In Section 5.3, we evaluate the effectiveness of our solution.

5.3 Quality of the generated data

We introduced features to the testbed aiming at certain effects. For example, the intrusion levels are presumed to lead to differences in the detection accuracy of an IDS. Similarly, using various types of components ideally leads to differences not just in the distribution of their generated data, but also in the effectiveness of an IDS at detecting intrusions. The following research questions intend to assess these effects.

5.3.1 Prerequisites

First, we introduce prerequisites for the following research questions.

Handling heterogeneous data

As we have described in the previous chapter (see Table 4.1 on page 24), the data that is produced by different routines in the logging component varies. Each of these routines represents a microservice, and in our real-world use case these microservices require or produce different data.

We regard each type of data produced by different log routines as separate datasets. Table 5.2 lists the names we use for those datasets. We refer to these names in our experiments. For a definition of the generated data, see Section 4.4 on page 31.

Name	Type of data	Name	Type of data
GAUS	Normal distribution	VONM	von Mises distribution
GUMB	Gumbel distribution	WALD	Wald distribution
LAPL	Laplace distribution	WEIB	Weibull distribution
LOGI	Logistic distribution	COLR	Colour background
PARE	Beta Prime distribution	CCOD	Country code request
RAYL	Rayleigh distribution	POIR	POI request
UNIF	Uniform distribution	ROUT	Routing request

Table 5.2: *Names we use to refer to different categories of data.*

We want to use the variability in the data generated by the testbed to show distinct effects. Hence, we score systems individually for each type of data in our experiments. The names used for the results match those of the respective dataset.

Libraries used

We make use of Python and *machine learning*, with our library of choice being *scikit-learn* [83]. Apart from classification, we utilise included pre-processing and metrics modules. From this library, we employ the *OneClassSVM* classifier, which is based on the *libsvm* [14] library for computations.

Standardisation of numerical features

The *machine learning* estimators we use require standardised data, ideally with a mean of zero [95]. Hence, we standardised numerical features as follows:

Unit position: The original position dimensions $x, y \in [0, 499]$ are scaled down and shifted to $x_s, y_s \in [-1, 1]$.

Colour values: The colour reading is sent as the red, green and blue (RGB) components of the colour, with the original values $r, g, b \in [0, 255]$. They are scaled down to $r_s, g_s, b_s \in [0, 1]$ to preserve the increase in colour intensity for higher values.

Encoding of categorical features

It is possible to encode categorical features as indices, mapping them to a list of n natural numbers for n possible forms. This is not sensible for the *scikit-learn* estimators, as they would interpret these values as being ordered [93]. Hence, we make use of the suggested method of *one-hot encoding* [93]. A field with n possible forms is encoded in a vector of n binary features. Each corresponds to one form of the field. All features are set to 0, with only one set to 1 for the respective form. An exemplary *one-hot encoding* for a field *level* with the possible forms "normal", "debug" and "error" could be:

level		*e*	*d*	*n*
normal	$\mapsto$	0	0	1
debug	$\mapsto$	0	1	0
error	$\mapsto$	1	0	0

In our tests, we identified another advantage of utilising *one-hot encoding*. We evaluated the training time for a *scikit-learn* classifier trained with categorical data that was either converted by mapping it or by utilising *one-hot encoding*. Our results (see Figure 5.14) show that *one-hot encoding* leads to significantly faster training. For 350,000 entries, it finished five times faster than when using a simple mapping.

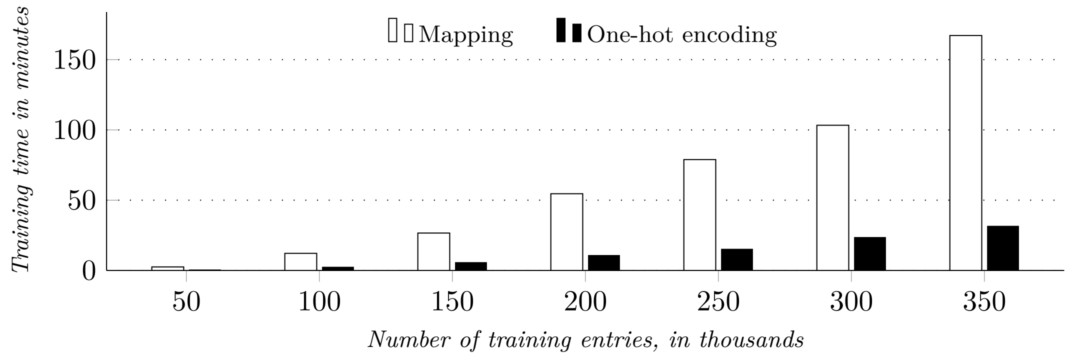


Figure 5.14: *Training time for a* scikit-learn *classifier, using mapping or* one-hot encoding.

Disregarded features

Fields that we know to be arbitrary, namely the *log_id* and the *time_unix*, are removed from the data. As we have not implemented sensible client-specific intrusions, the *client_id* is also left out. Finally, we already use the *microservice* field to differentiate between different data categories. Hence, it is the same for the data points of each respective classifier, which makes it obsolete.

Data generation

Our testbed allows multiple modes of operation (see Section 4.4.4 on page 39), the *detection mode*, in which an IDS is fed a single request at a time, and the *store mode*, that collects the request data on disk. For data generation, we use the *store mode*. We prepare multiple ROS *launch files* (see Section 4.4.1 on page 31) that are tuned to ensure that data for each component and each class is generated.

All components generating random data are seeded, so that multiple runs of the same *launch file* result in comparable behaviour and our experiments are reproducible.

Data selection process

Due to limitations in main memory and computation power, we cannot work with an arbitrary number of data points at a time. For the *machine learning* algorithm we use, all data points for training need to be processed at the same time. This means we have to sample those datasets that exceed our maximum data point limit.

Loading such a dataset entirely is not always possible, as we just discussed. The same problem applies to sampling. To still achieve a fair selection, we use an adapted *Reservoir sampling* algorithm originally devised by Knuth [58].

Finally, the data is split up into separate training and scoring sets. The training set contains only normal data points, with normal and intrusion data points being used for scoring (see Figure 5.15).

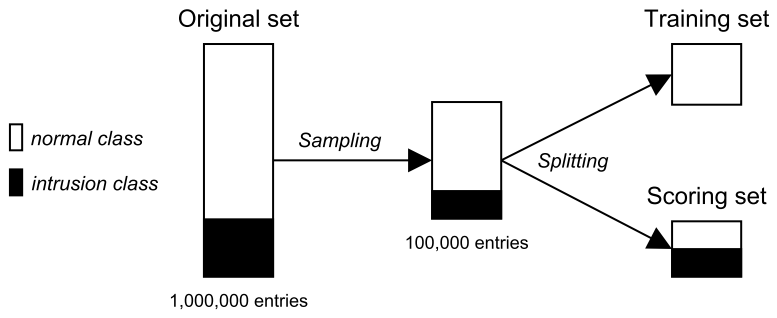


Figure 5.15: *The data selection process. The original set is sampled using* Reservoir sampling*, then it is split into separate training and scoring sets. Data points in these sets can be members of the normal class, denoted in white, or the intrusion class, denoted in black.*

Assessment measures

To assess the results of classifications, we employ the commonly used metrics *precision* and *recall*. Given the number of true positives tp, the number of false positives fp and the number of false negatives fn, they are defined as:

$$\text{Precision} = \frac{tp}{tp + fp} \qquad \text{Recall} = \frac{tp}{tp + fn}$$

Definition of positives

In intrusion detection, we can define either the normal class or the intrusion class as positive. We regard those data points as positives that are intrusions, contrary to the *scikit-learn* interpretation [94]. Accordingly, true positives are intrusions that were predicted as such. This interpretation is common in intrusion detection research [112].

We consider two aspects as important in intrusion detection when assessing classifier performance. The first aspect is how many intruded samples are correctly identified. This does not change with our definition of positives. The second important aspect is the percentage of correct predictions compared to all samples predicted as intrusions.

Consider an extreme example with a system classifying 90 % of instances correctly, regardless of class, with 100 normal instances and 10 intrusions being analysed. 90 of 100 normal instances are classified as normal, and 9 of 10 intruded instances as intruded. With the *scikit-learn* interpretation the IDS achieves a precision of $\frac{90}{90+1} \approx 98.90$ %, with our interpretation it is only $\frac{9}{9+10} \approx 47.37$ %. The recall percentage in both scenarios is identical, namely $\frac{90}{90+10} = \frac{9}{9+1} = 90$ %.

5.3.2 RQ 8: Are there significant differences in detection precision or recall for data generator types?

We utilise different distributions for our data generators, aiming at modelling the varying behaviour of different components. Ideally, this leads to differences in detection precision or recall of a classifier.

Our threshold for this is 5 percentage points. If we find the difference between the precision or recall of any two classifiers to be over 5 points, we assume the generated data leads to significant differences in detection.

Experimental set-up

We generated 1 million data points in total and randomly sampled three sets of 100,000 data points each (see Table 5.3). Each set contained data points of all data generator types. *OneClassSVM* classifiers with default parameters were trained for each data generator type and sample. The data for each classifier was split up into separate training and scoring sets (see *Data selection process* on page 59).

Measure	**Set 1**	**Set 2**	**Set 3**
Number of elements	100,000	100,000	100,000
Normal samples	71.41 %	71.41 %	71.48 %
Duplicates	0.55 %	0.52 %	0.50 %

Table 5.3: *The three dataset samples and their quality measures. Percentages are rounded to two decimal places.*

In total, we trained and scored classifiers for 10 data categories and repeated this for each dataset sample, resulting in 30 rounds.

Results

After three iterations with three different datasets, we arrived at an average precision and recall percentage of the classifiers for each data generator type (see Figure 5.16).

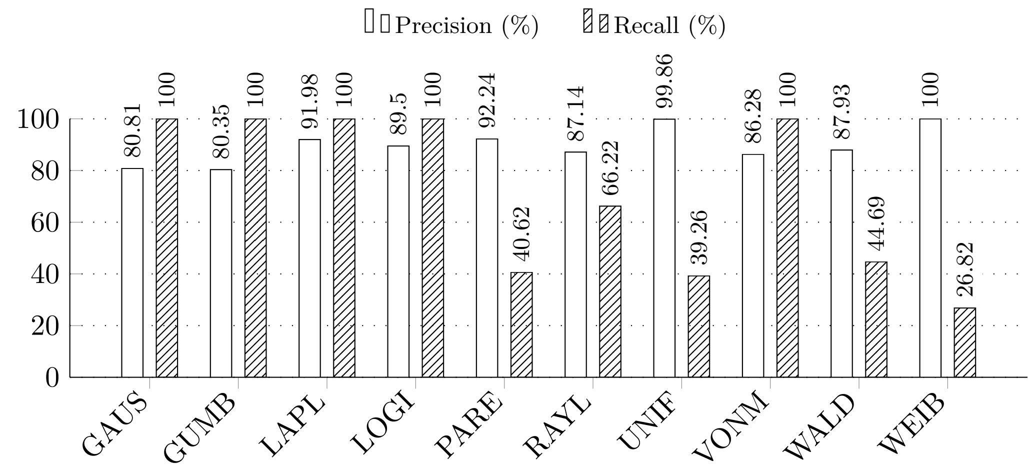


Figure 5.16: *Average precision and recall of a* OneClassSVM *classifier on three different datasets of the respective data categories.*

The difference between the lowest and highest precision is 19.65 points, for the recall it is 73.18 points. For most data categories our requirement is sufficiently met. We conclude that there are significant differences in detection precision and recall for different data generator types.

5.3.3 RQ 9: Can we show the effect of varying intrusion levels?

We implemented intrusion levels aiming at subjecting IDS to different difficulty levels. Data generated with intrusion level *hard* is supposed to lead to reduced detection accuracy compared to level *easy*, as the intruded samples for level *hard* lie much closer to the normal data.

Based on our current implementation, we do not expect the intrusion levels for categorical data in form of the positional requests to lead to any differences. The intrusions levels for these requests only influence the frequency of intrusion samples for each client. As our IDS prototype does not consider client-specific profiles, it is not influenced by this difference. Contrary to that, intrusion levels implemented for the data generators and colour values, which focus more on *anomaly-based* detection, are expected to have an effect on the outlier detection algorithm that we apply here.

Our threshold for this is again 5 percentage points. If we find the difference between the precision or recall of any classifier between the intrusion levels to be over 5 points, we assume changing the level leads to significant differences in detection accuracy.

Experimental set-up

We generated 1 million data points for intrusion levels *easy* and *hard* respectively, and randomly sampled three sets of $100,000$ data points for both levels (see Table 5.4). Each set contained data points of all data categories. *OneClassSVM* classifiers with default parameters were trained for each level, data category and sample. The data for each classifier was split up into separate training and scoring sets (see *Data selection process* on page 59).

Measure	**Intrusion level *easy***			**Intrusion level *hard***		
	Set 1	**Set 2**	**Set 3**	**Set 1**	**Set 2**	**Set 3**
Number of elements	$100,000$	$100,000$	$100,000$	$100,000$	$100,000$	$100,000$
Normal samples	73.08 %	73.27 %	73.34 %	72.97 %	73.00 %	72.90 %
Duplicates	0.38 %	0.41 %	0.37 %	0.38 %	0.38 %	0.38 %

Table 5.4: *The six dataset samples and their quality measures. Percentages are rounded to two decimal places.*

In total, we trained and scored classifiers for 14 data categories and both levels respectively, and repeated this for each dataset sample, resulting in 84 rounds.

Results

After three iterations with three different datasets, we arrived at an average precision and recall percentage of the classifiers for each data category and both intrusion levels (see Figure 5.17 and Figure 5.18).

As expected, precision and recall do not change significantly for the positional requests POIR and ROUT. We cannot explain the difference observed for CCOD data. For five data generators, we successfully showed the effect of increasing the intrusion level. GAUS, GUMB, LAPL, LOGI and VONM data do not show significant differences in precision and recall between intrusion levels. Finally, COLR data shows a visible effect of increasing the intrusion level.

In conclusion, additional and more specific experiments are necessary to judge the quality of individual intrusion levels for different data categories. The currently implemented broad intrusions seem to have different effects for different data categories, not showing the intended effect for all types of requests.

5.3.4 RQ 10: What false positive rate can we observe for different data categories?

We want to evaluate how precisely a classifier with no tuning and default parameters can detect intrusions in data generated by the testbed. For evaluation, we use the *false positive rate* FPR, defined as: $\text{FPR} = \frac{\text{number of false positives}}{\text{total number of negative samples}}$

A lower false positive rate means that our data is easy to predict, and less tuning is required to achieve sensible results. The best FPR is 0. We consider an FPR of over 0.5 to

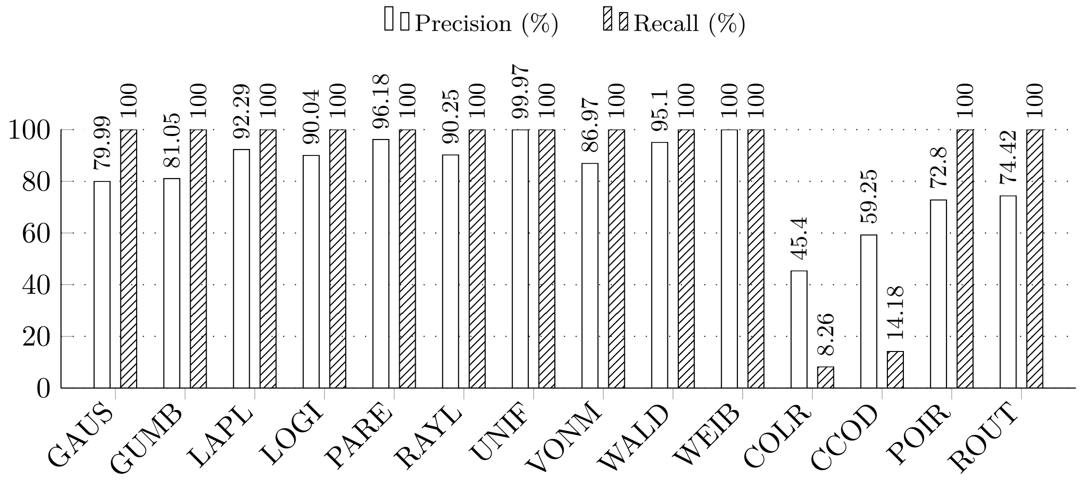


Figure 5.17: *Average precision and recall of a* OneClassSVM *classifier on three different datasets of the respective data categories with intrusion level* easy*.*

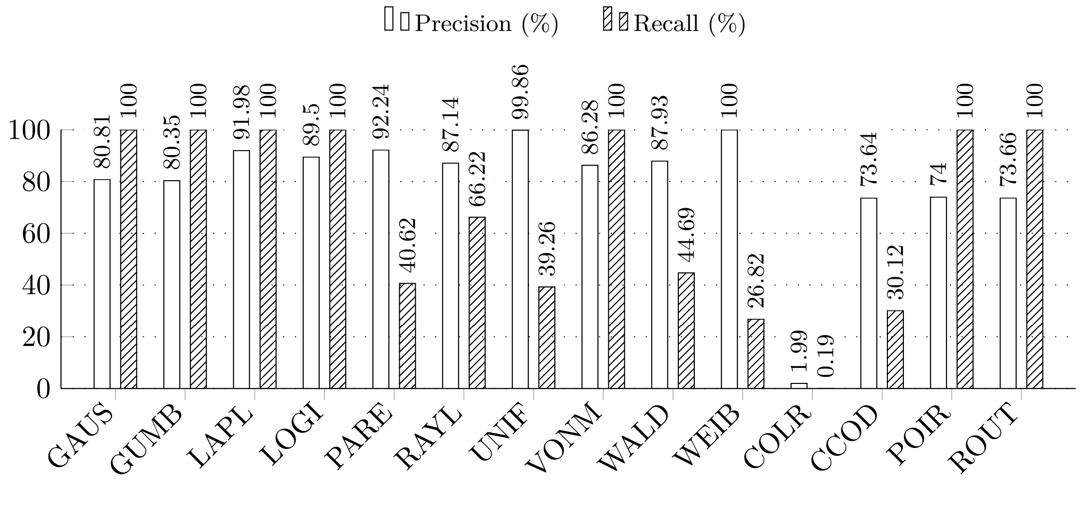


Figure 5.18: *Average precision and recall of a* OneClassSVM *classifier on three different datasets of the respective data categories with intrusion level* hard*.*

be an indicator that the data is too hard to predict, as with two possible classes a classifier could simply guess to achieve this FPR.

Based on experimental results by other authors [44, 101], we define an FPR of less than 0.05 to be achievable, one of less than 0.15 as normal. Accordingly, if we observe FPRs in $[0.15, 0.5]$ for our experiment, we consider it successful.

Experimental set-up

We generated 1 million data points for intrusion level *hard*, and randomly sampled three sets of $100,000$ data points each (see Table 5.5). Each set contained data points of all data categories. *OneClassSVM* classifiers with default parameters were trained for each data category and sample. The data for each classifier was split up into separate training and scoring sets (see *Data selection process* on page 59).

Measure	**Set 1**	**Set 2**	**Set 3**
Number of elements	$100,000$	$100,000$	$100,000$
Normal samples	73.12 %	73.07 %	73.10 %
Duplicates	0.40 %	0.36 %	0.39 %

Table 5.5: *The three dataset samples and their quality measures. Percentages are rounded to two decimal places.*

In total, we trained and scored classifiers for 14 data categories and repeated this for each dataset sample, resulting in 42 rounds.

Results

We calculated the average FPR for each data category over three rounds (see Figure 5.19). The classifiers for all data categories show a higher FPR than our threshold of 0.15. It is important to stress that the classifiers have not been tuned to improve their performance. For COLR, CCOD, POIR and ROUT data, we observe FPRs of approximately 0.5, indicating that these data categories are hard to predict and might require fine tuning. Still, our target condition is fulfilled so we consider the results a positive indicator for the difficulty level of detecting intrusions in our testbed.

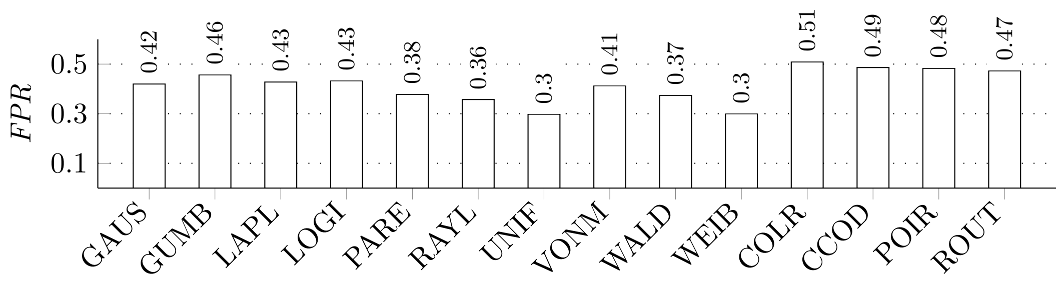


Figure 5.19: *Average false positive rate of a* OneClassSVM *classifier on three different datasets of the respective data categories.*

6 Limitations

In the following, we want to discuss those extensions of our testbed approach and evaluation which were not carried out as they were beyond the scope of our work.

6.1 Testbed

Our testbed concept and implementation were created from the ground up. Accordingly, they were limited in scope. That is most evident in the small number of components and variation of those. Additionally, each part of the testbed had to be narrowed down, discussed in the following.

6.1.1 Data generators

The data generated by the data generator components is restricted and lacks strong differentiation. Their distribution intervals can be enhanced with component-specific profiles to increase the challenge in learning their behaviour.

6.1.2 2D simulator

With the 2D simulator component, we try to emulate complex components in a client. As such, it sends requests mirroring domain knowledge-based patterns. Still, we only modelled this type of data with basic domain knowledge, omitting more intricate requests.

Simulated environment

Due to computational limits, the simulated environment is small, measuring only 500×500 pixels. This could and should be increased, given the appropriate computing power.

Equally, the simulated background colours are limited to seven discrete options. A more advanced scenario with continuous colour values was left out, which could potentially increase the difficulty level.

The simulated environment also lacks obstacles, routes, and other environment factors influencing the simulated units, which could more realistically depict the scenarios that we model.

Finally, there is currently only one unit being simulated in one separate environment for each client. A more realistic scenario would be to simulate all units in the same environment and let them interact with each other. This significantly more complex simulation was beyond the scope of our work.

Simulated unit

The simulated unit is, in its current form, rather simple. It can move according to a vector $\vec{v} = (v_x, v_y)$, which defines a velocity for the x and y direction, respectively. We did not implement functionality allowing acceleration, deacceleration, and steering patterns to be sent. Additionally, the movement generator is currently limited to one default behaviour profile. A possible extension to this would be individual “user profiles”.

The unit intelligence is also simple, lacking more sophisticated logic like path finding, lane keeping, or collision avoidance algorithms.

Finally, the data sent from the unit is currently restricted to its position and the colour it sees, to enable basic use cases. This data can be expanded to include the speed of the unit, its acceleration, distance covered since the last request, highest speed achieved, and many more. All these simply need to be calculated and sent.

6.1.3 Intrusions

Clearly, the number of intrusions we have modelled is limited and could be increased. More interesting is the prospect of introducing advanced intrusion types. We discuss some possibilities for expanding on the intrusions in Chapter 7.

6.1.4 Server

The testbed server aims for high performance and lightweight processing of incoming requests by various components. We tried to ensure a high throughput and to mitigate performance bottlenecks.

The current implementation is limited by storing all log data in the external memory. We did not make use of a high-performance database for storage that can increase throughput and durability, such as *redis* [91].

Additionally, the server is implemented in Python, utilising the *gevent* [5] library to support some form of multitasking with coroutines. This does improve performance compared to a single-threaded implementation. Still, a real parallel or distributed implementation would allow for performance gains.

Finally, the server functionality is currently restricted to storing or forwarding incoming requests. More complex interaction with clients including actual responses is missing.

6.2 Evaluation

We built the testbed to allow for the comparison of various IDS. To show its applicability in this scenario, we devised and evaluated multiple research questions. Still, our limited scope only allowed us to assess basic aspects. More complex and advanced IDS may perform differently and show unintended side effects or weaknesses of the testbed that we did not find with our simple tests.

Finally, we could not evaluate the data generated by the testbed with more advanced techniques for assessing the quality of test data. One example for such an assessment is “fuzzy qualitative modelling” proposed by Haider et al. [36]. They include multiple data points such as the “maximum number of possible attacks” and if “ground truth information [is] included to assist [the] labelling process” [36, p. 186] in their input to derive a quality measure. Without such validation, we cannot make an informed judgement about the quality and efficacy of the generated data.

7 Future work

This chapter is aimed at presenting areas of interest for future work that we have created with our thesis. Based on those, we discuss scenarios that we deem interesting to pursue.

7.1 Expanding the testbed

In its current form, our testbed is a first step towards a comprehensive solution for comparing IDS. The data generator components and 2D simulator can be expanded on, especially by implementing more advanced behaviour patterns. Additionally, the performance of all parts, including the server, could be increased.

For the 2D simulator component, having multiple units interact in a single, shared environment would allow for a multitude of new possibilities for intrusions, expected behaviours, and detection algorithms. One example is a peer group analysis algorithm that could try to match different units' behaviour and check for deviations, as shown by Bolton et al. [6].

Finally, the currently defined and implemented intrusions can be extended. Based on domain knowledge, more fine-grained intrusions may be implemented. Known attack patterns can be encoded in the testbed definition or be diversified to develop robust metrics for the evaluation of IDS. A complex but compelling idea is to model *advanced persistent threats* [104]. In short, these attacks aim not for fast and blunt access to a system, but for "long-term control and data collection" [104, p. 16]. That leads to the behaviour of the compromised system to be almost indistinguishable from a normal system. The implementation of these types of intrusions would allow the evaluation of more advanced detection systems, such as *Artificial Neural Networks* (ANN) [20] or *model-based reasoning* approaches [33].

7.2 Evaluating IDS with the testbed

Our testbed is a complex suite of tools and components, each well-defined and controllable. It allows for modelling custom scenarios for the evaluation of IDS, which may otherwise not be easy to create. We want to show the range of options that our testbed opens for evaluating various IDS.

Comparing different types of IDS

An essential feature of our testbed is that it allows for the comparison of different types of IDS. Many researchers choose the KDD Cup 1999 dataset [53] because it is widely used and as such allows for comparing their solution to others. As we have discussed (see Section 4.2.1 on page 25), this dataset is unfit for evaluating state-of-the-art IDS.

With the help of our testbed, a more up-to-date and flexible evaluation of IDS is possible. Hence, it allows for comparing the various systems that we have found in our literature survey. Such a comparison could offer interesting insights as to which system fares best under different conditions.

Additionally, the modular nature of the testbed server allows for quickly adding or swapping different IDS modules. This may be used for a more direct comparison of the performance of different IDS, especially facilitating the evaluation of real-time systems.

Varying intrusion scenarios

Once set up, the testbed can be configured for multiple intrusion scenarios that can be activated at will. Many more complex scenarios would be interesting to evaluate with advanced IDS.

An exemplary scenario is to introduce new intrusion patterns and evaluate how quickly different systems can adapt to those. Additionally, it would be interesting to develop edge cases for evaluation. For example, we envision implementing slightly anomalous, but legitimate behaviour patterns or, contrary to that, intrusions that are hard to differentiate from normal behaviour. With these scenarios, one can vary the parameters for different IDS and develop sensible thresholds to optimise the number of false positives and false negatives, depending on their requirements and the available resources.

Correlating intrusion traces in the training data to the false positive rate

A relevant topic of research for *anomaly-based* detection systems is the detection over noisy training data. Eskin [27] describes an advanced algorithm that can handle noisy training data when trying to detect anomalies. Naïve algorithms are shown to perform worse when intrusion traces are present in the training data [27, p. 6].

With our testbed, this research can be expanded on. A compelling experiment would be to correlate the number, type, or level of intrusions in the training data to the false positive rate of the system when used for detecting intrusions.

Generating data to evaluate specific use cases

When implementing the testbed, we modelled the automotive real-world scenario of our industry partner. With its current structure, the testbed can be adapted to various use cases that only loosely resemble the current configuration.

This adaptability enables generating data for several scenarios, which allows for faster and easier evaluation of various aspects. For example, we have shown that one-hot encoding can have significant performance benefits compared to a simple mapping of categorical data to integer labels. With the testbed, we were able to set up this experiment quickly and scale it arbitrarily, thereby allowing us to show the improved performance more reliably. This quality can be used for many other scenarios we would have liked to evaluate. One example would be to vary the distribution of different data categories in the test set and possibly reveal weaknesses in common practices of implementing IDS. Another interesting experiment would be to correlate the number of entries used for training to the detection accuracy of the trained classifier. Using less data for training can significantly shorten the training time, but arguably at the risk of worse classifier performance.

8 Conclusion

This thesis contributes two main aspects to the field of IDS: First, our extensive literature survey and comprehensive taxonomy of existing approaches to IDS gives an overview over the large variety of possible implementations. The results of our survey and the taxonomy are suitable as a basis for targeted discussion and research of IDS and improve on previous overviews. As we focussed on approaches that have been shown to work in practice, our work can be used as a starting point for the implementation of IDS in different scenarios.

Second, the testbed that we conceptualised and implemented enables the comparison of various IDS in a real-world scenario. It is adapted to our automotive use case but can be adjusted to numerous other scenarios with only minor edits. Its architecture and modularity allow for various configurations and seamless addition of new components.

In the evaluation, we find that the testbed fulfils its objectives. Its performance is consistent and with additional components the testbed scales linearly. This allows for complex simulations to be carried out. To assess the fitness for purpose of the testbed, we evaluate multiple quality metrics derived from our literature survey. We find that limitations of existing datasets used in research are solved by our implementation. When generating data repeatedly, the testbed exhibits high performance and reliable behaviour. This allows the reproduction of equally distributed data, which is beneficial for repeated evaluation and cross-validation. Additionally, we find that the testbed can be used to evaluate the various types of IDS we have identified in our survey. To conclude our evaluation, we assess the quality of the data generated by the testbed. Most of the data categories of the testbed fulfil our expectation of being of varying detection difficulty. The intrusion levels we have introduced lead to measurable differences in detection accuracy, allowing for varied evaluation of IDS. Lastly, we find that a naïve detection algorithm exhibits high false positive rates for all types of data generated. We argue that this confirms the challenge of detecting intrusions in the data generated by our testbed.

There are some limitations to our solution. This mainly regards our assumption that compromised clients can be identified based on their communication with a remote server. We consider this assumption fair, as existing vehicle defence systems have been proposed based on remote detection.

In conclusion, our testbed allows the flexible and reliable evaluation of IDS of various types and in multiple scenarios. Common problems of existing evaluation approaches are solved, which allows a more up-to-date and effective comparison of different IDS. Because of the reproducibility of the generated data, the same system can be tested multiple times to rule out possible side effects. Additionally, the generation of arbitrary amounts of new test data allows for effective cross-validation. Finally, other researchers can create custom evaluation scenarios with the testbed, without having to rely on outdated or fixed datasets.

The source code of our testbed is available for inspection, modification and use (see Appendix A). We hope that this work will advance the research of IDS.

List of Figures

List of Listings

List of Tables

Bibliography

[1] Agnar Aamodt and Enric Plaza. "Case-based reasoning: Foundational issues, methodological variations, and system approaches". In: *AI Communications* 7.1 (1994), pp. 39–59.

[2] Accenture. "Share of new cars sold that are connected to the internet worldwide from 2015 to 2025". In: *Connected vehicle – Succeeding with a disruptive technology*. 2015, p. 3. URL: `https://www.accenture.com/_acnmedia/Accenture/Conversion-Assets/DotCom/Documents/Global/PDF/Dualpub_21/Accenture-digital-Connected-Vehicle.pdf` (visited on 2018-04-24).

[3] Stefan Axelsson. *Intrusion detection systems: A survey and taxonomy*. Tech. rep. Chalmers University of Technology, 2000.

[4] Fabrice Bellard. *QEMU processor emulator*. URL: `https://www.qemu.org/` (visited on 2018-04-27).

[5] Denis Bilenko. *gevent*. URL: `http://www.gevent.org/` (visited on 2018-04-07).

[6] Richard J Bolton, David J Hand, et al. "Unsupervised profiling methods for fraud detection". In: *Proceedings of Credit Scoring and Credit Control VII*. 2001, pp. 5–7.

[7] Tim Bray. "The JavaScript object notation (JSON) data interchange format". In: 8259 (2017). URL: `http://www.rfc-editor.org/rfc/rfc8259.txt` (visited on 2018-04-21).

[8] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. *Extensible Markup Language (XML)*. Tech. rep. World Wide Web Consortium, 1997.

[9] Richard R Brooks, Sam Sander, Juan Deng, and Joachim Taiber. "Automobile security concerns". In: *IEEE Vehicular Technology Magazine* 4.2 (2009), pp. 52–64.

[10] Andrew Butterfield, Gerard E Ngondi, and Anne Kerr. *A Dictionary of Computer Science*. Oxford University Press, 2016.

[11] James Cannady. "Artificial neural networks for misuse detection". In: *Proceedings of the 21st National Information Systems Security Conference*. NIST. 1998, pp. 443–456.

[12] Canonical. *Ubuntu 16.04.4 LTS release notes*. URL: `https://wiki.ubuntu.com/XenialXerus/ReleaseNotes` (visited on 2018-04-27).

[13] Varun Chandola, Arindam Banerjee, and Vipin Kumar. "Anomaly detection: A survey". In: *ACM Computing Surveys* 41.3 (2009), pp. 1–58.

[14] Chih-Chung Chang and Chih-Jen Lin. "LIBSVM: A library for support vector machines". In: *ACM Transactions on Intelligent Systems and Technology* 2.3 (2011).

[15] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and William P Kegelmeyer. "SMOTE: Synthetic minority over-sampling technique". In: *Journal of Artificial Intelligence Research* 16 (2002), pp. 321–357.

[16] Nitesh V Chawla, Nathalie Japkowicz, and Aleksander Kotcz. “Special issue on learning from imbalanced data sets”. In: *ACM SIGKDD Explorations Newsletter* 6.1 (2004), pp. 1–6.

[17] Stephen Checkoway et al. “Comprehensive experimental analyses of automotive attack surfaces”. In: *Proceedings of the 20th USENIX Conference on Security.* USENIX. San Francisco, CA, 2011.

[18] Yuk Y Chung and Noorhaniza Wahid. “A hybrid network intrusion detection system using simplified swarm optimization (SSO)”. In: *Applied Soft Computing* 12.9 (2012), pp. 3014–3022.

[19] Dipankar Dasgupta. “Advances in artificial immune systems”. In: *IEEE Computational Intelligence Magazine* 1.4 (2006), pp. 40–43.

[20] Hervé Debar, Monique Becker, and Didier Siboni. “A neural network component for an intrusion detection system”. In: *Proceedings of the 1992 IEEE Computer Society Symposium on Research in Security and Privacy.* IEEE. 1992, pp. 240–250.

[21] Hervé Debar and Marc Dacier. *An experimentation workbench for intrusion detection systems.* Research Report. IBM Research Division, Zurich Research Laboratory, 1998.

[22] Hervé Debar, Marc Dacier, and Andreas Wespi. “A revised taxonomy for intrusion-detection systems”. In: *Annals of Telecommunications* 55.7 (2000), pp. 361–378.

[23] John E Dickerson and Julie A Dickerson. “Fuzzy network profiling for intrusion detection”. In: *Proceedings of the 19th International Conference of the North American Fuzzy Information Processing Society.* IEEE. 2000, pp. 301–306.

[24] Richard O Duda and Peter E Hart. *Pattern Classification and Scene Analysis.* Vol. 1. Wiley, 1973.

[25] Mahmoud H Eiza and Qiang Ni. “Driving with sharks: Rethinking connected vehicles with vehicle cybersecurity”. In: *IEEE Vehicular Technology Magazine* 12.2 (2017), pp. 45–51.

[26] Ericsson. *Connected vehicle cloud: Under the hood.* 2015. URL: `https://archive.ericsson.net/service/internet/picov/get?DocNo=28701-FGD101192` (visited on 2018-05-09).

[27] Eleazar Eskin. “Anomaly detection over noisy data using learned probability distributions”. In: *Proceedings of the 17th International Conference on Machine Learning.* Morgan Kaufmann Publishers, 2000, pp. 255–262.

[28] Eleazar Eskin, Andrew Arnold, Michael Prerau, Leonid Portnoy, and Sal Stolfo. “A geometric framework for unsupervised anomaly detection: Detecting intrusions in unlabeled data”. In: *Applications of Data Mining in Computer Security.* Springer US, 2002, pp. 77–101.

[29] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. "A density-based algorithm for discovering clusters in large spatial databases with noise". In: *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining.* Vol. 96. 34. ACM. 1996, pp. 226–231.

[30] Josh Faust. *ROS turtlesim.* URL: `http://wiki.ros.org/turtlesim` (visited on 2018-04-03).

[31] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. "A sense of self for Unix processes". In: *Proceedings of the 1996 IEEE Symposium on Security and Privacy.* IEEE. 1996, pp. 120–128.

[32] Pedro Garcia-Teodoro, Jesus Diaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. "Anomaly-based network intrusion detection: Techniques, systems and challenges". In: *Computers & Security* 28.1 (2009), pp. 18–28.

[33] Thomas D Garvey and Teresa F Lunt. "Model-based intrusion detection". In: *Proceedings of the 14th National Computer Security Conference.* Vol. 17. NIST. 1991.

[34] Jonatan Gomez and Dipankar Dasgupta. "Evolving fuzzy classifiers for intrusion detection". In: *Proceedings of the 2002 IEEE Workshop on Information Assurance.* Vol. 6. 3. IEEE. 2002, pp. 321–323.

[35] Frank E Grubbs. "Procedures for detecting outlying observations in samples". In: *Technometrics* 11.1 (1969), pp. 1–21.

[36] Waqas Haider, Jiankun Hu, Jill Slay, Benjamin P Turnbull, and Yi Xie. "Generating realistic intrusion detection system dataset based on fuzzy qualitative modeling". In: *Journal of Network and Computer Applications* 87 (2017), pp. 185–192.

[37] Xiaoshu Hang and Honghua Dai. "Applying both positive and negative selection to supervised learning for anomaly detection". In: *Proceedings of the 7th Conference on Genetic and Evolutionary Computation.* ACM. 2005, pp. 345–352.

[38] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. "Supervised learning". In: *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer, 2009. Chap. 2, pp. 9 sq.

[39] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. "Unsupervised learning". In: *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer, 2009. Chap. 14, pp. 485 sq.

[40] Marti A Hearst, Susan T Dumais, Edgar Osuna, John Platt, and Bernhard Scholkopf. "Support vector machines". In: *IEEE Intelligent Systems and their Applications* 13.4 (1998), pp. 18–28.

[41] Marcel Hellkamp. *Bottle.* URL: `http://bottlepy.org/` (visited on 2018-04-07).

[42] Paul Helman and Gunar Liepins. "Statistical foundations of audit trail analysis for the detection of computer misuse". In: *IEEE Transactions on Software Engineering* 19.9 (1993), pp. 886–901.

[43] Victoria Hodge and Jim Austin. “A survey of outlier detection methodologies”. In: *Artificial Intelligence Review* 22.2 (2004), pp. 85–126.

[44] Shi-Jinn Horng et al. “A novel intrusion detection system based on hierarchical clustering and support vector machines”. In: *Expert Systems with Applications* 38.1 (2011), pp. 306–313.

[45] IHS. *Average age of light vehicles in the U.S. from 2003 to 2016 (in years)*. Statista. 2016. URL: `https://www.statista.com/statistics/261881/average-age-of-light-vehicles-in-the-united-states/` (visited on 2018-04-24).

[46] Koral Ilgun, Richard A Kemmerer, and Phillip A Porras. “State transition analysis: A rule-based intrusion detection approach”. In: *IEEE Transactions on Software Engineering* 21.3 (1995), pp. 181–199.

[47] Kurt Jensen. “Coloured Petri nets”. In: *Petri nets: Central models and their properties*. Springer Berlin Heidelberg, 1987, pp. 248–299.

[48] Mon-Fong Jiang, Shian-Shyong Tseng, and Chih-Ming Su. “Two-phase clustering process for outliers detection”. In: *Pattern Recognition Letters* 22.6 (2001), pp. 691–700.

[49] Mon-Fong Jiang, Ching-Hung Wang, and Shian-Shyong Tseng. “Developing a sugarcane breeding assistant system by a hybrid adaptive learning technique”. In: *Proceedings of the 1996 IEEE International Conference on Systems, Man, and Cybernetics*. Vol. 2. IEEE. 1996, pp. 1196–1201.

[50] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. 2001. URL: `http://www.scipy.org/` (visited on 2018-03-30).

[51] Rick Joyce and Gopal Gupta. “Identity authentication based on keystroke latencies”. In: *Communications of the ACM* 33.2 (1990), pp. 168–176.

[52] Peyman Kabiri and Ali A Ghorbani. “Research on intrusion detection and response: A survey”. In: *International Journal of Network Security* 1.2 (2005), pp. 84–102.

[53] *KDD cup 1999 dataset description*. URL: `http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html` (visited on 2017-11-16).

[54] *KDD cup 1999 task description*. URL: `http://kdd.ics.uci.edu/databases/kddcup99/task.html` (visited on 2018-03-26).

[55] Gene H Kim and Eugene H Spafford. “The design and implementation of Tripwire: A file system integrity checker”. In: *Proceedings of the 2^{nd} ACM Conference on Computer and Communications Security*. ACM. 1994, pp. 18–29.

[56] Edwin M Knorr and Raymond T Ng. “Algorithms for mining distance-based outliers in large datasets”. In: *Proceedings of the 24^{th} International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers, 1998, pp. 392–403.

[57] Edwin M Knorr, Raymond T Ng, and Vladimir Tucakov. “Distance-based outliers: algorithms and applications”. In: *The VLDB Journal* 8.3-4 (2000), pp. 237–253.

[58] Donald E Knuth. “Algorithm R - Reservoir sampling”. In: *The Art of Computer Programming, Volume 2: Seminumeral Algorithms*. Addison-Wesley Professional, 1981. Chap. Random Sampling and Shuffling, p. 138.

[59] Levent Koc, Thomas A Mazzuchi, and Shahram Sarkani. “A network intrusion detection system based on a hidden naïve Bayes multiclass classifier”. In: *Expert Systems with Applications* 39.18 (2012), pp. 13492–13500.

[60] Christopher Kruegel, Darren Mutz, William Robertson, and Fredrik Valeur. “Bayesian event classification for intrusion detection”. In: *Proceedings of the 19th Annual Computer Security Applications Conference*. IEEE. 2003, pp. 14–23.

[61] Sandeep Kumar and Eugene H Spafford. “A pattern matching model for misuse intrusion detection”. In: *Proceedings of the 17th National Computer Security Conference*. NIST. 1994, pp. 11–21.

[62] Aleksandar Lazarevic, Vipin Kumar, and Jaideep Srivastava. “Intrusion detection: A survey”. In: *Managing Cyber Threats: Issues, Approaches, and Challenges*. Springer US, 2005, pp. 19–78.

[63] Leibniz Supercomputing Centre of the Bavarian Academy of Sciences and Humanities. *About the Leibniz Supercomputing Centre*. URL: `https://www.lrz.de/wir/lrz-flyer/lrz-flyer.pdf` (visited on 2018-05-08).

[64] Kingsly Leung and Christopher Leckie. “Unsupervised anomaly detection in network intrusion detection using clusters”. In: *Proceedings of the 28th Australasian Conference on Computer Science*. Vol. 38. Australian Computer Society. 2005, pp. 333–342.

[65] James Lewis and Martin Fowler. *Microservices: a definition of this new architectural term*. 2014. URL: `https://martinfowler.com/articles/microservices.html` (visited on 2018-03-23).

[66] Wei Li. “Using genetic algorithm for network intrusion detection”. In: *Proceedings of the United States Department of Energy Cyber Security Group 2004 Training Conference* 1 (2004), pp. 1–8.

[67] Yinhui Li et al. “An efficient intrusion detection system based on support vector machines and gradually feature removal method”. In: *Expert Systems with Applications* 39.1 (2012), pp. 424–430.

[68] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. “Intrusion detection system: A comprehensive review”. In: *Journal of Network and Computer Applications* 36.1 (2013), pp. 16–24.

[69] Ulf Lindqvist and Phillip A Porras. “Detecting computer and network misuse through the production-based expert system toolset (P-BEST)”. In: *Proceedings of the 1999 IEEE Symposium on Security and Privacy*. IEEE. 1999, pp. 146–161.

[70] Richard Lippmann, Joshua W Haines, David J Fried, Jonathan Korba, and Kumar Das. “The 1999 DARPA off-line intrusion detection evaluation”. In: *Computer Networks* 34.4 (2000), pp. 579–595.

[71] Antonio Loureiro, Luis Torgo, and Carlos Soares. “Outlier detection using clustering methods: A data cleaning application”. In: *Proceedings of KDNet Symposium on Knowledge-Based Systems for the Public Sector*. 2004.

[72] Teresa F Lunt. “IDES: An intelligent system for detecting intruders”. In: *Proceedings of the Symposium on Computer Security, Threat and Countermeasures*. 1990, pp. 30–45.

[73] Peter Mell, Vincent Hu, Richard Lippmann, Josh Haines, and Marc Zissman. “An overview of issues in testing intrusion detection systems”. In: *NIST Special Publication*. NISTIR 7007 (2003). URL: `https://csrc.nist.gov/publications/detail/nistir/7007/final` (visited on 2018-04-21).

[74] Regine Meunier. “The pipes and filters architecture”. In: *Pattern Languages of Program Design*. Addison-Wesley. 1995, pp. 427–440.

[75] Srinivas Mukkamala, Guadalupe Janoski, and Andrew Sung. “Intrusion detection using neural networks and support vector machines”. In: *Proceedings of the 2002 International Joint Conference on Neural Networks*. Vol. 2. IEEE. 2002, pp. 1702–1707.

[76] James R Norris. *Markov Chains*. 2. Cambridge University Press, 1998.

[77] *NSL-KDD dataset description*. URL: `http://www.unb.ca/cic/datasets/nsl.html` (visited on 2018-02-04).

[78] Hisashi Oguma et al. “New attestation-based security architecture for in-vehicle communication”. In: *Proceedings of the 2008 IEEE Global Telecommunications Conference*. IEEE. 2008, pp. 1–6.

[79] Sang H Oh and Won S Lee. “An anomaly intrusion detection method by clustering normal user behavior”. In: *Computers & Security* 22.7 (2003), pp. 596–612.

[80] Nam H Park and Won S Lee. “Statistical grid-based clustering over data streams”. In: *ACM SIGMOD Record* 33.1 (2004), pp. 32–37.

[81] Simon Parkinson, Paul Ward, Kyle Wilson, and Jonathan Miller. “Cyber threats facing autonomous and connected vehicles: Future challenges”. In: *IEEE Transactions on Intelligent Transportation Systems* 18.11 (2017), pp. 2898–2915.

[82] Animesh Patcha and Jung-Min Park. “An overview of anomaly detection techniques: Existing solutions and latest technological trends”. In: *Computer Networks* 51.12 (2007), pp. 3448–3470.

[83] Fabian Pedregosa et al. “Scikit-learn: Machine learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[84] Chao-Ying J Peng, Kuk L Lee, and Gary M Ingersoll. “An introduction to logistic regression analysis and reporting”. In: *The Journal of Educational Research* 96.1 (2002), pp. 3–14.

[85] Leonid Portnoy, Eleazar Eskin, and Sal Stolfo. “Intrusion detection with unlabeled data using clustering”. In: *Proceedings of the 2001 ACM CSS Workshop on Data Mining Applied to Security*. ACM. 2001.

[86] Morgan Quigley et al. “ROS: An open-source robot operating system”. In: *Proceedings of the 2009 IEEE International Conference on Robotics and Automation Workshop on Open Source Robotics*. Vol. 3. 3.2. 2009, p. 5.

[87] Montaser N Ramadan, Mohammad A Al-Khedher, and Sharaf A Al-Kheder. “Intelligent anti-theft and tracking system for automobiles”. In: *International Journal of Machine Learning and Computing* 2.1 (2012), p. 83.

[88] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim. “Efficient algorithms for mining outliers from large data sets”. In: *ACM SIGMOD Record*. Vol. 29. 2. ACM. 2000, pp. 427–438.

[89] Lee M Rossey et al. “LARIAT: Lincoln adaptable real-time information assurance testbed”. In: *Proceedings of the 2002 IEEE Aerospace Conference*. Vol. 6. IEEE. 2002, pp. 6–6.

[90] Joerg Sander. “Density-based clustering”. In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Springer US, 2010, pp. 270–273.

[91] Salvatore Sanfilippo. *redis*. URL: `https://redis.io/` (visited on 2018-04-20).

[92] Karen Scarfone and Peter Mell. “Guide to intrusion detection and prevention systems (IDPS)”. In: *NIST Special Publication* 800-94 (2007). URL: `https://csrc.nist.gov/publications/detail/sp/800-94/final` (visited on 2018-04-21).

[93] scikit-learn developers. *Encoding categorical features*. URL: `http://scikit-learn.org/stable/modules/preprocessing.html#encoding-categorical-features` (visited on 2018-04-24).

[94] scikit-learn developers. *Precision, recall and F-measures*. URL: `http://scikit-learn.org/stable/modules/model_evaluation.html#precision-recall-f-measure-metrics` (visited on 2018-04-25).

[95] scikit-learn developers. *Standardization, or mean removal and variance scaling*. URL: `http://scikit-learn.org/stable/modules/preprocessing.html#standardization-or-mean-removal-and-variance-scaling` (visited on 2018-04-24).

[96] David W Scott, Richard A Tapia, and James R Thompson. “Kernel density estimation revisited”. In: *Nonlinear Analysis* 1.4 (1977), pp. 339–372.

[97] Ramasubramanian Sekar et al. “Specification-based anomaly detection: A new approach for detecting network intrusions”. In: *Proceedings of the 9th ACM Conference on Computer and Communications Security*. ACM. 2002, pp. 265–274.

[98] R Shanmugavadivu and N Nagarajan. “Network intrusion detection system using fuzzy logic”. In: *Indian Journal of Computer Science and Engineering* 2.1 (2011), pp. 101–111.

[99] Shiuhpyng W Shieh and Virgil D Gligor. “A pattern-oriented intrusion-detection model and its applications”. In: *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE. 1991, pp. 327–342.

[100] Ali Shiravi, Hadi Shiravi, Mahbod Tavallaee, and Ali A Ghorbani. “Toward developing a systematic approach to generate benchmark datasets for intrusion detection”. In: *Computers & Security* 31.3 (2012), pp. 357–374.

[101] Siva S S Sindhu, Suryakumar Geetha, and Arputharaj Kannan. “Decision tree based light weight intrusion detection using a wrapper approach”. In: *Expert Systems with Applications* 39.1 (2012), pp. 129–141.

[102] Carlos Soares, Pavel Brazdil, Joaquim Costa, V Cortez, and André Carvalho. “Error detection in foreign trade data using statistical and machine learning methods”. In: *Proceedings of the 3rd International Conference on the Practical Applications of Knowledge Discovery and Data Mining*. 1999, pp. 183–188.

[103] Ole Tange. “GNU Parallel – The command-line power tool”. In: *;login: The USENIX Magazine* 36.1 (2011), pp. 42–47. URL: `http://www.gnu.org/s/parallel` (visited on 2018-04-03).

[104] Colin Tankard. “Advanced persistent threats and how to monitor and deter them”. In: *Network Security* 2011.8 (2011), pp. 16–19.

[105] Mahbod Tavallaee, Ebrahim Bagheri, Wei Lu, and Ali A Ghorbani. “A detailed analysis of the KDD CUP 99 data set”. In: *Proceedings of the 2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications*. IEEE. 2009, pp. 1–6.

[106] *Tor metrics*. URL: `https://metrics.torproject.org/` (visited on 2018-03-17).

[107] Michał J Trybulec. “Integers”. In: *Journal of Formalized Mathematics* 2 (1990), pp. 1–5.

[108] Alfonso Valdes and Keith Skinner. “Adaptive, model-based monitoring for cyber attack detection”. In: *Proceedings of the 2000 International Workshop on Recent Advances in Intrusion Detection*. Springer Berlin Heidelberg. 2000, pp. 80–93.

[109] Giovanni Vigna, William Robertson, Vishal Kher, and Richard A Kemmerer. “A stateful intrusion detection system for world-wide web servers”. In: *Proceedings of the 19th Annual Computer Security Applications Conference*. IEEE. 2003, pp. 34–43.

[110] Giovanni Vigna, Fredrik Valeur, and Richard A Kemmerer. “Designing and implementing a family of intrusion detection systems”. In: *ACM SIGSOFT Software Engineering Notes*. Vol. 28. 5. ACM. 2003, pp. 88–97.

[111] David Wagner and Drew Dean. “Intrusion detection via static analysis”. In: *Proceedings of the 2001 IEEE Symposium on Security and Privacy*. IEEE. 2001, pp. 156–168.

[112] Yun Wang. “A multinomial logistic regression modeling approach for anomaly intrusion detection”. In: *Computers & Security* 24.8 (2005), pp. 662–674.

[113] Eric W Weisstein. *Beta prime distribution*. MathWorld – A Wolfram Web Resource. URL: `http://mathworld.wolfram.com/BetaPrimeDistribution.html` (visited on 2018-04-10).

[114] Eric W Weisstein. *Gumbel distribution.* MathWorld – A Wolfram Web Resource. URL: http://mathworld.wolfram.com/GumbelDistribution.html (visited on 2018-03-30).

[115] Eric W Weisstein. *Inverse Gaussian distribution.* MathWorld – A Wolfram Web Resource. URL: http://mathworld.wolfram.com/InverseGaussianDistribution.html (visited on 2018-03-30).

[116] Eric W Weisstein. *Laplace distribution.* MathWorld – A Wolfram Web Resource. URL: http://mathworld.wolfram.com/LaplaceDistribution.html (visited on 2018-03-30).

[117] Eric W Weisstein. *Logistic distribution.* MathWorld – A Wolfram Web Resource. URL: http://mathworld.wolfram.com/LogisticDistribution.html (visited on 2018-03-30).

[118] Eric W Weisstein. *Normal distribution.* MathWorld – A Wolfram Web Resource. URL: http://mathworld.wolfram.com/NormalDistribution.html (visited on 2018-03-30).

[119] Eric W Weisstein. *Rayleigh distribution.* MathWorld – A Wolfram Web Resource. URL: http://mathworld.wolfram.com/RayleighDistribution.html (visited on 2018-03-30).

[120] Eric W Weisstein. *Uniform distribution.* MathWorld – A Wolfram Web Resource. URL: http://mathworld.wolfram.com/UniformDistribution.html (visited on 2018-03-30).

[121] Eric W Weisstein. *von Mises distribution.* MathWorld – A Wolfram Web Resource. URL: http://mathworld.wolfram.com/vonMisesDistribution.html (visited on 2018-03-30).

[122] Eric W Weisstein. *Weibull distribution.* MathWorld – A Wolfram Web Resource. URL: http://mathworld.wolfram.com/WeibullDistribution.html (visited on 2018-03-30).

[123] Darrell Whitley. “A genetic algorithm tutorial”. In: *Statistics and Computing* 4.2 (1994), pp. 65–85.

[124] *Wikimedia downloads.* URL: https://dumps.wikimedia.org/ (visited on 2018-03-17).

[125] Claes Wohlin. “Guidelines for snowballing in systematic literature studies and a replication in software engineering”. In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering.* ACM. 2014, p. 38.

[126] Xue Yang, Leibo Liu, Nitin H Vaidya, and Feng Zhao. “A vehicle-to-vehicle communication protocol for cooperative collision warning”. In: *Proceedings of the 1st Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services.* IEEE. 2004, pp. 114–123.

[127] Nong Ye, Yebin Zhang, and Connie M Borror. “Robustness of the Markov-chain model for cyber-attack detection”. In: *IEEE Transactions on Reliability* 53.1 (2004), pp. 116–123.

[128] Dit-Yan Yeung and Calvin Chow. “Parzen-window network intrusion detectors”. In: *Proceedings of the 16th International Conference on Pattern Recognition*. Vol. 4. IEEE. 2002, pp. 385–388.

[129] Tao Zhang, Helder Antunes, and Siddhartha Aggarwal. “Defending connected vehicles against malware: Challenges and a solution framework”. In: *IEEE Internet of Things Journal* 1.1 (2014), pp. 10–21.

[130] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. “BIRCH: An efficient data clustering method for very large databases”. In: *ACM SIGMOD Record*. Vol. 25. 2. ACM. 1996, pp. 103–114.

A Developed programs

Following is a quick overview of the programs we have developed for this thesis, as well as additional information that might be helpful when using them.

A.1 Aliases used in text

We sometimes use aliases in this thesis to refer to programs whose names are counterintuitive or might warrant further explanation. For completeness, we will list those aliases and their respective meaning (see Table A.1).

Alias	**Refers to**
DataGenerator	DistributionGenerator
DataPublisher	DistributionPublisher
data_generator.py	distribution_generator.py
Simulator	Turtlesim
Frame	TurtleFrame
Unit	Turtle
sim.py	py_turtlesim.py
communication_unit.py	logger.py

Table A.1: *The aliases used in this thesis and the respective class, module or program.*

A.2 Overview over programs and tools

Following is a list of programs and tools developed for this thesis (see Table A.2). Not included in the list are test modules.

Category	**File**	**Function**
Turtlesim.py	py_turtlesim.py	Host for the 2D simulator
	turtle_frame.py	Simulated environment
	turtle.py	Simulated unit
↪util	point.py	2D point
	point_f.py	2D point (float)
	rgb.py	RGB colour
ROS programs	logger.py	Client communication unit
	ros_tools.py	Interact with our ROS nodes
	GenValue.msg	Data generator ROS message
	launch_file_version_che...	Ensure launch file consistency

continued the next page...

Category	**File**	**Function**
	path_verify.py	Check global paths
↪generator	argument_constraint.py	Define generator arguments
	distribution_generator.py	Data generator
	distribution_publisher.py	Publishes data to ROS
	generators.py	Data generator definitions
↪mover	move_strategy.py	Interface for all movers
	basic_mover.py	File-based movement
	random_mover.py	Random movement
	turtle_control.py	Publishes movement to ROS
	numbers_to_velocity.py	Create movement from numbers
	move_helper.py	Movement utilities
	turtle_state.py	Container for turtle state
↪pipes	pose_processor.py	Transform poses to requests
	pose_pipe.py	Randomly pipe poses through pose processors
Web server	web_api.py	Endpoints and basic web server
	state_dao.py	Server data access object
	idse_dao.py	Data access object for the intermediate IDSE file format
	log_entry.py	Log entry for one request
	log_file_processor.py	Update and modify log files
	log_file_analysis.py	Log file and entry analysis
	log_file_tools.py	Split, sample, analyse
	server_tools.py	Interaction with the server
↪functionality	mapper_base.py	Base class for mappers
	country_code_mapper.py	Map poses to country codes
	poi_mapper.py	Map poses to POIs
	routing_mapper.py	*Deprecated*
↪ids	live_ids.py	IDS component of the server
	intrusion_classifier.py	Underlying classifier
	ids_converter.py	Encode log entries
	ids_data.py	Shared data
	ids_tools.py	Process, prepare, bootstrap
	dir_utils.py	Safe interaction with files
	ids_entry.py	Container for converted log entries
	ids_classification.py	Container for a classification
↪util	fmtr.py	Format numbers and text
	outp.py	Output functions

continued the next page...

Category	File	Function
	prtr.py	Pretty printing
	seqr.py	Convenient sequence handling
	stat.py	Statistics
Launch files	launch_file_orchestrator.py	Create randomised ROS launch files
	launch_file_switcher.py	Update and modify launch files
↪lfo_compo...	intrusion_definition.py	Process the user-defined intrusion
	vin_generator.py	Generate VIN tails
Experiments	experiment.py	Experiment runner
	experiment_modules.py	Single experiments

Table A.2: *The programs and tools we have developed for this thesis.*

A.3 Source code repository

Our source code is uploaded to GitHub and can be accessed at the following URL:

RRITbed .... `https://github.com/tum-i22/rritbed`

The commit that marks the end of the work documented in this thesis can be identified by the following commit hash: `e7ace1eec12c7f0dfac337adabbdfa105516bffc`

A.4 Library versions

Following is an alphabetical list of the main libraries that we used and their respective versions.

NumPy	`1.11.0`
scikit-learn	`0.19.1`