# A Real-Time Remote IDS Testbed for Connected Vehicles

Valentin Zieglmeier
Technical University of Munich
Munich, Germany
v.zieglmeier@tum.de

Severin Kacianka
Technical University of Munich
Munich, Germany
severin.kacianka@tum.de

Thomas Hutzelmann
Technical University of Munich
Munich, Germany
t.hutzelmann@tum.de

Alexander Pretschner
Technical University of Munich
Munich, Germany
alexander.pretschner@tum.de

## ABSTRACT

Connected vehicles are becoming commonplace. A constant connection between vehicles and a central server enables new features and services. This added connectivity raises the likelihood of exposure to attackers and risks unauthorized access.

A possible countermeasure 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 approaches with no sensible option for comparing them.

Our contribution to this problem comprises the conceptualization and implementation of a testbed for an automotive real-world scenario. That amounts to a server-side IDS detecting intrusions into vehicles remotely. To verify the validity of our approach, we evaluate the testbed from multiple perspectives, including 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 difficulties. This allows for comprehensive evaluation of real-time, remote IDS.

## CCS CONCEPTS

• **Security and privacy → Intrusion detection systems**; *Distributed systems security*;

## KEYWORDS

IDS, Intrusion detection, Testbed, Connected vehicles

**ACM Reference Format:**
Valentin Zieglmeier, Severin Kacianka, Thomas Hutzelmann, and Alexander Pretschner. 2019. A Real-Time Remote IDS Testbed for Connected Vehicles. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3297280.3297465


*SAC '19, April 8–12, 2019, Limassol, Cyprus*

ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3297280.3297465

## 1 INTRODUCTION

In 2015, 35 % of new cars sold were already connected to the Internet [1]. Accenture estimates that by 2020 that number will rise to 98 %, reaching 100 % in 2025 [1]. Manufacturers push for higher connectivity, as it offers advantages such as the ability to provide automated software updates which do not require customers to visit a repair shop for software updates [4]. Additionally, new features enabled by Internet connections are used as selling points. For example, vehicles may be monitored and controlled with a smartphone [4]. Finally, vehicle-to-vehicle communication allows for advanced autonomous features such as cooperative collision warnings [31]. However, this connectivity also raises the likelihood of exposing vulnerabilities, which increases the risk of attacks [23]. Checkoway et al. argue that the vehicle remote telematic systems providing a constant connection over cellular networks are one of the most important parts of the wireless attack surface [8].

For passenger cars, traditional defense measures may not always be feasible. The vehicles are expected to subsist for long periods of time, with the average age of cars in the United States rising from 10.6 years in 2010 to 11.6 years in 2016 [17]. It is reasonable to expect that it will rise even further in the future. Zhang et al. argue that this long lifespan makes it hard for manufacturers to predict the necessary hardware for on-board protection [32]. Because of strict limits in production budgets, manufacturers are likely to minimize the cost for each vehicle and only include the minimum required security hardware. Therefore, off-board protection seems promising to provide sufficient security over the whole lifespan of the vehicle [16]. An important challenge with this approach is balancing on-board processing load and the communication of the vehicle to the manufacturer server [32]. Ideally, the existing vehicle communication is utilized for this purpose, as it would neither require additional processing nor communication.

### 1.1 Definition

We define intrusions as deviations from the expected behavior of a system without the manufacturer's knowledge. On the one hand, this comprises malware, unauthorized access and anomalies, including software bugs, that mostly occur without the user's knowledge. On the other hand, our definition encompasses misuse and fraud that can take place with or without the user's knowledge. An example of intentional fraud by users are off-limits modifications intended to activate features that would otherwise be paid

upgrades. Different types of intrusions are often difficult or impossible to differentiate from each other and do not always correspond to illegitimate actions. Our definition follows the NIST [28].

### 1.2 Motivation and Requirements

To defend against intrusions, one can employ intrusion detection systems (IDSs). They can be used as a second line of defense after preventive protection mechanisms such as software integrity verification [22]. IDSs are built to detect intrusions that have already taken place, aiming to mitigate their consequences. For this purpose, they monitor events for possible violations of some defined policy or expected behavior [28]. A considerable advantage of IDSs 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 the problems that we discussed before. Additionally, IDSs 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.

We want to be able to evaluate real-time, remote IDSs for our scenario. This requires a testing environment that supports this real-time detection and emulates our remote scenario. It should simulate a complex system with interacting units that influence each other. Within clients, multiple components and sensors 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. Accordingly, we require an artificial testing environment, a so-called testbed, that is based on sophisticated real-time simulations of clients from our use case. This is necessary in order to sufficiently emulate the real-world use case. Finally, we require test scenarios that subject IDSs to various detection difficulty levels.

The testbed we envision can also be used to generate historical data for off-line detection. Additionally, naïve data generation can be emulated by simplifying the components and simulations. Accordingly, a complete testbed can be built to fulfill our requirements while also enabling simpler scenarios.

## 2 RELATED WORK

There exist a multitude of testbeds for IDSs in various environments. Most of these focus on different environments and use cases. An important research focus lies in testbeds for network IDSs, or NIDSs. These systems detect intrusions on different layers of Internet traffic, often focusing on the application layer. A well known example of this is LARIAT [27], which is based on the DARPA intrusion detection evaluations. It creates live traffic for protocols such as FTP and SSH and employs the NIDS as a separate node in the network that can intercept all traffic [27]. Shiravi et al. describe a more static approach, generating up-to-date datasets for multiple application protocols, including HTTP and SSH [29]. These datasets can then be used for off-line detection. Contrary to our focus, these systems aim for intrusion detection in Internet traffic. The general-purpose solutions these authors describe are too unspecific for our use case, requiring heavy adjustments or complete reworks to fit to that scenario. Our focus lies in a more specialized solution that better covers the specifics of our case.

Similarly, simulators such as ns-3 [14] are focused on discrete-event networks. They can be utilized when designing network protocols and interactions. This aspect can be interesting in the long run, but it does not solve the necessity of simulating the actual system behavior.

In the automotive environment, there is very limited research available. Daily et al. discuss a testbed for heavy vehicle electronic controls [9]. Their solution is focused on simulating sensor inputs, creating traffic for the in-vehicle network specific to commercial vehicles. Huang et al. describe a tool that generates attack traffic for the CAN bus inside vehicles without making use of simulated hardware [15]. Finally, the HCRL car-hacking dataset for intrusion detection [13] consists of real-world CAN traces that have been injected with attack traffic by the authors. These approaches are distinct from our scenario, as we are focused on server-side detection of intrusions into passenger vehicles.

To the best of our knowledge, there exists no work on the topic of real-time remote intrusion detection for connected vehicles.

## 3 THE TESTBED

Our objective is to enable the comparison of various types of IDSs for the connected vehicle use case provided by our industry partner. To allow for flexible use and adaptation for different contexts, we require an artificial testing environment that fits this use case.

### 3.1 Use case

Our use case is based on a client-server architecture as used by our industry partner in the automotive industry. Multiple vehicles communicate with a central, manufacturer-controlled server, consisting of functional components and a logging component. Vehicle requests are handled and then forwarded to the logging component that stores information about the request in a log store (see Figure 1). This data is used by the IDS. In the figure, arrows represent communication and components are dashed.

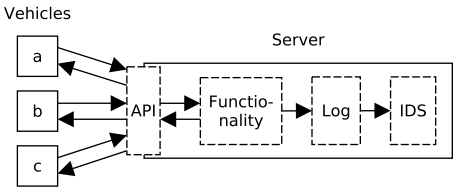


**Figure 1: A model of our use case.**

The server is our frame of reference, and we abstract the vehicles from an outside perspective as data providers that send requests to the server. Following similar designs by [11, 26, 32], we assume that normal and compromised vehicles can be differentiated based on their communication with the server. Much like the vehicles, we view the server functionality as a black box. It receives request data, processes it and sends data based on those requests to the logging component.

This component records the data that is available for intrusion detection. 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 its specific implementation.

## 3.2 Concept

For comprehensive evaluation of IDSs for the real-world use case described above, we design a testbed. It is aimed at modeling that use case as closely as possible. Hence, it is simulating a client-server architecture with individual clients, corresponding to the vehicles, 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 consists of a simulated logging component that generates the log data used for intrusion detection based on incoming requests from the clients. Additionally, an IDS can be added to the server for live detection. As we focus on intrusions in clients, the server functionality is omitted from our testbed.

*3.2.1 Clients.* As described before, each client consists of multiple components. These produce and consume data and are linked to a central communication unit that sends requests to the server. There are two types of components we can derive from real-world clients. Simple components generate various types of data periodically, i. e. temperature sensors. 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". 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 modeling such a component is what we refer to as "2D simulator". It consists of a simulated two-dimensional environment in which an independent unit representing the client can move around. This environment can have any defined size and different background colors. The colors 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. Finally, a central communication unit is responsible for collecting all generated data. It does only basic data transformations and then sends the result as a request to the server.

*3.2.2 Intrusions.* We define intrusions as deviations from the expected behavior of a system without the manufacturer's knowledge. That includes unauthorized access or fraud, but also software bugs (see Section 1.1). In our testbed, we conceptualize 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 color readings of the 2D simulator. These intrusions can be combined in various ways to form intrusion scenarios.

*3.2.3 Server.* The server we model consists of functional components and is being called by clients. 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. It 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. 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.

## 3.3 Implementation

To allow for the dynamic set-up of various combinations and configurations of clients, we use the "Robot Operating System" (ROS) [25]. This makes defining a new set-up easy and allows for switching configurations at will. Furthermore, while ROS is already used as a basis for automated driving functionality (e. g., [2]). Each client should have an individual identifier, behavior and layout, which refers to the components it contains (see Section 3.2.1). We use a special XML file, a so-called ROS *launch file*, to define the layout of our clients. Each of its components is an individual Python program. This means we can define different layouts of components, their behavior can be individually chosen through arguments, and the clients get a unique identifier.

*3.3.1 Clients.* The clients consist of multiple components, implemented as separate programs.

The data generators representing simple components are based on *NumPy* [18] random number generators. We use ten generators based on continuous probability distributions from that library.

The 2D simulator representing complex components is based on the ROS *turtlesim* [12]. We reimplemented it without a graphical user interface. Random movement commands are sent to the simulated unit. That moves according to the commands and publishes its position, speed and the current color reading. The colors represent the interpretation of certain positions by the unit moving around in the environment. Additionally, requests are 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. A "Country code" request is a simple inquiry for the associated country code for the current position of the unit: $(x, y)$. Similarly, a "Point of interest" (POI) request is an inquiry for a specific POI type at the current position and also contains the requested POI type: $(x, y, t)$. Finally, a "Route" request is an inquiry for routing to a specified target position: $(x, y, x_t, y_t) : x_t \neq x \wedge y_t \neq y$ with $(x_t, y_t)$ being the target position. Finally, the unit has basic movement intelligence. Certain zones can be marked as "illegal" and the unit will try to avoid those.

*3.3.2 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 detection difficulty levels. With this, we aim for adjusting the difficulty of detecting a compromised client.

For data generators, we currently model two types of intrusions 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, the mean $m$ of the distribution, and its spans $s_{left} := m - r_{min}, s_{right} := r_{max} - m$. Further given a factor $f$ that we define. 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 detection difficulty level, the factor $f$ differs. For level *easy*, *medium*, and *hard*, it is 5, 1.5, and 1.001, respectively. The smaller this factor, the higher the probability that the value could have been generated by the distribution function.

The second intrusion type, the *significant error generation*, is based on a generated value from the underlying number generator. Given $m, s_{left}, s_{right}, v, f$ (see above), an erroneous value $v_e$ is broadcast. For $v \geq m$, $v_e = m + s_{right} \cdot f + v^2$, otherwise $v_e = m - (s_{left} \cdot f + v^2)$.

The 2D simulator allows for multiple types of intrusions. Firstly, the environment background has different colors. This can be changed to represent erroneous readings. An area of the simulated environment is selected and colored in a different color. The unit can pass over this area just as it normally would. In both situations the unit sends information about its position and the color it detects. Because the environment is modified, the color it reads differs, representing an intrusion. Depending on the detection difficulty level, this area can be increased in size and its color can be modified (see Table 1). For different difficulty levels, we aim for varying the detectability of the erroneous color compared to the legal colors. Hence, we need a similarity measure to derive a similarity relationship between different colors

We define our colors in code as RGB (red, green, blue) with $r, g, b \in [0, 255]$. The background has four different legal color options. We chose *purple* (150, 140, 200), *yellow* (170, 250, 140), *green* (120, 180, 130) and *blue* (120, 180, 200). We can imagine the colors as points in a three-dimensional space and calculate the distance between two points $p$ and $q$ using the *Euclidean distance*: $d(p, q) = \sqrt{(p_r - q_r)^2 + (p_g - q_g)^2 + (p_b - q_b)^2}$

For our following calculations, we scale the values of the color 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: $d \in [0, \sqrt{3}]$. The goal when choosing our legal colors was to have moderately similar colors 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 colors. 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 color to all legal colors for easier detection difficulty 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 color for each difficulty level (see Table 1).

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 will try to avoid those. This reaction can be modified: Then, the unit stays in the illegal and continuously sends its color. The detection difficulty levels for the simulated environment apply here as well. The erroneous color marks the illegal zone, and, depending on the difficulty level, its color is closer to or further from the legal colors.

**Table 1: Erroneous color and its average distance to all legal colors, as well as size of erroneous area, per level.**

| **Difficulty level** | Easy | Medium | Hard |
|---|---|---|---|
| **Err. color (RGB)** | 255, 0, 0 | 200, 50, 50 | 170, 80, 80 |
| **Color distance** | 1.103 | 0.774 | 0.590 |
| **Area size** | 40 % | 20 % | 5 % |

Lastly, we have defined possible intrusions for the three types of requests that are sent based on the position of the unit (see Table 2).

**Table 2: Intrusions for positional requests.**

| **Request** | **Request sent** (**N**ormal, **C**ompromised) |
|---|---|
| C. code | **N**: $x, y$ : current position |
| | **C**: $x_c, y_c : n_x, n_y \geq 10 \wedge x_c = x \pm n_x \wedge y_c = y \pm n_y$ |
| POI | **N**: $t \in T$ : One of the legal types in $T$ |
| | **C**: $t_c \notin T$ |
| Route | **N**: $x_t, y_t : x_t \neq x \wedge y_t \neq y \wedge x, y$ : current position |
| | **C**: $x_{tc}, y_{tc} : x_{tc} = x \wedge y_{tc} = y$ |

These types of intrusions may be detectable with domain knowledge-based rules. Because of their nature, we have implemented a different type of detection difficulty levels for these requests. Depending on the level, a compromised unit sends one of these compromised requests with varying probability. For levels *easy*, *medium*, and *hard* this probability is 40 %, 20 %, and 5 %, respectively, making such a unit easier or harder to detect.

## 4 EVALUATION

We intend to assess different aspects of our testbed to show that it fulfills its purpose, formulated as research questions (RQ).

### 4.1 RQ 1: Do we solve the problems of existing datasets?

In our research of IDSs, 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 which can be difficult to obtain [5, 20]. Our testbed can arbitrarily generate new entries, meaning the appropriate number of entries can be generated as needed.

*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 [7]. We can tune the testbed, 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 [21, 30]. Mitigating this by deleting the redundant

**Table 3: Average precision and recall of a *OneClassSVM* classifier on three different datasets of the respective data categories.**

| Level | Measure | Laplace | Wald | Color | POI | Route | Country Code |
|---|---|---|---|---|---|---|---|
| Easy | Precision (%) | 92.29 | 95.1 | 45.4 | 72.80 | 74.42 | 59.25 |
| | Recall (%) | 100.0 | 100.0 | 8.26 | 100.0 | 100.0 | 14.18 |
| Hard | Precision (%) | 91.98 | 87.93 | 1.99 | 74.0 | 73.66 | 73.64 |
| | Recall (%) | 100.0 | 44.69 | 0.19 | 100.0 | 100.0 | 30.12 |
| Difference | Precision (*pt.*) | −0.31 | −7.17 | −43.41 | +1.2 | −0.76 | +14.39 |
| | Recall (*pt.*) | ±0 | −55.31 | −8.07 | ±0 | ±0 | +15.94 |

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.

As a quality measure for the class balance of a dataset we use the *dispersion index*, defined as $\frac{\text{variance}}{\text{mean}}$. This measure is 0 for an exactly balanced dataset. To measure 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: $i > 1 \wedge \exists\, l_{i-1}$ where the unit's identifier, its position, the data sent and the label in $l_i$ and $l_{i-1}$ are identical, then $l_i$ is a duplicate.

*Experimental set-up.* We generated a dataset containing 10 million data points and analyzed it. The dataset contains data points for 14 data categories with approximately 715,000 data points each. We consider each kind of data generated by a component of the testbed as a separate data category, allowing for more precise evaluation. That means we calculate the measures for each data category separately. For a more detailed explanation of these categories, see Section 3.3. 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 color requests are listed separately (see Table 4). All data categories succeed in meeting our strict duplicate targets. The class balance is not perfect, but for color requests we observe good and for positional requests ideal results. The *Generators* show an acceptable dispersion index below our threshold of 450.

**Table 4: The results for different categories of data, grouped.**

| Measure | Generators | Positional | Color |
|---|---|---|---|
| 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 % |

## 4.2 RQ 2: Can we show the effect of varying detection difficulty levels?

We implemented detection difficulty levels aiming at subjecting IDSs to different difficulty levels. Data generated with difficulty level *hard* is supposed to lead to reduced detection accuracy compared to *easy*, as the intruded samples for *hard* lie much closer to the normal data. To evaluate this, we created an IDS prototype based on machine learning. For this research question, we need to introduce some additional prerequisites.

*Handling heterogeneous data.* As we have described before, the generated data varies between different routines in the logging component. Each of these routines represents some server functionality, and in our real-world use case these require or produce different 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.

*Libraries used.* We make use of Python and machine learning, with our library of choice being *scikit-learn* [24]. Apart from classification, we utilize included pre-processing and metrics modules. From this library, we employ the *OneClassSVM* classifier, which is based on the *libsvm* [6] library for computations.

*Definition of positives.* We regard those data points as positives that are intrusions. This interpretation is common in intrusion detection research (see e.g. [19]).

As our IDS prototype does not consider client-specific profiles, we do not expect the detection difficulty levels for categorical data in the form of positional requests to lead to any differences. Contrary to that, difficulty levels implemented for the data generators and color values focus more on *anomaly-based* detection. They are expected to have an effect on the outlier detection algorithm that we apply here. If we find the difference of precision or recall between the difficulty levels to be over a certain threshold, we assume changing the level leads to significant differences in detection accuracy. Our threshold for this is 5 percentage points.

*Experimental set-up.* We generated 1 million data points for detection difficulty levels *easy* and *hard* respectively, and sampled three sets of 100,000 data points for both levels. *OneClassSVM* classifiers with default parameters were trained for each level, data category and sample in a total of 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 difficulty levels (see Table 3). As expected, precision and recall do not change significantly for

the positional requests *POI* and *Route*. We cannot explain the difference observed for *Country code* data. For the data generators *Pareto*, *Rayleigh*, *Uniform*, *Wald*, and *Weibull* we find a successful increase in difficulty, shown exemplary in the *Wald* column in Table 3. For *Gaussian*, *Gumbel*, *Laplace*, *Logistic* and *von Mises* data, we find no significant difference in precision and recall between detection difficulty levels. These results are shown exemplarily in the *Laplace* column of Table 3. The other data generator categories are omitted from the table for brevity. Finally, *Color* data shows a visible effect of increasing the difficulty level.

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

## 4.3 RQ 3: 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 behavior over multiple runs of the same set-up, it is essential that the generated data is similarly distributed.

For our data generators, we consider this to be the case. The underlying number generators produce 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 complex 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.

For evaluation, we generate three datasets. To minimize the influence of sampling errors on 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 $R^2$ 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 split the data up in bins of size 0.1. Then, we calculated the relative frequency of data points in each bin and compared the sets. For data points of the normal class, the $R^2$ value of Set 2 compared to the baseline set is approximately 0.9995, for Set 3 it is approximately 0.9996. For the intrusion class, 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 with almost ideal $R^2$ values.

*2D simulator positions.* Next, we evaluate the relative frequency of the simulated unit being located at various positions of the simulation. This is tricky to visualize, as we have a significant number of points to compare. We approach this problem with heat maps. For easier visualization, we divide the coordinate space into bins of $20 \times 20$ pixels in size. The relative frequency of each bin is signified by the shade of the cell that represents it (see Figure 2). For higher frequencies, the box shifts to a darker shade. The highest frequency is visualized in black. 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 center 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.

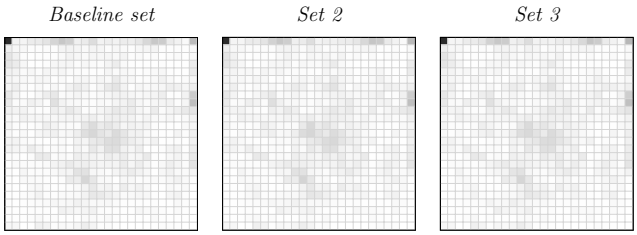


**Figure 2: Relative freq. of a coordinate bin in the simulation.**

*2D simulator requests.* Finally, we consider the requests made by the 2D simulator component, which consist of categorical data. We will discuss our evaluation in detail for point of interest (POI) pairs and list only the results for the remaining data categories.

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 3 we compare the relative frequency of possible forms of POI pairs in the three sets. We see almost identical distributions for different sets. This is confirmed by the $R^2$ value. Comparing Set 2 to the baseline set the $R^2$ is approximately 0.9995, for Set 3 it is approximately 0.9998.

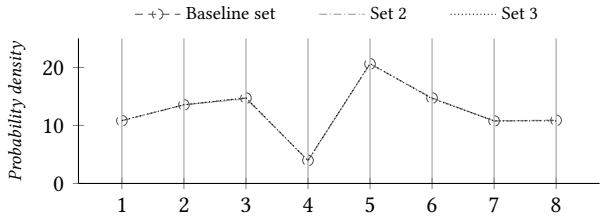


**Figure 3: Relative frequency of POI pairs.**

We find similar results for the other request types. For "Color" requests, the $R^2$ value of Set 2 compared to the baseline set is approximately 0.9999, for Set 3 it is approximately 0.9997. The categories "Country code" and "Route" have an almost ideal $R^2$ value of approximately 0.9999 for both Set 2 and Set 3 compared to the baseline set.

*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.

### 4.4 RQ 4: Can we evaluate various types of IDS?

There are various, heterogeneous IDS approaches. Existing evaluation options are often limited to specific types of systems. For example, authors utilizing *anomaly-based* detection can use general-purpose datasets such as the *Iris flower set* [10] for evaluation. For *signature-based* systems, such datasets offer no sensible evaluation option as they are missing intrusions that could be detected. We want to show that our testbed theoretically allows the evaluation of various types of IDSs.

*Anomaly-based detection.* IDSs utilizing *anomaly-based* detection require data with some form of anomalous behavior that they can detect. This use case is covered by our data generators. They are based on probability distributions, which allow us to define normal behavior 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 3.3.2).

*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 behavioral 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 labeling of testbed data, the detection accuracy can be measured effectively. For more advanced detection systems, our testbed does not offer adequate options for evaluation.

### 4.5 RQ 5: How well does the testbed scale?

We are interested in the scalability of our testbed regarding the system load when increasing the number of simulated components. Ideally, it scales almost linearly, as this allows running arbitrarily complex simulations if necessary. We regard the metrics *Start-up time*, *CPU load*, and *Main memory usage*. We perform a *regression analysis* to approximate the time complexity based on our measurements. For each analysis, we give the *mean squared error* MSE and *coefficient of determination* $R^2$ to assess the quality of the estimation compared to the measurements. For better estimations, the MSE lowers towards 0 and the $R^2$ increases towards 1.

For our experiments, we used a machine with a 12-core 2.5 GHz *QEMU* [3] virtual CPU and 32 GB of main memory. Because we could only measure reliably for component counts of up to 500, we do not generalize further. For component counts larger than 500, we would need more measurements to be able to sufficiently predict the limiting behavior.

The measurements are split up in cycles and rounds. Each cycle is identified by the number of components that are started. Each round represents one measurement. We measured repeatedly and took the average for our results. For the start-up time, the maximum standard deviation of all measurements was 0.26 seconds, or 1.3 %, with an average value of 0.07 seconds. For the CPU load measurements, we calculated a maximum standard deviation of 0.18 points, or 1.2 %, with an average value of 0.4 points. The maximum for the main memory measurements was 0.59 percentage points, or 1.58 %, with an average value of 0.08 points. Hence, we consider the measurements to be sufficiently reproducible.

*Results.* The measurements for start-up time, CPU load, as well as main memory usage grow linearly, so we estimate them with linear regression. We will describe our approach in detail for the main memory usage, and only list the results for the other measures. After multiple iterations with subsets of our measurements, we obtain for the main memory usage : $g_m(n) \approx 0.1491n+0.447$ with MSE $< 1.262$ and $R^2 > 0.997$ for the complete set of measurements (see Figure 4). In the figure, the dots are the mean, with the bars above and below indicating the maximum and minimum measurement.

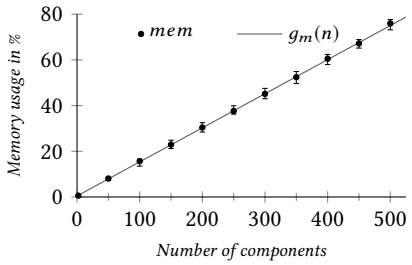


**Figure 4: Main memory usage and approximation $g_m(n)$.**

From our approximation, we can derive the limiting behavior: $f_m(n) = O(0.1491n + 0.447) = O(n), n \to 500$. Accordingly for the start-up time, we obtain: $g_s(n) \approx 0.0476n + 1.141$ with MSE $< 0.143$ and $R^2 > 0.998$. We conclude: $f_s(n) = O(0.0476n + 1.141) = O(n), n \to 500$. We assume that the constant 1.141 represents the cold-start time. For the CPU load, we obtain: $g_c(n) \approx 0.0127n + 9.756$ with MSE $< 0.031$ and $R^2 > 0.988$. We conclude: $f_c(n) = O(0.0127n + 9.756) = O(n), n \to 500$.

*Conclusion.* We can show for all our measures that a linear approximation can sufficiently explain our measurements. We derive a limiting behavior of $O(n)$ for component counts of up to 500, meaning our testbed scales linearly. For a valid estimation for higher $n$, we would need additional measurements.

## 5 LIMITATIONS

Our testbed concept and implementation were created from the ground up. To be able to simulate our use case, we had to simplify it and create exemplary implementations based on a more basic concept of the use case. This abstraction means that the system by itself is not representative for the real world. To reduce this risk, we closely modeled the scenario after the real-world use case, with the testbed generating data that resembles the actual data found in the system employed today. Using synthetic data is delicate and not the perfect solution. It is a necessary compromise though, as real-world data cannot be accessed in relevant quantities. Other authors have made this compromise for the same reason [15]. When using data

from the testbed it is important to keep those risks in mind. Finally, the IDS we used for evaluation is only a simplified machine learning based implementation. This does not pose a threat to the validity of our results, as we only measured differences in detection difficulty, not absolute ease of detection.

## 6 CONCLUSION AND FUTURE WORK

More and more vehicles are connected, exposing them to the risk of attacks. IDSs can be employed as a countermeasure. To be able to evaluate different IDSs, we conceptualized and implemented a testbed. It enables the comparison of various IDSs in a real-world scenario. The testbed is adapted to our automotive use case but can be quickly adjusted to numerous other scenarios.

In the evaluation, we find that the testbed fulfills its objectives. Its architecture and modularity allow for various configurations and the evaluation of real-time, remote IDSs. To assess the fitness for purpose of the testbed, we evaluate multiple quality metrics derived from related works. 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 behavior. 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 IDSs we have identified in literature. We also assess the quality of the data generated by the testbed. The detection difficulty levels we have introduced lead to measurable differences in detection accuracy, allowing for varied evaluation of IDSs. This confirms that IDSs are challenged when trying to detect intrusions in the data generated by our testbed. To conclude our evaluation, we evaluate the performance of the testbed. We find that with additional components the testbed scales linearly. This allows for complex simulations to be carried out. We have discussed some limitations to our solution above. This mainly regards the number of different components and the generated data.

Our testbed is a first step towards a comprehensive solution for comparing IDSs. More advanced behavior patterns can be implemented, including multiple units interacting in the 2D simulator component within a shared environment. Additionally, more subtle intrusions may be implemented based on domain knowledge. Still, the current state of the testbed is sufficient to evaluate most types of IDSs employed today while offering multiple improvements over existing solutions.

In conclusion, our testbed allows the flexible and reliable evaluation of IDSs 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 IDSs. Because of the reproducibility of the generated data, the same IDS can be tested multiple times to ensure consistent performance. 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[1].

[1] https://github.com/tum-i22/rritbed

## ACKNOWLEDGMENTS

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) under grant no. PR1266/3-1, Design Paradigms for Societal-Scale Cyber-Physical Systems, and the Federal Ministry of Education and Research (BMBF) under grant no. 5091121.

## REFERENCES

[1] Accenture. 2015. *Connected vehicle – Succeeding with a disruptive technology.* 3. www.accenture.com [Online; acc. 2018-04-24].
[2] M. Aeberhard et al. 2015. Automated Driving with ROS at BMW. *ROSCon* (2015).
[3] F. Bellard. 2018. QEMU processor emulator. www.qemu.org [Online; acc. 2018-04-27].
[4] R. R. Brooks et al. 2009. Automobile security concerns. *IEEE Vehicular Technology Magazine* 4, 2 (6 2009), 52–64.
[5] J. Cannady. 1998. Artificial neural networks for misuse detection. In *Proc. of the 21st National Information Systems Security Conference*. NIST, 443–456.
[6] C.-C. Chang et al. 2011. LIBSVM: A library for support vector machines. *ACM Trans. Intelligent Systems and Technology* 2, 3 (4 2011).
[7] N. V. Chawla et al. 2004. Special issue on learning from imbalanced data sets. *ACM SIGKDD Explorations Newsletter* 6, 1 (2004), 1–6.
[8] S. Checkoway et al. 2011. Comprehensive experimental analyses of automotive attack surfaces. In *Proc. of the 20th USENIX Security Symposium*.
[9] J. Daily et al. 2016. Towards a cyber assurance testbed for heavy vehicle electronic controls. *SAE Int. J. of Commercial Vehicles* 9, 2 (2016), 339–349.
[10] R. O. Duda et al. 1973. *Pattern Classification and Scene Analysis*. Vol. 1. Wiley.
[11] Ericsson. 2015. Connected vehicle cloud: Under the hood. archive.ericsson.net [Online; acc. 2018-05-09].
[12] J. Faust. 2018. ROS turtlesim. wiki.ros.org [Online; acc. 2018-04-03].
[13] HCRL. 2018. Car-hacking dataset for intrusion detection. ocslab.hksecurity.net [Online; acc. 2018-09-19].
[14] T. Henderson et al. 2006. Ns-3 project goals. *ACM International Conference Proceeding Series* 202 (2006).
[15] T. Huang et al. 2018. ATG: An attack traffic generation tool for security testing of in-vehicle CAN bus. In *Proc. of ARES 2018*. ACM, 32:1–32:6.
[16] T. Hutzelmann et al. in press. A comprehensive attack and defense model for the automotive domain. *SAE Int. J. of Transportation Cybersecurity and Privacy* (in press).
[17] IHS. 2016. Average age of light vehicles in the U.S. from 2003 to 2016 (in years). www.statista.com [Online; acc. 2018-04-24].
[18] E. Jones et al. 2001. SciPy. www.scipy.org [Online; acc. 2018-03-30].
[19] J. Kim et al. 2007. Immune system approaches to intrusion detection–a review. *Natural Computing* 6, 4 (2007), 413–466.
[20] E. M. Knorr et al. 2000. Distance-based outliers: algorithms and applications. *The VLDB Journal* 8, 3-4 (2000), 237–253.
[21] Y. Li et al. 2012. An efficient intrusion detection system based on support vector machines and gradually feature removal method. *Expert Syst. Appl.* 39, 1 (2012), 424–430.
[22] H. Oguma et al. 2008. New attestation-based security architecture for in-vehicle communication. In *Proc. of GLOBECOM 2008*. 1–6.
[23] S. Parkinson et al. 2017. Cyber threats facing autonomous and connected vehicles: Future challenges. *IEEE Trans. Intell. Transportation Syst.* 18, 11 (2017), 2898–2915.
[24] F. Pedregosa et al. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12 (10 2011), 2825–2830.
[25] M. Quigley et al. 2009. ROS: An open-source robot operating system. In *Proc. of ICRA 2009 Workshop on Open Source Robotics*, Vol. 3. IEEE, 5.
[26] M. N. Ramadan et al. 2012. Intelligent anti-theft and tracking system for automobiles. *Int. J. of Machine Learning and Computing* 2, 1 (2012), 83.
[27] L. M. Rossey et al. 2002. LARIAT: Lincoln adaptable real-time information assurance testbed. In *Proc. of the 2002 IEEE Aerospace Conference*, Vol. 6. 6–6.
[28] K. Scarfone et al. 2007. Guide to intrusion detection and prevention systems (IDPS). *NIST Special Publication* 800-94 (2007). csrc.nist.gov
[29] A. Shiravi et al. 2012. Toward developing a systematic approach to generate benchmark datasets for intrusion detection. *Comp. & Secur.* 31, 3 (2012), 357–374.
[30] S. S. S. Sindhu et al. 2012. Decision tree based light weight intrusion detection using a wrapper approach. *Expert Syst. Appl.* 39, 1 (2012), 129–141.
[31] X. Yang et al. 2004. A vehicle-to-vehicle communication protocol for cooperative collision warning. In *Proc. of MobiQuitous 2004*. 114–123.
[32] T. Zhang et al. 2014. Defending connected vehicles against malware: Challenges and a solution framework. *IEEE Internet of Things Journal* 1, 1 (2014), 10–21.