AVMaestro: A Centralized Policy Enforcement Framework for Safe Autonomous-driving Environments

Ze Zhang[1], Sanjay Sri Vallabh Singapuram[1], Qingzhao Zhang[1], David Ke Hong[1], Brandon Nguyen[1], Z. Morley Mao[1], Scott Mahlke[1] and Qi Alfred Chen[2]

***Abstract*— Autonomous vehicles (AVs) are on the verge of changing the transportation industry. Despite the fast development of autonomous driving systems (ADSs), they still face safety and security challenges. Current defensive approaches usually focus on a narrow objective and are bound to specific platforms, making them difficult to generalize. To solve these limitations, we propose AVMaestro, an efficient and effective policy enforcement framework for full-stack ADSs. AVMaestro includes a code instrumentation module to systematically collect required information across the entire ADS, which will then be feed into a centralized data examination module, where users can utilize the global information to deploy defensive methods to protect AVs from various threats. AVMaestro is evaluated on top of Apollo-6.0 and experimental results confirm that it can be easily incorporated into the original ADS with almost negligible run-time delay. We further demonstrate that utilizing the global information can not only improve the accuracy of existing intrusion detection methods, but also potentially inspire new security applications.**

I. Introduction

With the emergence of autonomous vehicles (AVs), the world is undergoing a tremendous transportation revolution. Despite the recent developments of AVs contributed by both academic institutions and leading industries, autonomous driving systems (ADSs) are still under active research and face some safety and security challenges. As of today, protecting AVs from both external malicious attacks and internal system errors has became a crucial aspect of designing reliable ADSs. However, there is no satisfying defense method applied on existing ADSs, making self-driving cars extremely vulnerable to sensor spoofing and jamming attacks [21], [18]. Although prior works [16], [12], [8], [22] have proposed ideas to detect sensor intrusions and system anomalies, their solutions are very hard to generalize to other software platforms. Furthermore, these techniques are only implemented and evaluated on simple cyber-physical systems such as drones and robotic vehicles that are far less complex than state-of-the-art ADSs. Therefore, how to enforce safety and security mechanisms on top of a full-stack ADS effectively and efficiently still remains an open question. Given the fact that more and more attacks have been developed to threaten self-driving vehicles, implementing each individual solution on an ADS software is too expensive to be practical. As a result, there is an urgent need to develop a unified platform that can be effectively applied to end-to-end ADSs to protect AVs from a wide spectrum of known threats.

To address above limitations, we propose AVMaestro, a practical policy enforcement platform to provide a safe execution environment for autonomous driving systems. AVMaestro is a flexible framework that can be easily applied to different ADSs without interrupting their original functionalities. Through a centralized data examination module, it provides a fine-grained method to manage the global information used by the entire system and enables following opportunities: 1. collecting necessary information at a unified place to implement various defensive techniques [16], [7] against malicious attacks; 2. examining the global information to detect data corruptions and issue alerts if any anomalous activity is observed at run time. 3. intelligently updating system-level configurations under different environmental conditions to achieve safer road performance.

To strengthen the communication between the data examination module and the rest of the ADS system, we also develop a code instrumentation module with two sets of algorithms to help the AVMaestro systematically access an ADS's internal variables and configuration parameters that can further improve the effectiveness of user-defined defense techniques. In summary, our proposed AVMaestro framework makes following contributions:

- We propose a new ADS communication paradigm with a centralized data examination module built on top of the modular ADS architecture, providing a single entry point for developing and verifying safety and security related implementations.
- We develop two sets of compiler analysis and instrumentation algorithms to provide inter-module communications with system-level internal variables and configuration parameters.
- We deploy two attack detection mechanisms using AVMaestro to validate their effectiveness. 86.3% and 93.2% true positive rate are achieved for two techniques in our attack injection experiments. We further propose novel upgrades on these techniques to reduce false positives and catch stealthy attacks by exploiting the system-level information provided by the AVMaesto.
- We evaluate AVMaestro on top of Apollo-6.0 [2]. Experimental results demonstrate that the AVMaestro framework only incurs 7.2 to 11.7 ms (1.89% - 3.06%) end-to-end delay depending on different sizes of workloads, which is almost negligible to the original ADS.

[1]Ze Zhang, Sanjay Sri Vallabh Singapuram, Qingzhao Zhang, David Ke Hong, Brandon Nguyen, Z. Morley Mao and Scott Mahlke are with the Department of Electrical Engineering and Computer Science, University of Michigan, MI, USA. {zezhang, singam, qzzhang, kehong, brng, zmao, mahlke}@umich.edu

[2]Qi Alfred Chen is with the Department of Computer Science, University of California, Irvine, CA, USA. alfchen@uci.edu

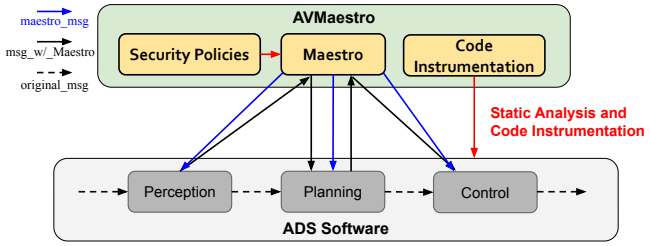


Fig. 1: AVMaestro high-level design.

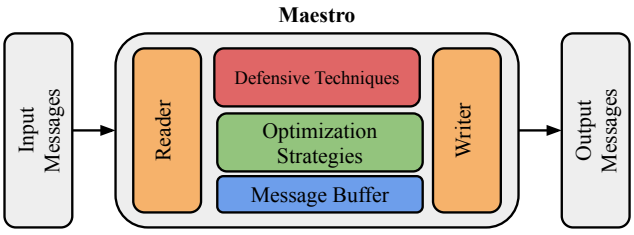


Fig. 2: Structure of online Maestro module.

II. Background and Motivation

A commonly used software architecture [2], [1] for ADSs is composed of different modules with each module dedicated to a specific job, including object detection, motion prediction, trajectory planning, and vehicle control. At run time, each module in the system subscribes its required inputs either from previous modules or from equipped sensors, and executes implemented algorithms to calculate results. Once the output is generated, it will be published as a message to its subscribing modules with the help of an underlying pub-sub communication system.

Over the past few years, various attacks have been developed to threaten self-driving vehicles. Based on previous researches, nearly all of sensors equipped in AVs are vulnerable to sensor spoofing and jamming attacks, including GPS [21], IMU [17], camera [15], RADAR [20], LiDAR [4], and ultrasonic sensors [20]. Meanwhile, potential defense mechanisms have also been discussed. In general, intrusion detection techniques can be classified into two major categories. First, model-based approaches use predictive models such as state space models [19], [5] or machine learning models [9], [3] to make predictions for vehicle's expected behavior in near future. If the difference between the predicted value and the actual sensor measurement is greater than a threshold, an anomaly is detected. Second, rule-based approaches [7] will collect and analyze historical sensory data and module outputs to generate a series of invariant rules based on correlations and physical laws. If any run-time behavior violates these rules, a possible error is marked.

Unfortunately, most of proposed defense methods are platform specific and are only tested on simple cyber-physical systems such as drones and robotic vehicles. Their evaluated results cannot represent actual behaviors of advanced self-driving cars. Thus, how to enforce extra safety and security policies on production-level autonomous driving systems in a realistic manner still remains an open question. Based on our study, there is no effective method incorporated in the current ADS design to protect AVs' run-time environments from either internal system errors or external malicious attacks. To overcome this limitation, we propose AVMaestro: a highly efficient and effective policy enforcement platform for providing a safe autonomous driving environment.

III. AVMaestro Design

Fig.1 illustrates the high-level diagram of the AVMaestro framework which includes two major components. A data examination module named *Maestro* is built on top of the original ADS, serving as a centralized controller and providing a convenient entry point for all defensive methods. At run time, the *Maestro* module will hold user-provided implementations and keep monitoring all communication messages published from other modules in the system, making sure the vehicle operates in a legal state. In case a suspicious behavior is detected by implemented techniques, the *Maestro* will trigger corresponding actions defined in policies such as issuing alerts, performing a controlled stop, or enforcing user-specified actions by overwriting specific message fields.

With the above setting, the *Maestro* module by default can access all variables that are predefined in inter-module communication messages. However, we find that a majority of safety and security techniques either rely on or benefit from certain system-level internal variables and configuration parameters. To provide the *Maestro* module with these information, we also develop a code instrumentation module that includes two sets of static analysis and code instrumentation algorithms. Its goal is to automatically collect and update necessary information by instrumenting proper message communication APIs in the original ADS.

Note that the AVMaestro framework itself does not include any defensive methods. Instead, it brings all information together and provides a centralized place to implement and verify them. More importantly, the message-level information combined with the system-level information can not only support a variety of defensive techniques, but also further improve them and potentially inspire new security applications. In following sections, we will provide more detailed explanations on each developed module.

A. Maestro Module

A majority of defensive techniques [5], [16] need to examine vehicular operations across different modules at the same time. Thus, enforcing them inside of each separate module is almost impossible because one module by default can only access a fixed set of inputs. Besides, finding an appropriate implementation location is also challenging due to the asynchronous nature of each module. To provide an effective place to deploy safety and security policies without affecting original ADS's functionalities, we propose to fundamentally upgrade the existing ADS communication behavior with a centralized message management and data validation component, named *Maestro* that can analyze and update all system-level information at a unified place.

Our proposed *Maestro* module (shown in Fig.2) can be easily realized in different ADS platforms. To begin with, it includes a reader and a writer that are responsible for

receiving and sending messages from/to other modules in a target ADS, respectively. Depending on the run-time system that an ADS is built on, these two parts can be implemented using the standard message communication APIs. In addition, the *Maestro* holds all user-defined implementations, making sure the autonomous vehicle operates in safe states. To minimize the run-time overhead, two optimization strategies are developed. First, since each module publishes its messages asynchronously, the *Maestro* checks the sequence number from different messages to avoid wasting computation resources from processing duplicated messages. Second, instead of monitoring all communication messages at a fixed frequency, the *Maestro* allows users to select only a subset of messages and change its execution frequency to reduce the message forwarding overhead. Finally, a configurable message buffer is allocated to keep track a short history of important variables. The message buffer plays a crucial role in most of intrusion detection methods because they rely on historical data as reference points to either make predictions for the expected behavior or accumulate some errors to catch anomalies. Furthermore, it also helps us reduce false positives resulted from sensor fluctuations.

While the *Maestro* is activated, all other modules in the ADS firstly send their output messages to it for examination. Based on the collected information and instrumented policies, the *Maestro* determines whether all components in the system work as expected. In normal situations, it simply forwards the messages to their destinations. In case an anomaly is detected, the *Maestro* module needs to overwrite certain message fields to enforce user-specified countermeasures before sending them out. In addition, the *Maestro* also sends out its own messages to update certain configuration variables used by other components. The communication diagram with the *Maestro* module is shown in Fig.1.

Using a centralized approach to enforce system-level safety and security has proven to be a successful methodology among many other domains including software defined networks (DSN) and internet of things (IoT). Although the ADS software share many similar characteristics as those applications (asynchronous module execution, pub-sub message communication etc.), no satisfying solution has been applied to address these challenges. Therefore, we believe adopting a centralized approach to perform fine-grained data examination is a valuable design space to explore and can potentially impact the future development of ADSs.

B. Code Instrumentation Module

Code instrumentation module is developed to empower the communication between the *Maestro* module and the rest of the system by automatically bringing the latest value of hidden variables to the corresponding communication messages. This instrumentation process gives the *Maestro* extra capability to dynamically manage thousands of system-level variables that can further benefit user-defined policies.

1) Internal Variable Instrumentation: To make the *Maestro* access internal variables from a module, our proposed algorithm will search the ADS's code base to systematically update these variables to the output messages of the corresponding module. Specifically, a list of variables that are associated with user-defined policies is firstly generated. Then, the algorithm searches the entire ADS code base as well as all communication messages to find if any variable's name matches with the items in the list. If no match is found, the listed variable is therefore not appeared in the current ADS. In this case, our proposed algorithm just reports an error and then aborts. If the match is found in one of the communication messages, no further action is needed because the target variable is already included in one of the outgoing messages.

```
1  if (FLAGS_use_navigation_mode &&
2       FLAGS_enable_navigation_mode_position_update) {
3    ...
4    double curr_vehicle_heading = 0.0;
5    if (localization->pose().has_heading()) {
6      curr_vehicle_heading = localization->pose().heading();
7    }
8    else {
9      curr_vehicle_heading = QuaternionToHeading(
10       orientation.qw(), orientation.qx(),
11       orientation.qy(), orientation.qz());
12     }
13     // other uses of the target variable are neglected
14     /* Instrumented Code */
15     cmd->set_curr_vehicle_heading(curr_vehicle_heading);
16 }
```

Listing 1: Instrumented code for variable customization.

Algorithm 1: Internal variable instrumentation

```
for v in variable_list do
    if v not in code_base then
        Error("Unsupported Variable!")
    else if v in message_declaration then
        continue
    else
        message_declaration.add(msg, v.module, v.name)
        scope = {}
        for def in definitions(v) do
            scope = Union(def.reaching_definitions, scope)
        end
        insertPts = scope.get_non_overlapping_intervals()
        for curPt in insertPts do
            insert_msg_update_API(msg, v, curPt.last)
        end
    end
end
```

However, if the match is only found in code base, confirming the target variable is a local module variable, following instrumentation needs to be performed: First, depending in which module the match is found, a new variable declaration with the same name of the target variable is added in the outgoing message of the corresponding module. Second, we perform a backward data-flow analysis on the target variable to find its *definitions* associated with its *scope*. Lastly, a message updating API is inserted before exiting the *scope* of the target variable. The updating API needs to be instrumented at the end of the variable's *scope* so that the *Maestro* module can access its latest value from the published message. If multiple *definitions* exist in parallel, the updating API will be inserted at the end of each definition's scope to ensure it can be executed no matter which path is taken. The proposed algorithm is summarized in Algorithm 1.

Listing 1 shows the instrumentation result with a local variable named *curr_vehicle_heading*. From the static analy-

```
1  // Global Use of configuration variable
2  double speed_limit = FLAGS_planning_upper_speed_limit;
3  // After code instrumentation
4  double speed_limit =
5      maestro_msg.FLAGS_planning_upper_speed_limit();
6  ------------------------------------------------------------
7  // Direct Use of configuration variable
8  speed_pid_controller_.SetPID(
9      lon_controller_conf.low_speed_pid_conf());
10 // After code instrumentation
11 speed_pid_controller_.SetPID(
12     maestro_msg.low_speed_pid_conf());
13 ------------------------------------------------------------
14 // Indirect Use of configuration variable
15 /* In Init Function */
16  query_relative_time_ = control_conf->query_relative_time();
17 /* In Use Function */
18 target_point =
19     trajectory_analyzer.QueryNearestPointByAbsoluteTime(
20     Clock::NowInSeconds() +  query_relative_time_);
21 // After code instrumentation
22 // inserted at the beginning of each use function
23  query_relative_time_ = maestro_msg.query_relative_time();
```

Listing 2: Instrumented code for configuration customization.

Algorithm 2: Configuration instrumentation

```
for config in configuration_list do
    m_msg.add_declaration(config.name)
    m_msg.initialize(config.name, config.val)
    for user in config.uses do
        if user in init_functions then
            var = user.name
            for local_user in var.uses do
                func = local_user.getFunction
                insertPt = func.getEntryBlock.begin
                createLoad(var, m_msg.config, insertPt)
            end
        else
            latest_config = m_msg.config
            user.replaceUseWith(latest_config)
        end
    end
end
```

sis, we find the *definitions* of the target variable is located at line 4, 6, and 9 (shown with blue texts), with the *scope* from line 4 to line 16. With this information, the message updating API (shown with red texts) is inserted at line 15 to get the latest value of the target variable. To apply this instrumentation method on different ADS platforms, we simply update the message API and the declaration format, whereas everything else remains the same.

2) Configuration Instrumentation: Configuration parameters are statically declared in configuration files and directly loaded to each module at run time. They are usually used as fixed thresholds in rule-based detection techniques [5]. For example, the minimal distance to keep away from obstacles should always no less than the value defined in configurations. In addition, dynamically updating them can customize vehicle behaviors which can achieve safer road performance (e.g. reduce the upper limit of planning speed at nights). To utilize these variables in policies, we declare a new message type for the *Maestro*, which contains all configuration variables that users want to manage.

Similar to the previous step, a list containing all configuration variables appeared in user-defined policies is firstly generated. For each variable in the list, a new variable declaration with the same name is allocated in the *Maestro*'s output message. By default, they are initialized with the same value as the original configuration setting to maintain the same behavior. Then, our algorithm performs a def-use analysis on the code base, which will identify all the *uses* of each target variable in the list. According to our study, we find that configuration variables can be used in three different ways: *Global Use*, *Direct Use*, and *Indirect Use*. *Global Use* simply uses the configuration variable as a global variable. In majority settings, *Direct Use* loads the configuration value from a specific location. For *Indirect Use*, however, the configuration value is firstly read from the file and then stored into a local variable at the module initialization stage. Later on, this local variable is used to represent the original configuration wherever is necessary.

Our proposed instrumentation algorithm (summarized in Algorithm 2) for the *Global Use* and the *Direct Use* is straightforward to realize. It simply replaces the original *use* of the configuration variable with the corresponding variable included in the *Maestro*'s output message, as shown in the first two part of Listing 2. The original use is shown at line 2, 9 with blue texts and the updated use is shown at line 5, 12 with red texts. However, if the *use* of the variable appeared in constructors or initialization functions, indicating the *Indirect Use* case, the above solution will not work because these functions will only be executed once even before the commutation starts. To solve this, we need to trace further by identifying all the *uses* of the corresponding local variable. For every function that the local variable is used, a variable assignment instruction (line 23 in Listing 2) is inserted at the beginning of the function to overwrite the stale value, making sure that the latest value from the *Maestro* module is used as the configuration setting.

Both of instrumentation techniques only insert message communication APIs at specific locations in the target ADS's code base. Thus, they will not affect the original module behavior nor introduce any vulnerabilities.

IV. Implementation

Implementation. To evaluate the AVMaestro, we implemented the *Maestro* module on top of Apollo Auto [2]. Our static analysis algorithms are implemented as module passes using the LLVM compiler infrastructure [13]. We firstly use the *wllvm* (Whole Program LLVM) to generate a combined bitcode file for each module in Apollo, and then executes the analysis pass on top of it. The analysis results are fed into a Python script to perform the source-code instrumentation.

Experimental setup. All experiments are running on an Intel i7-6700K CPU clocked at 4.0 GHz with 16 GB of RAM and an NVIDIA GTX 1080 GPU equipped with 8GB memory, matching the minimal system requirements by Apollo. To generate maps, traffics, and sensor inputs, we bridge the Apollo with the LGSVL Simulator [14].

Attack Injection To validate AVMaestro's effectiveness, we inject corrupted signals to the ADS and see how it responds. To do this, we intercept the sensor data published from the LGSVL simulator, modify them based on specific attack objectives and then forward the corrupted data to the Apollo like the prior work [10]. For example, to simulate a GPS spoofing attack, we can falsify the position variables (x, y, and z) included in the GPS message.

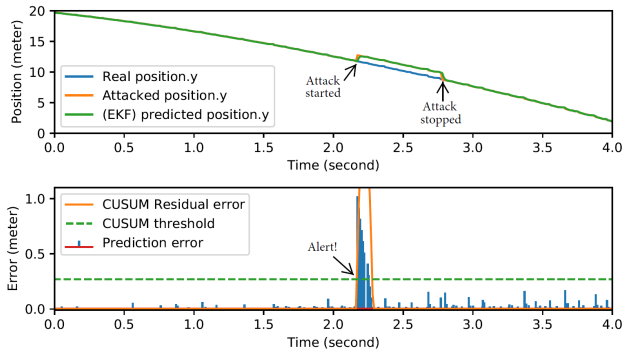


Fig. 3: Detection results on 1-meter GPS spoofing attack.

V. Evaluation

Our proposed AVMaestro framework supports a variety of safety and security related implementations, including anomaly and intrusion detection, cross-module data validation, behavior customization, as well as attack mitigation. In our evaluation, we will particularly focus on defensive methods targeting sensor intrusion attacks. As we mentioned in Section II, there are two major categories for attack detection techniques: model-based detection and rule-based detection. We select one technique from each category to illustrate how to effectively deploy them using AVMaestro, and how functionalities provided by the *Maestro* module can further improve them. In addition, we discuss how to define policies to make the autonomous vehicle recover from an obstacle relocation attack. Since AVMaestro includes a run-time component, we also measure its performance overhead under the worse case scenario to make sure it can be practically deployed in ADSs.

A. Model based intrusion detection

To implement this type of attack detection technique, we use the non-linear physical model presented in the SAVIOR [16] work. We also adopt the Extended Kalman Filter (EKF) and the CUSUM algorithm to make predictions and calculate residual errors. The CUSUM threshold is selected based on an offline learning process and set to 0.32.

We validate this approach's effectiveness by injecting a GPS spoofing attack, a realistic and well-studied sensor intrusion technique. The falsified signal will cause a 1-meter deviation from the victim vehicle's real position. As shown in Fig.3, the *Maestro* is able to detect the spoofing attack right after the injection while the vehicle is cruising. In total, we perform the attack injection experiment 1000 times across different maps and road segments, and are able to achieve **86.3%** true positive rate (TPR) in **30ms** with only **6.02%** false positive rate (FPR).

However, recent work [6] points out using a fixed threshold is vulnerable to stealthy attacks which causes a small deviation that will not be detected by the default defense technique. Over a period of time, these small deviations will accumulate to large errors and eventually crash the vehicle. Similarly, we found the performance of our proposed policy significantly drops if the falsified location is within the standard deviation of normal data. For example, when

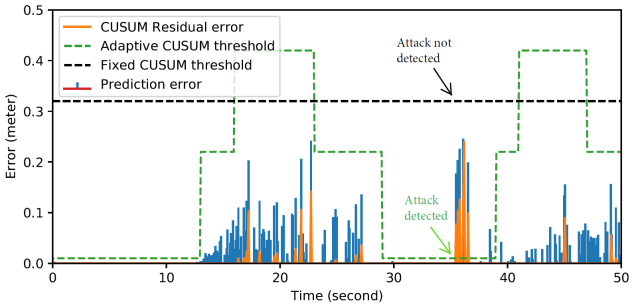
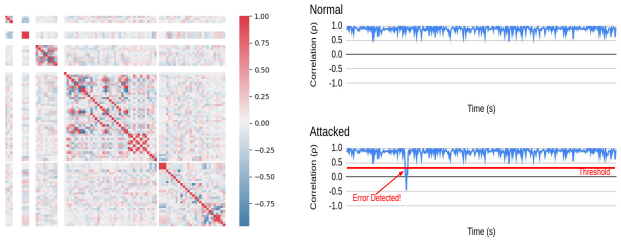


Fig. 4: Detection results on 0.2-meter GPS spoofing attack.



(a) (b)

Fig. 5: Correlation results.

Variable 1	Variable 2	Correlation	FPR
position.y	linear_velocity.y	-0.741	8.77%
position.z	orientation.qz	0.801	12.65%
orientation.qy	euler_angles.y	0.877	3.18%
position_std_dev.x	orientation_std_dev.x	0.845	2.22%
orientation.qw	euler_angles.y	-0.801	6.54%
linear_velocity.x	euler_angles.z	-0.964	2.55%
linear_acceleration.x	linear_acceleration_vrf.x	0.899	4.66%
angular_velocity.z	angular_velocity_vrf.z	0.999	0.3%
orientation_std_dev.z	linear_velocity_std_dev.z	0.699	9.20%
linear_velocity_std_dev.x	linear_velocity_std_dev.y	0.915	0.83%

TABLE I: Selected variables for attack injection.

we launch the attack causing 0.2-meter deviation, the overall TPR drops to only 11.7%. One way to detect these stealthy attacks is to apply adaptive thresholds. Because the CUSUM error largely depends on different driving situations such as going straight or turning, accelerating or cruising, we can lower the threshold if the AV is in stable states to minimize the attack window. Although driving states are not explicitly defined, they can be inferred from system-level variables.

Inspired by this idea, we export the *scenario_type* variable (indicating the current driving scenario) from the scenario manager in the planning module, and combine it with vehicle's acceleration and orientation statistics to categorize three different driving states (cruising, accelerating, and turning). For each driving state, we select the CUSUM threshold separately. As shown in Fig.4, the *Maestro* module detects stealthy attacks when the vehicle is in cruising state after applying adaptive thresholds. Overall, we observe **76.4%** TPR (achieving **6.5x** improvements compared to the fixed threshold) and **8.59%** FPR under stealthy attacks.

B. Invariant rule based intrusion detection

Natural redundancies exist in vehicles because the same physical phenomenon could result symptoms in multiple sensors [7]. One intuitive example is that pressing the accelerator pedal will cause increase in both engine RPM and vehicle speed. Thus, we can utilize correlations appearing in different sensors and system variables to detect malfunctions.

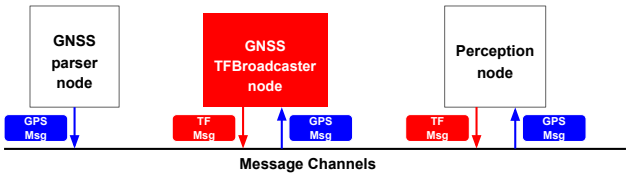


Fig. 6: Obstacle relocation attack [10].

We collect control outputs and localization information to identify correlations among sensors and actuators. In addition, we export all related internal variables from the localization and the control module to discover system-level correlations. Overall, **184 variables** are identified and their pairwise correlation under the normal operation is shown as the correlation matrix in Fig.5a. Using the same method as the prior work [7], we define a strong relationship between two variables if the absolute value of their correlation coefficient is greater than 0.5. In total, we identify **146** correlated pairs, which reveals more than **10x** information compared to the original work that only focuses on sensory data.

To detect errors, we examine the run-time correlation between two variables in a small time window (10 consecutive data points). If its run-time correlation significantly differs from the normal value, an anomaly is detected. We select 10 correlated pairs (shown in Table I) related to control, speed, and location variables to perform tests because they will directly compromise vehicles' safety once corrupted. Similar to the original work [7], we replace one of the signal values in a correlated pair with random noises to simulate sensor jamming attacks. Each pair is tested 100 times with different lane segments. As shown in Fig.5b, while the attack is present, a spike will appear in run-time correlation results, which can be detected using the proposed invariant rule. In summary, the *Maestro* module is able to detect attacks with **93.2%** TPR in **60ms**.

However, one limitation with this attack detection technique is the false positives due to natural fluctuations in sensory data. On average, we observe 5.09% FPR ranging from 0.3% to 12.65% shown in Table I. Fortunately, with more correlations identified, we find one variable usually correlates with multiple variables. Considering the worst case in Table I (shown in red texts), the *position.z* also correlates with 2 more variables, *angular_velocity.z* and *angular_velocity_vrf.z*. Instead of relying on a single correlation result to determine the anomaly, we can cross compare all related run-time coefficients and perform a majority vote. In addition, we observe the false positives resulted from sensor fluctuations are short-term effects, but intrusion attacks need to last a lot longer to become effective. Using the message buffer from the *Maestro* module, we also apply a time window and collect 20 consecutive coefficient results to check whether the anomaly happens in transient or not. With these two methods, the FPR can be reduced to only **1.02%**, achieving about **5x** reduction. Note that the time window can also be applied to the previous method to effectively eliminate false positives.

C. Attach Mitigation

In this section, we illustrate how users can define policies to detect and mitigate cross-module data corruptions

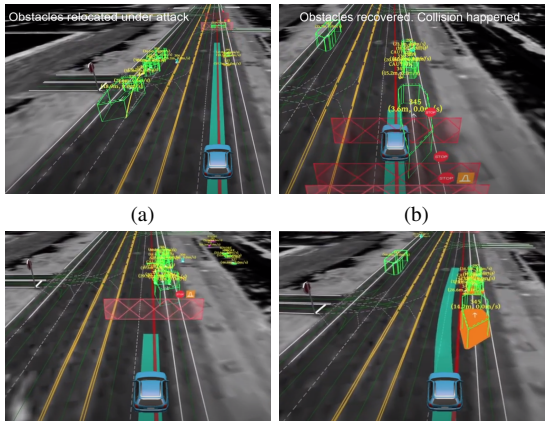


Fig. 7: Behavioral differences under the attack.

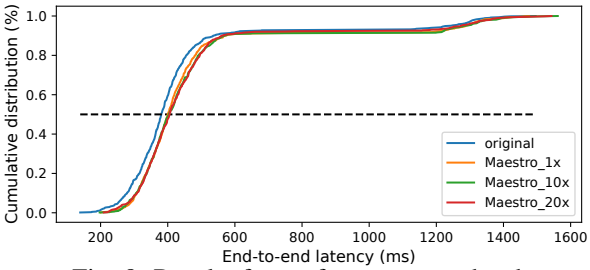


Fig. 8: Results for performance overhead.

discovered in the prior work [10], [11]. As shown in Fig.6, the proposed threat corrupts some sensitive variables in the TF message (*tf_msg.x* and *tf_msg.y*) through exploiting the publish-overprivileged variables. Under this obstacle relocation attack, the TF message will transmit falsified information, making the perception module erroneously locate perceived obstacles on the other side of the road (Fig.7a). When the victim vehicle suddenly realizes the previously detected obstacles are actually in front of the car and therefore tries to stop, it is already too late, and the vehicle collides with one of the obstacles (Fig.7b).

To mitigate this attack, we define an invariant rule to keep checking the consistency of target variables between two consecutive modules to make sure their values remain the same. In case a mismatch is detected, the policy will replace the corrupted variable with the correct information published from the source module. With the defense method instrumented, the *Maestro* module can detect the attack in **12ms** and relocate obstacles with the correct information (Fig.7c). Consequently, the AV successfully side passes obstacles and reaches the destination (Fig.7d). Although this approach can not be applied to zero-day attacks, AVMaestro still provides an effective way to handle well-studied threats. In this work, we only present one example due to the limited space, but the proposed defense policy can be generalized to mitigate other cross-module data corruptions as well.

D. Performance Overhead

AVMaestro imposes small run-time delays on the original ADS because each message needs to go through the *Maestro* module first before arriving at its destination. To evaluate the run-time overhead, we feed the *Maestro* with all policies presented in earlier sections. In addition, we extend the in-

strumented code by 10x and 20x to test *Maestro*'s scalability. Fig.8 plots the cumulative distribution graph (CDG) and the horizontal axis shows the overall latency from receiving sensor inputs to generating control commands. This is a worst case scenario in which all modules need to run sequentially. In Apollo, there are 5 modules on the communication critical path, meaning that the *Maestro* needs to trigger 4 times before generating final outputs. Realistically, since each module runs asynchronously in the system, the message forwarding overhead can be greatly masked. On average, the end-to-end latency of the original ADS is 381.9ms, whereas *Maestro-1x*, *Maestro-10x*, and *Maestro-20x* (with the corresponding code size) incur the mean latency of 389.1ms, 391.4ms, and 393.6ms, respectively. We observe the overhead roughly grows in a linear manner with *Maestro*'s code size, meaning our proposed framework is scalable. Overall, AVMaestro only imposes 1.89% to 3.06% delay on the original ADS, depending on workload sizes.

In addition, we monitor the CPU, DRAM, and GPU usages, and do not observe any noticeable change while the *Maestro* is in use. Based on these results, we believe the AVMaestro framework adds almost negligible overhead to the original autonomous driving software.

VI. Conclusion

In this paper, we present AVMaestro, a highly efficient and effective policy enforcement framework to provide a safe execution environment for autonomous driving systems. The entire design of the AVMaestro framework includes a code instrumentation module and a centralized data examination module. Through evaluations, we demonstrate that AVMaestro prototype can be easily applied to a production-level ADS with almost negligible run-time overhead. More importantly, it enables the deployment of various safety and security techniques and can further improve their accuracy with system-level information. We believe the AVMaestro will inspire new security applications on state-of-the-art ADSs. As our next steps, we are investigating a more comprehensive defense system built on top of the AVMaestro.

Acknowledgment

This research was supported in part by ONR N00014-18-1-2020, NSF CNS1850533, CNS-1932464, CNS-1929771, CNS-2145493, and USDOT UTC Grant 69A3552047138.

References

[1] AUTOWARE.AI. `https://www.autoware.ai/`, 2021.

[2] Baidu Apollo Auto. `https://github.com/ApolloAuto/apollo`, 2021.

[3] Anatolij Bezemskij, George Loukas, Diane Gan, and Richard J Anthony. Detecting cyber-physical threats in an autonomous robotic vehicle using bayesian networks. In *2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 98–103. IEEE, 2017.

[4] Yulong Cao, Chaowei Xiao, Benjamin Cyr, Yimeng Zhou, Won Park, Sara Rampazzi, Qi Alfred Chen, Kevin Fu, and Z Morley Mao. Adversarial sensor attack on lidar-based perception in autonomous driving. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2267–2281, 2019.

[5] Hongjun Choi, Wen-Chuan Lee, Yousra Aafer, Fan Fei, Zhan Tu, Xiangyu Zhang, Dongyan Xu, and Xinyan Deng. Detecting attacks against robotic vehicles: A control invariant approach. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 801–816, 2018.

[6] Pritam Dash, Mehdi Karimibiuki, and Karthik Pattabiraman. Out of control: stealthy attacks against robotic vehicles protected by control-based techniques. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 660–672, 2019.

[7] Arun Ganesan, Jayanthi Rao, and Kang Shin. Exploiting consistency among heterogeneous sensors for vehicle anomaly detection. Technical report, SAE Technical Paper, 2017.

[8] Wei Gao and Thomas H Morris. On cyber attacks and signature based intrusion detection for modbus based industrial control systems. *Journal of Digital Forensics, Security and Law*, 9(1):3, 2014.

[9] Jonathan Goh, Sridhar Adepu, Marcus Tan, and Zi Shan Lee. Anomaly detection in cyber physical systems using recurrent neural networks. In *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*, pages 140–145. IEEE, 2017.

[10] David Ke Hong, John Kloosterman, Yuqi Jin, Yulong Cao, Qi Alfred Chen, Scott Mahlke, and Z Morley Mao. AVGuardian: Detecting and mitigating publish-subscribe overprivilege for autonomous vehicle systems. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 445–459, 2020.

[11] Ke Hong. *Performance, Security, and Safety Requirements Testing for Smart Systems Through Systematic Software Analysis*. PhD thesis, University of Michigan, Ann Arbor, MI, 2019.

[12] Khurum Nazir Junejo and Jonathan Goh. Behaviour-based attack detection and classification in cyber physical systems using machine learning. In *Proceedings of the 2nd ACM International Workshop on Cyber-Physical System Security*, pages 34–43, 2016.

[13] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[14] FENG Minjian, ZHANG Hui, JU Zhiyang, and XU Qing. Localization estimation algorithm under cyber delay attack for autonomous vehicle based on lgsvl/apollo. *Journal of Automotive Safety and Energy*, 12(1):62, 2021.

[15] Jonathan Petit, Bas Stottelaar, Michael Feiri, and Frank Kargl. Remote attacks on automated vehicles sensors: Experiments on camera and lidar. *Black Hat Europe*, 11:2015, 2015.

[16] Raul Quinonez, Jairo Giraldo, Luis Salazar, and Erick Bauman. Savior: Securing autonomous vehicles with robust physical invariants.

[17] Timothy Trippel, Ofir Weisse, Wenyuan Xu, Peter Honeyman, and Kevin Fu. Walnut: Waging doubt on the integrity of mems accelerometers with acoustic injection attacks. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 3–18, 2017.

[18] Yazhou Tu, Zhiqiang Lin, Insup Lee, and Xiali Hei. Injected and delivered: Fabricating implicit control over actuation systems by spoofing inertial sensors. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1545–1562, 2018.

[19] David I Urbina, Jairo A Giraldo, Alvaro A Cardenas, Nils Ole Tippenhauer, Junia Valente, Mustafa Faisal, Justin Ruths, Richard Candell, and Henrik Sandberg. Limiting the impact of stealthy attacks on industrial control systems. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1092–1105, 2016.

[20] Chen Yan, Wenyuan Xu, and Jianhao Liu. Can you trust autonomous vehicles: Contactless attacks against sensors of self-driving vehicle. *DEF CON*, 24, 2016.

[21] Kexiong Curtis Zeng, Shinan Liu, Yuanchao Shu, Dong Wang, Haoyu Li, Yanzhi Dou, Gang Wang, and Yaling Yang. All your gps are belong to us: Towards stealthy manipulation of road navigation systems. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1527–1544, 2018.

[22] Qingzhao Zhang, David Ke Hong, Ze Zhang, Qi Alfred Chen, Scott Mahlke, and Z Morley Mao. A systematic framework to identify violations of scenario-dependent driving rules in autonomous vehicle software. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 5(2):1–25, 2021.