EECS 598 Team 5 Project Report

Yatian Liu, Haoliang Cheng, Kyle Liebler, Rohan Chandy

1 Introduction

The goal of our project is to design an voice-activated embedded device that could place orders on DoorDash. Our device is built upon an ESP32, a low-cost, low-power microcontroller with integrated Wi-Fi and Bluetooth connectivity. This is important because we want our system to act as a standalone device, essentially being able to operate independent of anything other than a Wi-Fi connection and power source. In addition to the ESP32, we integrated an ADMP401 microphone to record user commands. The voice commands we provide to our users are as follows:

- Add <Quantity> <Item>
- Clear Cart
- Place Order <Password>

This hardware alone provides us with the base functionality to place an order on DoorDash, but as part of our requirements, we implemented additional functionalities. For example, our system supports the use of DTMF tones of which the tones and associated functionalities are listed below:

- DTMF tone 1 = “Add one sandwich”
- DTMF tone 2 = “Clear cart”
- DTMF tone 3 = “Place order”

Also, our system needs to provide LED feedback when 1 kHz and 3.5 kHz frequencies are played and blink according to their intensity. Finally, we integrate hardware and software defenses to protect against a variety of different attacks. Our hardware defense is a TEMT6000 light sensor which could detect and prevent laser-based attacks mounted on the microphone, and our software defense is requiring a password to place an order and set audio detection threshold. Both of these defenses aim to prevent attackers from injecting commands into our system without the user’s consent. Figure 1 is a picture of our final design.

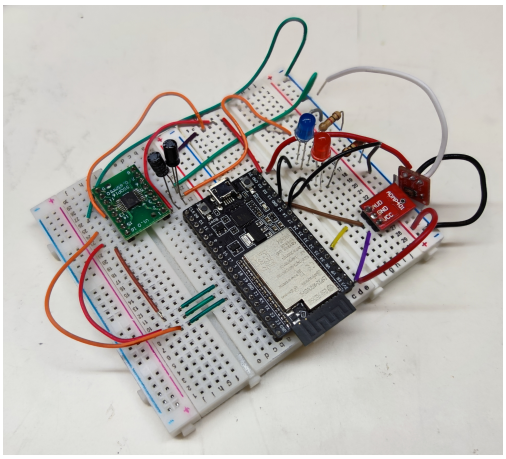

Figure 1: Device

2 Related Work

To develop our system, we referenced research papers and resources provided by the EECS 598 course staff. Firstly, our laser attack was directly based off the laser-injection methods introduced in the ”Light Commands: Laser-Based Audio Injection Attacks on Voice-Controllable Systems” paper [1]. Similarly, we also modeled our laser driver circuit off the circuit described in Lab 4 (Laser Lab) of EECS 598 [2].

3 Implementation

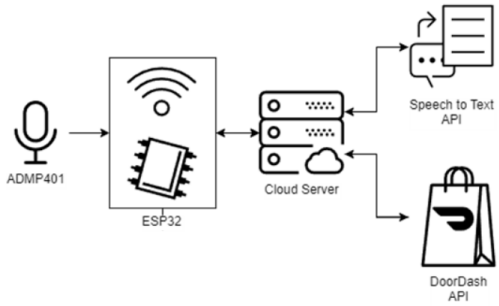

Figure 2: System diagram

As the system diagram shows, our voice-controlled DoorDash assistant can be divided into two subsystems. The embedded subsystem collects data from the microphone, does the initial frequency analysis, and sends the data to a cloud server via WiFi. The cloud subsystem receives data from the embedded subsystem and interacts with Google’s speech-to-text API and the DoorDash API. With both systems working together, the device is able to convert microphone data to text commands and send those commands to the DoorDash server. We will describe the implementation details of these two subsystems separately.

3.1 Embedded Software and Hardware Implementation

The core component of the embedded subsystem is the ESP32 microcontroller. We chose this chip because we wanted to build an IoT device with WiFi capability and it is one of the most popular microcontrollers

with built-in WiFi hardware. Besides the WiFi-related hardware, it also has other built-in peripherals like ADCs and DACs, which allows us to sample the microphone's output without external hardware. The main programming framework for the ESP32 is ESP-IDF [3], which is free and open source and mainly written in C. It is based on FreeRTOS and provides many useful built-in functions, but because of the syntax of C, the embedded code is harder to write compared to C++ and Python code. The ESP32 microcontroller is connected to three external devices: the ADMP401 microphone, the red LED, and the blue LED. Each LED is accompanied by a 1 kΩ resistor for regulating current. The microphone's output is connected to one of the ADC input pins, and the two LEDs are connected to GPIO pins.

To achieve fast sampling of the microphone's data, we use the I2S Direct Memory Access (DMA) mode of the built-in ADC. In this mode, the ADC will write its output to a DMA memory buffer by itself, and the main CPU only needs to copy data from the DMA buffer to the main memory periodically, which reduces the load on the main CPU. By using this method, our sampling rate for the microphone reaches 48 kHz, which is a standard high-quality audio sampling rate. We set the length of one data batch to 0.1 s, and we run the audio sampling task periodically at 10 Hz using FreeRTOS.

After getting 0.1 s of data, the microcontroller will calculate the FFT of the data on board using a free and open source DSP library called ESP-DSP [4]. This is used to extract the strength of the DTMF tones and the strength of 1 kHz and 3.5 kHz signal for controlling the LEDs. For the detection of DTMF tones, we choose the greatest value of the 4 lower DTMF frequencies' strength $a_{f_{\text{lower}}}$ and the greatest value of the 4 higher DTMF frequencies' strength $a_{f_{\text{higher}}}$, and consider a DTMF tone to be present when both $a_{f_{\text{lower}}}$ and $a_{f_{\text{higher}}}$ are greater than a preset threshold of 3 dB. The strength of the 1 kHz and the 3.5 kHz frequency bins are saved in two global variables. There are two separate FreeRTOS tasks for controlling the blinking of the two LEDs using the two global variables, which will be explained later.

To simplify the implementation of sound event detection, we fix the length of a command to 4 s. For every 0.1 s of data $\{d_n\}$, we calculate $l = \max\{|d_n|\}$, and think a sound event is happening if l is greater than a preset threshold. Whenever a sound event is detected the microcontroller will record 4 seconds of audio, which is 40 periods of sampling, and send them together to the server using HTTP POST requests. In addition to the raw audio data, the DTMF tone detection results for each 0.1 s of sample data are also sent to the server for command recognition.

We control the two LEDs' blinking frequency by delaying for a specific time, and since the period of the LED blinking varies, it is hard to put it inside the audio sampling task with a fixed period of 0.1 s. To solve this problem, we put the control of the two LEDs' blinking into two separate FreeRTOS tasks, so that they can set their own delay and period independently. However, having multiple tasks also introduces new problems. In particular, the LED blinking tasks and the audio sampling task need to share the data on whether a sound event is happening and being recorded and the strength of the 1 kHz and the 3.5 kHz frequency bins, which leads to race conditions if no protection is added. For this, we just use the standard solution of adding a mutex for the shared global data of the three tasks, and the tasks will grab the mutex before accessing the shared data and release the mutex after accessing the shared data. The formula for calculating the LED's blinking frequency is given as

$$f_{\text{blink}} = \frac{l - l_{\text{low}}}{l_{\text{high}} - l_{\text{low}}} \cdot (f_{\text{high}} - f_{\text{low}}) + f_{\text{high}},$$

where l is the strength of the 1 kHz or the 3.5 kHz frequency bin in dB, $l_{\text{low}} = 10\,\text{dB}$ is the lower level threshold, $l_{\text{high}} = 40\,\text{dB}$ is the upper level threshold, $f_{\text{low}} = 2\,\text{Hz}$, and $f_{\text{high}} = 20\,\text{Hz}$. If $l < l_{\text{low}}$, the LED will not blink; if $l > l_{\text{high}}$, the LED will blink with frequency f_{high}.

3.2 Server Software Implementation

The source code of our server software can be found at `https://gitlab.eecs.umich.edu/eecs_598/assistant-api` [5]. To interact with our hardware, a cloud server was hosted in the form of a RESTful API through a provider called render. Our server serves as an intermediary micro-service which contains its own logic and facilitates interactions between other services.

At our server's core is a Flask application with four main routes and an HTML page:

- /api - Health check endpoint.
- /api/transcribe - Development endpoint to see audio transcriptions. Input is an HTTP POST request of the audio bit stream from the microphone sent over the ESP32. Transcription is sent to Google Speech-to-Text and returned as the response.
- /api/save - Development endpoint used to help see the recordings and transcriptions server side. It saves the audio and transcriptions to a disk and serves them on an HTML page at the root directory on the server to be viewed.
- /api/dash - Handles the full order flow by acquiring audio transcriptions and sending the result to our command parser. The resolved command is then executed and the corresponding API call is made to DoorDash if the command is valid.

To serve our requests, some of these endpoints utilize external services such as Google's Speech-to-Text API and DoorDash's private API.

To interact with Google Speech-to-Text, a couple functions were written to convert the audio bit stream into a legible format for Google's API as well as to configure and send the request.

To interact with DoorDash's private API a more complicated design was needed. First, API endpoints and request payloads were scraped using Chrome DevTools and Postman. These payloads were retrieved as JSON and contain GraphQL queries. To avoid having to deserialize GraphQL, these payloads were left as is and individually saved to their own files in JSON format. With payloads saved for all necessary functions (adding individual items to cart, clearing the cart, ordering, getting the cart items, getting the cart id), they can be loaded from their files, modified (setting item quantity, cart id, etc.), serialized, and sent off as HTTP requests.

In order to utilize our interactions with external services correctly in the context of processing requests to our API, there is a command parser. The command parser searches through the retrieved audio transcription for keywords resolving to commands in many ways, some depending on sequential input and some order-invariant. The intent is to execute syntactically correct commands and ignore erroneous ones. After a command is discovered and the command's parameters are successfully parsed, the correct payload is loaded, modified, serialized, and executed as described previously. It is either loaded directly, or in the case of a specific item, the command's parameters are passed to an item payload resolver to retrieve the correct payload for that item. The command parser also contains additional logic to handle DTMF tones. The bits at the end of the data sent by the ESP32 contain information on whether a DTMF tone was detected for every 0.1 seconds of audio data. The command parser calls a utility function which aggregates these detected tones and resolves them to the most prominent tone found for that request if a threshold is reached. In this situation the command parser ignores any speech command and executes the command associated with the DTMF tone identified.

3.3 Attack Mechanisms

We choose a laser-based attack as our attack mechanism. In order to modulate the driving voltage of the laser diode using audio files from a computer, we design a driver circuit to efficiently couple the AC output of a computer's audio jack with a DC offset and amplify the peak-to-peak voltage of the laser input. The driver circuit is from a document by Texas Instruments [6], which is shown in Figure 3:

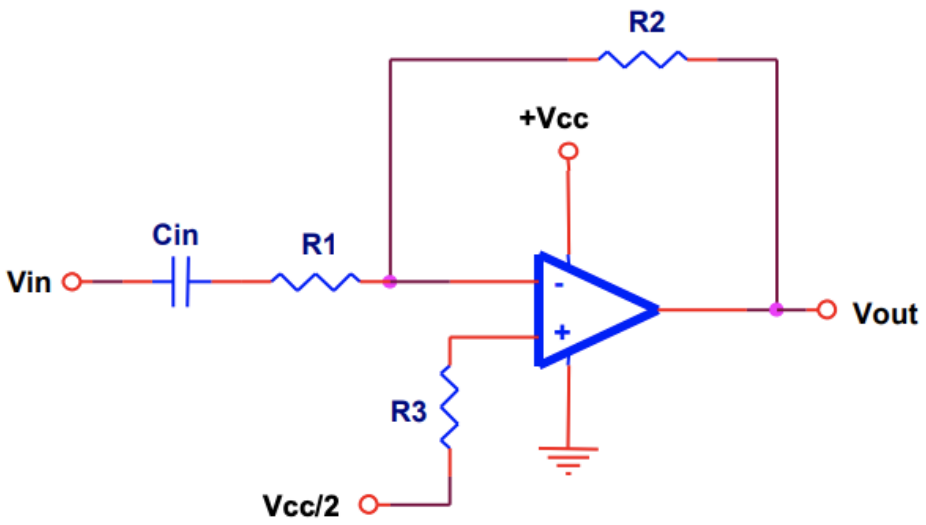

Figure 3: Laser Driver Circuit

We connected V_{in} to our audio input, and V_{out} to the laser diode. V_{cc} is set to 9 V and $V_{cc}/2$ is obtained from a voltage divider. We used an op-amp based inverting amplifier, and used the ratio of R_1 and R_2 to control the gain of the amplifier and thus the output peak-peak voltage. We tried different sets of R_1 and R_2 values, and got the following results:

1. $R_1 = 4.7\,\text{k}\Omega$ and $R_2 = 4.7\,\text{k}\Omega$: The peak-peak voltage is too small for the laser diode to inject clear signal.
2. $R_1 = 4.7\,\text{k}\Omega$ and $R_2 = 15\,\text{k}\Omega$: The peak-peak voltage is too large which leads to laser diode saturation and signal distortion.
3. $R_1 = 4.7\,\text{k}\Omega$ and $R_2 = 10\,\text{k}\Omega$: The peak-peak voltage is appropriate to let laser diode to inject a clear signal.

Therefore, we use $R_1 = 4.7\,\text{k}\Omega$ and $R_2 = 10\,\text{k}\Omega$ as the final setup.

Finally, we use Google Translate text-to-speech function to transform the text commands we want to inject to audio.

3.4 Defense Mechanisms

3.4.1 Hardware Defense

Targeting at laser attacks, we choose to use a light sensor (TEMT6000) as our hardware defense for our system. When the sensor value is higher than the threshold we set, we disable the function of recording

audio and sending recorded data to the server. Thus, when a laser attack is performed on our system, no matter what commends the adversary intended to inject, it will be disabled by the light sensor.

We used a high resolution external ADC to sample the reading of the light sensor since the two ADC channels for ESP32 are all occupied exclusively, one by microphone DMA sampling and the other by WiFi communication. Also, we can get more accurate reading from the light sensor so that we can distinguish between a laser attack and background light intensity. In our experiment, we found that under normal situation with all lights on in a room, the light sensor reading is about 150. When the laser is focused on the microphone, the light sensor reading increases to above 1000. Therefore, we can clearly detect the laser attack and we choose the threshold value to be 750.

3.4.2 Software Defense

We have two types of software defense: **Password** and **Detection Threshold**.

Password Defense:

We add a password confirmation after the command "Place Order". If the password defense is turned on, only "Place order" followed by the correct password will be executed. The user can customize their own private password, which prevents the adversary from maliciously placing their order and cause real-world damages.

Detection Threshold Defense:

Our voice-controlled system uses the microphone to continuously sample audio, but only when the maximum amplitude in 0.1 s is above the **defined threshold**, the ESP32 will start to record the sound, send it to our server and process the sound. Therefore, this function allow us to filter low amplitude sounds.

From our experiment, we find that the amplitude of the laser injected signal for a 5 mW laser is always smaller than the normal human voice. Therefore, we can manually set the wake-up threshold value to be above the amplitude of laser, but below the amplitude of human voice. In this scenario, when a laser attack happens, the laser injected signal can not wake up our system, and thus can not transmit any information to our server.

4 Evaluation of Performance and Security

To evaluate the performance of our system, we tested it under a variety of different scenarios and use cases. Details of each experimental setup and results are listed below.

4.1 Voice Control Performance

First, to measure the accuracy of our system, we tested all of our supported commands (clear, add, and order) in a quiet environment. Both normal voice as well as DTMF tones to input commands were tested and yielded almost perfect and fully perfect results respectively.

Next, we tested all the commands with background noise simulated from an audio recording playing from a Dell XPS 13 laptop at 50% volume. From the results, we can see that the add command's performance greatly decreased. This is because add is the most complex command of the three. For the add command to be parsed, the word add, the quantity, and the item need to be correctly transcribed. On the other hand, for the clear command, only the word clear needs to be detected, and for the order command, only the words order and the correct password. Also, since we are taking advantage of Google's Speech-to-Text API,

context greatly influences how the audio is transcribed. For example, if the add command is intended to be "add 2 fries", if Google initially transcribes it to "add to" (very similar sounding phrases), we noticed that the last word would become "fry's". This obviously led to both the quantity and item being incorrect and the overall command failing.

All of our results are shown in the table below.

Command	Normal Voice	50% Noise	DTMF
Clear	10/10	4/5	9/9
Add	8/10	2/5	9/9
Order	9/10	4/5	9/9

4.2 Security Performance

To measure the security performance of our system, we launched laser-based attacks on it and measured the success of the injected commands with defenses enabled and disabled. Our main defense was the light sensor, so with defenses off, the light sensor was unplugged, and with defenses on, the light sensor was plugged back in. As you can see from the results below, the laser attacks from the driver circuit we developed were almost 100% effective with no defenses, and with the defenses re-enabled, the system always detected the attack and prevented any injected commands from being executed.

All of our results are shown in the table below.

Command	Defense Off	Defense On
Clear	5/5	0/5
Add	5/5	0/5
Order	4/5	0/5

5 System Performance in the Demo and Tournament

In the demo, our system was able to successfully recognize normal voice and DTMF inputs as well as blink the LEDs according to the intensity of the respective 1 kHz and 3.5 kHz tones. We were unable to showcase the laser-based attacks and defenses due to limited time but we proved with our results in the Security Performance section that both our laser attacks and defenses were effective.

For the tournament, our team had to perform both attacks and defenses. Regarding the attack, we launched a laser-based injection attack on the opposing team with their defenses disabled and enabled. With the security procedures disabled, we were able to inject three commands correctly (voice and DTMF commands).

With the software defenses on, we were unable to inject any commands because their system utilized a liveness test where the user had to answer simple math questions. Since we couldn't see the problem, we had no way of passing this check. Similarly, their hardware defense was sensor fusion, utilizing two microphones to verify each others output, and if the recorded audio was drastically different, an attack was detected. Since we didn't have a setup to deploy and synchronize two laser injections, we were also unable to beat this defense.

For our defense, the opposing team used an ultrasonic-based injection, completely circumventing our light-sensor hardware defense. Although, for our software defense, without the attackers knowing the password, they were unable to place any orders, yielding our password successful in protecting a privileged operation.

6 Conclusion

The main limitations we faced during our development was in regards to the ESP32. For example, even though the ESP32 is equipped with two ADCs, only one of them was available for free use as the other was exclusively occupied by Wi-Fi communication. Therefore, we had to incorporate an external ADC into our design to support both the ADMP401 microphone and TEMT6000 light sensor. Also, due to the limited computational power of ESP32's CPU, sending 0.1 s of data using an encrypted HTTPS request takes more than 0.1 s and thus we could not send the data in real time. We have to record 4 seconds of audio and then send the audio to the server, and in the sending process we can not sample new data from the microphone, which adds extra delay and makes our system less responsive. Similarly, the limited memory forced us to move much of our computation to server side as opposed to running on board, adding delays to the execution of user commands.

Therefore, if the application scenario allows for a higher budget, using more expensive hardware such as a Raspberry Pi would have provided more functionality and faster processing, allowing for greater flexibility in our design. We would have been able to record longer and more complex commands and process them in real time at a faster rate.

Finally, our defenses could have been improved to provide security against a robust set of attacks. Our light sensor was able to reliably detect laser-based attacks but was not impervious to other attack-vectors such as ultrasonic. On the hardware side, we could have integrated an ultrasonic sensor or added a low-pass filter to detect and prevent high frequency signals from being demodulated into the human-audible range. On the software side, we could have added liveness tests such as having the user repeat which LED is flashing, repeat a random word or number generated by the system, or do a simple math problem. As the attacker in laser or ultrasonic based attacks isn't in audible range of the system, these measures would detect if a user is actually in proximity of the system and prevent any malicious interactions.

References

[1] T. Sugawara, B. Cyr, S. Rampazzi, D. Genkin, and K. Fu, "Light commands: Laser-based audio injection attacks on voice-controllable systems," 2020. [Online]. Available: https://arxiv.org/abs/2006.11946

[2] K. Fu, "Lab 4: Laser injection against mems microphones," 2022.

[3] "Espressif iot development framework." [Online]. Available: https://github.com/espressif/esp-idf

[4] "Espressif dsp library." [Online]. Available: https://github.com/espressif/esp-dsp

[5] "Assistant api," 2022. [Online]. Available: https://gitlab.eecs.umich.edu/eecs_598/assistant-api

[6] B. Carter, "A single-supply op-amp circuit collection." [Online]. Available: https://mil.ufl.edu/4924/docs/TI_SingleSupply_OpAmp.pdf

[7] "Doordash assistant embedded software," 2022. [Online]. Available: https://gitlab.eecs.umich.edu/eecs_598/doordash-assistant