Exposing the Imposter's Hunger for Power: Hardware Keylogger Attack Detection

A Major Qualifying Project (MQP) Report
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the Degrees of Bachelor of Science in

Computer Science
as well as
Electrical and Computer Engineering

By:

Brianna Roskind

Project Advisors:

Prof. Robert Walls (CS)
Prof. Patrick Schaumont (ECE)

Date: April 2023

Abstract

This Major Qualifying Project constructed a proof of concept circuit and analysis method for detecting the presence of a hardware keylogger connected in series between a USB keyboard and USB port on a computer. Different methods of keylogger detection were reviewed, and the behavior of two keyloggers were examined, which led to the selection of power signature analysis to detect a keylogger. Data collected in a laboratory environment unveiled the need for a custom circuit in order to have higher resolution power signature data available for analysis. Statistical measurement methods for histogram analysis were examined, including their short-comings, leading to the creation of an augmented form of the KL-algorithm, and the construction of a threshold detector. With threshold detection, no false positives (mistaken detections) occurred. Of the two keyloggers tested, one was detected 100% of the time within a 5 minute period, and the other was detected 100% of the time within a 10 minute period.

Throughout this project, experience in circuit design and analysis, literature review, Python programming, and technical writing was acquired. Future directions for this MQP were also identified, including creating an ASIC chip for production use.

Acknowledgements

I would like to thank my two advisors in this MQP, Professor Robert Walls and Professor Patrick Schaumont of Worcester Polytechnic Institute (WPI), for their time, encouragement, and support during this multi-semester project. I'd also like to thank Craig Williams, Senior Manager at Amazon for Threat Intel, Detection, Intel, and Response, for spending time discussing hardware malware with me, and providing pointers to online resources. Additionally, I'd like to express my gratitude to WPI for providing me with the opportunity to execute this project for my MQP.

Contents

List of Tables

List of Figures

1 Introduction

There are now small and relatively inexpensive USB devices which can be placed between a keyboard and a computer to covertly record keystrokes and transmit any recorded data to an attacker. The name of this device is a hardware keylogger. A typical “insertion attack” might consist of an attacker paying a cleanup crew to insert a keylogger at night while cleaning the office. Attackers may target “high value” assets such as a CEO, a highly trusted executive, or even a console that is routinely logged into by numerous employees. Examples of sensitive data that are of interest to such bad actors include passwords and even typed emails or documents.

There is not currently a foolproof way to detect the existence of a hardware keylogger. The most common method of hardware keylogger detection is simply for the owner of a computer to be vigilant and check their USB ports every time they come back from leaving their computer unattended. Since this completely relies on a human not forgetting to check, and successfully catching any keyloggers when they check, this is a very error prone process. Additionally, there is also not a good way for the operating system (OS) to detect whether a keylogger has been added. This is due to the fact that most keyloggers will pass on the device name and information of the USB keyboard it is connected to when it is registering with a USB port. Keyloggers, if they are smart, will not let the OS. know that they have a different device name.

This MQP proposes use of power draw analysis for rapidly mitigating such malicious hardware insertions. Manual checking is a “best intentions” approach which requires a user to perform daily inspection, and we need a tireless “automated mechanism” for detection. Power draw analysis, for keylogger detection, mixed with a user warning system, would create a much less error prone solution. An automated method removes errors from the need for consistent and thorough manual searching. It would also save humans the time from needing to constantly check. One of the reasons power draw analysis is superior to manual checking is that power utilization statistics of a true keyboard would be difficult for an attacker to mimic correctly. A keylogger is, by itself, a processor. In this study we have shown that a keylogger’s variable power draw statistics have proved to (currently) be detectable in several commercially available keyloggers.

Creating a reliable method of keylogger detection is not an easy task. Keyloggers are designed to be difficult, if not impossible, to detect via standard methods such as USB protocol based inspection. Additionally, the physical design of keyloggers is constantly advancing to make them almost impossible to visually detect. They are often designed to appear as “ordinary looking” USB cables! To ensure the reliability of our detection method, our solution needs to be tamper-resistant, as even those with minimal physical access to an area are a threat. Employees such as janitor's can successfully insert and retrieve keyloggers from USB ports without raising much or any suspicion. Therefore, our ultimate goal is to make our design part of the motherboard or part of the USB port. This way, a keylogger cannot be placed between our detection mechanism and the computer, and the detection mechanism cannot be easily tampered with or removed. However, keyboards themselves can add complexity to keylogger detection via power draw analysis. The basis of our project is that keyloggers have a detectable power draw. It turns out that LEDs on keyboards, such as the caps lock key, increase power usage as well when they are on! Therefore a threshold alone is not enough to detect a keylogger. This means we need a more sophisticated mechanism for detection, while still ensuring that the end product is cheap to manufacture, in terms of hardware design elements.

The study contributes both hardware and software proof of concept elements to the field of hardware keylogger detection. This study supplies a hardware design to measure keyboard and keylogger power draw distributions. We also provide modifications to the KL-divergence algorithm to better compare and analyze these power distributions for keylogger detection. In our analysis we are comparing histograms that represent probability distribution functions (PDFs) that can be relatively sparse. The KL-divergence algorithm works best for probability distribution functions that don't have any empty buckets. Therefore we first add one to each bucket to allow the sparse PDFs to be meaningfully compared. Second, we take the absolute value of the terms summed by the KL-algorithm, such that we penalize differences between PDFs, whether positive or negative. Thirdly, we include a method of analysis of convolutions of these power distribution graphs to best align distributions after they are individually normalized. All of these components work together to continuously detect anomalies in the keyboard power utilization which are indicative of a keylogger.

The following sections of this report outline the observations, conclusions, and testing done to establish and verify our detection method via power signature analysis. In section 2 we discuss the background of Keyloggers, with explanations on properties, prices, and the risks they pose. In section 3 we discuss the threat model and problem definition. In section 4 we review the lab observations that led us to creating a custom circuit. Section 5 outlines the design and cost breakdown of our circuitry for keylogger detection. Section 6 outlines the original KL-divergence algorithm along with our augmented KL-algorithm. Section 7 discusses our coding implementation of a threshold detection system, and the efficacy of the results. Section 8 outlines a brief conclusion of the work completed and outlines ideas for future work in the area of combatting hardware keyloggers.

2 Background

2.1 Background on Keyloggers

A keylogger is a physical USB device, smaller than the butt of a cigarette, that can be connected in series with a USB keyboard, with the purpose of recording keystrokes, and even sometimes performing keystroke insertion. The typical cost of a keylogger today (in 2023) is approximately $45 to $100 USD (U.S. Dollars). The website that we found to be trustworthy in selling keyloggers was https://www.keelog.com. All of the keyloggers we use for study and reference in this MQP came from this site.

2.1.1 Keylogger Examples

While keyloggers in general are very small, they can come in a variety of sizes. For example, a large keylogger may be 36 mm x 20 mm x 12 mm. For reference, that is a little longer than a grape but still shorter than a walnut (Tumor Size, n.d.). Keyloggers of this size can range from $45 to $100, depending on the extra abilities of the keylogger. As can be seen in Figure 1, the KeyGrabber USB, which has 16 megabytes of memory and 128-bit encryption, costs only $45 (KeyGrabber USB 16MB - USB Hardware Keylogger With 16MB Flash Drive, n.d.). In Figure 2, we can see the KeyGrabberTimeKeeper USB, which has 16 gigabytes of memory and date and time stamping capabilities, in addition to 128-bit encryption, costs $100 (KeyGrabber TimeKeeper USB MCP 16GB - Premium USB Hardware Keylogger With 16GB Flash Drive, Time-Stamping and Mac Compatiblity, n.d.).

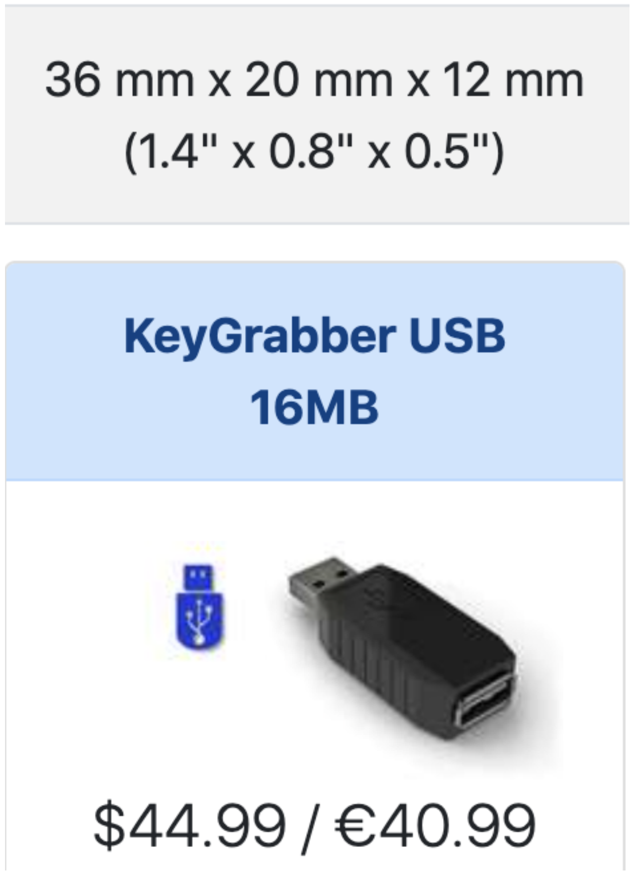


Figure 1: KeyGrabber USB from www.keelog.com.

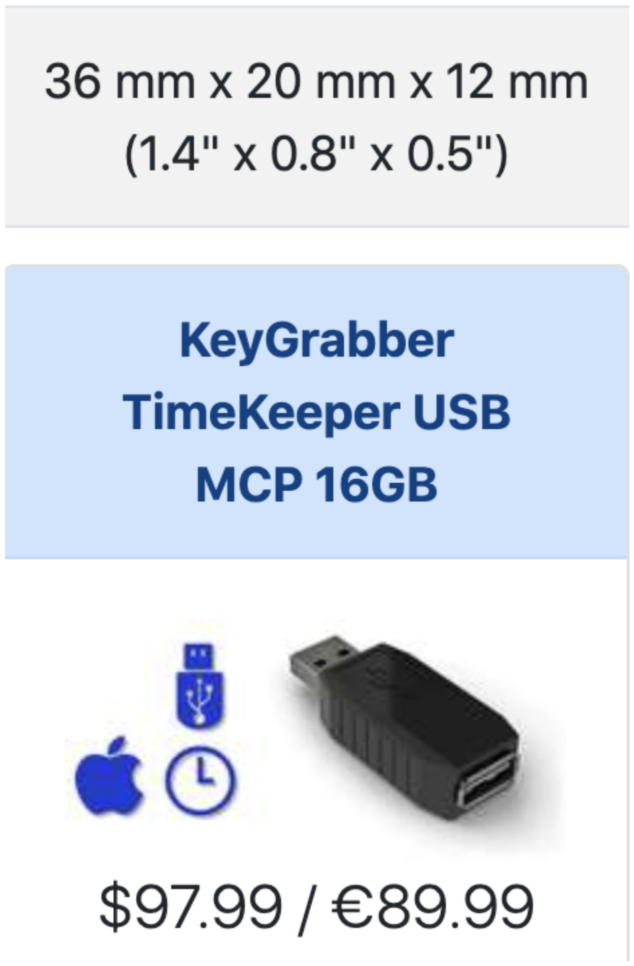


Figure 2: KeyGrabber TimeKeeper USB from www.keelog.com.

On the other hand, a small keylogger may be 10 mm x 16 mm x 11 mm, and be priced anywhere from $45 to $75. For reference, this is longer than a pea and shorter than a peanut (Tumor Size, n.d.). The website, www.keelog.com, refers to keyloggers of this size as the "smallest keylogger on the market." Looking at Figure 3, the AirDrive Forensic Keylogger, which has 16 megabytes of memory, works as a WiFii hotspot, can have data retrieved remotely, and has memory protection in the form of hardware encryption, costs only $45. In Figure 4, we can see the AirDrive Forensic Keylogger Pro, which in addition to what the non-pro version supports, also works as a WiFi device and can send email reports with the recorded keystroke data, supports time-stamping and live data-streaming, costs $53. An even more complex version can be seen in Figure 5, the KeyGrabber Forensic Keylogger Max, which has 16 gigabytes of memory, "Sophisticated USB frame capture algorithm with 32X oversampling," 128-bit encryption, leaves no wireless footprint, works as a keylogger and a HID keystroke generator, and has a built-in scripting language interpreter, costs $75.

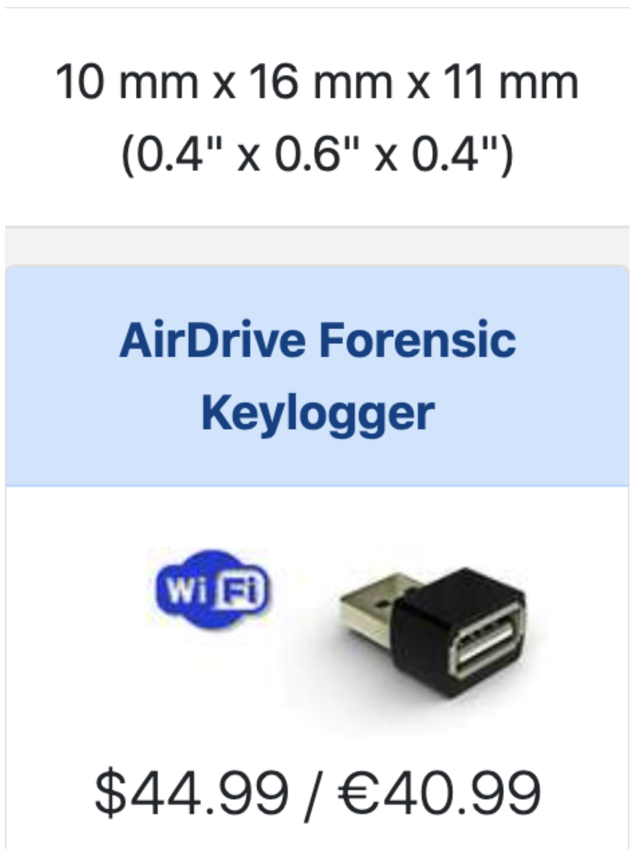
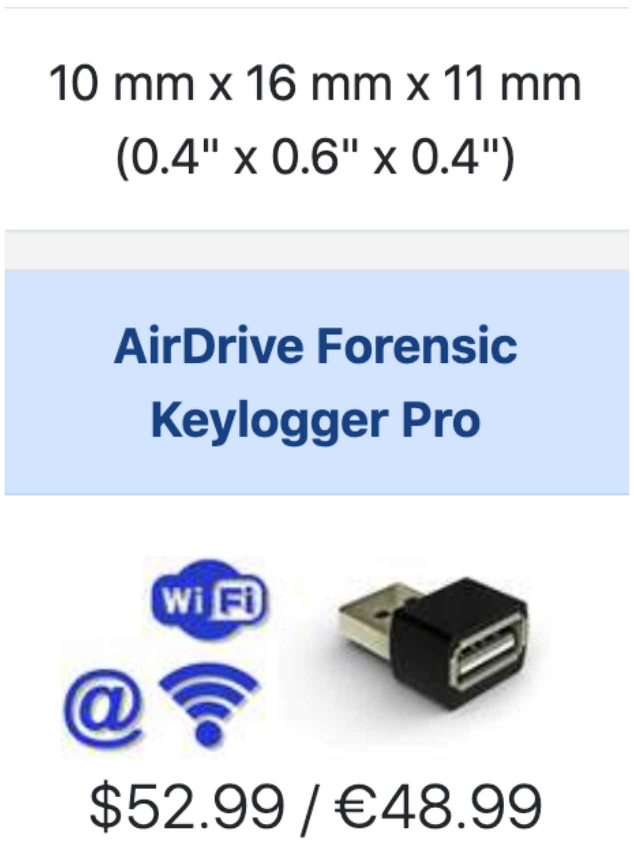


Figure 3: AirDrive Forensic Keylogger from www.keelog.com.



10 mm x 16 mm x 11 mm
(0.4" x 0.6" x 0.4")

AirDrive Forensic Keylogger Pro

$52.99 / €48.99

Figure 4: AirDrive Forensic Keylogger Pro from www.keelog.com.

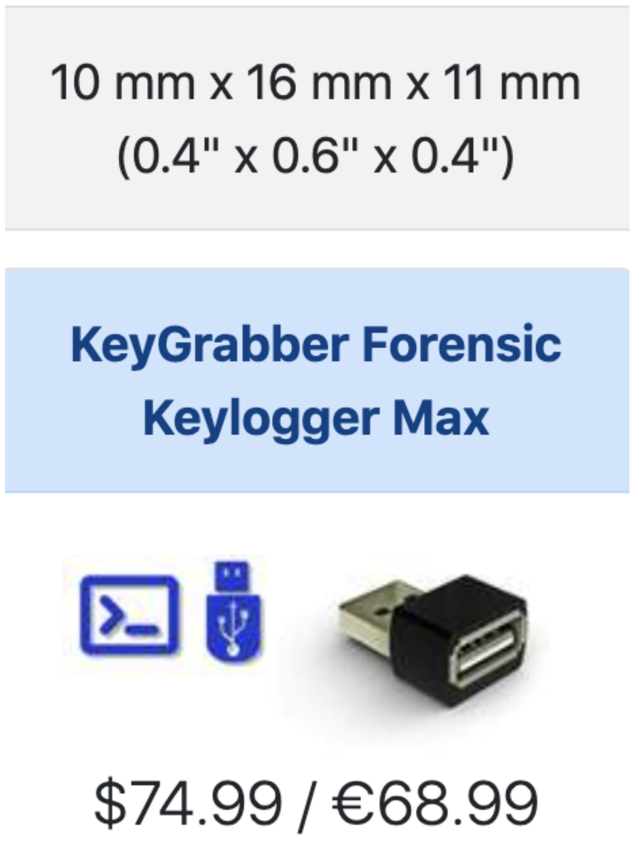


Figure 5: KeyGrabber Forensic Keylogger Max from www.keelog.com.

2.1.2 Keylogger Features

Possible features on a keylogger include but are not limited to the following:

- Built-in FPGA chip
- Built-in memory
- Built-in RTC + battery
- Date and time-stamping
- E-mail reporting
- HID injection scripts
- Keystroke generation
- Keystroke logging

- Live data streaming
- Mac Compatibility Pack (MCP)
- Programmable scripting language
- Remote configuration
- Retrieve data remotely
- Remotely download log
- Text Menu Mode
- USB flash drive mode
- Wi-Fi Access Point
- Wi-Fi Device

The typical memory capacity on a hardware keylogger ranges from 16 MB to 16 GB. For reference, 1 MB is approximately 500 pages of text, or “1 thick book” (Wynn, 2023).

Exfiltration of logged keystrokes may be done via WiFi, manual retrieval of the device or via direct use of the keyboard connection. WiFi exfiltration of keystrokes makes it easier for the malicious party to obtain the data without having to physically revisit the insertion point. This prevents the perpetrator from being caught in the act, and may allow the device to go undetected even longer, since no one will be caught visually examining or touching the USB device.

3 Threat Model and Problem Definition

3.1 System/Attacker Model

3.1.1 Corporate Environment

Computers in corporate workspaces are not fully guarded 24/7. Companies may have security measures to keep unauthorized personnel from getting in, but they cannot easily identify insiders with malicious intent. The insiders are often part of cleaning crews or other lower paid personnel who can be bribed by a bad actor to put a keylogger in place in return for monetary compensation. These companies do not have a good way to prevent bad actors from compromising such members of their personnel. Larger companies can have too many employees to ensure everyone is completely loyal to the company, and all it takes is for one insider to be compromised.

The keylogger, once put in place, will often aim to capture passwords or confidential emails as they are typed by the victim. Therefore, anyone in the company who has passwords, or knows confidential information is a potential target.

However, not all computers are at risk from simple hardware keyloggers, as the real target is computers that use USB keyboards.

3.1.2 Attacker's Access / How Information is Stolen

Attackers can gain access to the workspace by bribing lower paid personnel to discreetly put a keylogger in place. Once the keylogger is in place, the attacker can wait days, weeks, or months before attempting to retrieve data from the keylogger. Retrieval and exfiltration of the keylogger data can happen in a few ways:

1. If the USB keylogger does not have any fancy wireless abilities, a maintenance crew member can be bribed to remove the keylogger and return it to the attacker.

2. If the USB keylogger has wireless abilities, a cleaning crew member can be bribed to carry a cell phone belonging to the attacker with them when they are cleaning. When the phone is close enough to the keylogger to connect, the keylogger can then export the information it collected to the phone.

3. There are even fancier keyloggers that can send out automated email reports of the data they recorded.

3.2 Problem Definition

This MQP focuses on the mitigation, and specifically the detection, of USB hardware keyloggers. A secondary element of mitigation involves notifying a victim of the likely presence of a malicious device, as a visual inspection (and manual removal) should be sufficient for neutralizing the threat. We perceive that notification is a straightforward software element that could be implemented in and around an operating system, and thus we do not focus on that aspect.

What we are exploring is a proof of concept solution, with external inline USB monitoring to detect a keylogger. We anticipate that the commercial realization of our model would likely include implementation of the monitoring system inside of a desktop / laptop computer or a docking station. This way, a USB keylogger could not be placed to circumvent the monitoring circuitry. A complete commercial implementation would typically involve custom ASIC (Application Specific Integrated Circuit) circuitry, integrated into the USB hardware system on the motherboard. We consider the proof of concept to have been the critical element of this MQP, and we don't offer detailed ASIC design elements.

4 Observations

4.1 Digital Multimeter Empirical Methodology

We needed a way to test our hypothesis that there is a noticeable difference in power draw when a keylogger is in series with a USB keyboard.

In order to obtain the power draw of the keyboard / keylogger, we needed to create a small breakout circuit so that we could tap the wires going through the USB. A USB has four channels running through it: power (5 V), ground, source-clock, and source-data. For this portion of the experiment, we will only need to know about the power and ground wires. Our goal was to use the following equation (see Equation 1) to calculate the power drawn by the keyboard.

$$P = I * V \tag{1}$$

Looking at Equation 1, we can see that there are two measurements we need to calculate power (P). We need to find I, the current going through the USB port to the keyboard, and V, the voltage being delivered to the keyboard. Finding V is straightforward, as we can just measure the voltage difference between the power and ground wires of the USB breakout circuit. Finding I is a little more complicated, but we can use Equation 2 to calculate I.

$$I = V/R \tag{2}$$

Looking at Equation 2, to calculate the current, I, we can see that there are two more intermediate measurements we need. These measurements represent a resistor being put in series with the power line of the USB breakout. We will use the value of the resistor in ohms as the denominator (R), and the voltage drop across the resistor as the numerator (V). Finding the voltage drop across the resistor just means we need a measure of the voltage before and after the resistor, and we will subtract these values. The circuit diagram of this setup can be seen in Figure 6.

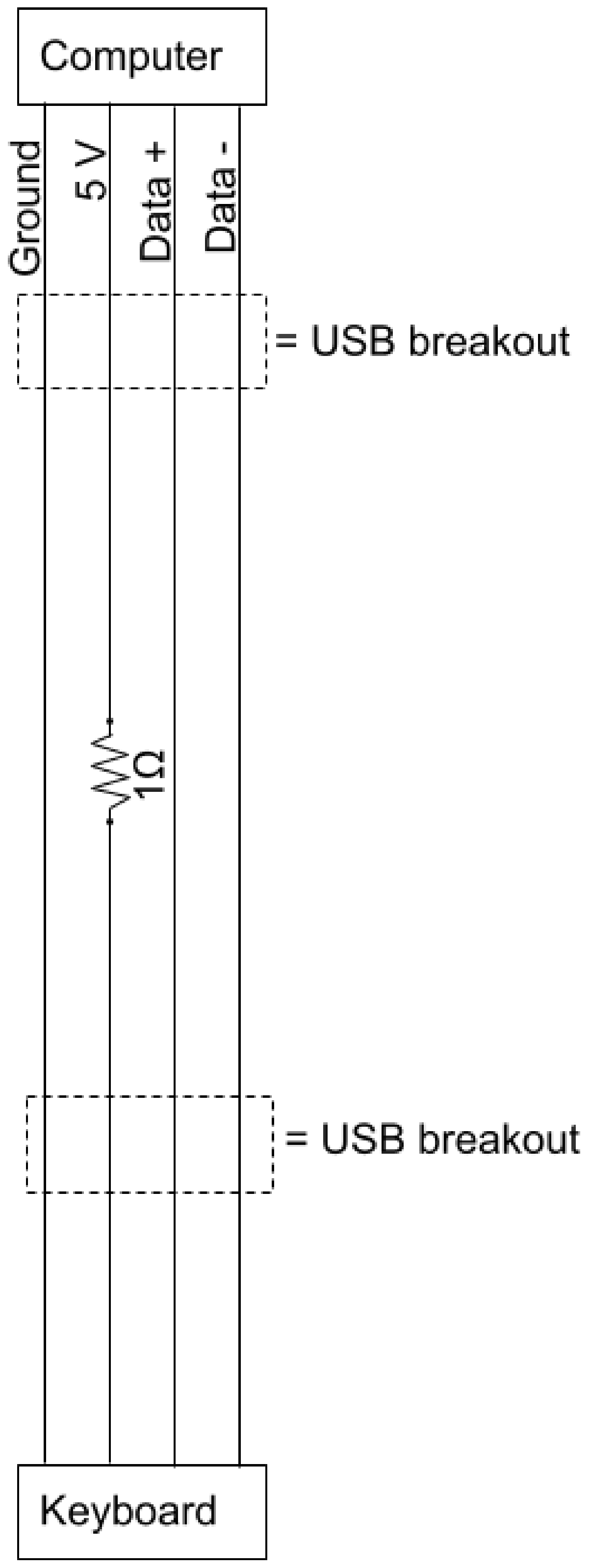


Figure 6: Circuit used to expose wires for digital multimeter.

To test our hypothesis, we went to the ECE lab in Atwater Kent to use a digital multimeter to gather the aforementioned voltage measurements we needed to calculate power draw. We built the circuit, from the circuit diagram in Figure 6, on a breadboard. The circuit can be seen in Figure 7 with the digital multimeter set up to measure the voltage drawn by the keyboard. To do this, we placed the positive probe of the Digital Multimeter on the 5V power line (close to the computer USB breakout), and placed the negative probe of the Digital Multimeter on the ground line. The circuit can be seen in Figure 8 with the digital multimeter set up to measure the current drawn by the keyboard (voltage drop across a 1 ohm resistor). To do this, we placed the positive probe of the Digital Multimeter on the 5V power line (close to the computer USB breakout, before the 1 ohm resistor), and placed the negative probe of the Digital Multimeter on the power line (close to the keyboard USB breakout, after the 1 ohm resistor).

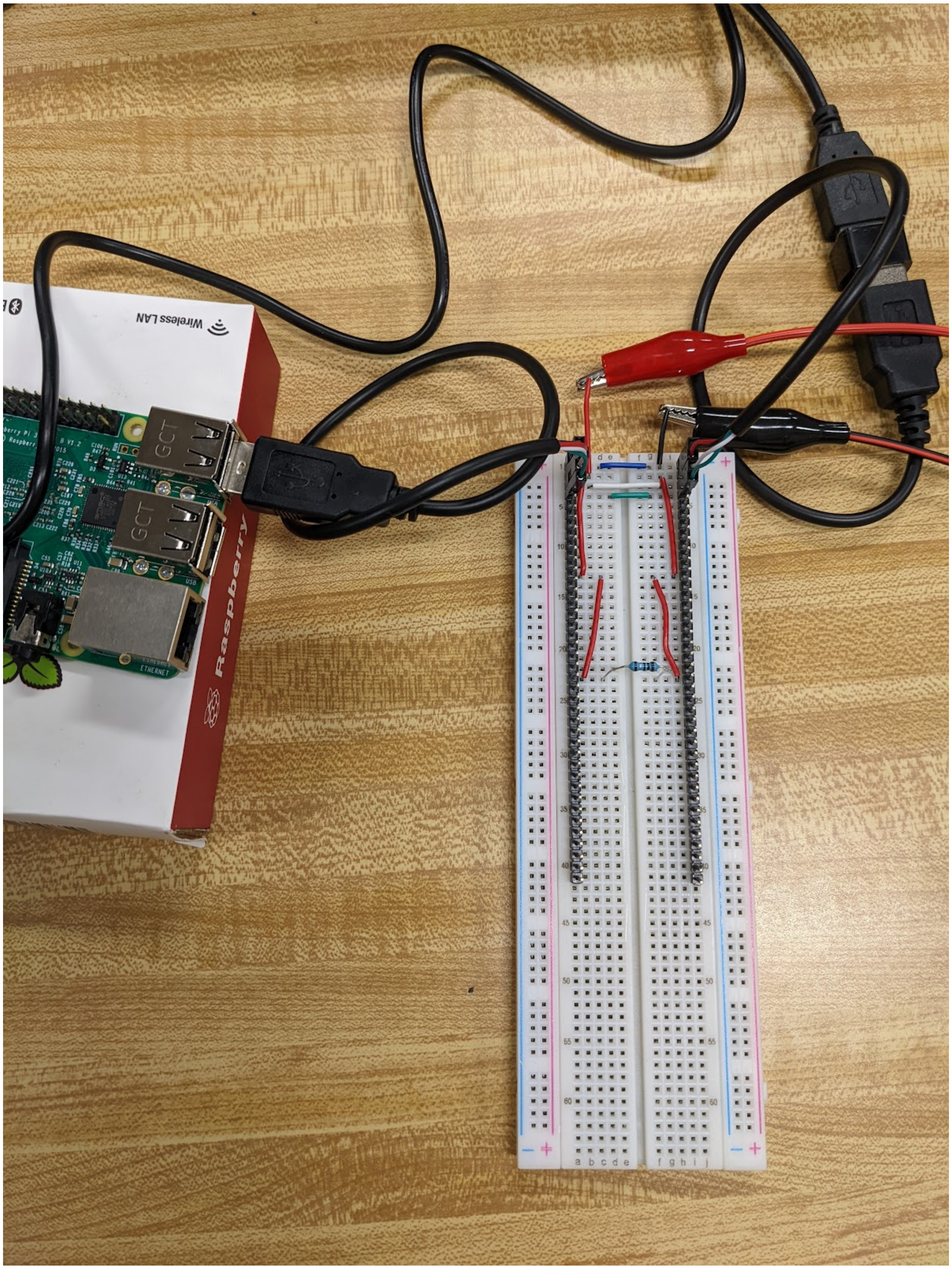


Figure 7: Digital Multimeter setup measuring voltage drawn by the keyboard (with a keylogger in place).

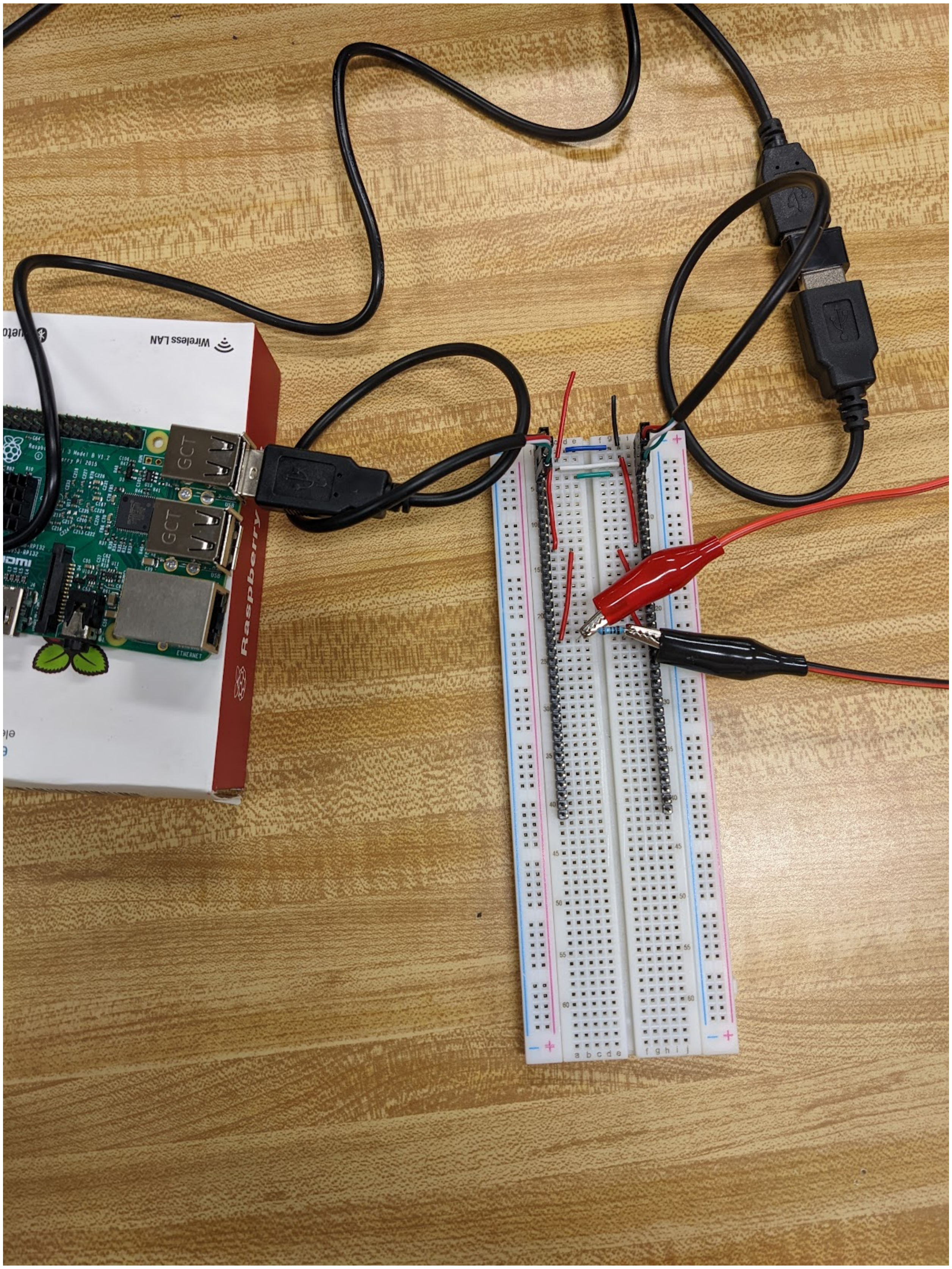


Figure 8: Digital Multimeter setup measuring current drawn by the keyboard (with a keylogger in place).

In the lab, we performed tests on multiple DELL 04G481 Keyboards (See Figures 9 and 10)

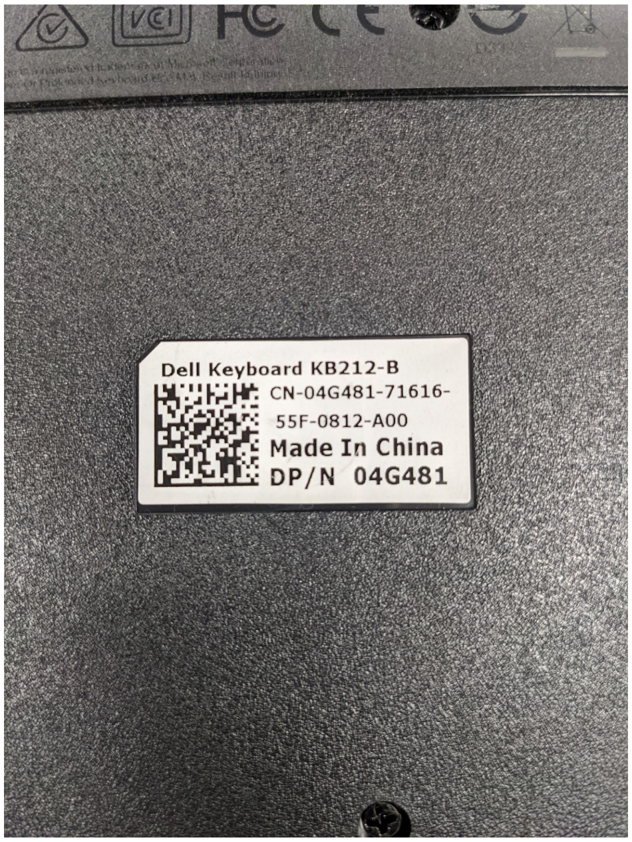


Figure 9: Back of Lab Keyboard (DELL 04G481).

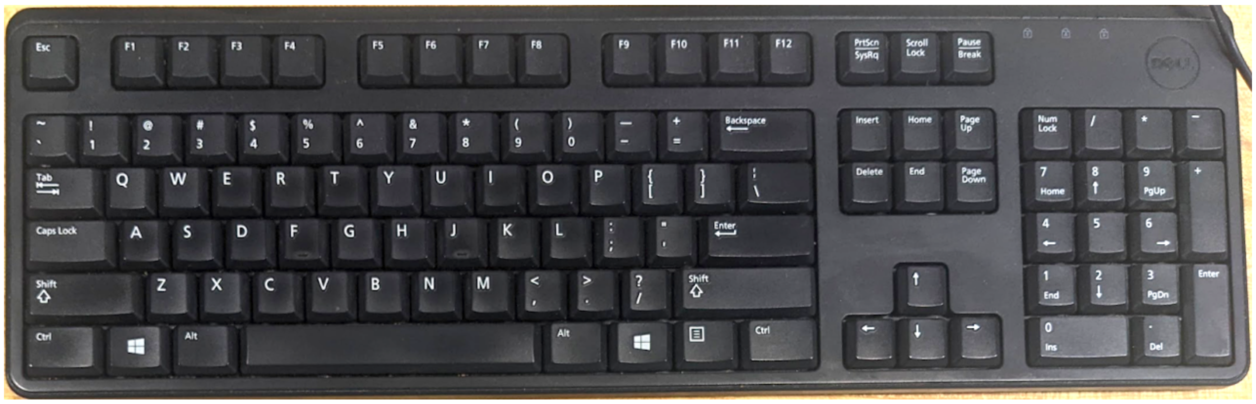


Figure 10: Front of Lab Keyboard (DELL 04G481).

4.2 Digital Multimeter Data Analysis

With our circuit fully connected, we tried pressing different keys on the lab keyboard to see what caused a difference in the current drawn (See Table 1). We chose to focus on current drawn rather than power drawn for this small test since the USB will try to output a steady 5V, and it does that by allowing the current to fluctuate more (i.e. the current draw is more sensitive to change).

	Noticeable Difference in Current Draw Seen
Pressing caps lock (turning on and off LED)	Yes
Pressing num lock (turning on and off LED)	Yes
Pressing scroll lock (turning on and off LED)	Yes
Pressing any non-LED key once	No
Pressing any non-LED key in quick succession	No
Pressing and holding any non-LED key	No
Holding shift while pressing any non-LED key	No

Table 1: Tests of pressing different keyboard keys to observe differences in current drawn.

As can be seen in Table 1, pressing keys that turned on LEDs was the only cause of any noticeable difference in the current drawn. However, we noticed that the difference did vary depending on the brightness of the LED being turned on. Pictures of a few lab keyboards with all LEDs turned on (caps lock, num lock, and scroll lock) can be seen in Figures 11 - 13.

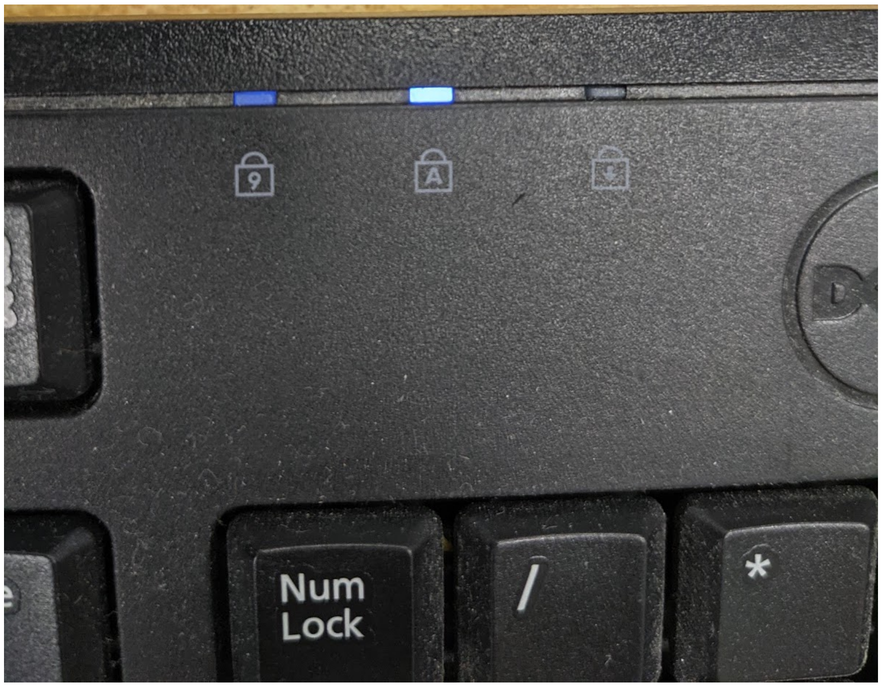


Figure 11: Keyboard from Lab Bench #23 has the scroll lock LED completely burnt out, and a dim num lock LED.

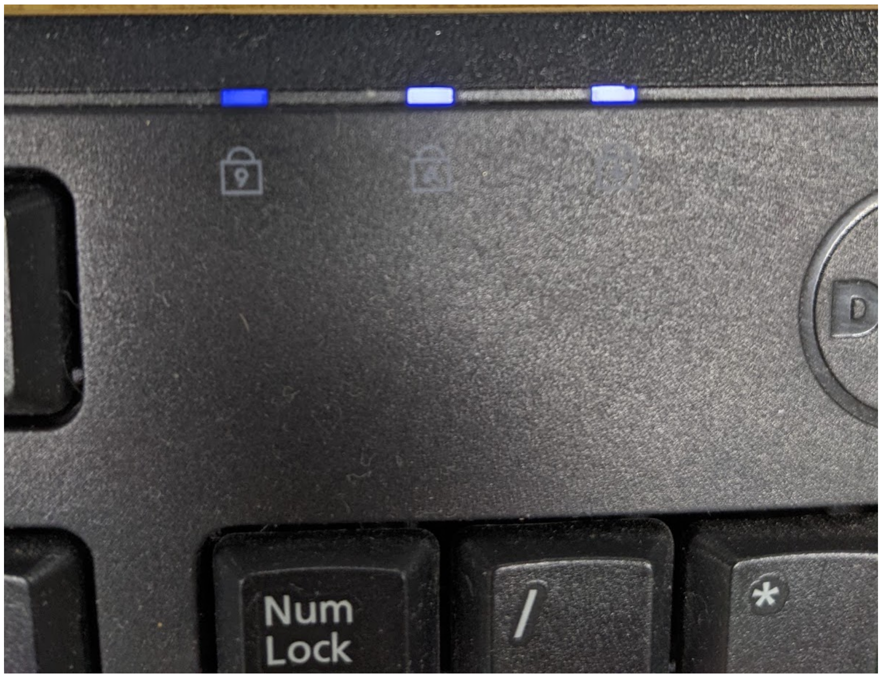


Figure 12: Keyboard from Lab Bench #22 has num lock (on the far left) noticeably dimmer than the other two LEDs.

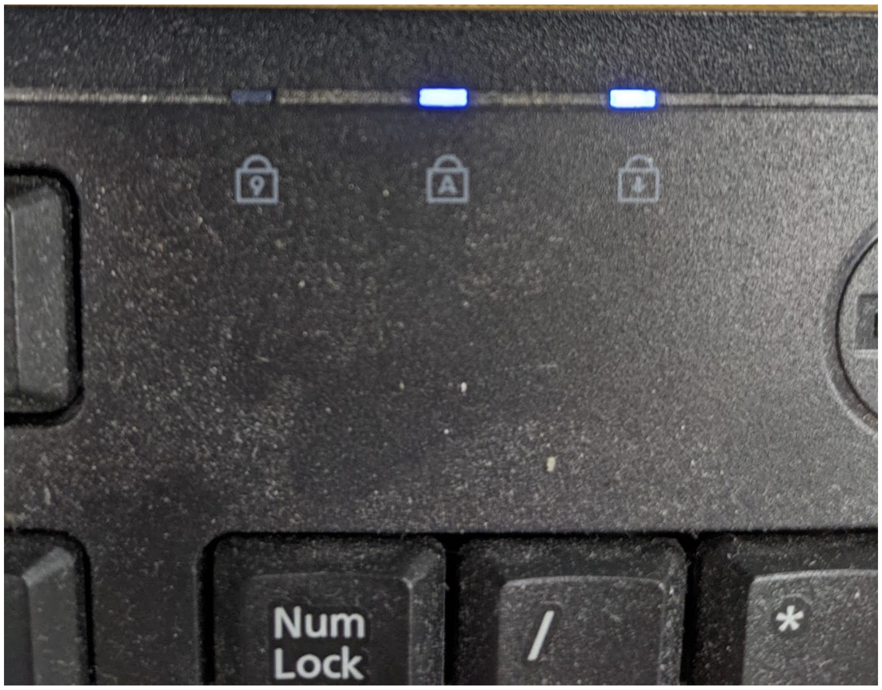


Figure 13: Keyboard from Lab Bench #24 has the num lock LED completely burnt out.

We next ran tests in this setup to measure the voltage and current draws with and without the keylogger when caps lock is on and when caps lock is off (See Tables 2, 3, and 4). We specifically chose to test the keyboard with the brightest caps lock LED (from lab bench #23, see 11), since we wanted to see how much of a difference a fresh / not burnt out LED would make in the power draw.

Without Caps Lock					
	Power Drawn	Voltage Drawn	Current Drawn	Voltage drop across resistor	(1 ohm) Resistor strength
Without Keylogger	10 mW	5.3 V	1.9 mA	4.5 mV	2.4 ohms
With Pico Keylogger	58 mW	5.3 V	11 mA	27 mV	2.4 ohms
With Forensic Keylogger	39 mW	5.2 V	7.5 mA	18 mV	2.4 ohms

Table 2: Caps lock off measurements with and without keylogger.

With Caps Lock					
	Power Drawn	Voltage Drawn	Current Drawn	Voltage drop across resistor	(1 ohm) Resistor strength
Without Keylogger	19 mW	5.3 V	3.6 mA	8.7 mV	2.4 ohms
With Pico Keylogger	68 mW	5.2 V	13 mA	31 mV	2.4 ohms
With Forensic Keylogger	49 mW	5.3 V	9.2 mA	22 mV	2.4 ohms

Table 3: Caps lock on measurements with and without keylogger.

	Without Caps Lock	With Caps Lock
Without Keylogger	10 mW	19 mW
With Pico Keylogger	58 mW	68 mW
With Forensic Keylogger	39 mW	49 mW

Table 4: Summary of Power draw with and without caps lock and/or keylogger.

As Seen in Table 4, we observed that LEDs on a keyboard increase the average power utilization levels. A LED of similar size to one on a caps lock key adds approximately 10 mW per LED to the keyboard voltage draw. This means that with enough LEDs (caps lock, scroll lock, num lock, etc.) a keyboard can gain an additional 30 mW of voltage draw due to human usage rather than keylogger insertion. That means that a keyboard with no keylogger inserted, but with caps lock, num lock, and scroll lock on, could draw as much power as 40 mW. However, as can be seen in table 4, a keyboard with no LED keys on, and the forensic keylogger plugged in, would only draw 39 mW of power. This cross over means that we cannot define a clear threshold saying, “if your average power exceeds X, then there must be a keylogger,” without knowing more information about the state of the keyboard (whether those LEDs were on).

Since our goal in this MQP is to create a non-invasive method of detection, we did not want to rely on asking the keyboard for the current state of certain keys or keeping a record of such key presses. Therefore a simple power threshold, as mentioned previously, will not be a workable solution.

The measurements we were taking with the digital multimeter were not showing us a record of the changes in power for small time increments. To combat this, we tried using an oscilloscope in the lab to visualize any such changes, by setting a positive edge trigger. However there was too much noise and not enough precision to make sense of the images.

We decided that in order to have a better insight on the power draw, we should look at power draw distributions collected by an analog to digital converter (ADC). To do this, we would need to build our own circuit and write code to gather information from our ADC Circuit.

5 Design of Circuit

To get a closer look at power distributions, with more precision of measurements, we needed to make a custom circuit with an ADC (analog to digital converter). The circuit diagram for this custom circuit can be seen in Figure 14.

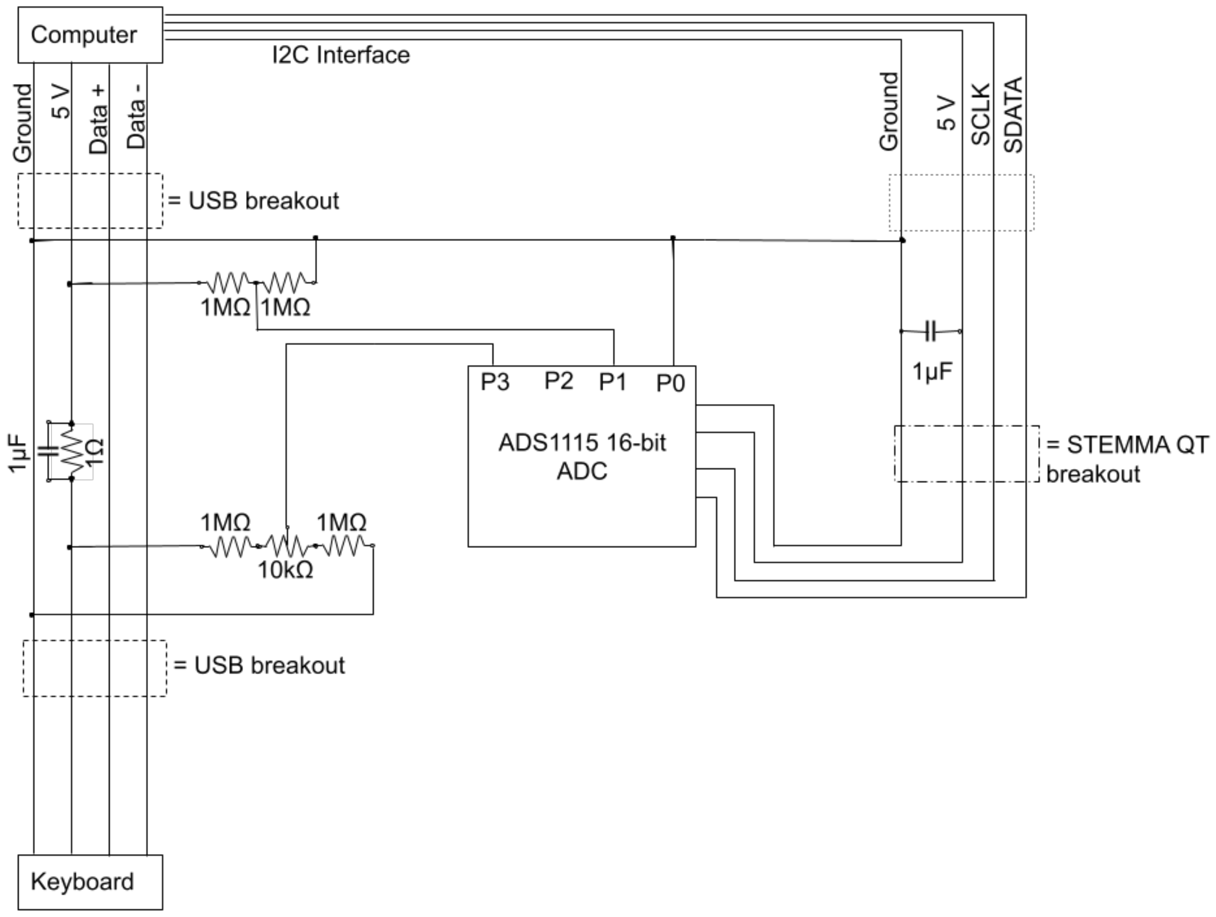


Figure 14: Custom circuit diagram for collecting ADC measurements.

5.1 Materials to Build Setup

- Raspberry Pi 3 Model B (labeled as computer in circuit diagram)
- USB keyboard
- Breadboard
- Male USB breakout

- Female USB breakout
- A 10K potentiometer
- 5 Resistors
  - A single 1 Ω resistor
  - Four 1 MΩ resistors
- 2 Capacitors
  - Two 1 μF capacitors
- ADS1115 16-Bit ADC with 4 Channels
  - Capable of taking up to 128 samples per second (Sps)
- Stemma Qt breakout
- 4 Male to female wires

5.2 Circuit Rationale

Our goal was to use equation 1 to calculate the power drawn by the keyboard. Further explanation of how we can get the current (I) and voltage (V) measurements can be found in section 6. However, rather than having the digital multimeter take measurements, we feed the measuring points into our analog to digital converter. The Analog to digital converter has 4 channels, however, we will only use three of them. Our ground reference will go in channel 0 (P0 in Figure 14). Our voltage draw reference will go in channel 1 (P1 in Figure 14). We note that we need to sample the voltage drawn, rather than just assuming it outputs a constant 5V, since there will be variance and changes in this voltage output as it tries to correct itself to always stay at 5V. Our current draw reference (far side of the resistor) will go in channel 3 (P3 in Figure 14). We chose these specific channels since our chosen ADC was able to perform subtractions between these (Channel 1 - Channel 0 to get the voltage draw (V), and Channel 1 - Channel 3 to get the current draw (I) (since we are using a 1 ohm resistor)). Not all channels on the ADC are wired to do these high precision subtractions.

$$P = I * V \tag{1}$$

We tried to design our circuit to be taking measurements and power from the same source. This is why the ADC has power provided to it via an I2C interface connected to the computer, or in our case, the Raspberry Pi 3. We also assume for our set up that if a keylogger is plugged into this circuit, it is plugged in on the keyboard side of the circuit, not the computer side (see Figure 14).

The voltage that is supplied to power the ADC will act as the ceiling power that our ADC can detect. We include a bypass capacitor in between the power and ground supplied to the ADC to ensure voltage stability. Since we are supplying approximately 5V from the computer to the ADC, our ADC will not be able to detect above 5V. Since the voltage measurements that the ADC is sampling are near 5V (taken around the 1 ohm resistor before the keyboard), we will use voltage dividers to half these voltage values. This way we can assure full accuracy of measurements without hitting the 5V ceiling.

The top voltage divider, connecting to P1, is a normal voltage divider consisting of two 1 M ohm resistors. However, in the bottom voltage divider, we placed a potentiometer of 10 k ohms between the two 1 M ohm resistors. The potentiometer is there to account for the fact that the resistors we were using were not all exactly equal to 1 M ohm in strength. All resistors have some error. However, since we are taking two measurements relatively close to each other (to get the current), we need to have high precision and equal treatment of the two voltage measurements through their respective voltage dividers. Essentially we need the ratio of resistance before and after the voltage sample (taken at the midpoint of the voltage divider) to be the same. The potentiometer allows us to tune, by hand, the resistance of the second voltage divider to match the first one. Once our circuit is set up, and we can take samples from the ADC, we will adjust the potentiometer as follows.

1. Connect the computer side USB breakout to the computer, but leave the keyboard side USB breakout disconnected from any keyboard or keylogger.
2. Run a script that prints out the measurements from the ADC, and adjust the potentiometer so that the voltage drop across the 1 ohm resistor (P3 - P1) is as close to being centered at zero as possible.
3. The potentiometer can be adjusted using a screwdriver for more precision.

The 2 USB breakouts connect the four USB channels (Ground, Power, Data+, and Data-) through our breadboard from the computer to the keyboard. In our case, our computer (Raspberry Pi 3) had a USB protocol that would always supply 5V over the power line to the keyboard, which is why we have labeled the power line 5V. We also made sure to connect the USB ground to all other ground references so we avoid creating a ground loop.

To monitor current we added a 1 ohm resistor in line with the 5V (power) line in between the USB breakouts. We used Equation 2 to calculate the current. However, we specifically chose to use a 1 ohm resistor (measuring 1.0 ohms), which means R = 1 in Equation 2. This means that Equation 2 simplifies to I = V. For further explanation of how taking these measurements work, refer to section 4.1.

$$I = V/R \tag{2}$$

To smooth out this current measurement, we also added a capacitor in parallel to the 1 ohm resistor. This helps to smooth out any sharp spikes in voltage that may occur between sampling measurements.

5.3 Circuit Cost Breakdown

This circuit is a relatively inexpensive set up for detection.

5.3.1 Proof of Concept Circuit Cost

The bulk of the cost for our circuit is in the analog to digital converter (ADC). The other elements are relatively inconsequential. Below is the cost analysis of elements needed for the prototype circuit.

- 1 16-bit ADC ≈ \$14.95 per ADC on Adafruit
  - https://www.adafruit.com/product/1085
  - (ADS1115 16-Bit ADC - 4 Channel With Programmable Gain Amplifier, n.d.)
- 2 USB breakouts:
  - 1 Male breakout ≈ \$1.95 per Male breakout
    - https://www.adafruit.com/product/4448
    - (USB Type A Plug Breakout Cable With Premium Female Jumpers, n.d.)
  - 1 Female breakout ≈ \$1.95 per Female breakout
    - https://www.adafruit.com/product/4449
    - (USB Type A Jack Breakout Cable With Premium Female Jumpers, n.d.)
- 1 10kΩ ≈ \$0.80 per potentiometer
  - https://www.amazon.com/MCIGICM-Breadboard-Trim-Potentiometer-Arduino/dp/B07S69443J

    - (MCIGICM (10 Pcs) 10K Ohm Breadboard Trim Potentiometer Kit With Knob for Arduino, n.d.)

- 2 1μF capacitors $\approx$ \$0.07 per capacitor
- 5 Resistors:
    - 4 1MΩ resistors $\approx$ \$0.02 per resistor
    - 1 1Ω resistor $\approx$ \$0.06 per resistor

5.3.2 Circuit on Chip Estimated Cost

As previously noted, the most expensive part of the circuit is the ADC. Looking closer at our ADC board we bought from Adafruit, we noticed that the ADC chip used was the Texas Instruments ADS1115. We have found the ADS1115 Texas instrument chip for as little as \$1.00 in bulk (Ads1115 Price and Stock, n.d.). This means that the cost of the production full circuit on chip may be as low, if not lower than, \$2.00.

6 Modifications to KL-divergence algorithm

6.1 Methods of Analyzing Histograms

We aggregated a time-series of power utilizations into histograms, and used that to define both the signature of a keyboard (personal DELL 0XD31W keyboard, as well as Lab DELL 04G481), and the signature of a key-logger plus keyboard. We noticed that the histograms had visually different shapes depending on caps lock (see Figure 17) vs no caps lock (see Figures 15 and 16).

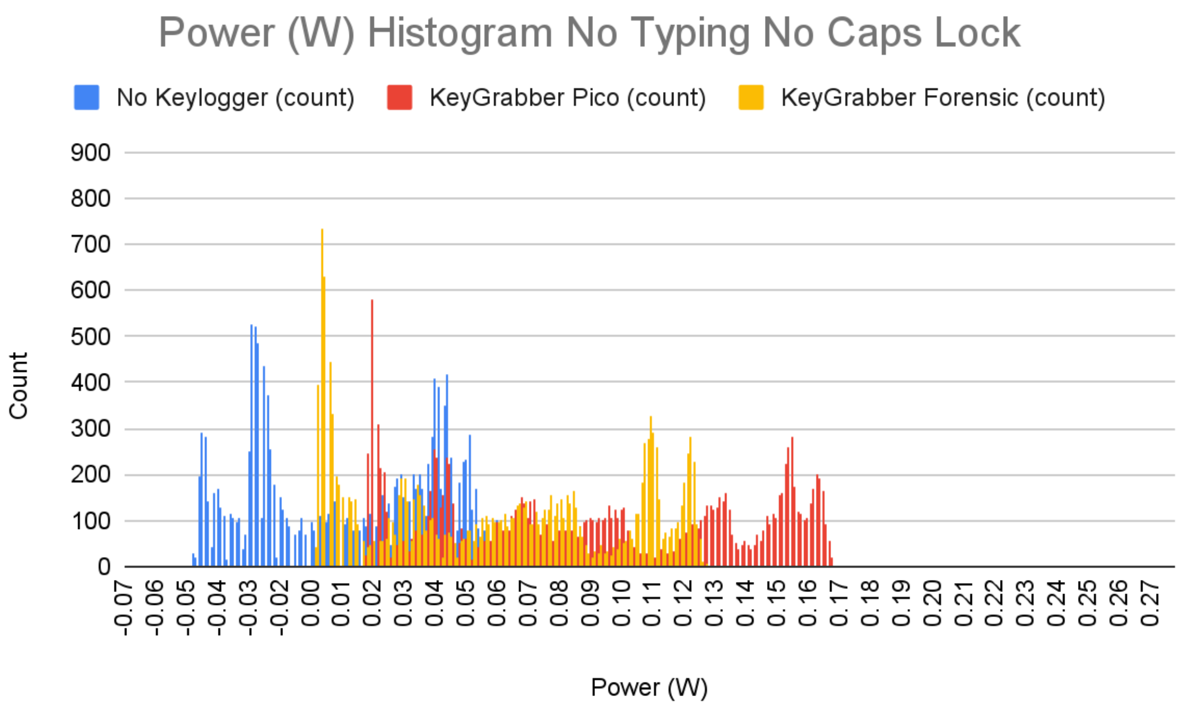


Figure 15: Power (W) histogram over 5 minutes without typing or caps lock (DELL 0XD31W keyboard).

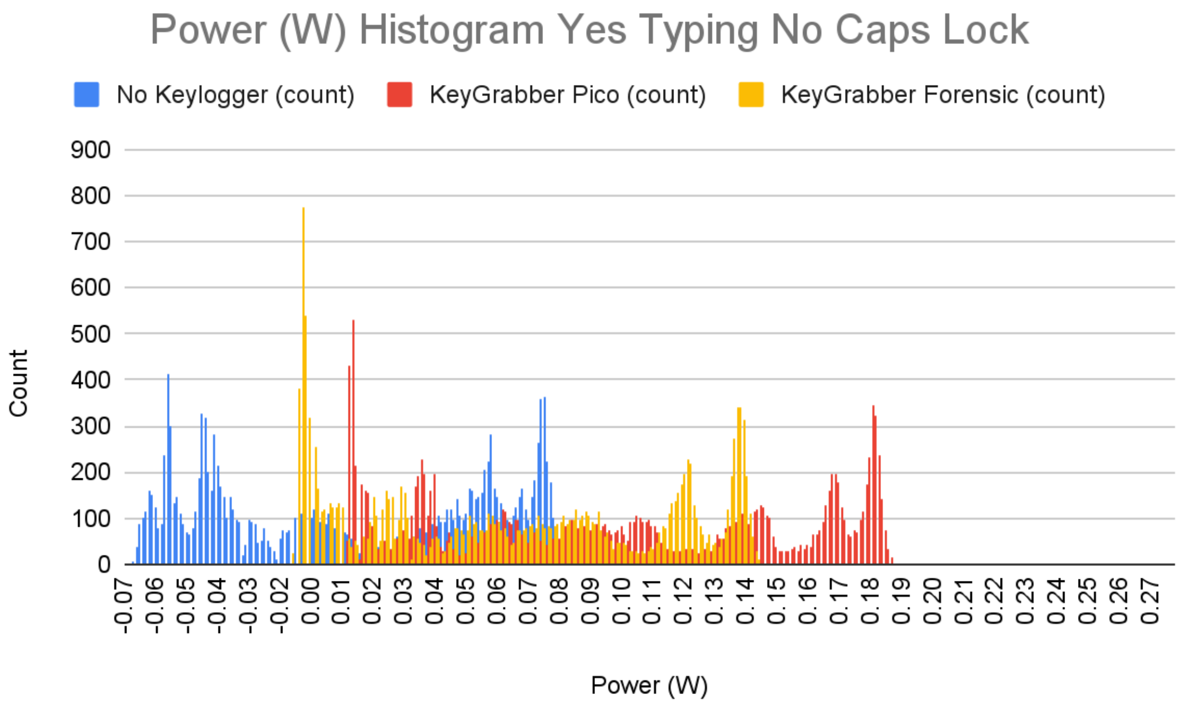


Figure 16: Power (W) histogram over 5 minutes with typing but without caps lock (DELL 0XD31W keyboard).

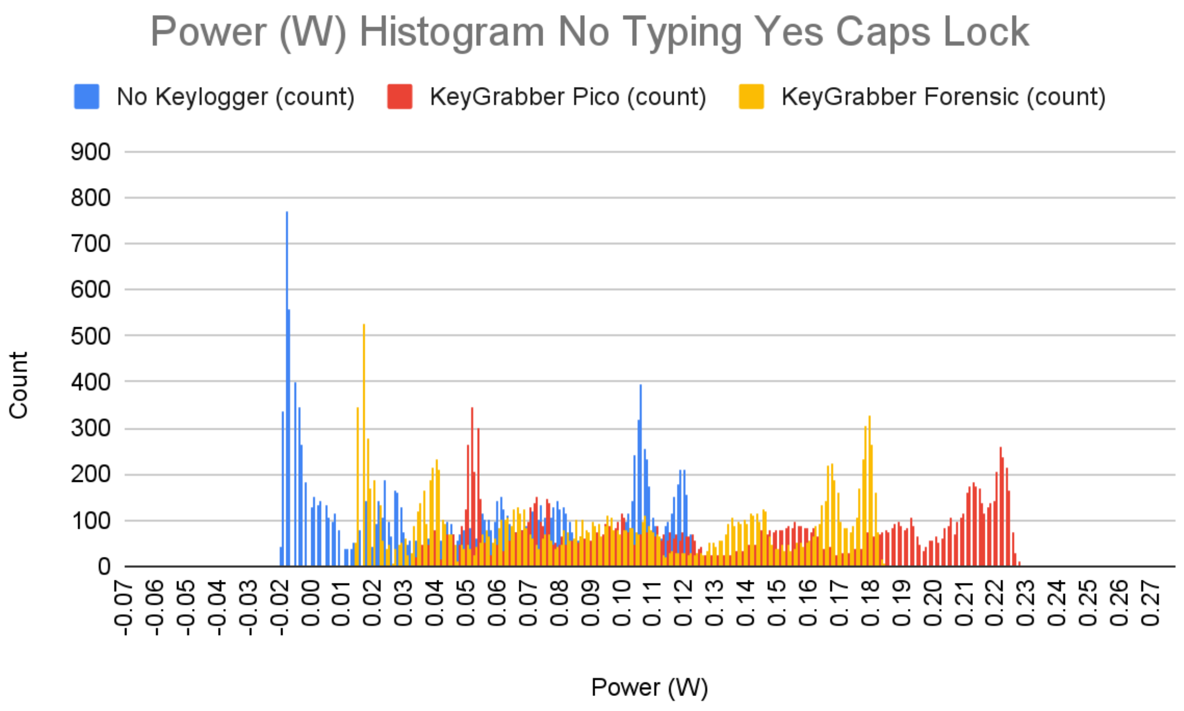


Figure 17: Power (W) histogram over 5 minutes with caps lock but without typing (DELL 0XD31W keyboard).

When comparing figures 15 and 16 (no typing vs typing, both without caps lock), we can see a small difference in the histograms. When looking closely, each of the histograms in figure 16 (blue, yellow, and red), are spread out more than they are in figure 15. However, the main shape, number of modes and spacing of modes, is still very similar. In our methods of analyzing histograms, simply typing as compared to a baseline of no typing always scores as being recognized as a similar histogram. Therefore, we will focus our explanation in this section on comparing the no typing no caps lock histogram (Figure 15) with the no typing yes capslock histogram (Figure 17), as they are the most visually different.

Therefore, we were looking for a method to confirm that the blue (no keylogger) histogram in Figure 17 (with caps lock) was more similar to the blue (no keylogger) histogram in Figure 15 (without caps lock), than any of the other yellow or red (keylogger) histograms. The KL-divergence (Kullback–Leibler divergence) algorithm came up during literature review as being a good option for comparing histograms since it does not make any assumptions about the distribution of the data being analyzed. Most other methods did make assumptions about the distribution being normal or Gaussian, which was not the case with our data.

6.2 Original KL-divergence algorithm

6.2.1 What is KL-divergence

KL-divergence, otherwise known as Kullback–Leibler divergence, is a statistical distance for measuring how different two probability distributions are from each other. KL-divergence is, most simply put, a measure of relative entropy. Entropy is a measure of disorder or randomness. Therefore, KL-divergence asks, "If we use one distribution as a model for the other, how much randomness do we experience?" The general equation for KL-divergence can be seen in equation 3 (MacKay, 2003).

$$D_{KL}(P||Q) = \sum_{x} P(x) * \log(\frac{P(x)}{Q(x)}) \tag{3}$$

6.2.2 Properties of KL-divergence

Most importantly, KL-divergence does not make any assumptions about the shape of these probability distributions (normal / Gaussian distribution, etc.).

However, by observing equation 3, we can see that $D_{KL}(P||Q) \neq D_{KL}(Q||P)$ (MacKay, 2003). This means that the magnitude of the KL-score of two distributions is not reflexive. This inequality is due to two reasons:

1. The KL divergence sum uses P(x) as a multiplier for $D_{KL}(P||Q)$ and Q(x) as multiplier for $D_{KL}(Q||P)$.
2. $\log(\frac{P(x)}{Q(x)}) = -\log(\frac{P(x)}{Q(x)})$

Therefore, when we apply the KL-algorithm to our data, we will use the same P, as our reference distribution, and change out the Q for different distributions we want to compare to it.

There are two special cases for which the partial KL-score is interpreted separately from the sum in equation 3. If the denominator of the logarithm (Q(x)) or the numerator of the logarithm (P(x)) has a value of 0, that partial sum is evaluated as 0. This proved to be very relevant in our work, as we had numerous buckets with 0 samples, and this pattern of 0s was very significant in discriminating the sample histogram from the baseline.

6.2.3 Interpreting KL-divergence

Since KL-divergence is a measure of entropy, a low KL-divergence score means that the two distributions are very similar. On the other hand a large KL-divergence value means that the two probability distributions in question are very different.

6.3 Modification Specifics

In analyzing our data, we always normalize our dataset to fit with the minimum and maximum sitting at 0 and 1 respectively. The data is then placed into a discrete number of buckets between these two values, to be processed by our augmented KL-algorithm.

There are three main changes we made to the KL-divergence algorithm for our specific histogram analysis.

1. Add one to each bucket.
2. Take the absolute value of the logarithm.
3. Convolution is used to find the best fit KL-score.

Our first change is adding one to each bucket in our histogram. Our power-utilization histograms had numerous quantized buckets containing zero samples. The presence of these zeros provides significant information, and is critical to our histogram differentiation analysis. As was previously noted in section 6.2.2, if either of the corresponding buckets in the two histograms being compared is 0, the partial original KL-score sum for that bucket will be 0. This means that if one histogram has a 0 in that bucket and the other has a large value, the original KL-score will not reflect that there was any difference. Recall that a low KL-score means the two distributions in question are more similar. By adding one to each bucket, turning information-rich "zeros" into "ones," the KL-score is well defined for taking the log of the ratio. This means that our augmented KL-score is able to interpret the dissimilarity of two histograms for a bucket which was originally empty.

Our second change is to take the absolute value of the logarithm. The logarithm of the KL-score (see equation 3) takes the log of the ratio of P(x) / Q(x). It is important to note that the log of any positive fraction less than 1 is a negative value. This means that if P(x), the numerator, is ever less than Q(x), the denominator, the partial sum for the original KL-score will be negative. Since a lower KL-score means a higher similarity between distributions, the ability to subtract difference obscures the true similarity measure of the histograms. In our case, we want to determine if the histograms are as close to matching as possible. This means we want to penalize any type of difference, whether a distribution has less or more values in a bucket than the baseline distribution. By taking the absolute value of the logarithm in the original KL-score, we effectively penalize all differences. Our augmented KL-algorithm can be seen in equation 4. Note that we always have the same number of samples in the Baseline and Test datasets. The BaselineCount seen in equation 4 is the total number of recorded samples plus the number of fictional samples (1s added to each bucket).

$$D_{KL}(Baseline||Test) = \sum_{x} \frac{Baseline(x)}{BaselineCount} * \log(\frac{Baseline(x)}{Test(x)}) \tag{4}$$

Finally, we use convolution to get the best fit for our KL-score. Essentially if the modes of the two distributions are slightly misaligned due to noise on the far ends factoring into the normalization, we want to be able to align the shapes when comparing them. To convolve the two distributions, we essentially move the Q distribution across the P distribution, taking the KL-score at each step of the convolution (See Figures 19 - 20). We note that the time to perform these convolutions using bucket sizes of 0.1 takes approximately 0.3 seconds. It is also notable that while we are using bucket sizes of 0.1 for our trials, this could be implemented with even more discrete bucket sizes, in which case convolution is even more necessary (See Figure 21).

DELL 0XD31W keyboard

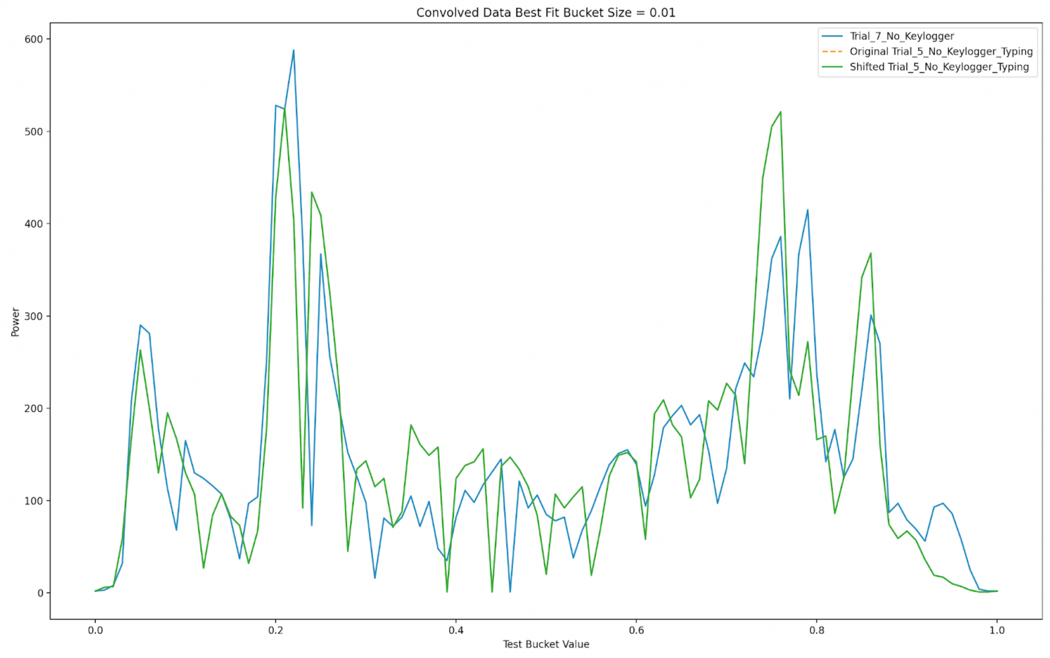
Trial_5_No_Keylogger_Typing v.s. Trial_7_No_Keylogger

Original score = 92.383

Shifted score = 92.383

Shifted right-starting index = 100

Total time to compute: 0.035528 s



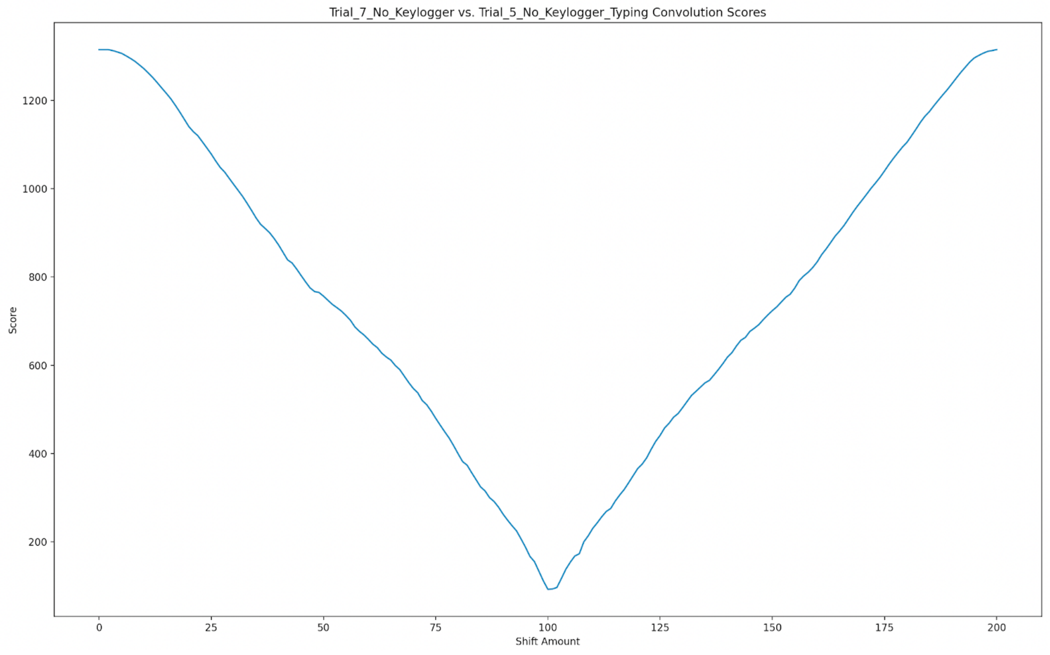


Figure 18: Convolution of No keylogger Q(x), with bucket size of 0.01, in this test, results in keeping the distribution centered as it was.

DELL 0XD31W keyboard

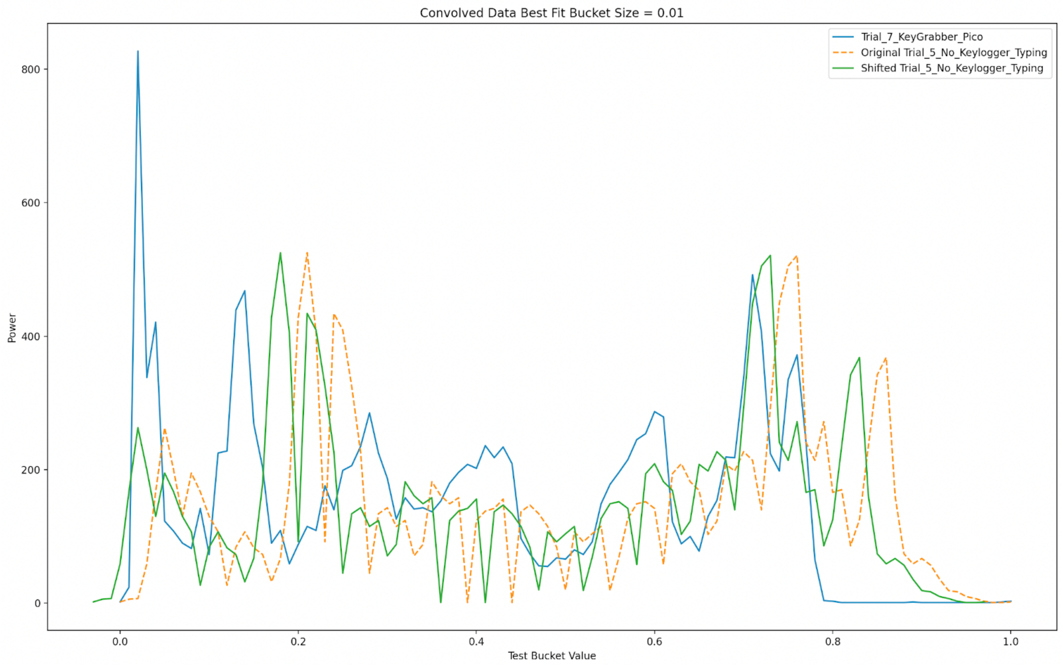
Trial_5_No_Keylogger_Typing v.s. Trial_7_No_Keylogger

Original score = 209.43

Shifted score = 185.24

Shifted right-starting index = 97

Total time to compute: 0.036254 s



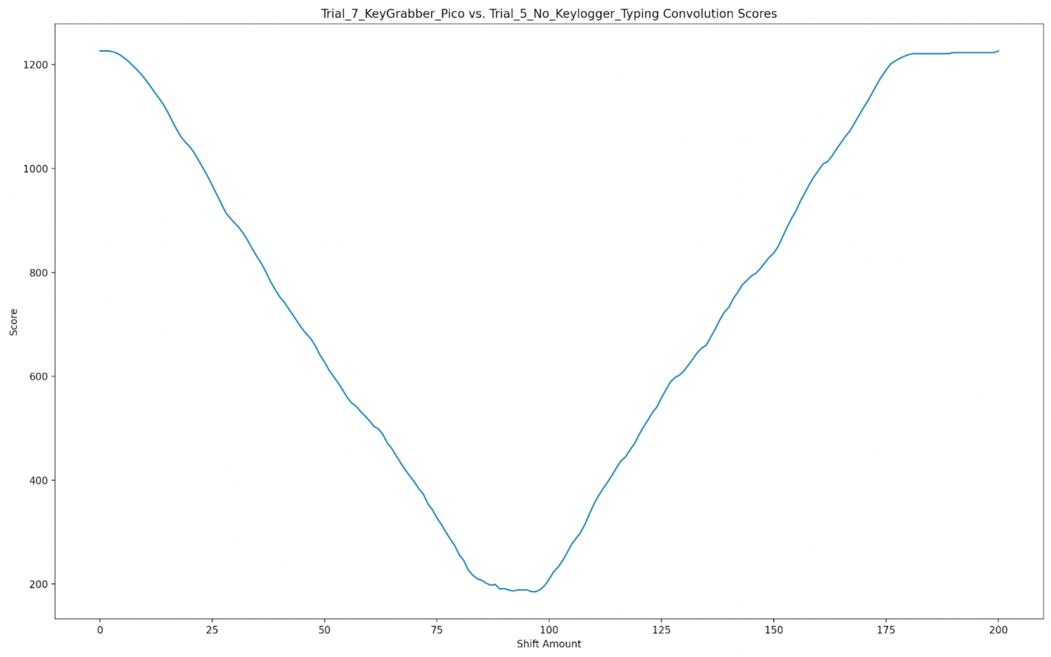


Figure 19: Convolution of Keygrabber Pico Q(x), with bucket size of 0.01, results in shifting the distribution to the left to minimize the KL-score.

DELL 0XD31W keyboard

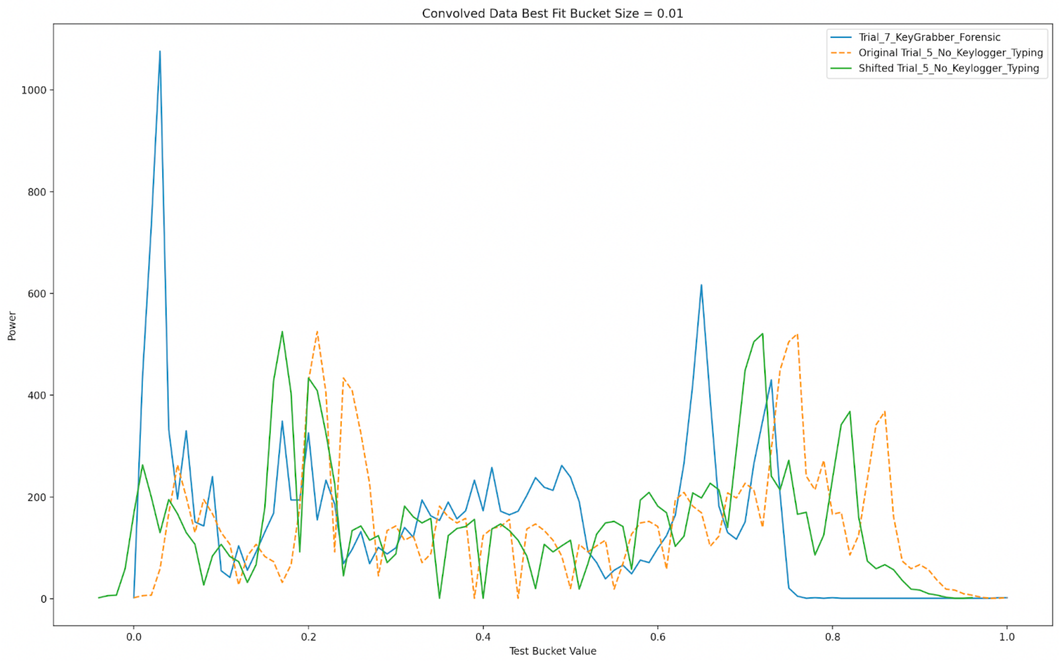
Trial_5_No_Keylogger_Typing v.s. Trial_7_No_Keylogger

Original score = 235.85

Shifted score = 206.04

Shifted right-starting index = 96

Total time to compute: 0.033694 s



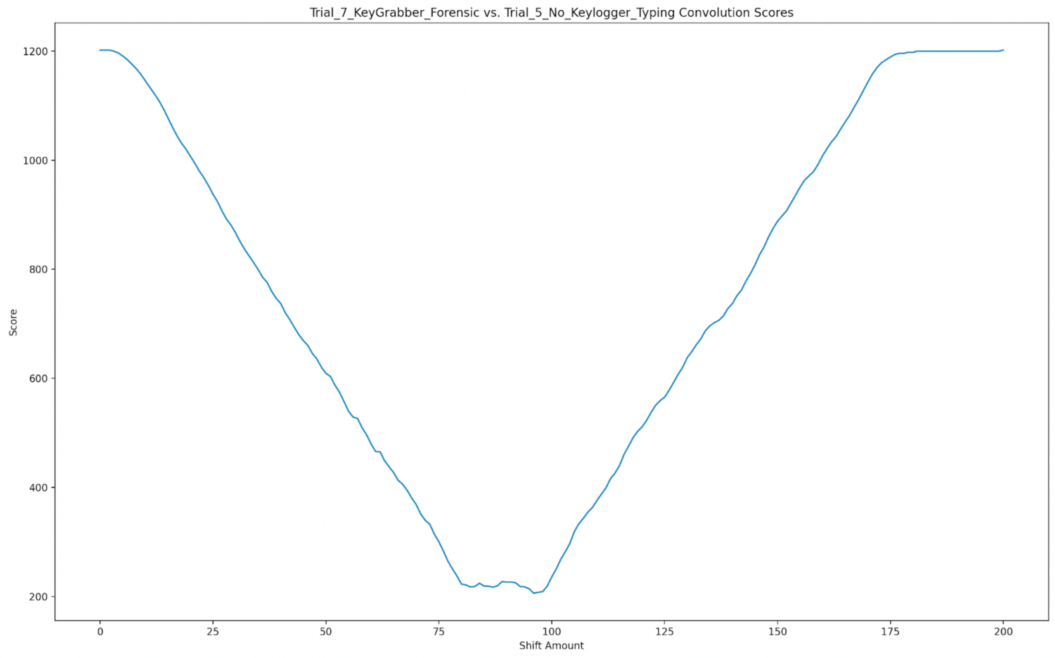


Figure 20: Convolution of Keygrabber Forensic Q(x), with bucket size of 0.01, results in shifting the distribution to the left to minimize the KL-score.

DELL 0XD31W keyboard

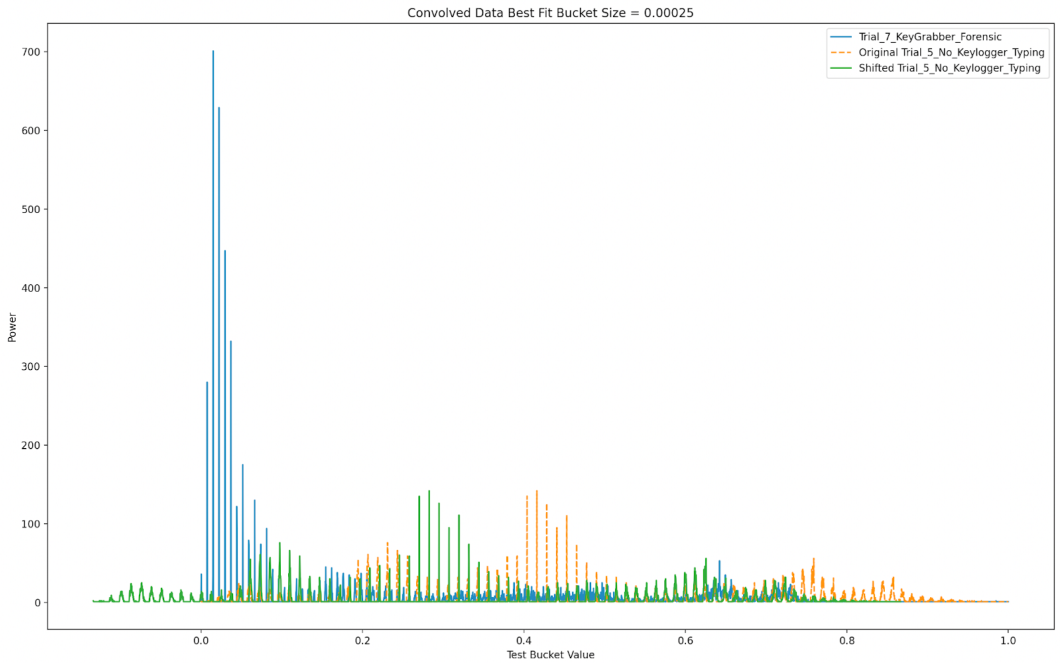
Trial_5_No_Keylogger_Typing v.s. Trial_7_No_Keylogger

Original score = 6500.2

Shifted score = 5757.1

Shifted right-starting index = 3466

Total time to compute: 46.573 s



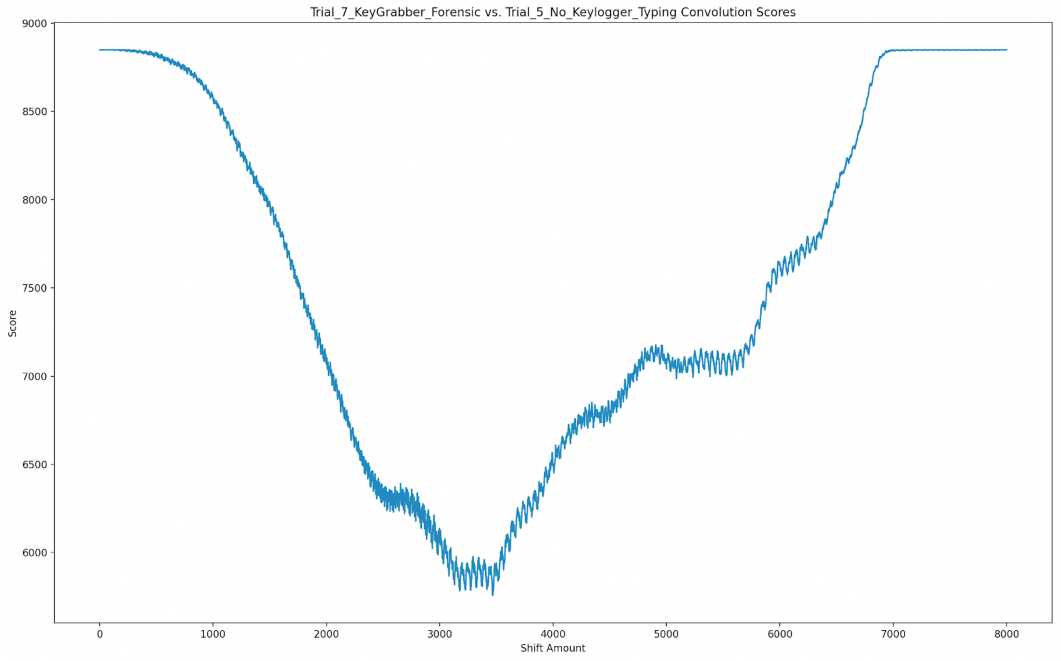


Figure 21: Convolution of Keygrabber Forensic Q(x), with bucket size of 0.00025, results in shifting the distribution to the left to minimize the KL-score.

7 Detection Results

7.1 Implementation

We implemented our augmented KL-divergence method, see equation 4, in Python. Once we were able to see the KL-scores calculated for different distributions (using DELL 0XD31W keyboard) with and without a keylogger in place, we set on tuning a threshold for keylogger detection. If the score is above our threshold, we would notify the user that there has been a keyboard change. This can mean that a new device was plugged in, or even a keylogger has been added to the keyboard. The Python code for implementation of the algorithm can be found in the appendix (primary Python file is detect_keylogger.py).

We needed to slightly tune our threshold KL-score for detecting a keylogger. Our method for calculating the threshold in this proof of concept was to take 3 five minute samples of the keyboard in its normal state (no keylogger inserted), but with caps lock on, which is when the distribution is most different from a no caps lock on scenario. We then took the maximum and minimum of the augmented KL-scores from these 3 trials and applied them in equation 5. Note that the threshold score is calculated by taking a percentage of the range of noKeyloggerScores and adding it to the maximum noKeyloggerScore we have seen. We tuned the denominator used, and found that 2 (using 50% of the range) resulted in the best threshold. Our goal in tuning was to have 0 false positives (for keylogger detected) and allowing minimal false negatives.

$$thresholdScore = \left(\frac{NoKeyloggerScoreMax - NoKeyloggerScoreMin}{2}\right) + noKeyloggerScoreMax \tag{5}$$

7.2 Efficacy of Detection

With our threshold in place we ran multiple tests without a keylogger, with the pico keylogger, and with the forensic keylogger, with and without caps lock. Thanks to our threshold we received no false positives (mistaken detection) (See Figures 22 and 23). With our tuned threshold we were able to detect the pico keylogger 100% of the time, regardless of whether the caps lock LED was on or off (See tables 24 and 25) (Hardware Keylogger - KeyGrabber Pico, n.d.). When capslock was turned off, our tuned threshold was able to detect the forensic keylogger 9 out of 10 times, with only 15,000 samples per decision, but it was even more detectable when caps lock was on (Hardware Keylogger - KeyGrabber Forensic Hardware Keylogger, n.d.). This means that while the forensic keylogger may have a very low KL-score for one five minute period, it will be detectable in the following five minute period. Put in other words, in a fifty minute period of detection, the forensic keylogger may be undetected for five minutes (See Figure 26). The threshold used in the below testing was 1.21707.

```
# (USING 15,000 samples) For No Keylogger NO Caps Lock:
# The augmented KL-score is: 0.5351992773825941
# The augmented KL-score is: 0.31641487489734677
# The augmented KL-score is: 0.25765348294024815
# The augmented KL-score is: 0.2962992215354933
```

Figure 22: No Keylogger No Caps Lock has no false positives (mistaken detection).

```
# (USING 15,000 samples) For No Keylogger YES Caps Lock:
# The augmented KL-score is: 1.138321700508211
# The augmented KL-score is: 0.9835992318366247
# The augmented KL-score is: 1.0780442542603117
# The augmented KL-score is: 0.9808159751397285
```

Figure 23: No Keylogger Yes Caps Lock has no false positives (mistaken detection).

```
# (USING 15,000 samples) For Keylogger Pico NO Caps Lock:
# The augmented KL-score is: 2.5202791557622737 ← KEYLOGGER DETECTED
# The augmented KL-score is: 2.1647753994249204 ← KEYLOGGER DETECTED
# The augmented KL-score is: 1.8265117643781354 ← KEYLOGGER DETECTED
# The augmented KL-score is: 2.5372695255955566 ← KEYLOGGER DETECTED
```

Figure 24: Keylogger Pico No Caps Lock is always correctly detected.

```
# (USING 15,000 samples) For Keylogger Pico YES Caps Lock:
# The augmented KL-score is: 1.721395640080059 ← KEYLOGGER DETECTED
# The augmented KL-score is: 2.185020219277311 ← KEYLOGGER DETECTED
# The augmented KL-score is: 1.504407197004179 ← KEYLOGGER DETECTED
# The augmented KL-score is: 2.146501677832364 ← KEYLOGGER DETECTED
```

Figure 25: Keylogger Pico Yes Caps Lock is always correctly detected.

```
# (USING 15,000 samples) For Keylogger Forensic NO Caps Lock:
# The augmented KL-score is: 2.6653660035459885 ← KEYLOGGER DETECTED
# The augmented KL-score is: 2.037338861949006 ← KEYLOGGER DETECTED
# The augmented KL-score is: 0.7569435724672036
# The augmented KL-score is: 1.9337000080055862 ← KEYLOGGER DETECTED
```

Figure 26: Keylogger Forensic No Caps Lock is detected within a 10 minute period (2 sampling sessions. The highlighted line represents a 5 minute period in which the keylogger was not detected.

As we can see in figure 26, the forensic keylogger occasionally dips below the threshold by a little bit, but the majority of the time, it is above the threshold. Therefore, while one 5 minute detection period did not detect the keylogger, the next and previous 5 minute detection periods did.

8 Conclusion

We were able to create a custom circuit to sample power draw from a USB keyboard, and used our custom augmented KL-algorithm to perform threshold detection of keyloggers. We also verified that it was strictly insufficient to merely monitor average power consumption, or median power consumption, as a simple thresholding mechanism. This is due to keyboard LED's, such as capslock, drawing considerable power, with multiple LEDs adding up to more power draw than a keylogger.

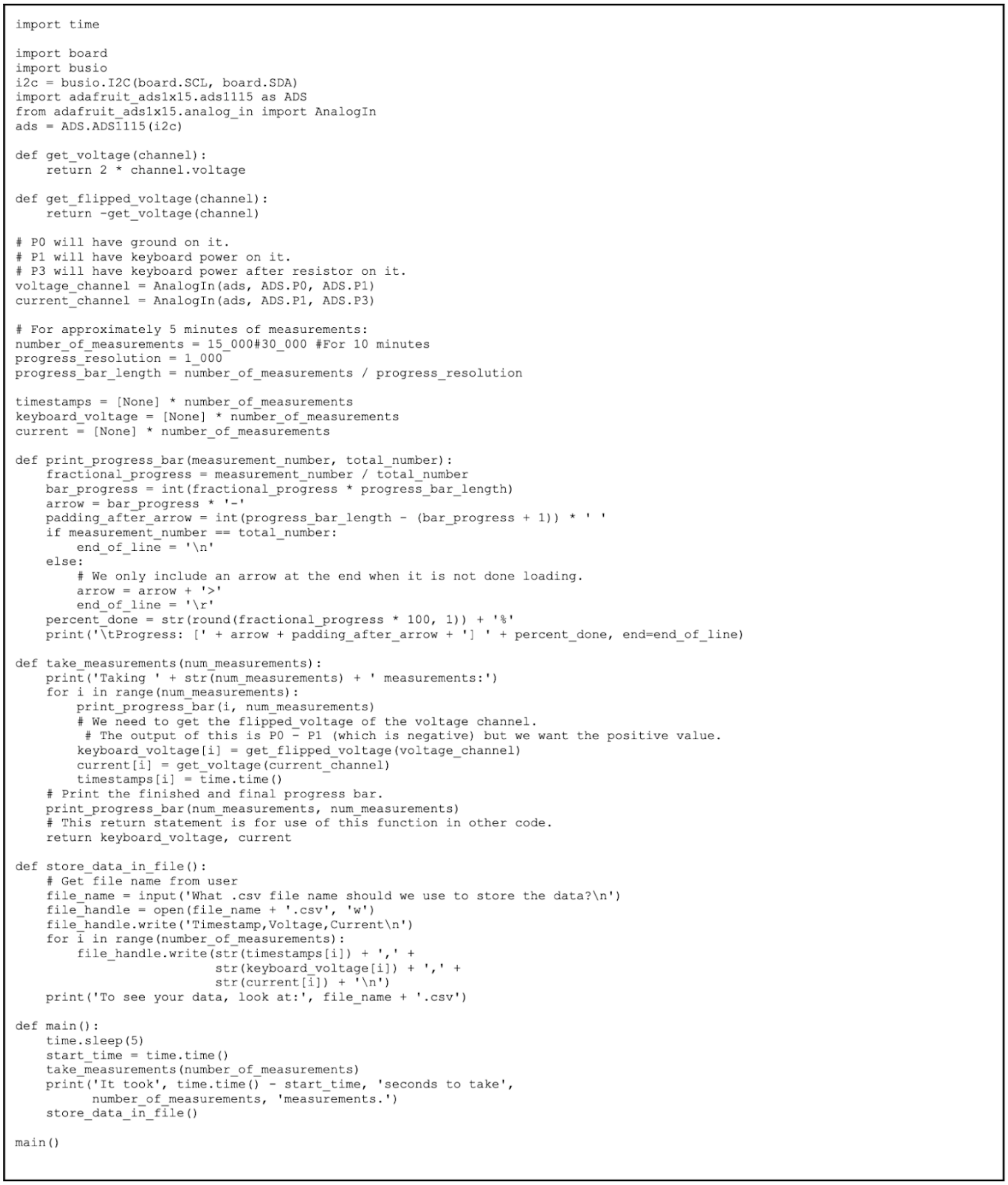
8.1 Future Study

Future work on this project includes testing on more keyboards, further validating the methods we have demonstrated. Once validation is complete, we'd suggest the implementation of our custom monitoring circuit and code as a small ASIC (Application-Specific Integrated Circuit) chip that could be built into a computer. Eventually we hope to see standard USB ASIC circuitry meticulously monitor power usage and implement the techniques discussed in this project. We hope to see such circuitry and associated software then integrated into operating systems such as LINUX.

9 Appendix

```
import time
import csv
import math

import board
import busio
i2c = busio.I2C(board.SCL, board.SDA)
import adafruit_ads1x15.ads1115 as ADS
from adafruit_ads1x15.analog_in import AnalogIn
ads = ADS.ADS1115(i2c)

from record_ADC_measurements import take_measurements

number_of_trials = 1
number_of_measurements = 15_000
number_of_buckets = 11

keyboard_voltage = [None] * number_of_measurements
current = [None] * number_of_measurements
power = [None] * number_of_measurements
normalized_power = [None] * number_of_measurements

def get_baseline_data(file_name):
    with open("NoKeyloggerNoCapslock2_new.csv", 'r', newline="") as baseline_file_handle:
        baseline_data_reader = csv.reader(baseline_file_handle, delimiter=",")
        headers = next(baseline_data_reader)
        if ("Timestamp" != headers[0]
            or "Voltage" != headers[1]
            or "Current" != headers[2]):
            raise Exception(
                "Baseline data file is not in correct format. "
                "Using 0-based indexing, please make sure "
                "the 0th header is 'Timestamp', "
                "the 1st header is 'Voltage', and "
                "the 2nd header is 'Current'.")
        # Don't include the header in the row count.
        baseline_row_count = sum(1 for row in baseline_data_reader)
        baseline_voltage = [None] * baseline_row_count
        baseline_current = [None] * baseline_row_count
        # Set reader back at first line of data.
        baseline_file_handle.seek(0)
        # Ignore the header:
        next(baseline_data_reader)
        lines_already_read = baseline_data_reader.line_num
        print("Reading baseline data:")
        for row in baseline_data_reader:
            #print(len(row))
            # Subtract 1 from the line number since we want 0-based indexing.
            line_number = baseline_data_reader.line_num - lines_already_read - 1
            #print(line_number, end=",")
            if 0 == line_number:
                print("\tFirst line:" + str(row))
            if baseline_row_count - 1 == line_number:
                print("\tLast line:" + str(row))
            # Assume voltage is 1st row (0-indexed)
            baseline_voltage[line_number] = float(row[1])
            # Assume current is 2nd row (0-indexed)
            baseline_current[line_number] = float(row[2])
        return baseline_voltage, baseline_current

def calculate_power(keyboard_voltage, current):
    for i in range(number_of_measurements):
        power[i] = current[i] * keyboard_voltage[i]
    return power

def normalize_measurements(power):
    # Get min
    min_power = min(power)
    # Get max
    max_power = max(power)
    power_range = max_power - min_power
    if 0 == power_range:
        print("TODO: ALL SAMPLES WERE THE SAME (FIX: ASSUME RANGE=1)")
        # Avoids a possible 0-division error in the for loop.
        # This means there is no range, so we will pretend the range is 1.
        # This is a temporary fix, and should be looked at more in depth.
        #TODO
        power_range = 1
    for i in range(number_of_measurements):
        normalized_power[i] = (power[i] - min_power) / power_range
    return normalized_power

def bucketize_measurements(normalized_power):
    # Step 0: All buckets start with a 1 in them.
    buckets = number_of_buckets * [1]
    updated_sample_count = len(normalized_power) + len(buckets)
    for i in range(number_of_measurements):
        measurement = normalized_power[i]
        if 0.9 < measurement and 1 >= measurement:
            buckets[10] = buckets[10] + 1
        if 0.8 < measurement and 0.9 >= measurement:
            buckets[9] = buckets[9] + 1
        if 0.7 < measurement and 0.8 >= measurement:
            buckets[8] = buckets[8] + 1
        if 0.6 < measurement and 0.7 >= measurement:
            buckets[7] = buckets[7] + 1
        if 0.5 < measurement and 0.6 >= measurement:
            buckets[6] = buckets[6] + 1
        if 0.4 < measurement and 0.5 >= measurement:
            buckets[5] = buckets[5] + 1
        if 0.3 < measurement and 0.4 >= measurement:
            buckets[4] = buckets[4] + 1
        if 0.2 < measurement and 0.3 >= measurement:
            buckets[3] = buckets[3] + 1
        if 0.1 < measurement and 0.2 >= measurement:
            buckets[2] = buckets[2] + 1
        if 0 < measurement and 0.1 >= measurement:
            buckets[1] = buckets[1] + 1
        else:
            buckets[0] = buckets[0] + 1
    return buckets, updated_sample_count

def prepare_data(keyboard_voltage, current, title):
    print("For", title,  "measurements:")
    # Calculate power of each sample.
    power = calculate_power(keyboard_voltage, current)
    print("\tFinished calculating power.")
    # Normalize piece of data with existing data between 0 and 1
    normalized_power = normalize_measurements(power)
    print("\tFinished normalizing power.")
    # Bucketize the existing data
    bucketized_data, updated_sample_count = bucketize_measurements(normalized_power)
    print("\tFinished bucketizing measurements with an extra 1 in each bucket.")
    return bucketized_data, updated_sample_count

def calculate_augmented_KL(questionable_bucketized_data, updated_questionable_sample_count,
                           baseline_bucketized_data, updated_baseline_sample_count):
    # NOTE: It is expected that the data inputted into this
    #       has already gone through the prepare_data function.
    augmented_KL_score = 0
    for i in range(number_of_buckets):
        baseline_bucket_probability = baseline_bucketized_data[i] / updated_baseline_sample_count
        questionable_bucket_probability = questionable_bucketized_data[i] / updated_questionable_sample_count
        partial_score = baseline_bucket_probability * abs(math.log2(
            baseline_bucket_probability / questionable_bucket_probability))
        augmented_KL_score = augmented_KL_score + partial_score
    print(questionable_bucketized_data)
    print(baseline_bucketized_data)
    print("The augmented KL-score is:", str(augmented_KL_score))
    return augmented_KL_score

def detect_keylogger(baseline_bucketized_data, updated_baseline_sample_count):
    # Loop forever:
    for i in range(number_of_trials):
        # Collect sample of data
        keyboard_voltage, current = take_measurements(number_of_measurements)
        # Prepare the data to have the KL-score calculated:
        questionable_bucketized_data, updated_questionable_sample_count = prepare_data(
            keyboard_voltage, current, "questionable")
        # Calculate augmented KL-score:
        augmented_KL_score = calculate_augmented_KL(
            questionable_bucketized_data, updated_questionable_sample_count,
            baseline_bucketized_data, updated_baseline_sample_count)
        no_keylogger_score_max = 0.31641487489734677
        no_keylogger_score_min = 0.25765348294024815
        threshold_score = (no_keylogger_score_max - no_keylogger_score_min) / 2 + no_keylogger_score_max
        if augmented_KL_score > threshold_score:
            print("KEYLOGGER DETECTED")#, end="\r")
        else:
            print("NORMAL BEHAVIOR")

def main():
    # Import baseline data from file.
    baseline_voltage, baseline_current = get_baseline_data("NoKeyloggerNoCapslock2_new.csv")
    # Prepare the baseline_data for KL score.
    baseline_bucketized_data, updated_baseline_sample_count = prepare_data(
        baseline_voltage, baseline_current, "baseline")
    # See if there is a keylogger.
    detect_keylogger(baseline_bucketized_data, updated_baseline_sample_count)

main()
```

Figure 27: The above is the detect_keylogger.py code. This is the file that gets run to try to detect a keylogger.



```python
import time

import board
import busio
i2c = busio.I2C(board.SCL, board.SDA)
import adafruit_ads1x15.ads1115 as ADS
from adafruit_ads1x15.analog_in import AnalogIn
ads = ADS.ADS1115(i2c)

def get_voltage(channel):
    return 2 * channel.voltage

def get_flipped_voltage(channel):
    return -get_voltage(channel)

# P0 will have ground on it.
# P1 will have keyboard power on it.
# P3 will have keyboard power after resistor on it.
voltage_channel = AnalogIn(ads, ADS.P0, ADS.P1)
current_channel = AnalogIn(ads, ADS.P1, ADS.P3)

# For approximately 5 minutes of measurements:
number_of_measurements = 15_000#30_000 #For 10 minutes
progress_resolution = 1_000
progress_bar_length = number_of_measurements / progress_resolution

timestamps = [None] * number_of_measurements
keyboard_voltage = [None] * number_of_measurements
current = [None] * number_of_measurements

def print_progress_bar(measurement_number, total_number):
    fractional_progress = measurement_number / total_number
    bar_progress = int(fractional_progress * progress_bar_length)
    arrow = bar_progress * '-'
    padding_after_arrow = int(progress_bar_length - (bar_progress + 1)) * ' '
    if measurement_number == total_number:
        end_of_line = '\n'
    else:
        # We only include an arrow at the end when it is not done loading.
        arrow = arrow + '>'
        end_of_line = '\r'
    percent_done = str(round(fractional_progress * 100, 1)) + '%'
    print('\tProgress: [' + arrow + padding_after_arrow + '] ' + percent_done, end=end_of_line)

def take_measurements(num_measurements):
    print('Taking ' + str(num_measurements) + ' measurements:')
    for i in range(num_measurements):
        print_progress_bar(i, num_measurements)
        # We need to get the flipped_voltage of the voltage channel.
         # The output of this is P0 - P1 (which is negative) but we want the positive value.
        keyboard_voltage[i] = get_flipped_voltage(voltage_channel)
        current[i] = get_voltage(current_channel)
        timestamps[i] = time.time()
    # Print the finished and final progress bar.
    print_progress_bar(num_measurements, num_measurements)
    # This return statement is for use of this function in other code.
    return keyboard_voltage, current

def store_data_in_file():
    # Get file name from user
    file_name = input('What .csv file name should we use to store the data?\n')
    file_handle = open(file_name + '.csv', 'w')
    file_handle.write('Timestamp,Voltage,Current\n')
    for i in range(number_of_measurements):
        file_handle.write(str(timestamps[i]) + ',' +
                          str(keyboard_voltage[i]) + ',' +
                          str(current[i]) + '\n')
    print('To see your data, look at:', file_name + '.csv')

def main():
    time.sleep(5)
    start_time = time.time()
    take_measurements(number_of_measurements)
    print('It took', time.time() - start_time, 'seconds to take',
          number_of_measurements, 'measurements.')
    store_data_in_file()

main()
```

Figure 28: The above code is from record_ADC_measurements.py. You need to have this file in place to run detect_keylogger.py (since it has functions imported).

10 References

1. ADS1115 16-Bit ADC - 4 Channel with Programmable Gain Amplifier. (n.d.). Adafruit Industries. Retrieved April 23, 2023, from https://www.adafruit.com/product/1085
2. ads1115 Price and Stock. (n.d.). Findchips. Retrieved April 23, 2023, from https://www.findchips.com/search/ads1115
3. Hardware Keylogger - KeyGrabber Forensic Hardware Keylogger. (n.d.). Keelog. Retrieved April 23, 2023, from https://www.keelog.com/keygrabber-keylogger/
4. Hardware Keylogger - KeyGrabber Pico. (n.d.). Keelog. Retrieved April 23, 2023, from https://www.keelog.com/keygrabber-pico/
5. KeyGrabber TimeKeeper USB MCP 16GB - Premium USB Hardware Keylogger with 16GB Flash Drive, Time-stamping and Mac Compatiblity. (n.d.). Keelog. Retrieved April 23, 2023, from https://www.keelog.com/keygrabber-timekeeper-usb-mcp-premium-usb-hardware-keylogger-with-flash-drive-time-stamping-and-mac-compatiblity/
6. KeyGrabber USB 16MB - USB Hardware Keylogger with 16MB Flash Drive. (n.d.). Keelog. Retrieved April 23, 2023, from https://www.keelog.com/keygrabber-usb-usb-hardware-keylogger-with-flash-drive/
7. MacKay, D. J. (2003, September 25). Information Theory, Inference and Learning Algorithms - David J. C. MacKay. Google Books. Retrieved April 23, 2023, from https://books.google.com/books?id=AKuMj4PN_EMC
8. MCIGICM (10 Pcs) 10K Ohm Breadboard Trim Potentiometer kit with Knob for Arduino. (n.d.). Amazon.com. Retrieved April 23, 2023, from https://www.amazon.com/MCIGICM-Breadboard-Trim-Potentiometer-Arduino/dp/B07S69443J
9. Tumor size. (n.d.). Mayo Clinic. Retrieved April 23, 2023, from https://www.mayoclinic.org/diseases-conditions/breast-cancer/multimedia/tumor-size/img-20006260
10. USB Type A Jack Breakout Cable with Premium Female Jumpers. (n.d.). Adafruit Industries. Retrieved April 23, 2023, from https://www.adafruit.com/product/4449

11. USB Type A Plug Breakout Cable with Premium Female Jumpers. (n.d.). Adafruit Industries. Retrieved April 23, 2023, from https://www.adafruit.com/product/4448

12. Wynn, L.S. (2023, March 28). How Much Text is in a Kilobyte or Megabyte? (with pictures). EasyTechJunkie. Retrieved April 23, 2023, from https://www.easytechjunkie.com/how-much-text-is-in-a-kilobyte-or-megabyte.htm