12-2024

Classifying Supersonic Frequencies for Active Acoustic Side-Channel Exploitation

Destin Hinkel

CLASSIFYING SUPERSONIC FREQUENCIES FOR ACTIVE ACOUSTIC SIDE-CHANNEL EXPLOITATION

A Thesis

Submitted to the Graduate Faculty of the
University of South Alabama
in partial fulfillment of the
requirements for the degree of

Master of Science

in

Computer Science

by
Destin A. Hinkel
B.M., University of South Alabama, 2018
M.M., University of South Alabama, 2022
December 2024

ACKNOWLEDGEMENTS

I would first like to thank my committee chair, Dr. George Clark. Thank you for always holding me accountable and pushing me to achieve. I would also like to thank Dr. J. Todd McDonald and Dr. Arie VandeWaa for serving on my committee and my Scholarship for Service cohort for being the best support system a student could ask for.

I would also like to thank my former colleagues and continued friends in music education. Music and education will always have a special place in my heart. The support of the music faculty and friends I have had during my time at the University of South Alabama will forever positively impact my life.

Finally, and most importantly, I would like to thank my family: my wife Jessica and our two puppies, Zoey and Gracie. Thank you, Jessica, for your unwavering love and support throughout this endeavor. This has been a difficult road at times, but you never failed to pick me up through the failures and cheer me on through the successes. I am beyond grateful for the support system that I have been blessed with.

This research was supported in part by the National Science Foundation under grant DGE-2142948.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AST	Audio Spectrogram Transformer
CNN	Convolutional Neural Network
FFT	Fast Fourier Transform
GHz	Gigahertz
HMM	Hidden Markov Model
Hz	Hertz
IFFT	Inverse Fast Fourier Transform
kHz	Kilohertz
LFCC	Linear Frequency Cepstral Coefficient
MHz	Megahertz
ML	Machine Learning
OFDM	Orthogonal Frequency Division Multiplexing
RPM	Rotations per Minute
SCA	Side-Channel Attack
TDoA	Time Difference of Arrival
WPM	Words per Minute

ABSTRACT

Destin A. Hinkel, M.S., University of South Alabama, December 2024. Classifying Supersonic Frequencies for Active Acoustic Side-Channel Exploitation. Chair of Committee: George Clark, Ph.D.

Computing side-channel research explores the manner in which physical emanations from systems can be used to reconstruct data. Acoustic side-channels are those physical emanations that produce a sonic frequency that is subsonic, supersonic, or considered in the range of human hearing [1]. Acoustic side-channel attacks (SCAs) are typically performed passively: a listening device captures aural frequencies from a machine via a microphone that are transmitted to the attacker for analysis [1]–[3]. Machine learning models have been presented to classify individual keystrokes according to variations in acoustic frequency [4]. Furthermore, the SonarSnoop framework presents a novel active approach that involves both generating and recording aural frequencies acting as a type of sonar system to record physical motion [5].

This research attempts to develop a supervised machine learning model to classify finger motion to collect login credentials typed on a laptop keyboard. The active acoustic side-channel has been used to track two-dimensional finger motion, but three-dimensional finger tracking using active acoustics is novel. The model as trained in this study incorrectly inferred labels on unseen data; however, we found and demonstrated that training with more samples per label may result in greater success during inference.

CHAPTER I

INTRODUCTION

Side-channels in computing represent potentially devastating avenues through which physical emanations representing bits of information can be gathered from systems even when they are air gapped. Clues about specific computer operations can be observed in the form of thermal, electromagnetic, or acoustic emanations created by user interaction or operations of the system itself. In recent years, the field of acoustics has represented an increasingly explored area of side-channel research [1]–[3], [5]. Acoustic side-channels are typically observed passively by a listening device that introduces a microphone to capture operating sounds produced by various components of the system, such as keyboard clicks, coils, printers, or fans [1]–[4]. The exfiltrated data are then analyzed to reconstruct passwords, a cryptographic key, or other sensitive data whose processing caused the unique operation and acoustic output of the associated hardware component. For example, Genkin et al. proposed reconstructing cryptographic keys based on the coil whine of air-gapped systems while performing encryption operations [1]. Furthermore, supervised machine learning (ML) has been used in relation to keyboard acoustic emanations where there is a predictable number of distinct sounds that can be classified, such as by keys or combinations of keys that are typed [4].

In 2019, Cheng et al. proposed a novel active attack framework that, like its predecessors, collected data through microphones to send to the attacker for analysis [5]. However, the framework diverges on the basis of the function of the generated sound. Instead of passively listening to the acoustic emanations of the devices, the SonarSnoop framework assumes control of the speaker system of a device to generate pulses of supersonic frequencies. These frequencies are recorded by a microphone and continuously looped to generate a miniature sonar system that can capture physical movement in a two-dimensional space [5], [6]. The authors applied this framework to capture unlock patterns for Android smartphones drawn by individual users. They were able to show that their framework could either retrieve the pattern outright or that the framework could significantly reduce the number of guesses needed to enter the correct pattern. When reducing the guesses below a threshold that would result in the phone becoming locked, the implementation of the SonarSnoop framework was considered successful. Although this used features of supersonic audio production and analysis that had previously been proposed, SonarSnoop claims to be the first active acoustic SCA framework [5].

In this thesis, a more generalized attack against a laptop system is proposed using the SonarSnoop framework introduced by Cheng et al., as well as a supervised ML model designed to classify the supersonic frequencies produced by keys as they are typed [5]. The goal is to successfully identify typed keys to obtain user login credentials or to reduce the possible set of passwords by a statistically significant margin. The proposed attack uses the active SonarSnoop framework, which can take advantage of the speakers and microphones of a laptop computer to create a miniaturized sonar system that can relay data

to the attacker for analysis and reconstruction of physical movements through audio classification.

1.1 Research Questions

This thesis will explore the application of a classification model first proposed in passive attack frameworks, as well as active acoustic developments presented by Cheng et al. in an effort to obtain authentication credentials entered via the keyboard of a laptop device [4], [5]. To that end, this research will attempt to answer the following question. How can the SonarSnoop attack framework be combined with an ML model to gather sensitive information?

There are three sub-questions that have been derived from asking about creating this attack. The first sub-question asks what specific implementation will allow the capture of user input login credentials from a laptop computer system. The second sub-question asks how this attack can be obfuscated in a real-world setting. The final sub-question asks what defenses could be used to prevent an active acoustic SCA.

Information is collected detailing the methods by which the attack can remain undetected by the user. This is a critical component mentioned in almost every acoustic SCA surveyed [1]–[3], [5], [6]. For active acoustic attacks, aural signals must be produced in the supersonic range that exists outside the typical upper limit of human hearing. Frequencies greater than 18 kHz are outside the upper limits of human hearing and are most appropriate for a project of this class [6]. Unlike other physical side-channels, aural signals are easily perceived if no steps are taken to obfuscate them. This thesis will

explore different ideas for obfuscation, such as the production of supersonic frequencies using the framework.

Several potential defenses against attacks of this class are proposed based on the original suggestions of Cheng et al. [5]. The approaches considered will include those that can be applied practically in software, since fundamental changes to hardware and user interaction are unlikely. For example, a suggestion from the SonarSnoop authors regarding defense involved completely disabling microphone and speaker devices. However, these features are widely used by commercial users and disabling them is an unlikely compromise between security and functionality. Practical improvements are proposed, starting with the suggestion of the SonarSnoop authors that supersonic audio channels should be jammed outside the range of human hearing [5].

1.2 Expected Outcome

The SonarSnoop framework's originally proposed attack was against an Android smartphone with the goal of capturing user-drawn unlock patterns [5]. The authors were able to show that this framework could either capture such an unlock pattern or reduce the number of guesses by a statistically significant amount by leveraging what the data indicated against known common unlock patterns. Using a similar approach against a laptop system that leverages known common passwords based on readily available resources from penetration testing, such as rockyou.txt, a similar result can probably be produced [7]. Furthermore, the FingerIO and CovertBand frameworks show that physical motion tracking is possible, and, in the latter, Nandakumar et al. specifically state that

extension to three dimensions would be feasible by adding a third microphone [6], [8]. Classification models have also been shown to be successful in identifying keystrokes in passive analysis, and modern image-based spectrogram classification models for aural frequencies have been used and applied successfully [4], [9]. Therefore, it would be reasonable to hypothesize that the attack will be successful in obtaining the appropriate login credentials that a user enters into a password field on a laptop computer after inference using a model trained on the active acoustic framework.

1.3 Thesis Outline

The remainder of this document is outlined as follows: Chapter II presents a review of the related literature on various passive acoustic SCAs, audio classification models used in passive attacks, the SonarSnoop active acoustic side-channel, projects that implement miniature sonar development for technological features, and other supporting work. Chapter III proposes a specific methodology to conduct an experiment to generalize the SonarSnoop framework and apply an ML model for classification. Chapter IV provides experimental results and evaluates the accuracy of the audio classification model. Finally, Chapter V presents conclusions and recommendations for future work in supersonic audio classification. The relevant code is also provided in the appendices.

CHAPTER II

BACKGROUND AND RELATED WORK

The concept of an active acoustic attack framework is relatively new, but is based on several years of research in acoustic side-channels [1], [4], [5]. The notion of an acoustic attack against computer systems has long been associated with simply listening to peripheral devices for emanations, such as the work by Backes et al. or, from 2004, Asonov and Agrawal [4], [10]. Researchers have explored this area more thoroughly in recent years, despite some noticeable skepticism mentioned in the surveyed works about the legitimacy of aural emissions as an attack vector [11]. A large amount of the literature surveyed represents a passive acoustic framework. The use of passive acoustic side-channel exploitation involves simply listening to information transmitted by a physical computing device, exemplified by Genkin et al., Guri et al., and de Gortari Briseno et al. [1]–[3]. Classification models have been proposed in early keyboard acoustic emanation work, but are unique to the analysis of passively captured keyboard frequencies [4].

The idea of an active acoustic framework, first described as an attack vector in SonarSnoop, presents a natural extension to the previously proposed passive acoustic side-channel, and the related literature shows the development of such an attack vector rooted in studies aimed at exploring active acoustic capabilities as a set of features for the

user [5]. The work most frequently cited by various studies exploring passive attack frameworks has been that of Genkin et al. [1]. In 2014, they set the stage for further exploration of acoustic side-channels, and their work is a major step towards proving the legitimacy of acoustics as a valid and potentially devastating side-channel.

2.1 Cryptographic Acoustic Side-Channel Analysis

Side-channels have been presented as vehicles to obtain physical emanations from a computer in a manner that allows reconstruction of a cryptographic key. Various side-channels include electromagnetic, thermal, power, and acoustics [11], [12]. Much of the related work in the area of side-channel analysis for the purpose of successfully extracting a cryptographic key deals with the former three in the aforementioned list. Kocher et al. pointed out that acoustic side-channels are an unlikely source of information leakage due to the relatively low frequency of emissions compared to higher frequency operations, such as processor clock rates [11]. Specifically, sonic emission frequencies are typically measured in hertz (Hz) and kilohertz (kHz), particularly in the 20 Hz to 20 kHz frequency range. In contrast, processor clocks are typically measured in megahertz (MHz) and gigahertz (GHz), and this makes a measurement of processor operations from an acoustic standpoint particularly difficult due to the sheer number of operations that can be executed in a single cycle at peak aural frequencies around 18 kHz [6].

Although acoustic leakage is considered a less likely source of relevant information, Genkin et al. presented a method to obtain a cryptographic key in 2014 [1], [11]. Furthermore, they show that their attack is applicable to other side-channels measured in

low-bandwidth (Hz-kHz) ranges, and the example that they provide utilizes power emissions as a side-channel. This paper is cited by nearly every other surveyed work, and it subsequently sparked an interest in acoustics as a cryptographic side-channel. To our knowledge, no other surveyed work presented a direct model to obtain a cryptographic key; however, similar frameworks were proposed to secure, interrupt, or infect cryptographic processes between computers and peripheral devices. For example, Anand and Saxena assert that an adversary can easily hijack the aural frequencies used for device pairing by applying a model proposed by Halevi and Saxena to recover the aural frequencies produced by vibration motors [13], [14]. A vibration side-channel is occasionally mentioned in other works such as SoK and Vibreaker [13], [15]. However, Figure 1 shows the aural frequencies emitted by motor vibrations, and some of these frequencies are clearly below the upper limit of human hearing [6], [13]. Therefore, it is reasonable for the purposes of this work to assume that the acoustic side-channel is a subset of the vibration side-channel. The authors proceed to present a model to defend against an attack of this class by presenting their Vibreaker model that floods the acoustic side-channel with noise intended to mask authorized information exchange [13].

2.2 Passive Acoustic Attacks

Passive acoustic attacks have been defined in the surveyed literature as attacks that involve the use of a microphone to record acoustic frequencies emitted by a computer system during its normal course of operation [5].

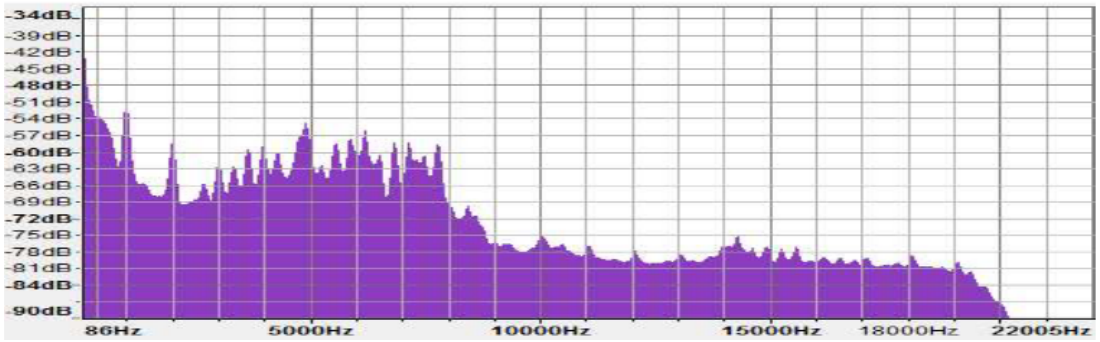

Figure 1. Frequencies Produced Pairing with PIN 4562 [13].

The vast majority of the surveyed literature fits within this framework and attacks typically depend either on the user interaction with the computer or the operation of the computer itself. Therefore, there is a clear distinction between attacks that specifically listen for user input using a keyboard and attacks that listen for frequencies emitted by the computer system itself or peripheral devices. In Figure 2, Backes et al. present a high-level overview of attacks that use this framework and employ ML techniques [10].

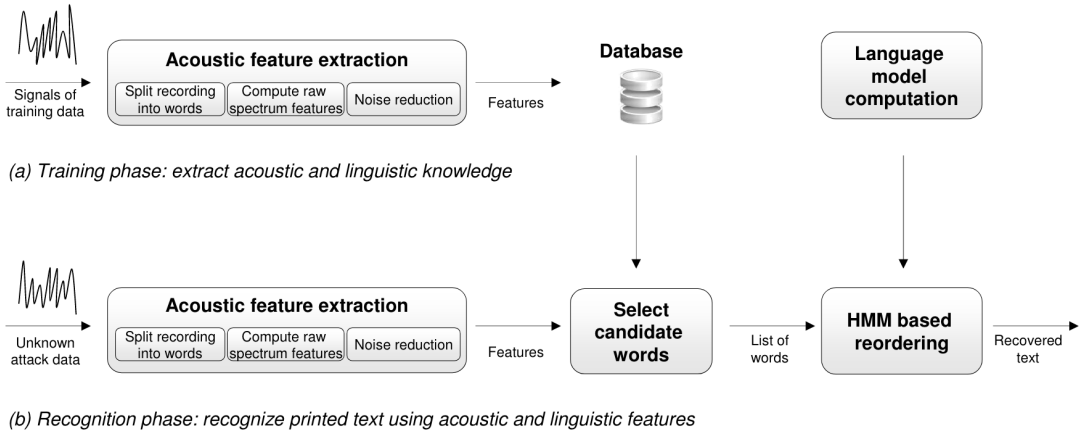

Figure 2. High Level Passive Attack Overview [10].

The attacks are broken into a training phase for the neural network ML model to develop a database of candidate words to cross-reference against features extracted from raw acoustic data. The Hidden Markov Model (HMM) language component leverages speech recognition technology to extract the most likely word used [10]. Section 2.2.1 further details ML techniques that involve audio classification.

2.2.1 Audio Classification

Reconstruction of keystrokes, described in further detail in Section 2.2.2, is a subset of a much larger field in audio classification, with many pre-trained models for such tasks readily available on the public ML database, HuggingFace [16]. Significant effort has been made to develop various models for audio classification, particularly in the range of audible speech. A prime example of a common use case for such a model is speech recognition software. This is typically accomplished by collecting large amounts of training data in the form of real speech and converting those data into vectors to run into the classification model [17]. In many cases, these are vector representations of spectrogram images generated from raw audio files [18]. Exemplified in Figure 3, deep learning with convolutional neural networks (CNNs) is applied to the image classification problem to classify spectrograms based on various labels that match the original audio files used as training data [19].

Although deep learning models are commonly used to train audio classification models, a shift toward transformer models has been observed following a proposal by Google in 2017.

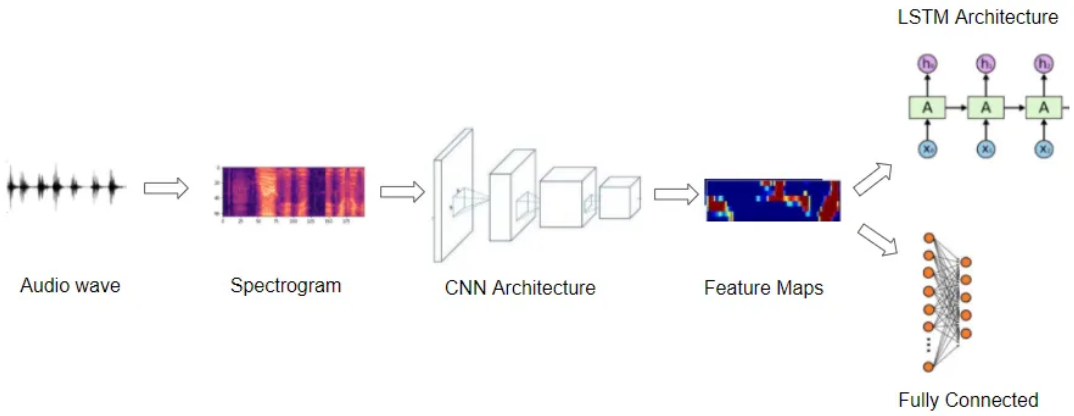

Figure 3. Deep Learning Pipeline for Audio Classification [18].

At a high level, these models can learn context around specific instances of data, making them extremely effective in language and audio classification applications [20]. Coinciding with an interest in image-based classification, an audio spectrogram transformer model (AST) was proposed in 2021 by Gong et al. Their model converts aural signals into audio spectrograms and uses image-based classification, rather than direct vectors based on audio files. Their model is exemplified in Figure 4, and the authors were able to demonstrate the ability to classify various audio data sets with extremely high levels of accuracy [9].

2.2.2 Keystroke Frequency Capturing

The first academic work surveyed that directly addresses the ability to passively monitor keyboard acoustic emissions was presented by Asonov and Agrawal [4]. This work is heavily referenced by subsequent acoustic attacks that build on its proposed framework. Acoustic attacks in this area are generally based on ML from neural networks to categorize various keystrokes, as exemplified in Figure 2 [4], [21], [22].

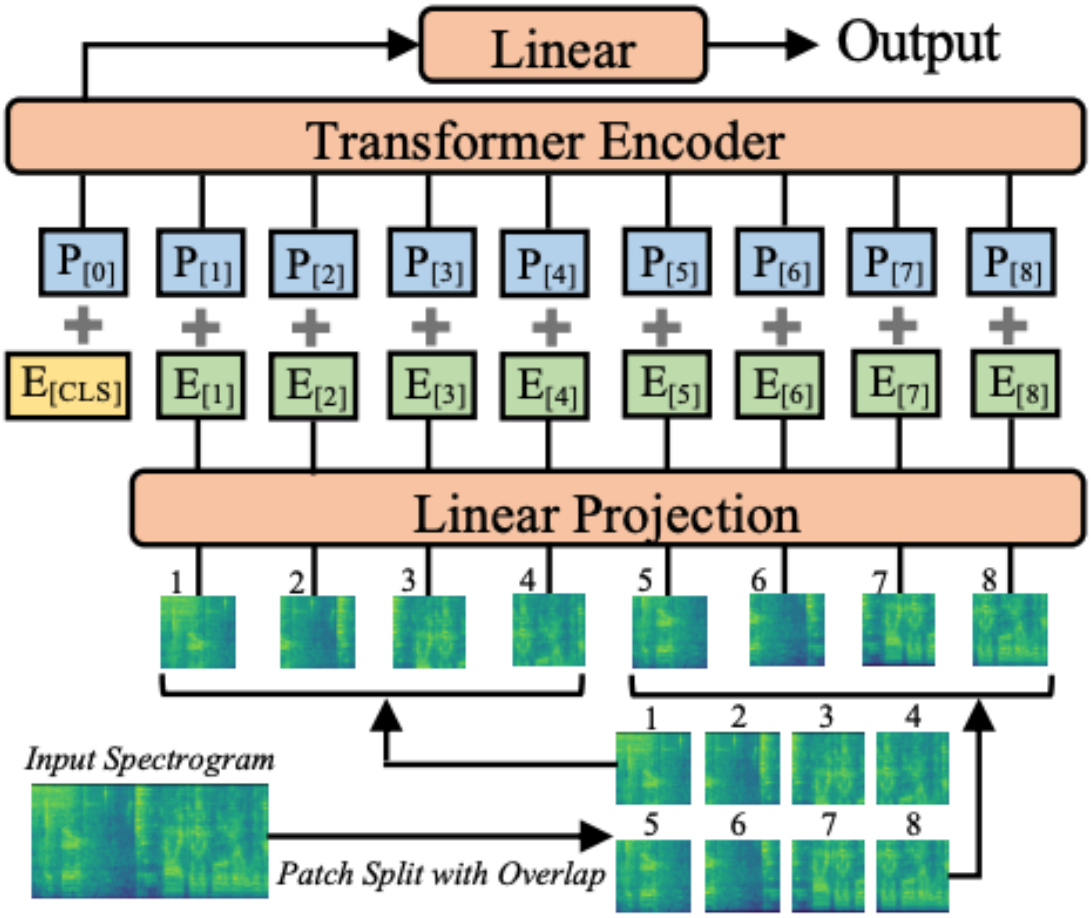

Figure 4. Overview of Audio Spectrogram Transformer (AST) [9].

For example, Asonov and Agrawal proposed an ML model that provided information on variations in the sound frequencies of 30 different keys on a mechanical keyboard for training [4]. Once the neural network ML model was trained, it was able to use a parabolic microphone placed up to 15 meters from a computer to acquire acoustic data representing individual keystrokes. They introduced a model that was able to reconstruct data to successfully represent typed passwords, and Halevi and Saxena reconstructed data based on this model [21], [22].

Asonov and Agrawal also addressed more specific questions that arise for specific implementation, including differences in keyboard and typing styles when classifying individual letters [4]. Specifically, they conclude that although collecting direct keys being typed has a lower quality when the model is applied to different types and sizes of keyboards, there is enough data leakage to be statistically significant. Furthermore, their experimentation showed that a more resilient model was developed when a variable typing force was applied during training [4]. Interestingly, one of their suggested countermeasures to this class of attack was the use of a touchscreen keyboard that can now be attacked using the active acoustic side-channel, discussed further in Section 2.3 [4], [5].

Halevi and Saxena, further addressing potential issues for analysis of different typing styles, introduced two common typing styles: "hunt-and-peck" typing and "touch" typing [21], [22]. In their papers, the "hunt-and-peck" method is defined as requiring the user to search for each key on a keyboard before typing the key. It is implied that this method is used by users with little or no formal typing training and is much slower than the alternative "touch" typing method. In contrast, "touch" typing is traditional typing that is much more prevalent in today's world [22]. A gap in the methodology proposed by Asonov and Agrawal along with Zhuang et al. is the complete omission of the typing style as a component of the framework [4], [23]. Therefore, the attacks presented within the methodology may not be feasible in a more realistic setting. Halevi and Saxena were able to show that typing style was a major component that affected the ability to successfully obtain data with a 64% success rate using "hunt-and-peck" and only a 40% success rate using "touch" typing. Thus, their later work proposed an attack that attempts to take into

account different typing styles [21], [22]. Although they employ neural network ML as a baseline for comparison, the authors show that their novel "Time Frequency Decoding" technique offers a superior rate of detection. The authors also manage to point out shortcomings of previous works by showing that typing style has a significant effect on the ability to successfully recover acoustic information that accurately represents keystrokes [4], [21]–[23].

Passive attacks on keystroke capture continue to cite Asonov and Agrawal's study as a seminal work in this area and have proposed important ways to improve the effectiveness of their proposed framework [4]. Backes et al. apply an ML model similar to that proposed by Asonov and Agrawal that classifies entire English words rather than individual letters to reconstruct acoustic side-channel data leaked by printers [10]. Zhu et al. still use a passive attack framework that records emitted frequencies, but the work represents an important change in the frameworks from passive to active [24]. The component "Time Difference of Arrival" (TDoA) uses a smartphone for acoustic tracking based on differences in signal arrival over time, which begins to approach the development of an active attack framework [24].

2.2.3 Operations Frequency Capturing

The remaining passive frameworks surveyed typically deal with aural emissions produced by regular computer component operations or those of peripheral devices, such as printers. Works such as those presented by Genkin et al. and Guri et al. deal with acoustic emanations produced by the internal components of computers by addressing "coil whine" or internal fans, respectively [1], [2]. Genkin et al. define "coil whine" as the

vibration of electronic components within a computer chassis that operates at up to 20 kHz [1]. Guri et al. used system fans to transmit data [2]. Both models require a microphone that is within 10 meters of the computer chassis, and a primary weakness in the frameworks is the speed at which data can be transmitted. Both attack frameworks require data to be consistently sent over a time frame of at least an hour, and Guri et al. specifically point out that data can only be transmitted at a rate of 900 bits/hour [2]. As mentioned in Section 2.1, processors and other internal components are typically clocked in the MHz to GHz range, while acoustic frequencies are in the Hz to kHz range. Thus, many computations can occur within a single acoustic clock, making single instructions virtually impossible to obtain [11]. Genkin et al. relies on pattern recognition in many repeated instructions, while Guri et al. assume access to a key that is simply being transmitted bit by bit over a time frame measurable in hours [1], [2]. Figure 5 provides an example of this: a series of bits produced over a period of 60 seconds.

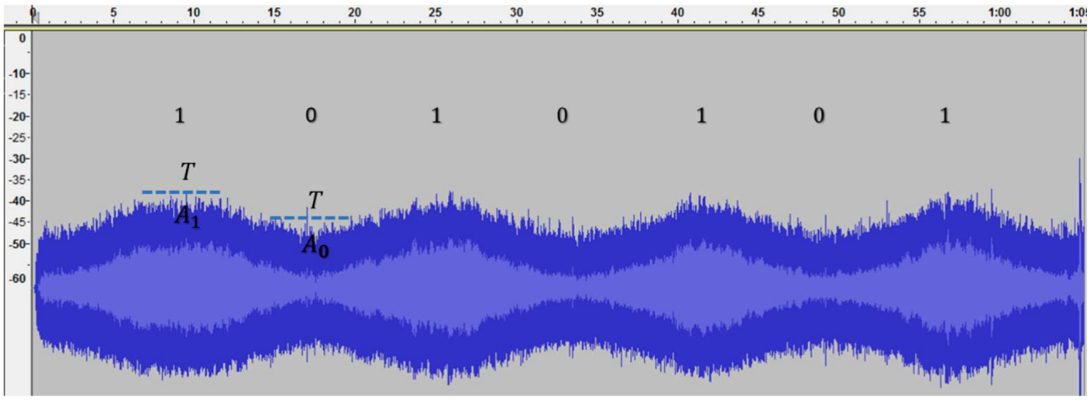

Figure 5. Bits Represented by Sounds from Fan Speed [2].

The peaks are sounds produced at 3500 rotations per minute (RPM) representing a binary "one", and the valleys are sounds produced at 3000 RPM representing a binary "zero" [2]. Neither framework lends itself well to real-time data collection or transmission, but they do show that both acquiring and transmitting a cryptographic key or password is feasible.

Other passive frameworks focus exclusively on the use of printers as a more efficient means of transmitting information. Backes et al. provided an excellent breakdown of the attack process originally proposed by Asonov and Agrawal, as exemplified in Figure 2. Their model follows a framework similar to that discussed in Section 2.2.2 where ML is applied to a language model and different sounds that represent printed characters [4], [10].

Most recently, de Gortari Briseno et al. proposed a novel attack with the potential to decrease the user's ability to detect more sensitive information being printed [3]. Surveyed attacks typically capture clear text representations of data, but the authors proposed a framework that inserted patterns into a printed document that were invisible to the naked eye. When the documents were printed, the abnormal operating sounds emitted to print these patterns were captured, and the authors showed that they were able to successfully reconstruct these frequencies to collect data [3]. Like some other authors of passive attack frameworks, they assume compromised access to a computer system [2], [10]. The framework presented by de Gortari Briseno et al. is particularly important because it emphasizes the importance of adversarial covert action. The authors not only mask the interception of acoustic data but also manage to mask the transmission of acoustic data

through a novel framework that is highly unlikely to alert a typical user to any abnormal operation [3].

2.3 Developing an Active Acoustic Attack Framework

In contrast to attacks that follow the passive framework detailed in Section 2.2, the authors of SonarSnoop propose the class of active acoustic attacks [5]. In this study, Cheng et al. define an "active acoustic attack" as an attack that introduces sound into the environment to be collected by a microphone [5]. More specifically, the authors propose the creation of a miniature sonar system that uses system speakers and internal microphones to collect data based on frequencies and time, as proposed by Zhu et al., which represents human-computer interaction in the real world [24]. The SonarSnoop attack framework claims to be the first attack framework to utilize an active sonar component [5]. Based on this survey of relevant works, their claim is true, but their work is not the first to implement sonar as a means to capture and use data representing movement in the real world.

When considering an active acoustic approach as defined by Cheng et al., the timing of signal arrival is a critical factor that allows the sonar feature to work as intended [5]. Although technically presented as a passive attack, the notion of detecting keys based on the "time difference of arrival" proposed by Zhu et al. is a step towards incorporating a sonar feature [24]. Similar ideas have been proposed to obtain a map of a specific room, and CovertBand proposes a framework that incorporates an active sonar component to detect movement in rooms within a building [8], [25]. A malicious threat actor could then

draw conclusions about the types of activity that takes place based on the location of people within a building. This would assist in potential active reconnaissance against an organization. Furthermore, the differences in time arrival of these audio signals would be reflected in the audio spectrogram image, so the differences between images is a potentially lucrative place to gain insight into keystroke patterns [9].

The idea of detecting physical motion using the timing of frequencies detected by a microphone is only one step short of applying a sonar feature that actively injects frequencies to detect physical objects and motion. The aforementioned sonar feature is prominently displayed in FingerIO and CovertBand to show that physical activities can be tracked at various micro- and macroscales, respectively. Furthermore, they clearly indicate that the presence of more microphones increases the accuracy of the sonar component in the detection of movement [6], [8]. For example, three microphones can be used to triangulate the position of a finger in a three-dimensional space; this cannot be achieved using only two microphones [6]. Cheng et al. also address this point by mentioning that the SonarSnoop framework uses two microphones to track finger movement on the smartphone's screen: a top microphone and a bottom microphone [5].

CovertBand is the framework closest to SonarSnoop in terms of its potential to be used as an active attack [5], [8]. The SonarSnoop framework takes CovertBand's detection technique a step further by weaponizing the sonar component in order to obtain authentication credentials to a user's Android smartphone. Like FingerIO, the SonarSnoop framework explores aural emissions at the micro level, but Cheng et al. introduce the feature proposed by Nandakumar et al. as an attack vector [5], [6]. Furthermore, the

duality of microphones is built into the smartphone, which lends itself well to the successful completion of the attack proposed in SonarSnoop. Thus, generalizing this attack to involve applying active sonar to a laptop system might necessitate the use of external microphones depending on the number of internal microphones within the laptop.

2.4 Proposed Defenses Against Acoustic Attacks

Prior to the introduction of active sonar, most defensive recommendations against acoustic attacks involved completely reducing or eliminating acoustic emissions. For example, Asonov and Agrawal proposed the use of plastic keyboards and the manufacture of keyboard plates that would cause each key to emit the same sonic frequency [4]. As technology has advanced, this advice has become less effective due to the ability of a microphone to perceive minute changes in frequency and spatial awareness features such as TDoA [23], [24]. A highly recommended countermeasure in the surveyed literature is acoustic signal jamming [2], [5], [8]. Specifically, various authors recommend emitting defensive acoustic frequencies in the 18-20 kHz range. However, CovertBand and SonarSnoop concede that jamming is not a perfect solution [5], [8]. For example, supersonic frequencies in the aforementioned range may still be audible to young children or animals and cause discomfort. Furthermore, an attacker could potentially mask adversarial frequencies in the audible range with music or ambient sounds, bypassing their need to attack through supersonic frequencies [8].

Hardware solutions are also a popular recommendation of various authors [1], [2], [4], [5], [8], [23]. One solution is to inject background noise to obfuscate all acoustic

emissions. Although this might be effective in a passive attack, active sonar attacks would be virtually unaffected by this technique [1], [8]. Cheng et al. suggested designing speakers that do not emit acoustic frequencies in the inaudible supersonic range, but the authors also acknowledge that this recommendation does not address existing hardware [5]. Other authors have suggested eliminating all speakers on secure computers or installing a switch to completely disable audio [2], [5]. Although this might be an effective recommendation for secure computers, the related literature has shown that many attacks occur against non-secure computers, against internal components of secure computers, or against smartphones that virtually always have built-in speakers [1], [2], [5], [23]–[27]. To the author's knowledge, no realistic defensive proposal has been made to defend against active acoustic attacks that is widely applicable.

2.5 Summary of Acoustic Side-Channel Related Works

Table 1 presents a categorized summary of key research contributions in the field of acoustic side-channel analysis. Cryptographic studies have demonstrated the feasibility of extracting cryptographic keys through low-bandwidth acoustic emissions. Audio classification highlights advancements in techniques that use deep learning models. Passive attacks encompasses research that uses ambient acoustic emissions to infer sensitive information. Active attacks involve studies in which acoustic signals are injected to gather information about the environment.

Table 1. Acoustic Side-Channel Research by Category.

Category	**Authors**	**Significance to Current Work**
Cryptographic	Genkin et al. [1]	Displays the possibility of acquiring a cryptographic key through low bandwidth acoustics.
	Anand & Saxena [13]	Establishes the acoustic side-channel as a subset of the vibration side-channel.
	Walker & Saxena [15]	Further references vibration side-channel in realistic attack success rates.
Classification	Salamon & Bello [19]	Proposal of CNN for non-speech sound classification.
	Vaswani et al. [20]	Proposes the first transformer model for context-based classification.
	Gong et al. [9]	Proposes an AST as a highly accurate method of image-based audio classification.
	Backes et al. [28]	Proposes a model to classify n-grams from supersonic frequencies produced by dot matrix printers.
Passive	Asonov & Agrawal [4]	First modern attack aimed at reconstructing keystrokes.
	de Gortari Briseno et al. [3]	Acquiring acoustic signals representative of data from invisible alterations in printed documents.
	Guri et al. [2]	Acquiring bits representing data from system fan acoustic emissions.
	Halevi & Saxena [21], [22]	Introduction of "hunt & peck" and "traditional" typing styles for analysis.
	Zhuang et al. [23]	Review the attack and implement ML for password identification.
Active	Cheng et al. [5]	First weaponization of the active component to capture Android unlock patterns.
	Nandakumar et al. [6]	First implementation of active acoustics for physical tracking.
	Zhu et al. [24]	Introduction of TDoA component used later in echo analysis.
	Nandakumar et al. [8]	Application of FingerIO analysis model to track location in a large area.

CHAPTER III

METHODOLOGY

This study seeks to create an active class attack by expanding the framework proposed by Cheng et al. in a more generalized setting against a laptop computer [5]. Specifically, the goal is to obtain 12 character passwords typed by a user on a keyboard by applying classification model trained on active acoustic captures of individual keys. The approach involves an experimental setup consisting of a laptop computer system and a two-speaker/three-microphone configuration to modify the active sonar component of the attack. A local application was developed that displays a simple text box and stores passwords entered within a database.

Data collection occurs in two steps. First, training/validation data are collected by creating audio files for each key, while the speaker/microphone setup generates and captures an orthogonal frequency division multiplexing (OFDM) signal. These data are converted to spectrogram images and used to train a CNN in audio classification and use a subset of data to test the accuracy of the model [18]. Second, the small password data set is entered into the local database, and the audio recordings are run through the trained model to infer the labels for each typed key. A single user conducts both phases of the data collection using a variety of approaches to be discussed in Section 3.3. Following initial

results, a second experiment was conducted using the same methodology. The second experiment collected samples for a subset of keys and inferred labels on only a single key to test the model's ability to successfully identify single keys with unseen data.

3.1 Research Objective

As stated in Section 1.1, the primary research objective of the study is to extend an active acoustic SCA to a laptop system. The aim is to generalize an attack of the class proposed by Cheng et al. and to test its efficacy against laptop systems using ML [5]. This informs an attempt to answer whether their model can be extended into a three-dimensional space. In addition, this study attempts to answer what specific implementation will allow this type of attack by designing an environment consistent with suggestions by Nandakumar et al. for extending their two-dimensional framework [8]. Furthermore, obfuscation techniques and potential defenses against attacks in this class are proposed in an effort to answer whether this attack could be performed in a real-world setting and what steps could be taken to prevent it. By answering these questions, the study hopes to contribute to the growing body of knowledge on the security vulnerabilities of laptop systems and to provide insight into how such systems can be better protected against potential attacks.

3.2 Research Approach

This thesis takes a quantitative approach that reuses the basic framework proposed by Cheng et al. [5]. However, specific changes in the setup are applied, particularly with the

number of microphones and the analysis method, to generalize the active acoustic SCA in three dimensions to a laptop system. Data are collected using a combination of microphones that record reverberations of OFDM signals produced by the computer's speakers and reflected by moving fingers in the environment. The OFDM signals pulse sound for 64 samples and are silent for 200 samples, creating a sound frame of 264 samples that lasts, with a sampling frequency of 44.1 kHz, 5.5 milliseconds [5].

FingerIO and SonarSnoop both use OFDM acoustic signals for two main reasons [5], [6]. First, OFDM signals can effectively combat the effects of environmental obstacles by using a large number of subcarriers, each with a different frequency, to transmit signals in parallel. This helps spread the signal energy over a wide frequency range, making the signal stronger and less susceptible to physical interference. For example, Cheng et al. recorded a vector of 64 subcarriers that spanned 375 Hz each [5]. Second, OFDM signals are well suited for use with digital signal processing techniques, making it easier to implement signal processing algorithms to detect and analyze physical movements. For example, the Inverse Fast Fourier Transform (IFFT) algorithm is applied to the vector created by Cheng et al. and is used to generate the waveform [5].

Figure 6 illustrates the proposed active acoustic attack framework divided into (1) data collection and (2) analysis. Since the attack is an active framework that seeks to apply miniaturized sonar technology, there is little concern about the acoustic emanations produced by typing on the keyboard itself; this represents a passive approach that has been adequately covered in the related literature [1], [4], [23], [24].

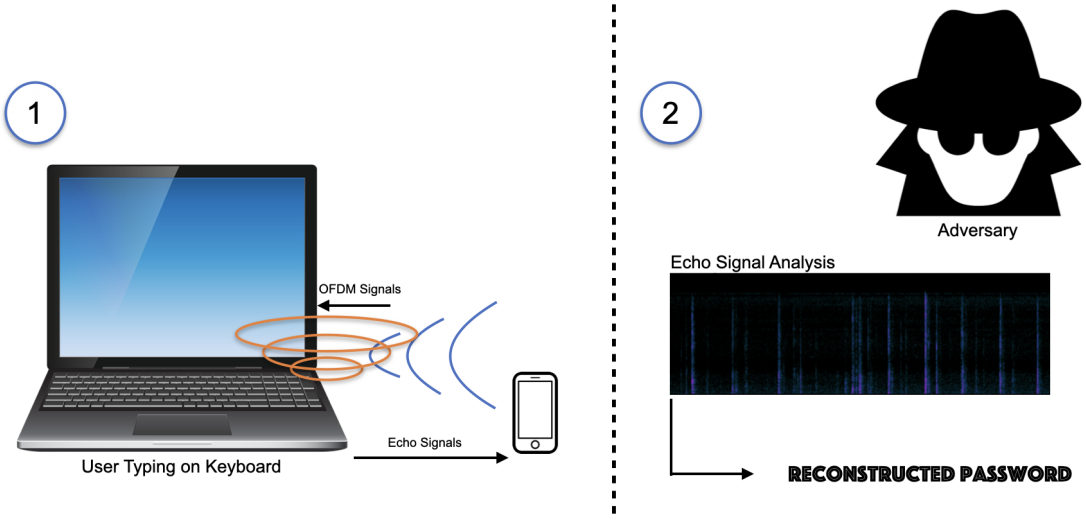

Figure 6. Active Acoustic Attack Framework.

Figure 6 illustrates that microphones record OFDM signals produced by speakers and calculate the degree of movement and location on the keyboard using differences in the signal relative to a static signal without movement after feature extraction [5]. To quantify the differences in the signals, a classification model is applied via a CNN to learn spectrogram representations of active acoustic emanations of individual pressed keys [18].

3.3 Experimental Setup

The experimental setting is a sound-controlled environment. As data are collected for both the training and testing phases, the individual researcher is the only person present. The physical setup involves a single 2023 MacBook Pro that uses a traditional scissor mechanism for the keyboard, and the laptop is placed on a desk [29]. The laptop has left and right internal speakers and both speakers emit a predefined OFDM aural

signal, exemplified in Appendix B, which pulsates in the 18-20 kHz range, which is above the upper limits of human hearing [5], [6]. This file was created using a Python script following the steps described in FingerIO [6]. There are a single Apple iPhone and Apple iPad placed on either side of the keyboard to the front and rear of the laptop. The reason behind using two devices, as Nandakumar et al. point out, is that two microphones allow tracking movement in two dimensions [8]. The mobile devices record uncompressed audio, allowing the capture of frequencies above the range of human hearing. For the purposes of this thesis, an iPhone and an iPad represent common devices that a user may have at the workstation. The experimental setup is shown in Figure 7.

The microphones are set to sample at 44.1 kHz to support analog sound waves up to 22.05 kHz [5], [6]. Cheng et al. use the Nyquist theorem to justify this sampling frequency, as it states that a sampling frequency should be double the maximum observed rate [5]. Although the SonarSnoop framework observed frequencies up to 24 kHz, recreating the SonarSnoop sampling frequency was not possible due to technical limitations of the iPhone and iPad. However, a sample rate of 44.1 kHz could be used and that was sufficient to observe frequencies in the 18-20 kHz range [5].

There is also a Sennheiser MKE 600 supercardioid microphone suspended one foot above the MacBook Pro that is connected via a third-generation Focusrite Scarlett 2i2 audio interface. There are two reasons for including a third microphone. First, the sonar component introduced by Nandakumar et al. has an operational distance of 1 meter, so the distance is theoretically optimal for this experiment [6]. Second, they mention that a minimum of three microphones are needed to sense movement in three dimensions [8].

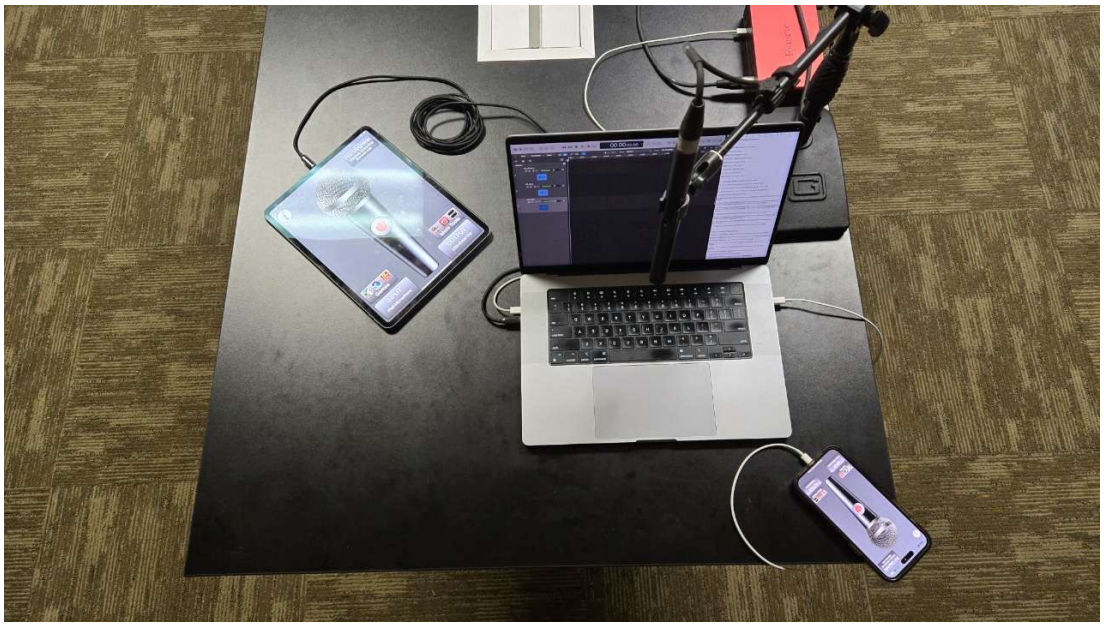

Figure 7. Experimental Setup for Aural Data Collection.

Unlike the attack proposed in SonarSnoop, where active sonar detects motion in two dimensions on a screen, the proposed attack will need to detect motion in three dimensions as the user types on the keyboard [5].

The keyboard plane represents the x and y axes, and the various keystrokes themselves create a z axis. Despite the likelihood that all three microphones will be needed to successfully train a model for finger tracking in three dimensions, the experiment initially uses individual and combined microphone spectrograms for training [8]. Both mobile devices are connected via cables to the laptop directly, and the microphone is suspended and connected to the Focusrite, which is connected to the laptop via a USB-C cable. Using the macOS Audio MIDI Setup utility, an aggregate audio device is created that combines all three microphones into a single logical device. That

logical device is imported into Apple's Logic Pro digital audio workstation, where three tracks are created: one for each microphone. The tracks for the iPhone, iPad, and MKE 600 are labeled "01_iPhone", "02_iPad", and "03_MKE", respectively. The audio tracks are recorded and exported both as individual WAVE files and as an aggregate file containing all microphone sources.

The reason for using Logic Pro is two-fold. First, Logic Pro enables the completion of export tasks extremely efficiently, so audio files can be saved quickly between sets of data collection. Second, Logic Pro is exceptionally stable on macOS. Initially, the free and open source digital audio workstation Audacity was to be used for data collection. However, it was quickly discovered that the macOS version of Audacity was particularly unstable and continuously crashed when the audio input device was imported. To avoid the additional overhead of having to export audio files from another machine, the decision was made to use Logic Pro, which proved to be much more stable. All data collection functions performed using Logic Pro can also be accomplished using GarageBand, Logic Pro's free counterpart, or Audacity on a Windows system. Both are particularly stable on their respective operating systems and are available free of charge.

The experimental interface for collecting data to train the CNN, further discussed in Section 3.4, is a simple text document that contains reminders about starting the setup and a field to type a single key for each audio capture. Once the training/validation data set is complete and the model is trained using those data, a simple local application prompts the user to enter a password in a text box while the audio setup is recording, shown in Figure 8. When the password is entered, the page refreshes and displays a blank prompt, and the

entered password is stored in a database in the order in which it was entered. Those audio files are entered into the model in order to infer labels and reconstruct passwords.

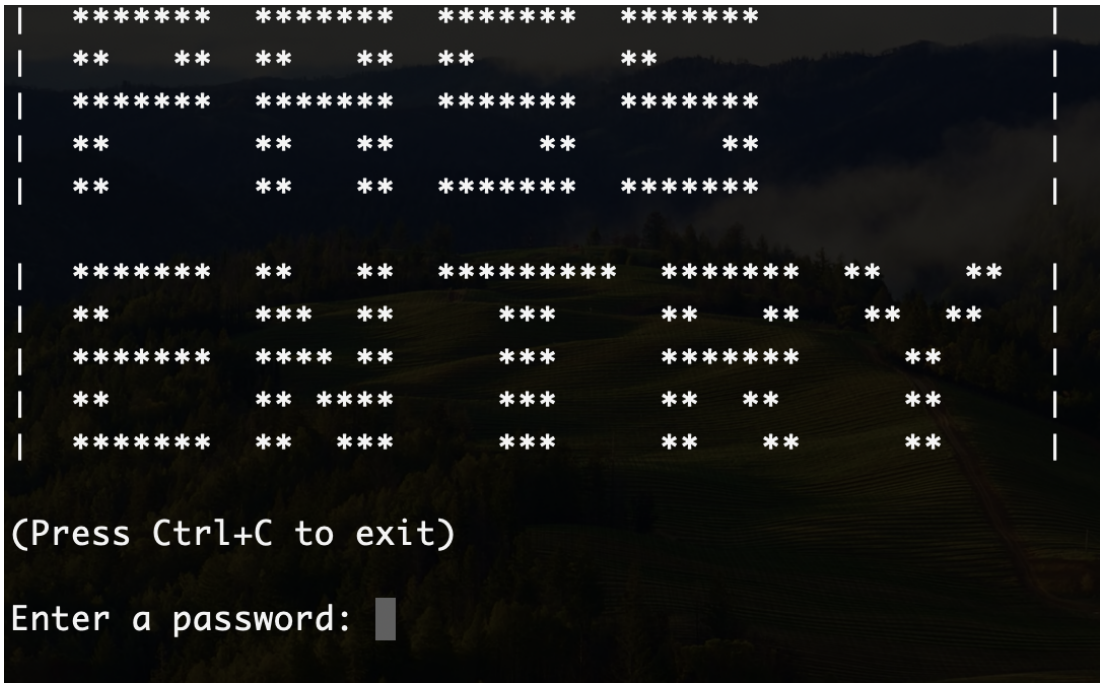

Figure 8. Application for Password Entry to Database for Analysis.

To comply with the latest NIST recommendations for the minimum length of passwords and to have a constant value, all entered passwords are exactly twelve characters long; contain a mixture of uppercase letters, lowercase letters, numbers, and special characters; and are defined in advance. The passwords entered are pulled, using the Perl scripts shown in Appendix A, from the readily available word lists rockyou.txt and common.txt found in default Kali Linux distributions, as long as they meet the aforementioned requirements [7]. A sample of 20 passwords is selected at random. Ten

passwords come from the rockyou.txt file in a fresh Kali Linux installation, and ten come from the common.txt file in the same distribution. Passwords are stored in a document that is easily accessible, and the researcher views them as they are being entered into the web application's text box.

3.4 Data Collection

Upon execution of the experiment, the MacBook Pro plays an OFDM audio file configured to emit OFDM signals at 18-20 kHz throughout the course of data collection [5]. The Python script to generate this file is based on that originally proposed by Nandakumar et al. and is exemplified in Appendix B [6]. Three microphone inputs are recording simultaneously in order to capture the OFDM signals as they reverberate off the fingers in a three-dimensional space. For every key, a five minute recording is created with the key being pressed every two seconds on each odd second. This process is repeated for every key on the keyboard, as well as for Shift combinations of keys, to create capital letters and special characters. In total, (47 * 2) + 1 = 95 recordings are created where 47 is the number of keys on the keyboard, 2 is the number of variations of keys (without the Shift key and in combination with the Shift key), and 1 is the space bar.

There also exist sensitive areas for microphone gain and laptop speaker volume interaction. Too high in both results in feedback, even when dealing with frequencies in the supersonic range. Too little in one or both results in a severely degraded capture of the OFDM signals. Thus, some tuning of the microphone gain and speaker volume is required. Ultimately, setting the MacBook Pro volume level to 75% and the Focusrite gain

to 60 dB provide the strongest capture of the OFDM signal, without causing any audio feedback. The Focusrite gain level can be controlled using the Focusrite Control 2 software (available for macOS and Windows), so consistent volume settings can be applied even across multiple data collection batches. This problem is eliminated entirely by changing the audio output device in Logic Pro, but a real attack may not have the flexibility to reroute the audio as needed.

For each five-minute audio recording, also known as a set of training data, different modes of key entry are performed. For the first minute, hands are placed in a traditional typing position and the user attempts to type the single key normally [21], [22]. In the second minute, the user continues to type the key with the same finger in traditional hand placement, but noise is introduced by moving the hands over the two-dimensional keyboard space. For the third minute, a "hunt-and-peck" typing style is applied with one hand only [21], [22]. For the fourth minute, the user resumes a traditional typing position, but applies a greater force when typing the key to incorporate the variable typing force that was part of Asonov and Agrawal's model [4]. For the last minute, the user resumes a traditional typing position and types with a passive amount of force as in the first minute. This process is repeated for all 95 recordings. Table 2 breaks down the data collection steps for the training data set. Figure 9 shows Logic Pro configured to record a set of data.

Each audio file is in its own folder named for the recorded key, and this serves as the label for the classification model. Once all the audio files are created, a Python script is applied to normalize the data. This script first converts the stereo file created by mobile devices to a 16-bit mono file for storage optimization purposes.

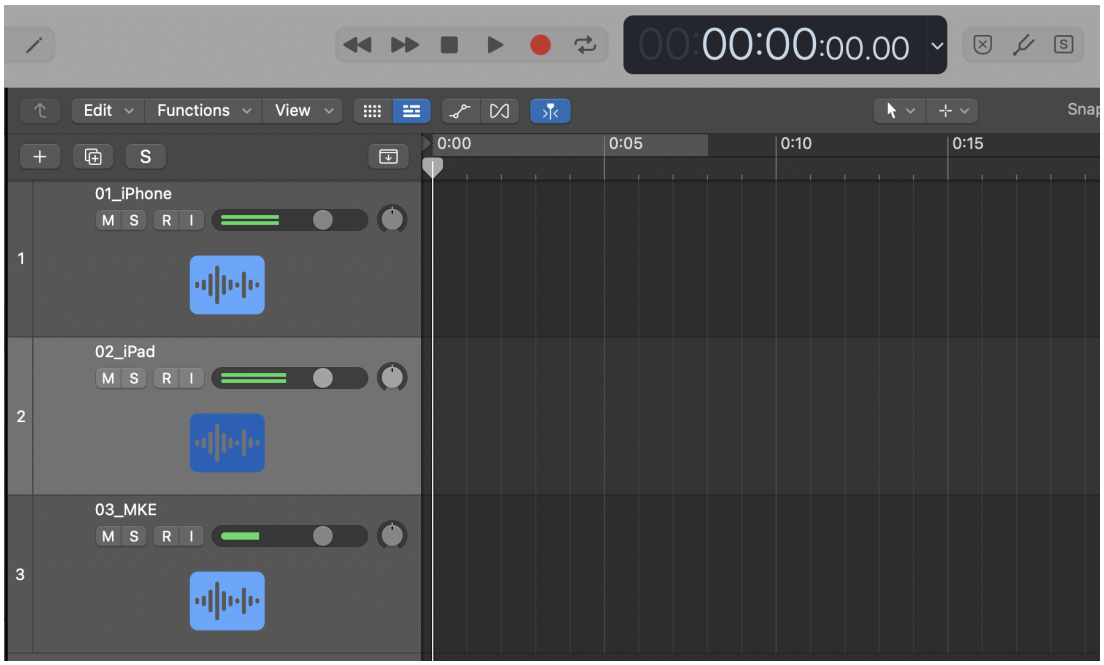

Figure 9. Logic Pro Configured to Record.

It also processes the WAVE file into a floating point representation and applies a high-pass filter to remove frequencies below 16 kHz. The script creates a backup of the normalized five-minute audio file and saves it in a backup subdirectory. Another Python script is applied to split the five-minute audio file into a collection of 150 two-second audio files that are split on even seconds. This makes entering keys on odd numbers during data collection critical; the user should not be entering data where a split is applied to the larger file. This process is applied to all four recordings that have been exported, creating 150 iPhone recordings, 150 iPad recordings, 150 MKE 600 recordings, and 150 aggregate recordings for a total of 600 WAVE files per key. Across all 95 iterations of this process, a total of 57,000 WAVE files are created over just under eight hours of total data collection.

Both Python scripts used to normalize and split are exemplified in Appendices C and D, respectively.

Table 2. Steps to Training Data Collection.

Step Number	**Action**
Step 1	Open the MIDI setup utility and enable both peripheral recording devices.
Step 2	Ensure there is a signal from all 3 inputs on the digital audio platform (either Logic Pro or Audacity).
Step 3	Start OFDM recording from laptop and make sure that the laptop is not muted.
Step 4	Prime all three tracks. Make sure track slider is at zero and header is set to "time". Have a text-box window accessible.
Step 5	Press record, click text-box window, type key every two seconds where $x\%2 \neq 0$, if x is the current second.
Step 6	Finish recording, bounce full track, 01, 02, and 03, in that order. Make sure destination folder is what letter was typed.
Step 7	Navigate to that directory on the command line. Make sure that the virtual environment is active.
Step 8	Run python3 normalize.py on the four exported tracks.
Step 9	Run python3 split.py on the four normalized tracks.
Step 10	Repeat for each letter until there are 600 samples in each folder.

After collecting all training/validation data, more normalization is performed to prepare the data to train the classification model, discussed in more detail in Section 3.5. Once the data set is input into the model, the trained model is used to infer labels in an attempt to reconstruct the captured passwords entered into the database. The same experimental setup is applied for password entry, and recordings are created using the three microphones. The passwords are in a document that is clearly visible and are entered

into the database sequentially. Each password is entered a total of six times. The first three entries are with traditional hand placement, and the second three use "hunt-and-peck" typing [21], [22]. The following describes three methods of differentiation in password entry that are used for both traditional and "hunt-and-peck." For the first time, the password is typed with a five-second pause between each key press. The second time, the password is typed with a two-second pause between each key press. The third time, there is no pause between key presses, and the password is entered at a medium speed of approximately thirty words per minute (WPM).

The audio is exported from Logic Pro and stored locally in a manner similar to the data set used to train the model. After each set of six entries of a password, the microphone recordings are saved for analysis, the experimental setup is reset, and the process begins again for the next set of six entries. Each audio file is manually divided into six files for each password entry. In total, twenty audio files are created that are then divided six ways to form 120 distinct audio files. Like with the data set for fine-tuning, there are four individual microphone recordings for each file (iPhone, iPad, MKE 600, and aggregate). This results in 24 audio files for each password entered and a total of 480 audio files.

3.5 Data Analysis

The first collected data set is used to train a deep learning model using a CNN architecture, based on the audio classification code proposed by Doshi [18]. The model converts WAVE files to tensors derived from spectrogram image representations of the audio and runs those through image-based classification. Several improvements to the

model are discussed in more detail in Section 4.1; however, examples of some preprocessing steps taken in the model include padding/truncating audio files to two seconds, performing a time shift to diversity samples, converting to a spectrogram focused on the 16-21 kHz frequency range, and equalizing the spectrogram to emphasize features [18]. The data set is divided into training data sets (66.66%) and validation data sets (33.33%), and the model is trained. Once the model is trained, the password data set is input into the model to infer labels for each entered key to reconstruct the entire password. The accuracy of the model is computed when reconstructing passwords, and an overall accuracy score is presented for the reconstructed passwords.

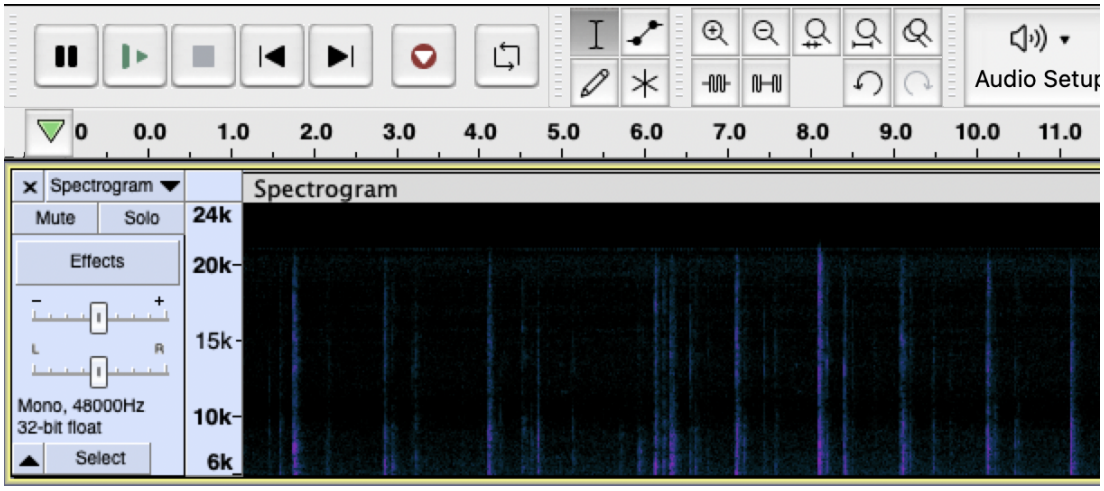

Figure 10. Sample Spectrogram View of Audio File in Audacity.

3.5.1 Keystroke Set for Model Training

Figure 10 shows a spectrogram in Audacity. An Apple iPhone 14 Pro Max was placed next to a 2023 Apple MacBook Pro to record keystrokes at a sample rate of 44.1

kHz. The author typed “The University of South Alabama” with a pause of approximately one second between each keystroke. When the recorded audio file was imported into Audacity, the resulting spectrogram clearly displayed the timing of the keystrokes. At the six-second mark, the space bar is pressed following the entry of “The,” and that feature is clearly visible. Furthermore, overtones of key press frequencies stretch slightly higher than 20 kHz, and that covers the range of 18-20 kHz that is explored due to its presence outside of the upper limits of human hearing. To see features outside the range of human hearing more clearly, a utility within the program can be used to apply a Hanning window to filter out frequencies below 18 kHz.

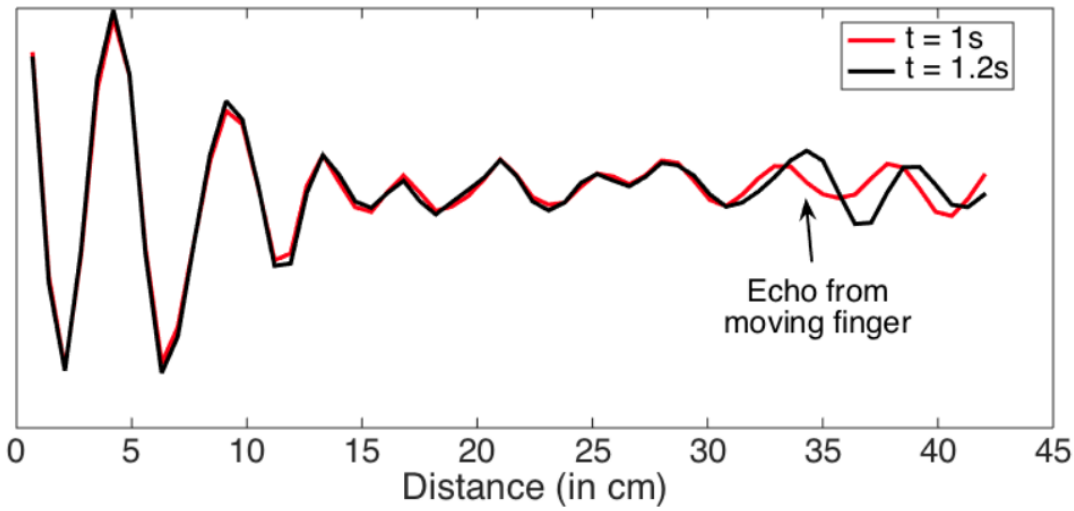

Figure 11. Comparison of Two Echo Profiles.

In previous active acoustic projects, pre-processing involved applying Fast Fourier Transform (FFT) and IFFT algorithms, applying Hanning windows, and using waveform and spectrogram views [5], [6], [8]. Between the normalization applied immediately after

data collection and in the model, all these functions are applied to the training data set, and this can be confirmed by applying a custom script, shown in Appendix E, to view the image representation of the spectrogram before passing it along to the model [9].

The active approach requires sound analysis to find differences in the time arrivals of the echos. Specifically, the OFDM signals produced by the laptop computer will reverberate from various objects in the vicinity of the laptop, and those reverberations, or echos, are picked up by the microphone for analysis. Echoes from any given snapshot in time of the microphone can be extracted to create an echo profile that displays amplitudes that correlate with distances. Nandakumar et al. assert that differences in amplitude, shown by a waveform, represent movement, and their comparison of echo profiles is shown in Figure 11 [6].

To find the exact location of the fingers on a keyboard, the echo profile matrix is analyzed to extract the distance of the moving finger from the microphone [6]. Since three microphones are used, it is theoretically possible to triangulate the location of fingers moving on a keyboard, and the third, supercardioid, microphone enables the ability to monitor a z axis to detect actual keystrokes. Note that keystrokes may be identified when only using the two smartphone microphones on either side of the laptop, so attempts will also be made to reconstruct passwords using only smartphone microphone audio recordings. The pre-processing of the model will create spectrograms of these supersonic audio captures, and the same theoretical analysis is applied. Figure 12 shows an example of a spectrogram from the data set used for image-based deep learning. The difference in other modes of analysis and the classification model proposed in this work is an attempt to

generalize these time differences in echo profiles and identify these features using image-based classification with spectrograms [19].

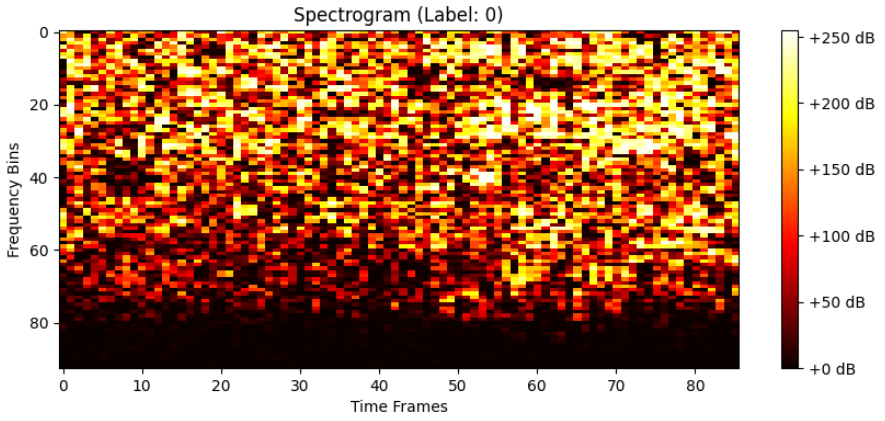

Figure 12. Sample Spectrogram for Lowercase 'a' Key Used for Deep Learning.

3.5.2 Inference

Inference on unseen audio spectrograms is performed by inserting files into the trained classification model. For inference on passwords, they are entered as individual letters, in the manner specified in Table 3. They are also inputted as individual microphone recordings and aggregate recordings of all three microphones. In this scenario, the ability of the model to classify three-dimensional motion is tested, despite previous statements that such a classification is unlikely to be achievable [5], [8]. Specifically, there are 480 password audio files to be entered: 24 aural files per password (since each is entered six times and typed differently each time). Each of those files is broken down into audio files that represent individual letters for each password, and the

entire password audio file is retained as well. Thus, for each entry of a password, there are twelve audio files (one for each individual keystroke). This adds up to 288 audio files for each of the 20 passwords (12 audio files per password * 6 entries of the password * 4 microphone settings) and a total of 6,240 audio files (288 * 20) input into the trained model. However, each password is entered as a batch of twelve audio files.

Table 3. Differentiation in Infer Data Collection.

Entry Number	**Action**
Entry 1	Traditional Typing, 5 seconds between keystrokes. Bounce at 1 minute.
Entry 2	Traditional Typing, 2 seconds between keystrokes. Bounce at 25 seconds.
Entry 3	Traditional Typing, 30 WPM. Bounce at 10 seconds.
Entry 4	"Hunt & Peck" Typing, 5 seconds between keystrokes. Bounce at 1 minute.
Entry 5	"Hunt & Peck" Typing, 2 seconds between keystrokes. Bounce at 25 seconds.
Entry 6	"Hunt & Peck" Typing, 30 WPM. Bounce at 10 seconds.

To test the effectiveness of the single-key inference model, a subset of keys is also collected. This subset includes only lowercase letters 'a', 'b', 'e', 'i', 'm', and 'p'. These keys represent distinct areas on the keyboard, and training a model on only these keys would allow for simpler inference to test the model's accuracy in a more coarse-grained setting.

3.5.3 Effectiveness Evaluation

Another stated goal of this thesis is to reduce the number of guesses required by an attacker to brute force a password. In other words, if we assume that a password of length 12 with uppercase letters, lowercase letters, numbers, and special characters is relatively secure, we would like to see if we can guess values of some password elements in order to lighten the load on the number of passwords needed for brute force. To evaluate this, we use the formula for password entropy.

The entropy formula is calculated as follows: $H = L * log_2 N$, where H is the entropy measured in bits, L is the character length of the password, and N is the number of possible symbols [30]. For our set of passwords, the entropy is $12 * log_2 94 = 78.66$ or 79 bits needed to store the password. The brute forcing of a 79-bit password would take 2^{79} attempts or roughly $6.04 * 10^{23}$. When considering the slow hashes commonly employed for password storage without other information, this is well above what could be achieved with modern computing resources in a reasonable amount of time [31], [32].

However, assuming that we can establish half of the password with a high degree of confidence using the classification model, our problem now is to crack a 6-character password. The entropy for a 6-character password, using the same number of possible symbols, is $6 * log_2 94 = 39.33$ or 40 bits needed to store the password. The brute forcing of a 40-bit password would take 2^{40} attempts or roughly $1.1 * 10^{12}$. Although there are still many attempts needed, brute force becomes much more manageable when only trying to crack a password of 6 characters [31]. Thus, if we can reduce the number of guesses needed to brute force a password by a meaningful amount, we will satisfy our goal of

making progress towards password leakage by taking advantage of the acoustic side channel.

3.6 Limitations

There are two main limitations to this study. First, only 14,100 data points are used to train the deep learning model; that is, 94 labels and 150 samples per label. There would likely be a higher level of accuracy when evaluating the passwords if there were more data points to train the model, but this becomes time-consuming since the data set must be created manually in real-time.

Second, breaking the typed passwords into individual letters to input into the classification model assumes that there is a capacity to extract information from the aural range of frequencies because likely keystrokes would need to be identified. For the purposes of this study, only when a key is typed would be known, and the pilot test, shown in Figure 10 shows that the overtones of the typed keys can be seen in supersonic ranges. However, this is not reliable due to the potential for frequency jamming in the aural range. We will discuss this and other potential defenses against an attack of this class in Section 4.5.1. For the purposes of this study, we assume the ability to successfully break passwords into separate keys so that a trained model can infer labels on individual keys.

CHAPTER IV

RESULTS

In this section, we give an overview of the classification model's construction process including details on hyperparameter tuning, testing accuracy, and validation efforts. We also discuss model inference on password recordings and task correction results on a subset of lowercase keys to demonstrate a trend towards inference success.

4.1 Classification Model Construction

Several pre-trained models are currently available online through repositories such as HuggingFace, but much of the ML application to aural data is in sound recognition [16]. Typically, these sounds fall below 8 kHz, so a sampling rate of 16 kHz is used for various available projects, including those applying more robust transformer models [5], [33]. Originally, the intention was to fine-tune the individual keystroke data set on a pre-trained model such as the AST, which was shown to have excellent results on data sets such as AudioSet and Speech Commands V2 [9]. However, like other pre-trained models, the AST was trained on audio sampled at 16 kHz. Since the supersonic keystroke data set was sampled at 44.1 kHz, it could not be used to fine-tune AST or any other readily available audio classification model.

Due to the availability of various audio classification models, we decided to customize the code to train on the supersonic keystroke data set within a CNN framework [18]. In its raw form, the model selected took a manifest containing audio file paths as input, applied pre-processing to generate spectrogram images from audio files, split the data into train/test subsets, trained the model using the data in a CNN, and output results of the test set in terms of accuracy and loss. Holistically, the final customized model performed the same actions with significant optimizations to each component to better train the supersonic keystroke data set.

4.1.1 Model Optimization

For the sake of model optimization, only the supersonic audio files from lowercase alphabet keystrokes were used. This allowed for faster model convergence over the full data set to obtain an optimized baseline. Furthermore, the complete audio data set includes four audio files for each individual keystroke. These files are the iPhone, iPad, MKE and combined audio from all three sources. Nandakumar et al. assert that three-dimensional motion should only be possible using a minimum of three microphones [6]. A validation set was specified following this process to be used with the full keyboard set, so the loss and accuracy shown in the optimization figures are from the training sets. Furthermore, optimizations were applied prior to early stopping code being added, so the models ran for 500 epochs each.

Table 4. Optimization Phase Results.

Model	Microphones Used	Training Accuracy	Test Accuracy
1	Individual & Combined	∼40%	34%
2	Combined	∼70%	56%
3	Individual & Combined	∼90%	71.05%
4	Combined	∼97%	90.47%
5	Individual & Combined	∼95%	77.17%
6	Combined	∼98%	88.71%

Since individual microphone recordings could possibly cause significant misclassification in the full data set, a second supersonic keystroke data set was created that contained only the combined audio files [8]. The two data sets were applied alternatively for each set of optimizations made with the full data set used for training first and the combined data set used for training in the subsequent model. Therefore, Model 1 and Model 2 were trained on the data set without optimizations, Model 3 and Model 4 were trained on the data set with the first round of optimizations, and Model 5 and Model 6 were trained on the data set with the second round of optimizations. Model 6, trained on the combined microphone data set, was used as the final classification model with only minor subsequent optimizations applied in attempts to improve the accuracy of inference. Table 4 shows the results of the optimization process run on the lowercase letter subset.

Humans hear sounds logarithmically rather than linearly. Consequently, a Mel scale spectrogram is typically used for audio classification tasks due to its emphasis on aural frequencies below 8 kHz [18]. However, this reduces the emphasis on supersonic ranges, as shown in Figure 13, which is counterproductive to our goals, since our data set is entirely composed of frequencies in the 18-20 kHz range.

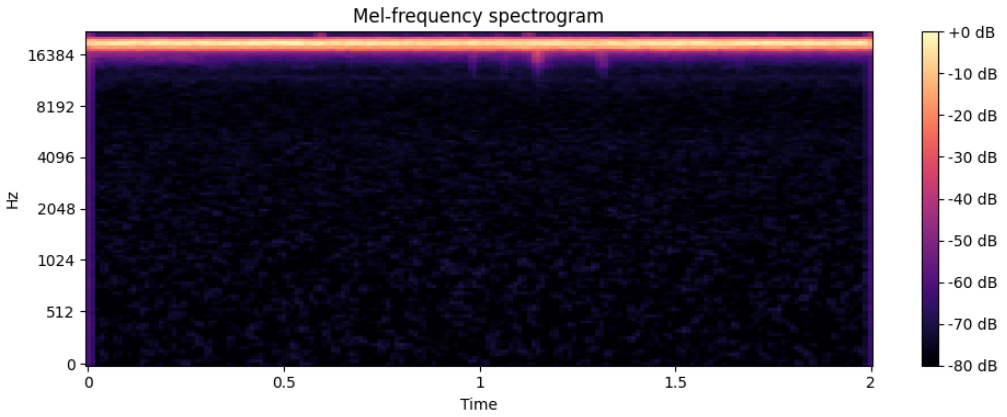

Figure 13. Supersonic Audio Displayed as a Mel Spectrogram.

Figure 14 shows the default model only changed to sample at 44.1 kHz used on the optimization subset of the supersonic keystroke data. The default number of epochs run was 500, which was too high for this data set and likely resulted in overfitting. However, the first model graph shows convergence around the accuracy of 40%, with the test subset achieving an overall accuracy of 34%.

Model 2, which used only the combined audio files, showed a significant improvement and converged around the accuracy of 70% during training with the accuracy of the test set of 56%. At this point, we began making optimizations for Model 3, taking note of the model's superior performance on the combined audio data set. The primary optimization at this stage was to generate linear spectrogram images rather than a Mel spectrogram from the audio files during model preprocessing. Once again, Model 4, using the combined audio data set, outperformed Model 3 which used the full data set. Model 3 converged around 90% accuracy with test set accuracy of 71.05%. Model 4, shown in Figure 15 converged around 97% accuracy with a test set accuracy of 90.47%.

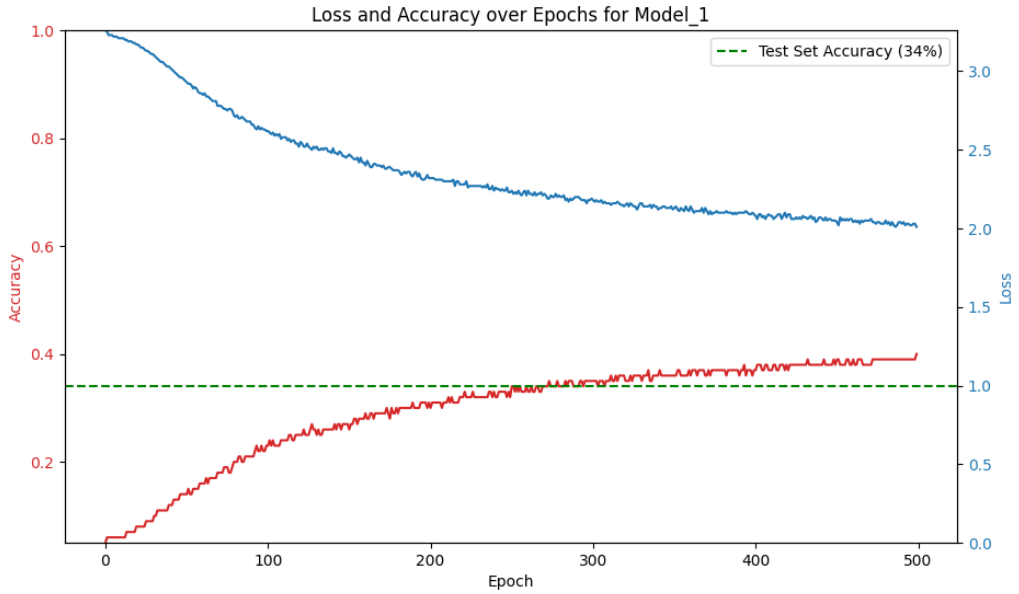

Figure 14. Model 1 Using Mel-Scale Spectrogram.

The final set of applied major optimizations consisted of using a filtered linear spectrogram focusing on the 16 - 21 kHz frequency range (shown in Figure 16), applying equalization to highlight areas of change over the course of the spectrogram, and adding a fifth convolution block to double the input features to 128. Model 5 showed a significant improvement over Model 3, both of which used the full audio data set. Model 5 converged around the accuracy 95% during training with a test set accuracy of 77.17%. Model 6, shown in Figure 17, converged around 98% accuracy during training with test set accuracy of 88.71%. This model was chosen as the primary model moving forward, and its results were displayed in a graph entitled “Lowercase Model.”

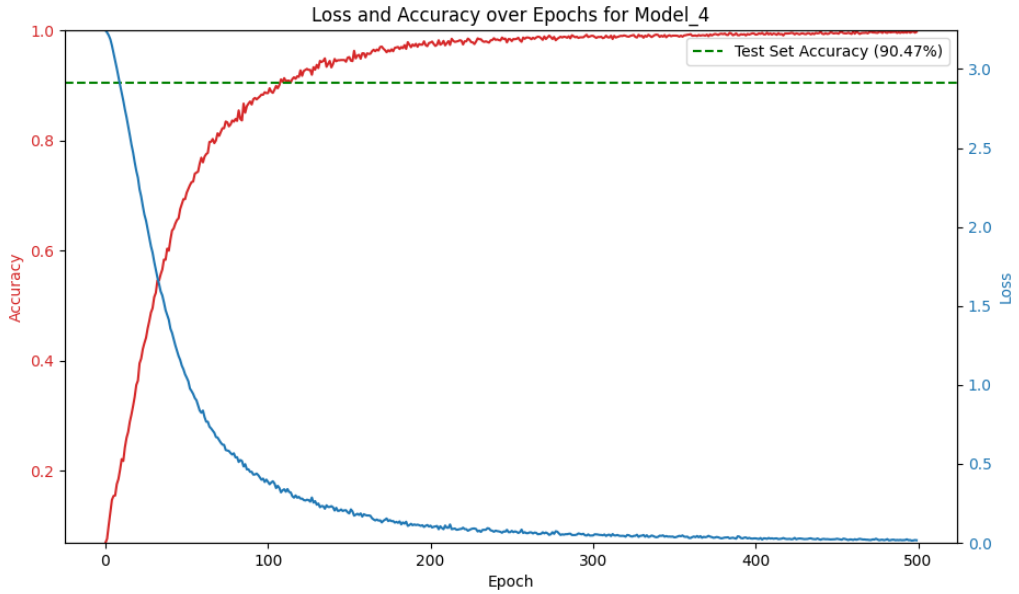

Figure 15. Model 4 Using Linear Spectrogram.

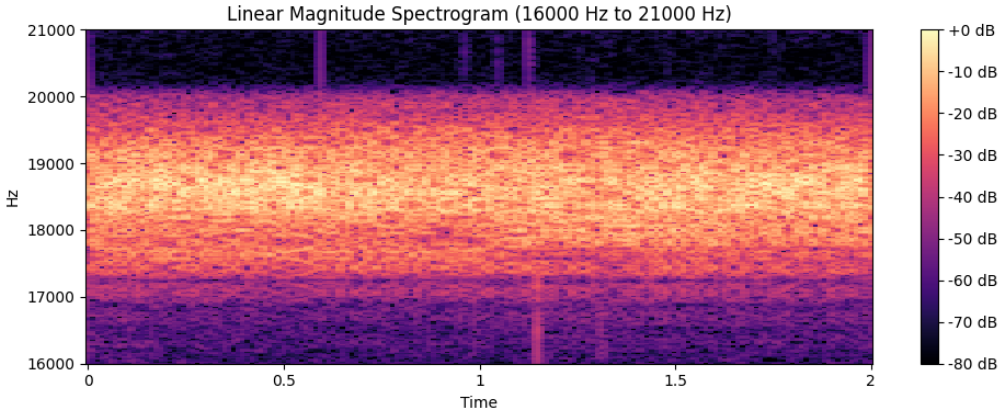

Figure 16. Supersonic Audio Displayed as Zoomed Linear Spectrogram.

4.2 Model Results and Accuracy

Model 6 was chosen as the classification model to evolve and only smaller optimizations were applied once the entire data set was used.

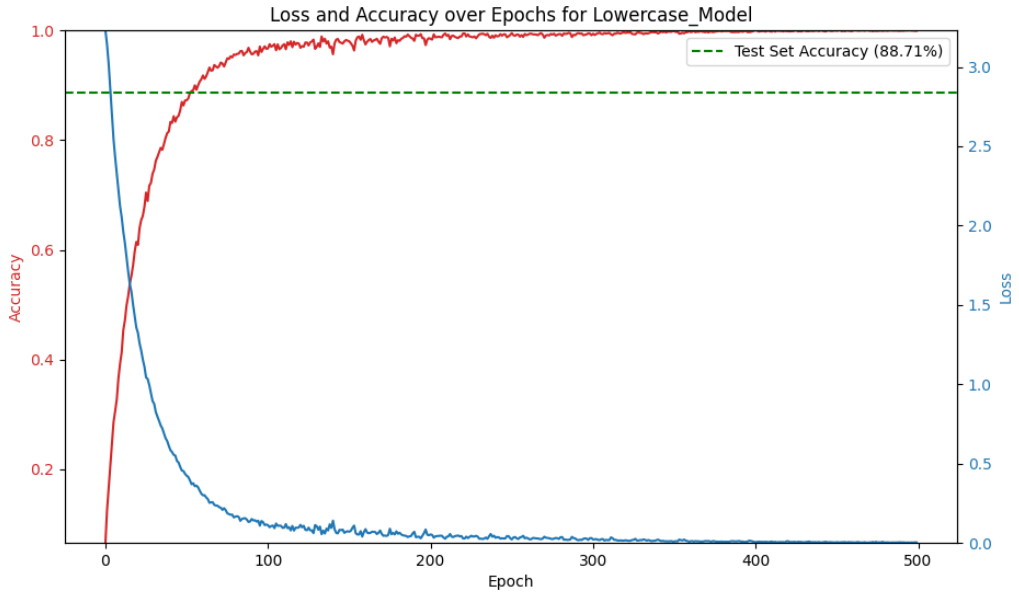

Figure 17. Model 6 Using Final Set of Major Optimizations.

Furthermore, due to consistently superior results during the optimization phase, we decided to use the combined audio file data set going forward rather than the data set also containing the individual microphone recordings. Checkpoints of the model's accuracy were taken with only lowercase letters used, the entire alphabet (incorporating the shift key for capital letters), alphanumeric keys, and the entire set of alphanumeric keys with special characters.

The model trained on the lowercase subset, shown in Figure 17, converged around 98% accuracy during training with test set accuracy of 88.71%. This model was also Model 6 during the optimization phase, so it shares its result with that shown in Section 4.1.1. The model trained on the entire alphabetic subset, shown in Figure 18, converged around 95% accuracy during training with a test set accuracy of 80.23%. The model trained on the alphanumeric subset, shown in Figure 19, converged around 90% accuracy

during training with test set accuracy of 83.71%. Finally, the full model, shown in Figure 20, converged around 95% accuracy during training with test set accuracy of 83.2%.

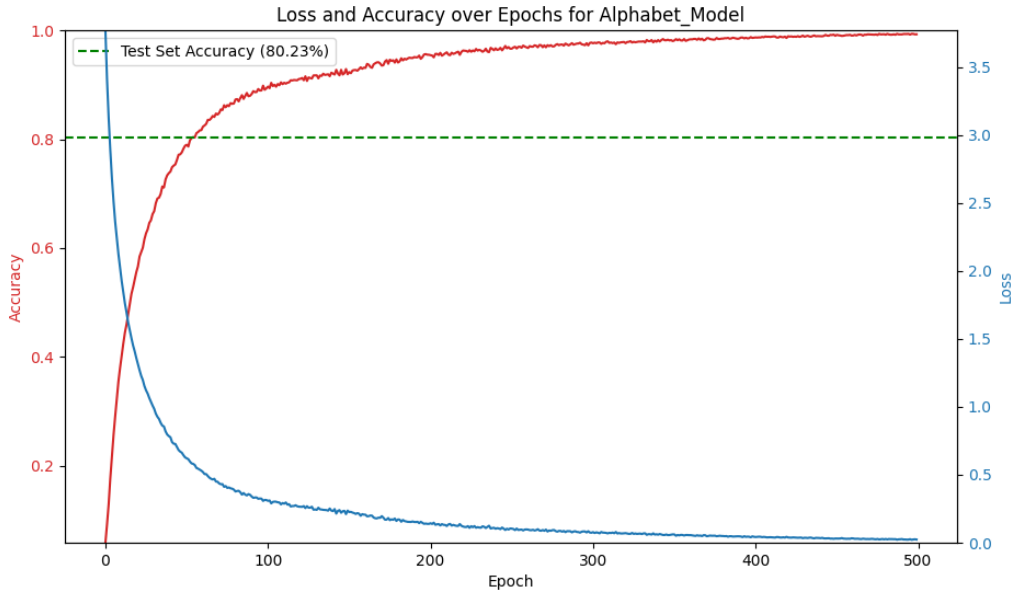

Figure 18. Model 6 with Only Alphabetic Characters Typed.

Further optimizations were added to the complete model in an effort to improve accuracy during testing. First, a sixth convolution block was added for a deeper pipeline to accommodate the increased number of labels in the full data set. Next, a validation set was defined and its results for each epoch were used to assess accuracy and inform early stopping training. The split between the training, validation and testing sets was as follows: 70% for the training set, 15% for the validation set, and 15% for the testing set. However, holistic accuracy, especially with the addition of multiple keystrokes for a character (capital 'A', for instance), does not necessarily indicate how well a model is performing.

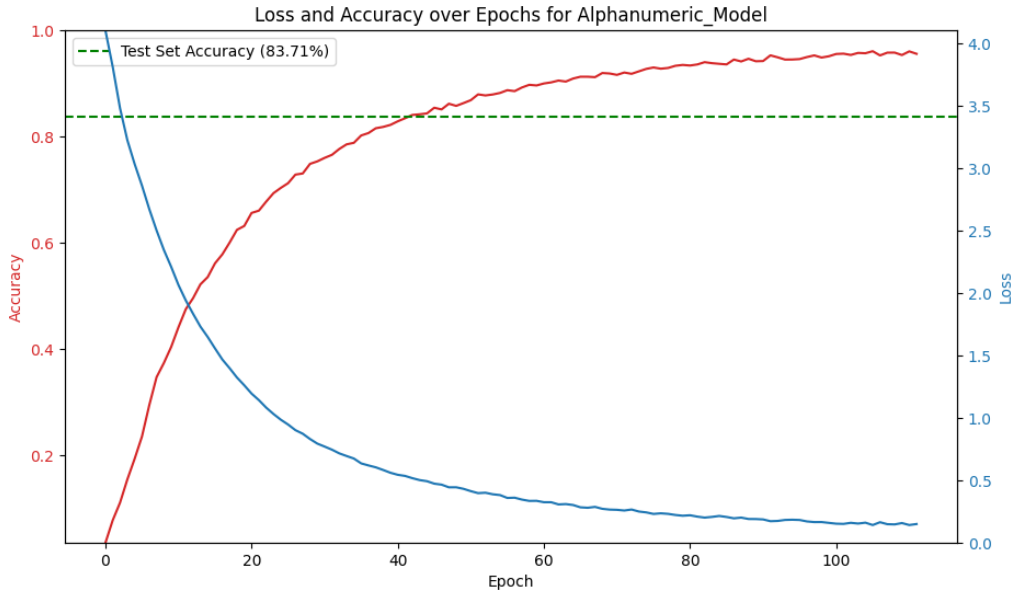

Figure 19. Model 6 with Alphanumeric Characters Typed.

Therefore, we first added metrics for precision, recall, F1-Score, and support for each class at the end of the test set. The entire report was included at the end of the model results along with the complete accuracy of the model in the test set. The confusion matrix for the final model, showing an optimal training/testing process, is shown in Figure 21.

4.3 Password Classification Results

The environment used for the inference of password files used the original script model functions and imported the trained model weights. Once the audio files were captured and stored, a manifest file was created with paths to the files (much like the training data), but without attached labels.

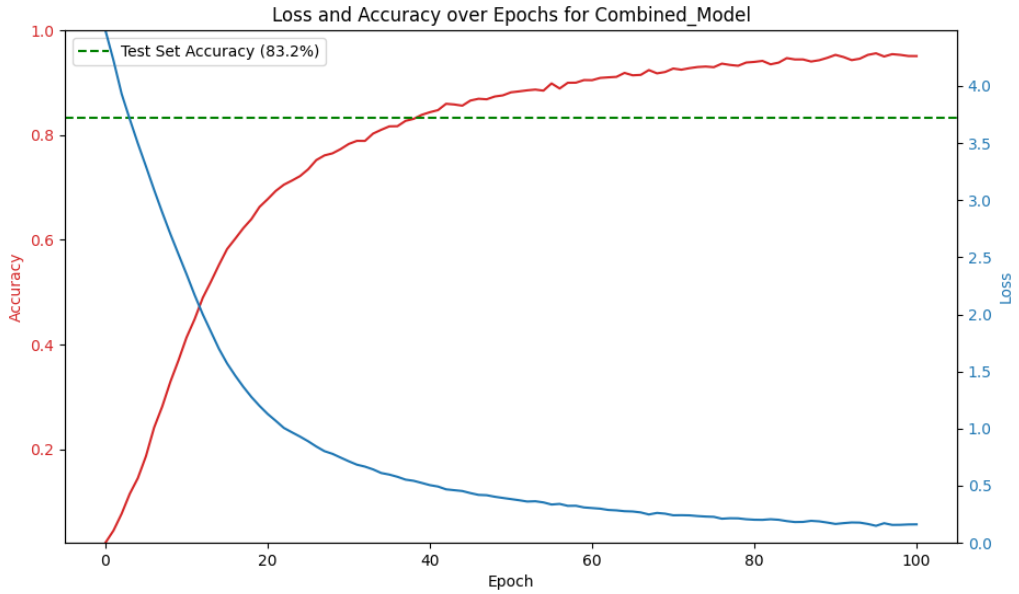

Figure 20. Model 6 with All Keys Included.

The files contained in that manifest are then run through an inference function, similar to the test function in the original script, that simply outputs the predicted label for each audio file. The predicted labels are concatenated to create a string, and the predicted password is output.

4.3.1 Inference Optimization

The inference script required that passwords be split prior to inference. However, the best known method for determining individual keystrokes was looking at the aural frequency range to split the audio files based on the actual keystrokes. However, all frequencies below 18 kHz are eliminated during audio file preprocessing, and looking into the audible range defeats the purpose of being able to conduct this attack completely in the supersonic frequency range without detection or obfuscation.

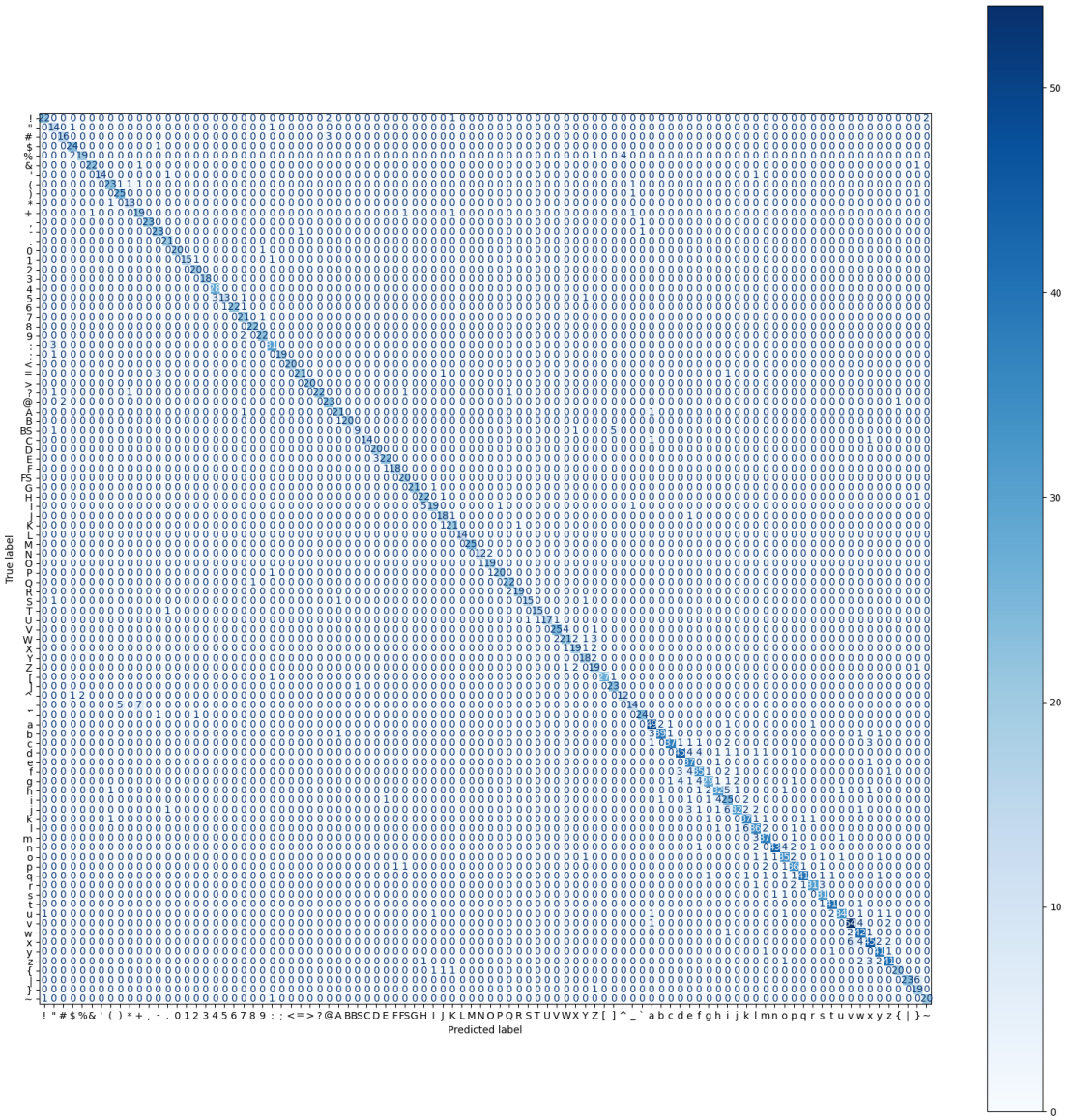

Figure 21. Full Model Confusion Matrix.

4.3.2 Initial Inference Results

For inference, the data were prepared similarly to the training/testing data set. As mentioned in Section 3.5.2, each password was entered in five- and two-second intervals as well as about 30 WPM. To test that the infer script was working properly, we started

with the two-second interval password entry. The reason behind this decision was that it was the most easily preprocessed data set using the normalize.py and split.py scripts, and the same preprocessing was applied to the train, validate, and test data sets. Initially, the infer script test was not promising because the inference did not return the expected results for the passwords entered, and this is shown in Figure 22. The first password tried was "autodiscover," and the trained model did not infer the correct labels for any key positions in the password. The expected resulting array would have been the following:
[a, u, t, o, d, i, s, c, o, v, e, r]. However, the actual resulting array was the following:
[N, #, *, (, #, z, z, *, #, #, 5, ( ], shown in Figure 22.

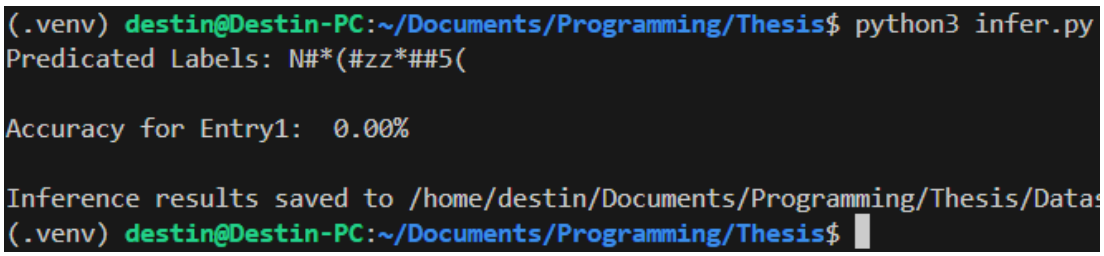

Figure 22. Full Model Inference Prediction for "autodiscover".

4.4 Proof of Concept

Inference that behaves in this manner is possibly due to inconsistencies in the gathering of the training data set or not enough samples per label. The confusion matrix shown in Figure 21 represents a robust model, but inconsistencies in microphone volume and placement, along with limited samples, probably contributed to the model being robust only for audio files collected in that batch without any ability to generalize effectively.

Following this discovery, steps were taken to adjust the microphone using Focusrite Control 2 software, which allowed the gain to be consistently set to 75% for each batch of data collection rather than relying on guesswork when manually turning the gain knob. Furthermore, marks were made on the desk for the iOS devices as well as on the microphone stand so that the microphone distance would be consistent for each batch of data collection. Later steps were taken that included collecting all data at once and training three models, each with a higher number of samples per label, to serve as a proof of concept for coarse-grained finger tracking.

Model optimizations and further data collection attempts were made after the discovery that the labels were not correctly inferred. The data collections were intended to serve as a proof of concept for the task of reconstructing the position of the finger or hand on a keyboard using supersonic aural emissions, reverberations, and observances. The data collected and used to train the model were a subset of the original full set of keys including only lowercase a, b, e, i, m, and p. These keys were chosen because of their distinct positions on a QWERTY keyboard. In these instances, the train/validate/test data set was collected immediately before the inference data set. Each data set was collected in the same session for consistency.

4.4.1 Model Optimizations

The model optimizations were applied in four major steps. Step one involved changing the number of convolutions blocks. Step 2 involved a change between the LFCCs and the optimized spectrograms. Step three involved changing the number of applied FFT

points. Step four involved changing the power applied to the spectrograms. After running optimizations and observing the overall validation accuracy, the following hyperparameter settings were chosen for the highest accuracy going forward: 4 convolution blocks, equalized spectrograms, 2048 FFT points, no hop length, and a power of three applied to all spectrograms. With all of these settings applied, the validation accuracy of the subset model of just ‘a’, ‘b’, ‘e’, ‘i’, ‘m’, and ‘p’ was around 90%. Figure 23 exemplifies a strong confusion matrix for the subset model.

4.4.2 Coarse-Grained Finger Position

In order to test the model’s ability to infer the basic finger position, five additional sets of ‘a’ labels were collected. Each set contained 10 samples that were intended to be inferred. The idea was to use the base model to infer on all five sets and then note the accuracy of the model. Then, the first infer set would include in training the model, and all five sets would again be inferred on by the new model. This process would continue until four infer sets had been included in the base model training and then the fifth infer set accuracy would be compared with previous accuracy levels.

When inferred by the base subset model, sets 1-5 infer ‘a’ with an accuracy of 0%, and this was expected. Inclusion of infer set 1 resulted in the following ‘a’ set accuracies: 2 = 86.67%, 3 = 93.33%, 4 = 93.33%, and 5 = 83.33%. Inclusion of sets 1 and 2 inferred the following accuracies of the ‘a’ set: 3 = 90%, 4 = 90%, and 5 = 86.67%. Inclusion of sets 1-3 of the inferred resulted in the following accuracies of the ‘a’ set: 4 = 96.67% and 5 = 93.33%.

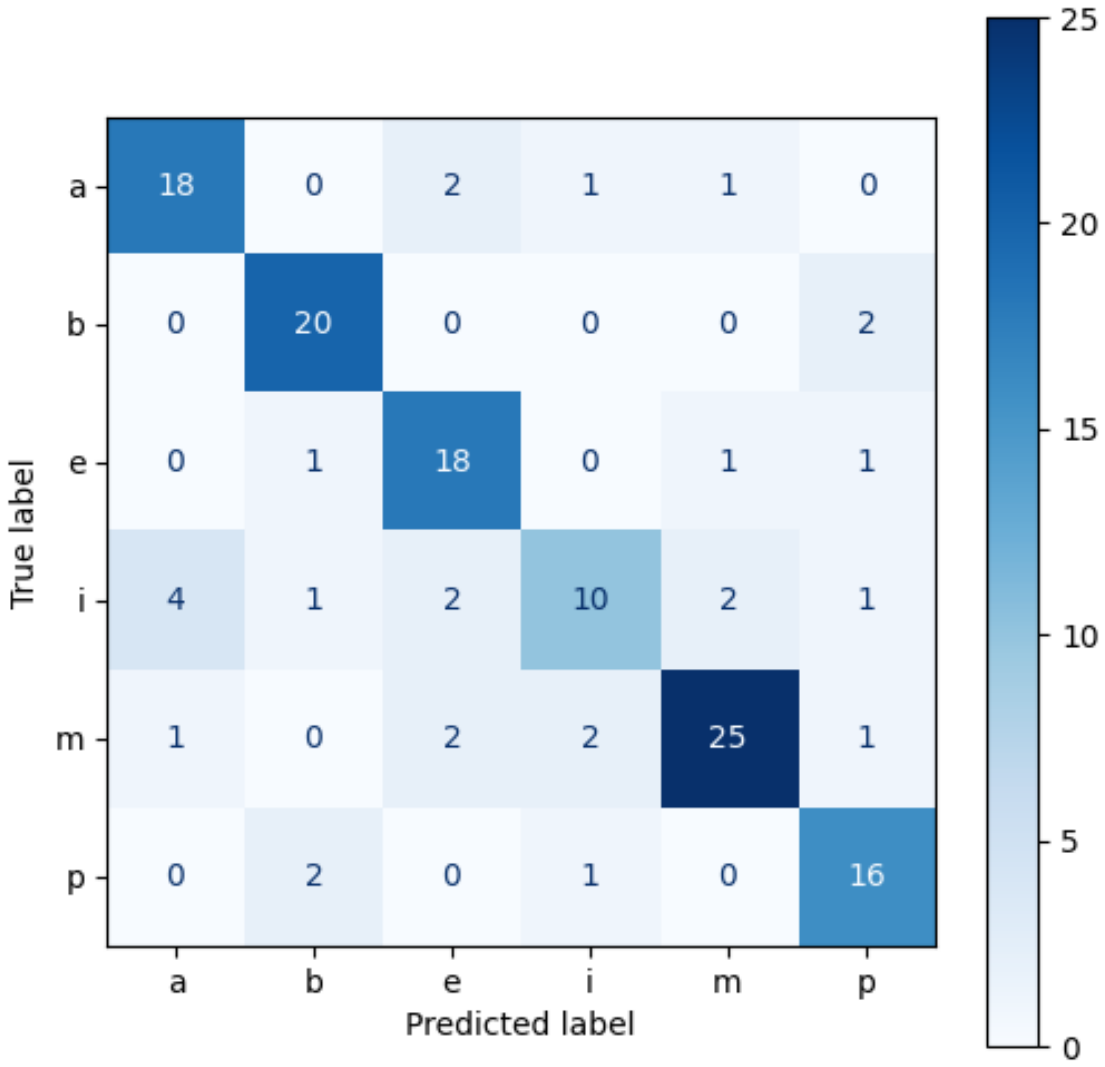

Figure 23. Confusion Matrix for abeimp Subset.

Finally, the inclusion of sets 1-4 inferred resulted in a 100% precision of ‘a’ set 5. These trends are shown in Table 5. This pointed towards more data being beneficial for inference, but was not strong enough to be conclusive due to the possibility of bias in the model.

Table 5. Subset Accuracy Inferring ‘a’ Key Sets.

‘a’ Set Included	**Set 1**	**Set 2**	**Set 3**	**Set 4**	**Set 5**
None	0%	0%	0%	0%	0%
1	100%	86.67%	93.33%	93.33%	83.33%
1-2	100%	100%	90%	90%	86.67%
1-3	100%	100%	100%	99.67%	93.33%
1-4	100%	100%	100%	100%	100%
1-5	100%	100%	100%	100%	100%

In order to point towards the necessity of more data, a final collection was carried out in order to collect more data points on the limited number of labels. The labels included in this new data set collected for training were still ‘a,’ ‘b’, ‘e’, ’i’, ‘m’, and ‘p’, and 450 samples per label were collected. The idea was to collect all of the training and inference data points in the same session to reduce the likelihood that external factors (such as microphone placement) impact the results. For training, three models were created. The first model was trained using the first 150 samples for each label, the second was trained using the first 300 samples for each label, and the third was trained using all 450 samples for each label. An inference set of 30 samples was collected for each label. The idea was to perform inference using each model and compare the accuracies of each label.

All three models had validation accuracies greater than 90%. The inference set ‘a’ had an accuracy of 13.33% for Model 1, 96.67% for Model 2 and 80% for Model 3. The inference set ‘b’ had an accuracy of 6.67% for Model 1, 40% for Model 2, and 33.33% for Model 3. The inference set ‘e’ had an accuracy of 96.67% for Model 1, 60% for Model 2, and 73.33% for Model 3. The inference set ‘i’ had an accuracy of 13.33% for Model 1, 10% for Model 2, and 56.67% for Model 3. The inference set ‘m’ had an accuracy of 40%

for Model 1, 26.67% for Model 2, and 46.67% for Model 3. The inference set 'p' had an accuracy of 86.67% for Model 1, 83.33% for Model 2, and 93.33% for Model 3. The inference set containing five samples each of the six labels had an upward trending accuracy of 23.33% for Model 1, 33.33% for Model 2 and 46.67% for Model 3. These trends are exemplified in Table 6.

Table 6. Key Set Accuracy for Increasing Samples.

Infer Set	**Model 1**	**Model 2**	**Model 3**
'a' Key	13.33%	96.67%	80%
'b' Key	6.67%	40%	33.33%
'e' Key	96.67%	60%	73.33%
'i' Key	13.33%	10%	56.67%
'm' Key	40%	26.67%	46.67%
'p' Key	86.67%	83.33%	93.33%
Combined (abeimp)	23.33%	33.33%	46.67%

4.5 Discussion

Overall, this thesis shows success in creating an ML model to classify supersonic audio, but the trained model does not infer labels on unseen data effectively. The confusion matrix in Figure 21 shows that the model classified the testing data successfully as expected. This also shows the effectiveness of using an OFDM signal for active acoustic side-channel analysis, since we are not concerned with the actual frequencies produced while a user is typing. Were the model to be trained with a user typing on a different keyboard, assuming the same microphone placement and OFDM signal generation, we would expect similar results from the model itself. However, the model fails in terms of

generalization and is unable to infer labels for unseen data collected as part of a different batch from the training/testing data set. This initially represented a failure in data consistency, so microphone volume and distance were standardized throughout the train/test data collection process to combat this issue.

The combined audio file data set was found to be much more reliable for training than the larger data set that contained individual microphone recordings. This corroborated the assertion of Nandakumar et al. that a three-dimensional motion would require at least three microphones to capture [6]. Furthermore, the revelation that the infer script was not working initially allowed us to discover that the success of this attack was also highly affected by microphone variables such as distance and gain level. Keeping those consistent during training is expected, but inference data sets using this model would be highly unpredictable in a real-world setting, and the model did not do quite as well as expected when generalizing to unseen data collected with different microphone settings.

When the trained model was used for the password inference of "autodiscover", the returned labels did not match the original labels at any position of the array. Small optimizations to the model were applied, such as experimentation with using linear frequency cepstral coefficient images (LFCCs) rather than spectrogram images to train the model, but only small improvements to the overall validation accuracy were observed. Furthermore, Figure 21 showed that the validation accuracy was generally acceptable across labels.

Furthermore, upon inspection of the spectrograms directly, using a Python script exemplified in Appendix E, inconsistencies were identified between the individual training

and the inferred samples. Furthermore, overlaying all training and inference data set samples for a key in a single image highlights differences holistically. Figure 24 shows the combined spectrogram of the ‘a’ keys used for training, and Figure 25 shows the combined spectrogram of all ‘a’ samples in the inference set. Although they appear somewhat similar, there are differences in frequency bins over time, possibly caused by environmental variables during data collection. A larger set of data used for training might help the model generalize to account for these variables. Therefore, the most likely cause of the inference issues was in the data set rather than the training model used.

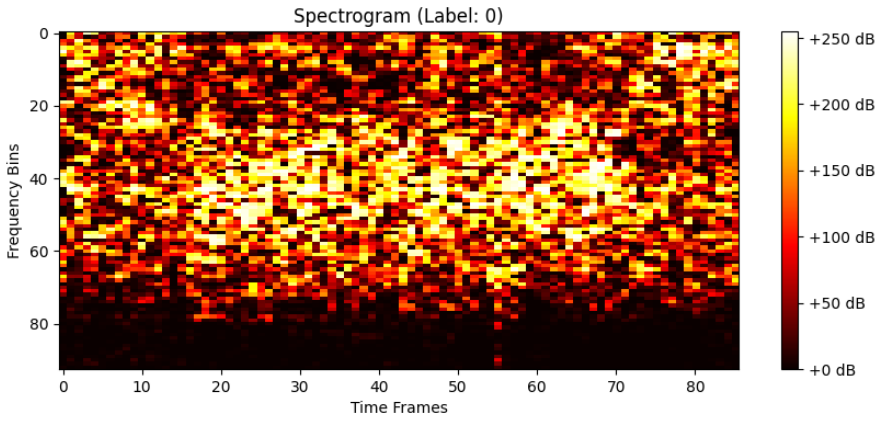

Figure 24. Combined Image of Training Spectrograms for ‘a’.

To test this theory, a subset of data was collected using the same methodology on just six keys. These keys, spaced out to represent discrete areas of the keyboard, were ‘a’, ‘b’, ‘e’, ‘i’, ‘m’, and ‘p’ [8]. Initially, 150 samples per label were collected, and the accuracy of the validation was consistent with the levels observed in the full data set and the lowercase letter subset.

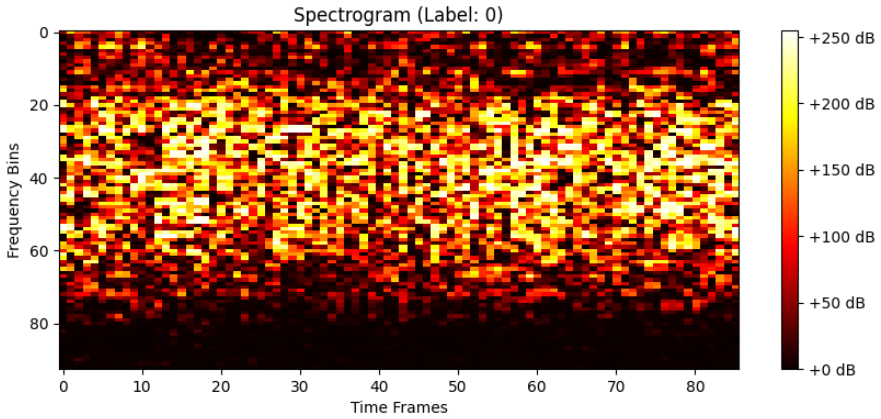

Figure 25. Combined Image of Inference Spectrograms for ‘a’.

Infer sets were then collected including 10 labels each for five sets of the ‘a’ key. The results, detailed in Section 4.4, indicated that the model was more able to pick up the position of the ‘a’ key with a small bias towards that key, which led to a more conclusive experiment.

Since the presence of more data had an observable affect on a single key, we hypothesized that we would be able to observe a general increase in inference success relative to the number of samples per label. Once again, a 6 label subset was collected, including the keys ‘a’, ‘b’, ‘e’, ‘i’, ‘m’, and ‘p’ were collected. In total, 450 samples were used in the training, which consisted of three rounds. Round 1 was trained with 150 samples per label, Round 2 was trained with 300, and Round 3 was trained with 450. In addition, 30 samples per label were collected as inference sets. All of these data were collected in a single session in an effort to maximize environmental control.

Once the models had been created, the inference on individual keys improved generally from Round 1 to Round 3 when testing those weighted models on the inference

sets, shown previously in Table 6. This indicates that the presence of more data improves inference accuracy, and we can generalize this to our full set of data and say that we most likely needed many more samples per label in order to successfully infer labels from the password sets.

4.5.1 Proposed Defenses

The state of the attack is highly dependent on the control of the microphone variables and the need for many more varied samples per label. Furthermore, single microphones did not pose a threat when evaluated in our model. Therefore, the current state of this attack may be challenging in a real-world environment due to the setup requirements. Current general recommendations for defense against acoustic SCAs, such as injection of background noise, may be effective but would interfere only with attack frequencies if injected at supersonic levels [1], [5]. Injecting movement or abnormal typing movements into the attack space would also be an adequate defense against this class of attack in its current state.

If this attack was to be condensed to be performed successfully using only one or two microphones, it could be possible if the model were trained using a transformer model, the implications in a real-world setting would be more severe [9]. A single mobile device microphone is much easier to obfuscate and would not require special placement. However, this is only a theoretical possibility and requires several more steps to be taken to generalize the current state of this acoustic SCA.

Theoretically, this active acoustic SCA remains resistant to most suggestions provided in the related literature due to its existence outside the field of audible frequencies [1], [2], [5], [23]. However, the current state of the attack is vulnerable to infer set variations in microphone level and hand position, most likely due to the limited number of samples per label. Therefore, unless a model is trained with many more samples per label, its potential effectiveness in a real-world situation is limited.

CHAPTER V

CONCLUSION

The potential severity of an acoustic side channel attack, although previously questioned, shows a serious potential to escalate as active attacks become more prevalent [11]. Active acoustic attacks essentially constitute miniaturized sonar, and the ability to map three-dimensional motion within a space as small as a keyboard means that any typed data could be compromised assuming that the attacker has the ability to reconstruct the acoustic data in some meaningful way.

5.1 Summary of Key Contributions

This work generalizes the SonarSnoop model of active acoustic SCAs to collect data in the three-dimensional space. Although the experiment was not successful in capturing and reconstructing entire password strings, the results of the experiment indicate that tracking individual fingers using an ML model and miniaturized sonar is possible and likely more successful with more data points. To our knowledge, this is the first study that attempts to utilize the active acoustic side-channel in a three-dimensional space [5], [8]. This study also shows that creating a deep learning model to classify supersonic

frequencies is achievable with a high rate of success, although generalizing the model to infer on unseen data is still a challenge.

5.2 Future Work

There are opportunities for significant improvements in the ML setup. First, the amount of data that run through the model could be increased to potentially increase accuracy. Due to time constraints, only 150 samples per key were used to train the full model. Based on our later experimentation, a higher number of samples per key would likely result in higher model accuracy. Another improvement to the model might come from the whole-word classification. There are several audio classification models that exist to classify speech to identify emotions, keywords, and whole words [17], [34], [35]. Theoretically, running spectrogram or vector-based representations of an active acoustic capture of these words could be classified in a similar manner, and they may return the same high level of success seen in the aforementioned studies. However, it is also possible that this approach would be less effective for randomized passwords due to its dependence on whole words rather than individual letters.

Another potential area of improvement is the adaptation of the CNN architecture to a transformer model. Gong et al. shared a transformer model for classifying aural frequencies with consistent accuracies of more than 90% when fine-tuned on various other data sets [9]. A transformer model might be particularly beneficial for the task of supersonic audio classification because it is a foundation model in that it can generalize learned patterns to recognize trends and adapt to specific downstream tasks [33]. This may

also allow the individual recordings (iPhone, iPad, MKE 600) to train a more effective model and even be successful in reconstructing passwords when used as the only microphone source for inference. Using a transformer model would also likely make the model effective in recognizing passphrases if such a data set were created [20]. The available AST code was pre-trained on audio at 16 kHz, and thus incompatible with the supersonic data set created. Developing such a model for supersonic audio classification from scratch requires much more specific expertise in transformers and is left as a task for future work in this area.

Another weakness in the ML setup is the extension of the use of individual letters rather than keywords. To divide the inference data set to input into the trained model, it was necessary to examine the aural frequency to observe when the actual keystrokes were being performed. Generally, if the attacker knows a user's typing speed, they could automate splitting the file appropriately. However, users occasionally pause while typing, and this would require manual inspection of the audio files in order to split into individual letters. It is possible that a transformer model could account for this by taking the entire password as input, but more training data (as well as the experience to develop such a model) would be required [9]. Obfuscation against attacks operating at an audible frequency is well documented, so this may be a critical factor that affects the ability of the attack to be recreated outside of a laboratory setting [4], [5].

REFERENCES

[1] D. Genkin, A. Shamir, and E. Tromer, "RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis," en, in *Advances in Cryptology – CRYPTO 2014*, J. A. Garay and R. Gennaro, Eds., vol. 8616, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 444–461, ISBN: 978-3-662-44370-5 978-3-662-44371-2. DOI: 10.1007/978-3-662-44371-2_25. [Online]. Available: http://link.springer.com/10.1007/978-3-662-44371-2_25 (visited on 02/16/2023).

[2] M. Guri, Y. Solewicz, A. Daidakulov, and Y. Elovici, "Fansmitter: Acoustic data exfiltration from (speakerless) air-gapped computers," *arXiv preprint arXiv:1606.05915*, 2016.

[3] J. de Gortari Briseno, A. D. Singh, and M. Srivastava, "InkFiltration: Using Inkjet Printers for Acoustic Data Exfiltration from Air-Gapped Networks," en, *ACM Transactions on Privacy and Security*, vol. 25, no. 2, pp. 1–26, May 2022, ISSN: 2471-2566, 2471-2574. DOI: 10.1145/3510583. [Online]. Available: https://dl.acm.org/doi/10.1145/3510583 (visited on 02/16/2023).

[4] D. Asonov and R. Agrawal, "Keyboard acoustic emanations," en, in *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, Berkeley, CA, USA:

IEEE, 2004, pp. 3–11, ISBN: 978-0-7695-2136-7. DOI: 10.1109/SECPRI.2004.1301311. [Online]. Available: http://ieeexplore.ieee.org/document/1301311/ (visited on 02/18/2023).

[5] P. Cheng, I. E. Bagci, U. Roedig, and J. Yan, "SonarSnoop: Active acoustic side-channel attacks," en, *International Journal of Information Security*, vol. 19, no. 2, pp. 213–228, Apr. 2020, ISSN: 1615-5262, 1615-5270. DOI: 10.1007/s10207-019-00449-8. [Online]. Available: http://link.springer.com/10.1007/s10207-019-00449-8 (visited on 02/16/2023).

[6] R. Nandakumar, V. Iyer, D. Tan, and S. Gollakota, "FingerIO: Using Active Sonar for Fine-Grained Finger Tracking," en, in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, San Jose California USA: ACM, May 2016, pp. 1515–1525, ISBN: 978-1-4503-3362-7. DOI: 10.1145/2858036.2858580. [Online]. Available: https://dl.acm.org/doi/10.1145/2858036.2858580 (visited on 02/16/2023).

[7] P. Rabadia and C. Valli, "Finding evidence of wordlists being deployed against SSH honeypots – implications and impacts," en, *12th Australian Digital Forensics Conference. Held on the 1-3 December*, vol. 2014 at Edith Cowan University, Western Australia. 2014, Medium: PDF Publisher: Security Research Institute (SRI), Edith Cowan University. DOI: 10.4225/75/57B3E7D5FB882. [Online]. Available: http://ro.ecu.edu.au/adf/141 (visited on 02/22/2023).

[8] R. Nandakumar, A. Takakuwa, T. Kohno, and S. Gollakota, "CovertBand: Activity Information Leakage using Music," en, *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 1, no. 3, pp. 1–24, Sep. 2017, ISSN: 2474-9567. DOI: 10.1145/3131897. [Online]. Available: https://dl.acm.org/doi/10.1145/3131897 (visited on 02/16/2023).

[9] Y. Gong, Y.-A. Chung, and J. Glass, *Ast: Audio spectrogram transformer*, 2021. arXiv: 2104.01778 [cs.SD].

[10] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, C. Sporleder, *et al.*, "Acoustic side-channel attacks on printers.," in *USENIX Security symposium*, vol. 9, 2010, pp. 307–322.

[11] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to differential power analysis," en, *Journal of Cryptographic Engineering*, vol. 1, no. 1, pp. 5–27, Apr. 2011, ISSN: 2190-8508, 2190-8516. DOI: 10.1007/s13389-011-0006-y. [Online]. Available: http://link.springer.com/10.1007/s13389-011-0006-y (visited on 02/19/2023).

[12] G. Joy Persial, M. Prabhu, and R. Shanmugalakshmi, "Side channel attack-survey," *Int. J. Adv. Sci. Res. Rev*, vol. 1, no. 4, pp. 54–57, 2011.

[13] S. A. Anand and N. Saxena, "Vibreaker: Securing Vibrational Pairing with Deliberate Acoustic Noise," en, in *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, Darmstadt Germany: ACM, Jul. 2016, pp. 103–108, ISBN: 978-1-4503-4270-4. DOI: 10.1145/2939918.2939934.

[Online]. Available: https://dl.acm.org/doi/10.1145/2939918.2939934 (visited on 02/18/2023).

[14] T. Halevi and N. Saxena, "On pairing constrained wireless devices based on secrecy of auxiliary channels: The case of acoustic eavesdropping," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10, Chicago, Illinois, USA: Association for Computing Machinery, 2010, pp. 97–108, ISBN: 9781450302456. DOI: 10.1145/1866307.1866319. [Online]. Available: https://doi-org.libproxy.usouthal.edu/10.1145/1866307.1866319.

[15] P. Walker and N. Saxena, "SoK: Assessing the threat potential of vibration-based attacks against live speech using mobile sensors," en, in *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, Abu Dhabi United Arab Emirates: ACM, Jun. 2021, pp. 273–287, ISBN: 978-1-4503-8349-3. DOI: 10.1145/3448300.3467825. [Online]. Available: https://dl.acm.org/doi/10.1145/3448300.3467825 (visited on 02/16/2023).

[16] *Huggingface*, https://huggingface.co/, Accessed: 2024-04-17.

[17] A. Baevski, H. Zhou, A. Mohamed, and M. Auli, *Wav2vec 2.0: A framework for self-supervised learning of speech representations*, 2020. arXiv: 2006.11477 [cs.CL].

[18] K. Doshi, *Audio deep learning made simple (part 1): State-of-the-art techniques*, May 2021. [Online]. Available: https://towardsdatascience.com/audio-deep-learning-made-simple-part-1-state-of-the-art-techniques-da1d3dff2504.

[19] J. Salamon and J. P. Bello, "Deep Convolutional Neural Networks and Data Augmentation for Environmental Sound Classification," en, *IEEE Signal Processing Letters*, vol. 24, no. 3, pp. 279–283, Mar. 2017, ISSN: 1070-9908, 1558-2361. DOI: 10.1109/LSP.2017.2657381. [Online]. Available: http://ieeexplore.ieee.org/document/7829341/ (visited on 03/10/2024).

[20] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, *Attention is all you need*, 2023. arXiv: 1706.03762 [cs.CL].

[21] T. Halevi and N. Saxena, "A closer look at keyboard acoustic emanations: Random passwords, typing styles and decoding techniques," en, in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, Seoul Korea: ACM, May 2012, pp. 89–90, ISBN: 978-1-4503-1648-4. DOI: 10.1145/2414456.2414509. [Online]. Available: https://dl.acm.org/doi/10.1145/2414456.2414509 (visited on 02/18/2023).

[22] T. Halevi and N. Saxena, "Keyboard acoustic side channel attacks: Exploring realistic and security-sensitive scenarios," en, *International Journal of Information Security*, vol. 14, no. 5, pp. 443–456, Oct. 2015, ISSN: 1615-5262, 1615-5270. DOI: 10.1007/s10207-014-0264-7. [Online]. Available: http://link.springer.com/10.1007/s10207-014-0264-7 (visited on 02/16/2023).

[23] L. Zhuang, F. Zhou, and J. D. Tygar, "Keyboard acoustic emanations revisited," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–26, 2009.

[24] T. Zhu, Q. Ma, S. Zhang, and Y. Liu, "Context-free Attacks Using Keyboard Acoustic Emanations," en, in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, Scottsdale Arizona USA: ACM, Nov. 2014, pp. 453–464, ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660296. [Online]. Available: https://dl.acm.org/doi/10.1145/2660267.2660296 (visited on 02/18/2023).

[25] O. Shih and A. Rowe, "Can a phone hear the shape of a room?" en, in *Proceedings of the 18th International Conference on Information Processing in Sensor Networks*, Montreal Quebec Canada: ACM, Apr. 2019, pp. 277–288, ISBN: 978-1-4503-6284-9. DOI: 10.1145/3302506.3310407. [Online]. Available: https://dl.acm.org/doi/10.1145/3302506.3310407 (visited on 02/16/2023).

[26] A. Hamed and N. Abdelbaki, "Acoustic Attacks in IOT Era: Risks and Mitigations," en, in *Proceedings of the 2020 5th International Conference on Cloud Computing and Internet of Things*, Okinawa Japan: ACM, Sep. 2020, pp. 13–19, ISBN: 978-1-4503-7527-6. DOI: 10.1145/3429523.3429530. [Online]. Available: https://dl.acm.org/doi/10.1145/3429523.3429530 (visited on 02/16/2023).

[27] I. Shumailov, L. Simon, J. Yan, and R. Anderson, *Hearing your touch: A new acoustic side channel on smartphones*, en, arXiv:1903.11137 [cs], Mar. 2019. [Online]. Available: http://arxiv.org/abs/1903.11137 (visited on 02/16/2023).

[28] M. Backes, M. Durmuth, S. Gerling, M. Pinkal, and C. Sporleder, "Acoustic Side-Channel Attacks on Printers," en,

[29] P. Dempsey, "The tear down: 2018 macbook pro touch bar keyboard," *Engineering & Technology*, vol. 13, no. 11/12, pp. 78–79, Dec. 2018, ISSN: 1750-9637. DOI: 10.1049/et.2018.1127.

[30] C. E. Shannon, "A mathematical theory of communication," in *Claude E. Shannon: Collected Papers*. 1993, pp. 5–83. DOI: 10.1109/9780470544242.ch1.

[31] A.-D. Vu, J.-I. Han, H.-A. Nguyen, Y.-M. Kim, and E.-J. Im, "A homogeneous parallel brute force cracking algorithm on the gpu," in *ICTC 2011*, 2011, pp. 561–564. DOI: 10.1109/ICTC.2011.6082661.

[32] J. Galbally, I. Coisel, and I. Sanchez, "A new multimodal approach for password strength estimation—part i: Theory and algorithms," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 12, pp. 2829–2844, 2017. DOI: 10.1109/TIFS.2016.2636092.

[33] R. Bommasani, D. A. Hudson, E. Adeli, *et al.*, *On the opportunities and risks of foundation models*, 2022. arXiv: 2108.07258 [cs.LG].

[34] S. R. Livingstone and F. A. Russo, "The ryerson audio-visual database of emotional speech and song (ravdess): A dynamic, multimodal set of facial and vocal expressions in north american english," *PLOS ONE*, vol. 13, no. 5, pp. 1–35, May 2018. DOI: 10.1371/journal.pone.0196391. [Online]. Available: https://doi.org/10.1371/journal.pone.0196391.

[35] W.-N. Hsu, B. Bolte, Y.-H. H. Tsai, K. Lakhotia, R. Salakhutdinov, and A. Mohamed, *Hubert: Self-supervised speech representation learning by masked prediction of hidden units*, 2021. arXiv: 2106.07447 [cs.CL].

[36] D. Hinkel, A. Buie, A. Surles, and G. Clark, "Attack vectors against ics: A survey," in *2023 Congress in Computer Science, Computer Engineering, & Applied Computing (CSCE)*, 2023, pp. 2494–2501. DOI: 10.1109/CSCE60160.2023.00401.

APPENDICES

Appendix A: Password Collection Script

```
perl -ne `print if /\b.{12}\b/' common.txt
```

Appendix B: OFDM Signal Generation

```
# ofdm.py
# Destin Hinkel
# Classifying Supersonic Frequencies for Active Acoustic Side-Channel Exploitation
# Run: python3 ofdm.py within .venv
# Script to generate a 1-hour long OFDM signal

import numpy as np
from scipy.io.wavfile import write

# Constants
SAMPLE_RATE = 44100
NUM_SUBCARRIERS = 64
SUBCARRIER_SPACING = 375  # Subcarrier spacing in Hz
OFDM_SIZE = int(SAMPLE_RATE / SUBCARRIER_SPACING)  # Adjusted OFDM size

# Parameters for the desired signal
START_FREQ = 18000
END_FREQ = 20000
START_INDEX = int(START_FREQ / SUBCARRIER_SPACING)
END_INDEX = int(END_FREQ / SUBCARRIER_SPACING)

# Time duration
DURATION = 60 * 60  # 1 hour

# Generate OFDM symbols
num_symbols = int(DURATION * SAMPLE_RATE / OFDM_SIZE)
ofdm_symbols = np.zeros((num_symbols, OFDM_SIZE), dtype=complex)

# For each OFDM symbol, randomly set subcarriers in the 18-20kHz range to +1 or -1
for i in range(num_symbols):
    ofdm_symbols[i, START_INDEX:END_INDEX] = np.random.choice([1, -1], size=(END_INDEX - START_INDEX))

# Create a Hermitian symmetric signal to ensure the output is real-valued
for i in range(num_symbols):
    ofdm_symbols[i, -END_INDEX:-START_INDEX] = np.conj(ofdm_symbols[i, START_INDEX:END_INDEX][::-1])

# Compute IFFT for each symbol to get time-domain samples
time_domain_samples = np.fft.ifft(ofdm_symbols, axis=1)

# Convert to real values (due to the conjugate symmetry, the output should be real)
time_domain_samples = time_domain_samples.real

# Apply a Hanning window to each symbol
window = np.hanning(OFDM_SIZE)
```

```
time_domain_samples = time_domain_samples * window

# Flatten to create the final signal
final_signal = time_domain_samples.flatten()

# Save as an audio file
filename = "ofdm_signal.wav"
write(filename, SAMPLE_RATE, (final_signal * 32767).astype(np.int16))

print(f"OFDM signal saved to {filename}")
```

Appendix C: Audio Normalization Script

```
# normalize.py
# Destin Hinkel
# Classifying Supersonic Frequencies for Active Acoustic Side-Channel Exploitation
# Run: python3 normalize.py within .venv
# Script to rechannel and strip unneeded audio frequencies from .wav files. Also creates backups.

import sys
import os
import warnings
import numpy as np
from scipy.io import wavfile
from scipy.signal import butter, sosfilt

def highpass_filter(data, cutoff, fs, order=5):
    nyquist = 0.5 * fs
    normal_cutoff = cutoff / nyquist
    sos = butter(order, normal_cutoff, btype='high', analog=False, output='sos')
    filtered_data = sosfilt(sos, data)
    return filtered_data

def convert_to_mono(data):
    if data.ndim == 2:
        data = data.mean(axis=1)
    return data

def get_next_backup_filename(directory, base_filename, ext):
    backup_directory = os.path.join(directory, "Backup")
    if not os.path.exists(backup_directory):
        os.makedirs(backup_directory)

    counter = 0
    while True:
        backup_filename = f"{base_filename}_backup_{counter:03d}{ext}"
        backup_filepath = os.path.join(backup_directory, backup_filename)
        if not os.path.exists(backup_filepath):
            return backup_filepath
        counter += 1

def process_wav_file(filepath, cutoff_frequency=18000):
    directory, filename = os.path.split(filepath)
    base_filename, ext = os.path.splitext(filename)

    with warnings.catch_warnings():
         warnings.simplefilter("ignore")
```

```
        sample_rate, data = wavfile.read(filepath)

    # Convert to mono if stereo
    data = convert_to_mono(data)

    # Apply high-pass filter
    filtered_data = highpass_filter(data, cutoff_frequency, sample_rate)

    # No scaling back to int16, keeping the output as float32
    filtered_data = filtered_data.astype(np.float32)

    # Create backup file in the Backup directory
    backup_filepath = get_next_backup_filename(directory, base_filename, ext)
    os.rename(filepath, backup_filepath)

    # Save the filtered audio with the original filename
    wavfile.write(filepath, sample_rate, filtered_data)

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python3 normalize.py <filename>")
        sys.exit(1)

    filepath = sys.argv[1]
    if not os.path.isfile(filepath):
        print("Error: The specified file does not exist.")
        sys.exit(1)

    process_wav_file(filepath)
```

Appendix D: Audio Split Script

```
# split.py
# Destin Hinkel
# Classifying Supersonic Frequencies for Active Acoustic Side-Channel Exploitation
# Run: python3 split.py within .venv
# Script to split combined audio files into two second individual files

import os
import sys
import numpy as np
from scipy.io import wavfile

def get_next_available_prefix(directory, base_filename):
    existing_files = [f for f in os.listdir(directory) if f.endswith('_' + base_filename + '.wav')]
    highest_number = -1  # Start from -1 so that the first available will be 0
    for file in existing_files:
        try:
            number = int(file.split('_')[0])
            highest_number = max(highest_number, number)
        except ValueError:
            continue
    return f"{highest_number + 1:03d}"

def split_wav_file(filepath, segment_length=2, max_prefix=999):  # Changed segment_length to 2
    directory, filename = os.path.split(filepath)
    if not directory:
        directory = '.'  # Set to current directory if empty
    base_filename = os.path.splitext(filename)[0]
    sample_rate, data = wavfile.read(filepath)

    file_length = data.shape[0] / sample_rate
    num_segments = int(np.ceil(file_length / segment_length))

    prefix = int(get_next_available_prefix(directory, base_filename))
    available_segments = max_prefix - prefix + 1
    num_segments = min(num_segments, available_segments)

    for i in range(num_segments):
        start_sample = int(i * segment_length * sample_rate)
        end_sample = int((i + 1) * segment_length * sample_rate)
        segment_data = data[start_sample:end_sample]

        new_filename = f"{prefix + i:03d}_{base_filename}.wav"
        new_filepath = os.path.join(directory, new_filename)
        wavfile.write(new_filepath, sample_rate, segment_data)
```

```
    # Delete the original file after splitting
    os.remove(filepath)

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python split_wav.py <filename>")
        sys.exit(1)

    filepath = sys.argv[1]
    if not os.path.isfile(filepath):
        print("Error: The specified file does not exist.")
        sys.exit(1)

    directory, filename = os.path.split(filepath)
    if not directory:
        directory = '.'  # Set to current directory if empty

    if filename not in os.listdir(directory):
        print("Error: File not found in the directory (case-sensitive check).")
        sys.exit(1)

    split_wav_file(filepath)
```

Appendix E: Spectrogram Viewer

```
# spectrogram.py
# Destin Hinkel
# Classifying Supersonic Frequencies for Active Acoustic Side-Channel Exploitation
# Run: python3 spectrogram.py within .venv
# Script to output a collection of spectrograms, given a manifest of .wav files
# CHANGEME Line 197 to specify a manifest
# CHANGEME Line 206 to specify an output directory

import os
import random
import torch
import torchaudio
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from torchaudio import transforms
from torch.utils.data import Dataset

# Class to Load in Audio Files from CSV & Return Tensor/SR
class AudioUtil():

    # Function to Load in Audio File
    @staticmethod
    def open(audio_file):
        sig, sr = torchaudio.load(audio_file)
        return (sig, sr)

    # Convert the Input Audio to a Single Channel
    @staticmethod
    def rechannel(aud):
        sig, sr = aud

        if (sig.shape[0] == 0):
            # Nothing to do
            return aud

        else:
            # Convert from Stereo (or more) to Mono by Selecting First Channel
            resig = sig[:1, :]

        return ((resig, sr))

    # Function to Pad Audio to Two Seconds
    @staticmethod
```

```
def pad_trunc(aud, max_ms):
    sig, sr = aud
    num_rows, sig_len = sig.shape
    max_len = sr//1000 * max_ms

    # If Signal Needs to be Truncated
    if (sig_len > max_len):
        sig = sig[:,:max_len]

    # If Signal Needs to be Padded
    elif (sig_len < max_len):
        pad_begin_len = random.randint(0, max_len - sig_len)
        pad_end_len = max_len - sig_len - pad_begin_len

        # Pad with 0s
        pad_begin = torch.zeros((num_rows, pad_begin_len))
        pad_end = torch.zeros((num_rows, pad_end_len))

        sig = torch.cat((pad_begin, sig, pad_end), 1)

    return (sig, sr)

# Function to Shift Signal to Left/Right w/ Wrapping
@staticmethod
def time_shift(aud, shift_limit):
    sig,sr = aud
    _, sig_len = sig.shape
    shift_amt = int(random.random() * shift_limit * sig_len)
    return (sig.roll(shift_amt), sr)

# Function to Generate a Spectrogram
@staticmethod
def spectro_gram(aud, n_fft, hop_length, power, freq_min=18000, freq_max=20000):
    sig, _ = aud
    sr = 44100

    # Calculate Frequency Resolution
    freq_res = sr / n_fft

    # Calculate Indices for Min & Max Frequencies
    idx_min = int(freq_min / freq_res)
    idx_max = int(freq_max / freq_res)

    # Generate Linear Spectrogram
    spec = transforms.Spectrogram(n_fft=n_fft, hop_length=hop_length, power=power)(sig)

    # Filter Frequencies Outside the 18000-20000 Hz Range
    spec_filtered = spec[:, idx_min:idx_max, :]
```

```
        return spec_filtered

    # Function to Perform Histogram Equalization on Linear Spectrogram
    @staticmethod
    def histogram_equalization(spec):

        # Convert spectrogram to numpy for processing
        spec_np = spec.numpy()

        # Flatten the spectrogram to 1D
        spec_flat = spec_np.flatten()

        # Calculate the histogram
        histogram, bin_edges = np.histogram(spec_flat, bins=256, range=(spec_np.min(), spec_np.max()),
density=True)

        # Compute the cumulative distribution function (CDF)
        cdf = np.cumsum(histogram) * np.diff(bin_edges)

        # Normalize the CDF
        cdf_normalized = (cdf - cdf.min()) / (cdf.max() - cdf.min()) * 255

        # Use linear interpolation of the CDF to find new pixel values
        spec_equalized = np.interp(spec_flat, bin_edges[:-1], cdf_normalized)

        # Reshape back to the original shape
        spec_eq_reshaped = spec_equalized.reshape(spec_np.shape)

        # Convert back to tensor
        spec_eq_tensor = torch.from_numpy(spec_eq_reshaped).float()

        return spec_eq_tensor

    # Function to Calculate LFCCs
    @staticmethod
    def lfcc(aud, n_fft, hop_length, power, f_min=18000, f_max=20000):
        sig, _ = aud

        lfcc_transform = transforms.LFCC(
            sample_rate = 44100,
            f_min = f_min,
            f_max = f_max,
            speckwargs={"n_fft": n_fft, "hop_length": hop_length, "center": False, "power": power}
        )

        # Apply LFCC Transformation
        lfcc = lfcc_transform(sig)
        return lfcc
```

```
# Class to Define the Sound Dataset
class SoundDS(Dataset):
    def __init__(self, df, data_path):
        self.df = df
        self.data_path = str(data_path)
        self.duration = 2000
        self.sr = 44100
        self.channel = 1
        self.shift_pct = 0.4

    def __len__(self):
        return len(self.df)

    def __getitem__(self, idx):
        audio_file = self.data_path + self.df.loc[idx, 'filepath']
        if 'label' in self.df.columns:
            class_id = self.df.loc[idx, 'label']
        else:
            class_id = None

        # Variables for Tuning
        n_fft = 2048
        hop_length = None
        power = 3

        aud = AudioUtil.open(audio_file)
        aud = AudioUtil.rechannel(aud)
        dur_aud = AudioUtil.pad_trunc(aud, self.duration)
        shift_aud = AudioUtil.time_shift(dur_aud, self.shift_pct)
        sgram = AudioUtil.spectro_gram(shift_aud, n_fft, hop_length, power)
        equalized = AudioUtil.histogram_equalization(sgram)
        #lfcc = AudioUtil.lfcc(shift_aud, n_fft, hop_length, power)
        return equalized, class_id

# Function to Save Spectrogram Images
def save_spectrograms(dataset, output_dir):
    os.makedirs(output_dir, exist_ok=True)
    for i in range(len(dataset)):
        spectrogram_tensor, label = dataset[i]
        spectrogram_np = spectrogram_tensor.detach().cpu().numpy()

        plt.figure(figsize=(10, 4))
        plt.imshow(spectrogram_np[0], cmap='hot', aspect='auto')
        plt.title(f'Spectrogram (Label: {label})')
        plt.xlabel('Time Frames')
        plt.ylabel('Frequency Bins')
        plt.colorbar(format='%+2.0f dB')
        output_path = os.path.join(output_dir, f'spectrogram_{i}.png')
```

```
        plt.savefig(output_path)
        plt.close()
        print(f'Saved spectrogram: {output_path}')

# Read in Manifest
manifest = '/home/destin/Documents/Programming/Thesis/Datasets/PATH/GOES/HERE/manifest.csv' # CHANGEME
to specify a manifest
df = pd.read_csv(manifest)
data_path = ''
df = df[['filepath', 'label']]

# Create Dataset
myds = SoundDS(df, data_path)

# Save Spectrogram Images
output_dir = '/home/destin/Documents/Programming/Thesis/Spectro_Images/Training_PATH/GOES/HERE' #
CHANGEME to specify where images should be output (preferably a preconfigured directory)
save_spectrograms(myds, output_dir)
```

Appendix F: Deep Learning Model

```
# model.py
# Destin Hinkel
# Classifying Supersonic Frequencies for Active Acoustic Side-Channel Exploitation
# Run: python3 model.py within .venv
# Script to train deep learning models and output model weights
# CHANGEME Line 289 to specify number of convolution layers
# CHANGEME Line 572 to specify a manifest of training files
# Optimized code is based on original source code published in Towards Data Science by Ketan Doshi
# https://towardsdatascience.com/audio-deep-learning-made-simple-sound-classification-step-by-step-
cebc936bbe5

import random, torch, torchaudio, os
import pandas as pd
import numpy as np
import torch.nn as nn
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import seaborn as sns
from torchaudio import transforms
from torch.utils.data import Dataset, random_split
from torch.nn import init
from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay

# Function to Calculate Mean & Std for All Datasets
def statistics(train_dl):

    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

    # Initialize Values
    sum = 0
    sum_sq = 0
    count = 0

    for data in train_dl:
        inputs, _ = data[0].to(device), data[1].to(device)
        sum += torch.sum(inputs)
        sum_sq += torch.sum(inputs ** 2)
        count += np.prod(inputs.size())

    # Compute the Mean and Std
    mean = sum / count
    std = (sum_sq / count - mean ** 2) ** 0.5
    return mean.cpu().numpy(), std.cpu().numpy()
```

```
# Function to Convert Subset Label IDs to Key Names (for inference)
def id_to_min_label(id):
    ids = {
        0: 'a',
        1: 'b',
        2: 'e',
        3: 'i',
        4: 'm',
        5: 'p',
    }

    if id in ids:
        return ids[id]
    else:
        raise ValueError(f"ID '{id} is not in the defined label map.")

# Function to Convert Label IDs to Key Names (for inference)
def id_to_label(id):

    # Mapping for the additional labels, inverted from `label_to_id`
    id_to_additional_label = {
        52: "`", 53: "1", 54: "2", 55: "3", 56: "4",
        57: "5", 58: "6", 59: "7", 60: "8", 61: "9",
        62: "0", 63: "-", 64: "=", 65: "[", 66: "]",
        67: "\\", 68: ";", 69: "'", 70: ",", 71: ".",
        72: "/", 73: "~", 74: "!", 75: "@", 76: "#",
        77: "$", 78: "%", 79: "^", 80: "&", 81: "*",
        82: "(", 83: ")", 84: "_", 85: "+", 86: "{",
        87: "}", 88: "|", 89: ":", 90: "\"", 91: "<",
        92: ">", 93: "?"
    }
    # Check if the id is in the additional mappings
    if id in id_to_additional_label:
        return id_to_additional_label[id]

    # Convert numerical id back to character (a-z, A-Z)
    if id < 26:
        # Convert numerical id to lowercase (0=a, 1=b, ..., 25=z)
        return chr(97 + id)
    elif id < 52:
        # Convert numerical id to uppercase (26=A, 27=B, ..., 51=Z)
        return chr(65 + id - 26)
    else:
        # Raise an error if the ID is not in the map
        raise ValueError(f"ID '{id}' is not in the defined label map.")

# Function to Save Models Iteratively
def save_model(model, save_directory):
```

```
    # Ensure Save Directory Exists
    os.makedirs(save_directory, exist_ok=True)

    # Combine Save Path with Model.pth
    model_save_path = os.path.join(save_directory, f"Model.pth")

    # Save the Trained Model's State Dictionary
    torch.save(model.state_dict(), model_save_path)
    print(f"Model saved as {model_save_path}")

# Class to Stop Training Early if No Improvement is Happening
class EarlyStopper:
    def __init__(self, patience=1, min_delta=0):
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.min_loss = float('inf')

    def early_stop(self, loss):
        if loss < self.min_loss:
            self.min_loss = loss
            self.counter = 0
        elif loss > (self.min_loss + self.min_delta):
            self.counter += 1
            if self.counter >= self.patience:
                return True
        return False

# Class to Load in Audio Files from CSV & Return Tensor/SR
class AudioUtil():

    # Function to Load in Audio File
    @staticmethod
    def open(audio_file):
        sig, sr = torchaudio.load(audio_file)
        return (sig, sr)

    # Convert the Input Audio to a Single Channel
    @staticmethod
    def rechannel(aud):
        sig, sr = aud

        if (sig.shape[0] == 0):
            # Nothing to do
            return aud

        else:
            # Convert from Stereo (or more) to Mono by Selecting First Channel
```

```
            resig = sig[:1, :]

        return ((resig, sr))

    # Function to Pad Audio to Two Seconds
    @staticmethod
    def pad_trunc(aud, max_ms):
        sig, sr = aud
        num_rows, sig_len = sig.shape
        max_len = sr//1000 * max_ms

        # If Signal Needs to be Truncated
        if (sig_len > max_len):
            sig = sig[:,:max_len]

        # If Signal Needs to be Padded
        elif (sig_len < max_len):
            pad_begin_len = random.randint(0, max_len - sig_len)
            pad_end_len = max_len - sig_len - pad_begin_len

            # Pad with 0s
            pad_begin = torch.zeros((num_rows, pad_begin_len))
            pad_end = torch.zeros((num_rows, pad_end_len))

            sig = torch.cat((pad_begin, sig, pad_end), 1)

        return (sig, sr)

    # Function to Shift Signal to Left/Right w/ Wrapping
    @staticmethod
    def time_shift(aud, shift_limit):
        sig,sr = aud
        _, sig_len = sig.shape
        shift_amt = int(random.random() * shift_limit * sig_len)
        return (sig.roll(shift_amt), sr)

    # Function to Generate a Spectrogram
    @staticmethod
    def spectro_gram(aud, n_fft, hop_length, power, freq_min=18000, freq_max=20000):
        sig, _ = aud
        sr = 44100

        # Calculate Frequency Resolution
        freq_res = sr / n_fft

        # Calculate Indices for Min & Max Frequencies
        idx_min = int(freq_min / freq_res)
        idx_max = int(freq_max / freq_res)
```

```
        # Generate Linear Spectrogram
        spec = transforms.Spectrogram(n_fft=n_fft, hop_length=hop_length, power=power)(sig)

        # Filter Frequencies Outside the 18000-20000 Hz Range
        spec_filtered = spec[:, idx_min:idx_max, :]

        return spec_filtered

    # Function to Perform Histogram Equalization on Linear Spectrogram
    @staticmethod
    def histogram_equalization(spec):

        # Convert spectrogram to numpy for processing
        spec_np = spec.numpy()

        # Flatten the spectrogram to 1D
        spec_flat = spec_np.flatten()

        # Calculate the histogram
        histogram, bin_edges = np.histogram(spec_flat, bins=256, range=(spec_np.min(), spec_np.max()),
density=True)

        # Compute the cumulative distribution function (CDF)
        cdf = np.cumsum(histogram) * np.diff(bin_edges)

        # Normalize the CDF
        cdf_normalized = (cdf - cdf.min()) / (cdf.max() - cdf.min()) * 255

        # Use linear interpolation of the CDF to find new pixel values
        spec_equalized = np.interp(spec_flat, bin_edges[:-1], cdf_normalized)

        # Reshape back to the original shape
        spec_eq_reshaped = spec_equalized.reshape(spec_np.shape)

        # Convert back to tensor
        spec_eq_tensor = torch.from_numpy(spec_eq_reshaped).float()

        return spec_eq_tensor

   # Function to Calculate LFCCs
    @staticmethod
    def lfcc(aud, n_fft, hop_length, power, f_min=18000, f_max=20000):
        sig, _ = aud

        lfcc_transform = transforms.LFCC(
            sample_rate = 44100,
            f_min = f_min,
            f_max = f_max,
```

```
            speckwargs={"n_fft": n_fft, "hop_length": hop_length, "center": False, "power": power}
        )

        # Apply LFCC Transformation
        lfcc = lfcc_transform(sig)

        return lfcc

# Class to Define the Sound Dataset
class SoundDS(Dataset):
  def __init__(self, df, data_path):
    self.df = df
    self.data_path = str(data_path)
    self.duration = 2000
    self.sr = 44100
    self.channel = 1
    self.shift_pct = 0.4

    # Map Labels to Indices so Subsets Work
    self.label_to_id = {label: idx for idx, label in enumerate(sorted(df['label'].unique()))}

  # Function to Return the Number of Dataset Items
  def __len__(self):
    return len(self.df)

  # Function to Get the i'th Dataset Item
  def __getitem__(self, idx):

    # Concatenate the audio directory with the relative path
    audio_file = self.data_path + self.df.loc[idx, 'filepath']

    # Check if Label Exists (training/testing/validation) or Not (inference)
    class_id = self.label_to_id.get(self.df.loc[idx, 'label'], None)

    # Variables for Tuning
    n_fft = 2048
    hop_length = None
    power = 3

    # Load Audio File & Apply Preprocessing Steps
    aud = AudioUtil.open(audio_file)
    aud = AudioUtil.rechannel(aud)
    dur_aud = AudioUtil.pad_trunc(aud, self.duration)
    shift_aud = AudioUtil.time_shift(dur_aud, self.shift_pct)
    sgram = AudioUtil.spectro_gram(shift_aud, n_fft, hop_length, power)
    equalized = AudioUtil.histogram_equalization(sgram)
    #lfcc = AudioUtil.lfcc(shift_aud, n_fft, hop_length, power) # CHANGEME if you want to try
classifying with LFCCs, but this was not originally effective
```

```
    return equalized, class_id

# Audio Classification Model
class AudioClassifier (nn.Module):

    # Function to Define the Model Architecture
    # Use only four layers if running on a,b,e,i,m,p subset
    # Use only five layers if running on lowercase subset
    # Use all six layers if running on full dataset
    def __init__(self, num_classes):
        super().__init__()
        conv_layers = []
        conv_features = 8

        # First Convolution Block with Relu and Batch Norm & Use Kaiming Initialization
        self.conv1 = nn.Conv2d(1, 8, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
        self.relu1 = nn.ReLU()
        self.bn1 = nn.BatchNorm2d(8)
        init.kaiming_normal_(self.conv1.weight, a=0.1)
        self.conv1.bias.data.zero_()
        conv_layers += [self.conv1, self.relu1, self.bn1]

        # Second Convolution Block
        self.conv2 = nn.Conv2d(8, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        self.relu2 = nn.ReLU()
        self.bn2 = nn.BatchNorm2d(16)
        init.kaiming_normal_(self.conv2.weight, a=0.1)
        self.conv2.bias.data.zero_()
        conv_layers += [self.conv2, self.relu2, self.bn2]
        conv_features *= 2

        # Third Convolution Block
        self.conv3 = nn.Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        self.relu3 = nn.ReLU()
        self.bn3 = nn.BatchNorm2d(32)
        init.kaiming_normal_(self.conv3.weight, a=0.1)
        self.conv3.bias.data.zero_()
        conv_layers += [self.conv3, self.relu3, self.bn3]
        conv_features *= 2

        # Fourth Convolution Block
        self.conv4 = nn.Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        self.relu4 = nn.ReLU()
        self.bn4 = nn.BatchNorm2d(64)
        init.kaiming_normal_(self.conv4.weight, a=0.1)
        self.conv4.bias.data.zero_()
        conv_layers += [self.conv4, self.relu4, self.bn4]
        conv_features *= 2
```

```
        # Fifth Convolution Block
        self.conv5 = nn.Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        self.relu5 = nn.ReLU()
        self.bn5 = nn.BatchNorm2d(128)
        init.kaiming_normal_(self.conv5.weight, a=0.1)
        self.conv5.bias.data.zero_()
        conv_layers += [self.conv5, self.relu5, self.bn5]
        conv_features *= 2

        # Sixth Convolution Block
        self.conv6 = nn.Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        self.relu6 = nn.ReLU()
        self.bn6 = nn.BatchNorm2d(256)
        init.kaiming_normal_(self.conv6.weight, a=0.1)
        self.conv5.bias.data.zero_()
        conv_layers += [self.conv6, self.relu6, self.bn6]
        conv_features *= 2

        # Linear Classifier
        self.ap = nn.AdaptiveAvgPool2d(output_size=1)
        self.lin = nn.Linear(in_features=conv_features, out_features=num_classes)

        # Wrap the Convolutional Blocks
        self.conv = nn.Sequential(*conv_layers)

    # Function to Perform Forward Pass Computations
    def forward(self, x):

        # Run the convolutional blocks
        x = self.conv(x)

        # Adaptive pool and flatten for input to linear layer
        x = self.ap(x)
        x = x.view(x.shape[0], -1)

        # Linear Layer
        x = self.lin(x)

        # Final Output
        return x

# Function for Training Loop
def training(model, train_dl, val_dl, num_epochs):

  # Calculate Global Statistics & Convert to Tensors
  global_mean, global_std = statistics(train_dl)
  global_mean = torch.from_numpy(global_mean).float()
  global_std = torch.from_numpy(global_std).float()
```

```
# Loss Function, Optimizer and Scheduler
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(),lr=0.001)
scheduler = torch.optim.lr_scheduler.OneCycleLR(optimizer, max_lr=0.001,
                                                steps_per_epoch=int(len(train_dl)),
                                                epochs=num_epochs,
                                                anneal_strategy='linear')

# Early Stopper
early_stopper = EarlyStopper(patience=10, min_delta=0.01)

# Lists to store validation metrics for plotting
val_losses = []
val_accuracies = []
epochs_list = []

# Repeat for Each Epoch
for epoch in range(num_epochs):
  model.train()
  running_loss = 0.0
  correct_prediction = 0
  total_prediction = 0

  # Training Phase
  for i, data in enumerate(train_dl):

      # Get Input Features and Target Labels, & Put Them on GPU (if available)
      inputs, labels = data[0].to(device), data[1].to(device)

      # Normalize Inputs with Global Values
      inputs = (inputs - global_mean) / global_std

      # Zero Parmeter Gradients
      optimizer.zero_grad()

      # Calculate Loss & Step
      outputs = model(inputs)
      loss = criterion(outputs, labels)
      loss.backward()
      optimizer.step()
      scheduler.step()

      # Keep Stats for Loss and Accuracy
      running_loss += loss.item()

      # Get the Predicted Class with Highest Score
      _, prediction = torch.max(outputs,1)
```

```
            # Count of Predictions that Matched Target Label
            correct_prediction += (prediction == labels).sum().item()
            total_prediction += prediction.shape[0]

        # Validation Phase
        model.eval()
        val_loss = 0.0
        val_correct = 0
        val_total = 0
        with torch.no_grad():
            for i, data in enumerate(val_dl):

                # Get Input Features and Target Labels, & Put Them on GPU (if available)
                inputs, labels = data[0].to(device), data[1].to(device)

                # Normalize Inputs with Global Values
                inputs = (inputs - global_mean) / global_std

                # Calculate Loss
                outputs = model(inputs)
                loss = criterion(outputs, labels)

                # Keep Stats for Loss and Accuracy
                val_loss += loss.item()

                # Get the Predicted Class with Highest Score
                _, prediction = torch.max(outputs, 1)

                # Count of Predictions that Matched Target Label
                val_correct += (prediction == labels).sum().item()
                val_total += prediction.shape[0]

        # Calculate Statistics at the End of Each Epoch
        avg_loss = running_loss / len(train_dl)
        avg_acc = correct_prediction / total_prediction
        val_avg_loss = val_loss / len(val_dl)
        val_avg_acc = val_correct / val_total

        # Store validation metrics for plotting
        val_losses.append(val_avg_loss)
        val_accuracies.append(val_avg_acc)
        epochs_list.append(epoch)

        #Print Statistics at the End of Each Epoch
        print(f'*** Epoch {epoch} ***\nTraining Loss: {avg_loss:.4f}\t\t\tValidate Loss: {val_avg_loss:.4f
    }\nTraining Accuracy: {avg_acc:.4f}\t\tValidate Accuracy: {val_avg_acc:.4f}\n')

        # Stop Early if Model Has Converged
```

```
    if early_stopper.early_stop(val_avg_loss):
      print("No improvement in validation loss detected. Stopping.")
      break

    # Plot validation loss and accuracy
  fig, ax1 = plt.subplots(figsize=(10, 6))

  # Plot Validation Accuracy
  color = 'tab:red'
  ax1.set_xlabel('Epoch')
  ax1.set_ylabel('Validation Accuracy', color=color)
  ax1.plot(epochs_list, val_accuracies, label='Validation Accuracy', color=color)
  ax1.tick_params(axis='y', labelcolor=color)
  ax1.set_ylim([min(val_accuracies), 1.0])

  # Create a second y-axis for validation loss
  ax2 = ax1.twinx()
  color = 'tab:blue'
  ax2.set_ylabel('Validation Loss', color=color)
  ax2.plot(epochs_list, val_losses, label='Validation Loss', color=color)
  ax2.tick_params(axis='y', labelcolor=color)
  ax2.set_ylim([0, max(val_losses)])

  # Final plot adjustments
  fig.tight_layout(pad=3.0)
  plt.title('Validation Loss and Accuracy over Epochs')
  plt.savefig('validation_plot.png')

  print('Finished Training')

# Function to Perform Testing
def testing(model, test_dl, class_names):

    # Calculate Global Statistics & Convert to Tensors
    global_mean, global_std = statistics(test_dl)
    global_mean = torch.from_numpy(global_mean).float()
    global_std = torch.from_numpy(global_std).float()
    num_classes = len(class_names)

    model.eval()
    all_predictions = []
    all_labels = []

    # Disable Gradient Updates
    with torch.no_grad():
        for data in test_dl:

            # Get Input Features and Target Labels, & Put Them on GPU
```

```
            inputs, labels = data[0].to(device), data[1].to(device)

            # Normalize Inputs with Global Values
            inputs = (inputs - global_mean) / global_std

            # Get Predictions
            outputs = model(inputs)

            # Get the Predicted Class with Highest Score
            _, prediction = torch.max(outputs,1)

            # Store Predictions and Labels for Later Analysis
            all_predictions.extend(prediction.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    # Print Classification Report for Individual Labels
    print("Classification Report:")
    print(classification_report(all_labels, all_predictions, target_names=class_names, digits=4))

    # Compute the confusion matrix
    cm = confusion_matrix(all_labels, all_predictions, labels=range(num_classes))

    # Convert Numeric Labels to Character Labels
    display_labels = [name for name in class_names]

    # Create the plot
    if num_classes == 6: # using subset
        plt.figure(figsize=(6, 6))
    elif num_classes == 26: # using lowercase
        plt.figure(figsize=(14, 14))
    elif num_classes == 94: # using full set
        plt.figure(figsize=(20, 20))
    else:
        plt.figure(figsize=(20, 20))

    # Save and display the CM
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=class_names)
    disp.plot(cmap='Blues', ax=plt.gca())
    plt.savefig('confusion_matrix.png')
    plt.close()

# Main Model Script
if __name__ == "__main__": # prevents infer.py from also training

  # Read in Manifest
  manifest = '/home/destin/Documents/Programming/Thesis/Datasets/Training_Full/manifest.csv' # CHANGEME
 to desired dataset manifest (generated with manifest.py)
  df = pd.read_csv(manifest)
```

```
  df.head()

  data_path = ''

  df = df[['filepath', 'label']] # labels can be anything, the model will run labels as index numbers
  df.head()

  myds = SoundDS(df, data_path)
  num_classes = len(myds.label_to_id)

  # Random Split of 70:15:15 Between Training, Validation, & Testing Datasets
  num_items = len(myds)
  num_train = round(num_items * 0.7)
  num_val_test = num_items - num_train
  num_val = round(num_val_test * 0.5)
  num_test = num_val_test - num_val
  train_ds, val_test_ds = random_split(myds, [num_train, num_val_test])
  val_ds, test_ds = random_split(val_test_ds, [num_val, num_test])

  # Create Training/Testing Data Loaders
  train_dl = torch.utils.data.DataLoader(train_ds, batch_size=16, shuffle=True)
  val_dl = torch.utils.data.DataLoader(val_ds, batch_size=16, shuffle=False)
  test_dl = torch.utils.data.DataLoader(test_ds, batch_size=16, shuffle=False)

  # Create the Model & Put on GPU if Available
  myModel = AudioClassifier(num_classes)
  device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
  myModel = myModel.to(device)
  next(myModel.parameters()).device

  # Training
  num_epochs=500 # default 500, may stop early due to validation
  training(myModel, train_dl, val_dl, num_epochs)

  # Define a Path for Saving the Model
  save_model(myModel, "/home/destin/Documents/Programming/Thesis/Saved_Models") # CHANGEME if you want
to save the weighted model elsewhere

  # Testing
  class_ids = sorted(df['label'].unique())
  class_ids = [str(id) for id in class_ids]
  testing(myModel, test_dl, class_ids)
```

Appendix G: Infer Script

```
# infer.py
# Destin Hinkel
# Classifying Supersonic Frequencies for Active Acoustic Side-Channel Exploitation
# Run: python3 infer.py within .venv
# Script to infer labels on unseen datasets by loading in weighted models
# CHANGEME Line 52 to specify a manifest file for the inference dataset
# CHANGEME Line 59 to specify the number of classes trained on for the model weights
# CHANGEME in model.py, make sure that you're using correct number of convolution layers - 4 layers for
 6 subset, 5 for lowercase, 6 for full
# CHANGEME Line 68 to specify a model .pth file

import os
import torch
import pandas as pd
from model import SoundDS, AudioClassifier, statistics, id_to_min_label, id_to_label

# Function to Perform Final Inference
def inference (model, inf_dl, num_classes):

  # Calculate Global Statistics & Convert to Tensors
  global_mean, global_std = statistics(inf_dl)
  global_mean = torch.from_numpy(global_mean).float()
  global_std = torch.from_numpy(global_std).float()

  model.eval()
  password = []

  # Disable gradient updates
  with torch.no_grad():
    for data in inf_dl:

      # Get the input features
      inputs = data[0].to(device)

      # Normalize Inputs with Global Values
      inputs = (inputs - global_mean) / global_std

      # Save Predictions
      outputs = model(inputs)

      # Get the Predicted Class with Highest Score
      _, prediction = torch.max(outputs,1)

      # Map the Integer Label ID to Alphanumeric Label
```

```
      if num_classes == 6: # if we are using the subset
        letter = id_to_min_label(prediction.item())
      else:
        letter = id_to_label(prediction.item())

      # Append letter to full password
      password.append(letter)

  return (password)

# Read in Manifest
manifest = '/home/destin/Documents/Programming/Thesis/Datasets/PATH/GOES/HERE/manifest.csv' # CHANGEME
for different infer sets
df = pd.read_csv(manifest)
df = df[['filepath', 'label']] # dataloader takes in "label" values, which should be NULL but aren't
used if defined

# Instantiate Inference Dataset
data_path=''
inf_ds = SoundDS(df, data_path)
num_classes = 94 # CHANGEME for different model weights (6, 26, 94), if this script is crashing, this
is probably the reason - make sure this number matches the weight

# Create Data Loader
inf_dl = torch.utils.data.DataLoader(inf_ds, batch_size=1, shuffle=False)

# Initialize GPU & Load in Model Weights
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = AudioClassifier(num_classes)
model.to(device)
model.load_state_dict(torch.load('/home/destin/Documents/Programming/Thesis/Saved_Models/Training_Full.
pth', weights_only=True)) # CHANGEME for different models

# Run inference on trained model with the password set and output labels for each character of the
password
predicted_labels = inference(model, inf_dl, num_classes)

# Calculate Percentage of Correct Classification (only works on 6 label subset excluding abeimp,
otherwise calculate manually)
correct_label = os.path.basename(os.path.dirname(manifest))
correct_label = correct_label.replace('Key', '').strip() # Label should just be the name of the key
correct_predictions = sum(1 for label in predicted_labels if label == correct_label)
accuracy = (correct_predictions / len(predicted_labels)) * 100

# Write Results to File
output_dir = os.path.dirname(manifest)
output_file = os.path.join(output_dir, 'inference_results_full.txt')
with open(output_file, 'w') as f:
```

```
    f.write(f"Predicated Labels: {''.join(predicted_labels)}\n")
    f.write(f"Accuracy for {correct_label}: {accuracy: .2f}%\n")

print(f"Inference results saved to {output_file}")
```

Appendix H: Final Password List

```
MasterPass.txt

From: common.txt
01. autodiscover
02. openvpnadmin
03. activeCollab
04. clientscript
05. single_pages
06. config.local
07. default_icon
08. swfobject.js
09. open-account
10. m6_edit_item

From: rockyou.txt
01. Beachdemon21
02. tummycha!R53
03. ANDYje$s1992
04. papiC*R!L022
05. BaByG&Rl19(5
06. !mC0NFU$E*69
07. p-n@ng!0@MIW
08. hMD0p7Rf&)(c
09. fki!u"6,r'KN
10. !(l?4&v"n^v@
```

BIOGRAPHICAL SKETCH

Destin A. Hinkel is completing the Master of Science program in Computer Science at the University of South Alabama (USA) in Mobile, Alabama. Destin previously attended USA for both a Master of Music (M.M.) and a Bachelor of Music (B.M.) in Music Education. During his time as a graduate assistant for the USA Department of Music, he was able to take prerequisite courses for the computer science program.

In 2022, Destin was offered the NSF Scholarship for Service award, which allowed him to return to graduate school in computer science full-time while preparing for a career in the federal government. In the summers between his academic years of funding, Destin worked for the Government Accountability Office (GAO) in Washington, D.C. While at GAO, Destin was involved with drafting federally published documents.

While at USA for his graduate studies in computer science, Destin was published through IEEE along with two other graduate students in a conference paper entitled "Attack Vectors Against ICS: A Survey" [36]. He has also taken opportunities to become involved with the DayZero cybersecurity club and participate in cyber competitions such as the DoE CyberForce Competition and the National Collegiate Cyber Defense Competition. Upon graduation, Destin plans to pursue a career performing cybersecurity work for the United States Federal Government.