Jamming Attack on Voice Activated Systems

Author: Cyrus Daruwala, Eugene Luo, Spencer Yu

Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—We developed a system capable of detecting wake words for a particular voice activated system (Siri). To demonstrate this, we are focusing on Siri's always on detection and reducing the frequency by 45% of which these it activates as a result. This problem requires tight latency bounds (~400ms after the user starts talking). Our solution uses commodity hardware to perform this attack, and uses time synchronization, dynamic time warping, and Mel-Frequency Cepstral Coefficients to achieve our final solution.**

***Index Terms*—Activated, Alexa, Defence, Dynamic Time Warping, Design, IOT, Jamming, Mel-Frequency Cepstral Coefficient, NLP, Security, Siri, Smart Speaker, Speech Recognition, Voice Controllable Systems, Voice Recognition**

I. Introduction

Voice recognition systems are becoming more commonplace in the everyday household, and are increasingly used to control connected IOT systems. If a malicious program were to prevent critical systems in a house from working, such as changing the temperature of a smart thermostat, this could lead to discomfort or even physical damage. We are aiming to show the relevance of these attacks by demonstrating them on the Siri-always-listening system using commodity hardware (i.e. a laptop). Our product is a low footprint background application that will run on a laptop, since our attack requires only a microphone and a speaker to run properly. The intended audience of this product is companies designing smart speaker technology. We will reduce the accuracy of which Siri interprets the "Hey Siri" phrase from 90% to 45%, and our attack has to be carried out between 0.2-0.4s after the user has started to speak for it to be effective at jamming the wake word. We would also like our output to be no more than 75dB, as this is the volume of loud conversation.

There are no competing products on the market but numerous research papers have demonstrated the viability of attacking smart speakers. The "Dolphin Attack" paper[1] shows how it is possible to use ultrasonic commands to covertly activate and control voice controllable systems, and was the inspiration for our project. Attacks have also shown that these attacks can be embedded[2] into various sound files, and embedded sound exploits can be used for tracking human movement through sonar[3]. However, these attacks all require specialized environments or setups. We are aiming to produce an attack that uses only commodity hardware to reduce the effectiveness of voice controlled systems, rather than hardware such as ultrasonic speakers or extremely specific environmental conditions.

II. Design Requirements

Our requirements are divided into three major categories: determining suitable jamming inputs, meeting latency bounds, and optimizing the accuracy and performance of our attack. Suitable jamming inputs are necessary to ensure we are actually able to reduce the accuracy of Siri's always listening system. Latency bounds should be met in order to ensure that Siri is receiving conflicting inputs. The trivial attack would be to just play a loud noise, but since we are going for a more refined attack our results are going to be affected more by false positives or negatives.

A. Determining Jamming Inputs

To motivate our approach on how to jam Siri, we first had to do research into how black box systems handle the always listening problem. Apple released a paper[4] about how they implemented the "Hey Siri" system. It samples every 0.2 seconds from the surrounding environment, and uses a 5 hidden layer DNN to decompose sound into various features. These features are then passed into a probability distribution with a threshold. If this threshold is exceeded, then Siri assumes the phrase heard was "Hey Siri".

Our actual jamming input should ideally lower the certainty score of the user's original input when the jamming input overlaps with the original input. We aim to do this by skewing the probability distribution by only sharing some features between the jamming input and the user's input.

Since Siri has around a 5% failure rate[5] at detecting a singular word, the two word phrase "Hey Siri" will have an average rate 90.25% rate of success. We aim to reduce this rate by roughly half, to a 45% detection rate for "Hey Siri". We will verify that we can actually reduce the frequency of which "Hey Siri" can be detected by testing our system using human inputs against our completed system.

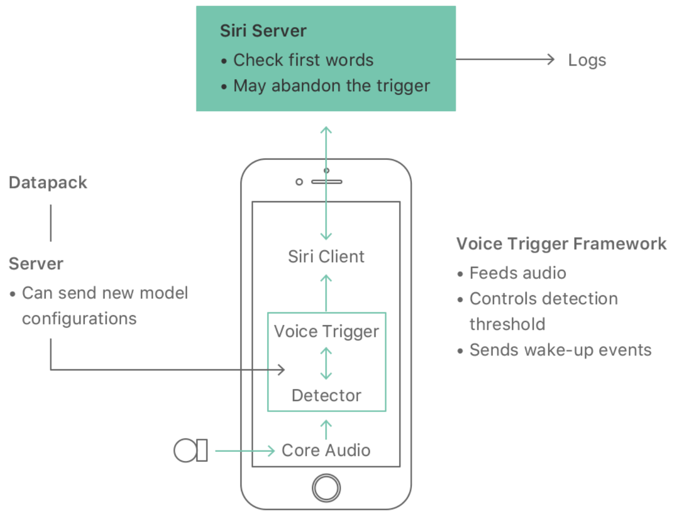


Figure 1: Siri's always-on "Hey Siri" detection system[4]

B. *Meeting Latency Bounds*

1. *User Bounds:*

Since we are aiming to output our jamming output at the same time as the "s" sound in Siri, the time at which the "s" sound in Siri and the jamming input are heard should differ by no more than 0.1 seconds.

Justification of the metric: We collected 20 samples of different people saying "Hey Siri" at different speeds. The graph below shows the distribution of the time taken by the different users to say the "s" sound from when the user starts saying "Hey Siri":

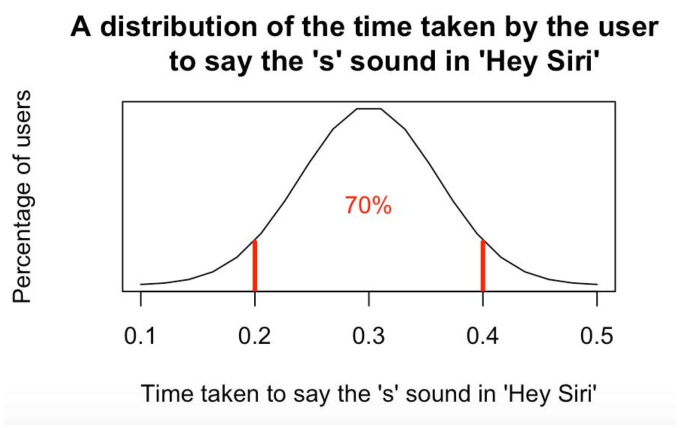


Figure 2: Distribution of time to say the "s" sound

On average, we found that the average time between when the user starts talking and the "s" sound in Siri is heard is 0.31s with a standard deviation of 0.06s. This means that our attack has to be timed between 0.2 - 0.4s (so that the time at which the "s" sound in Siri and the jamming input are heard differ by no more than 0.1 seconds). Approximately 70% of our samples fall within the 0.2 - 0.4s range, which means that aiming for an approximate time of 0.3s for our system would allow us to jam approximately 70% of all "Hey Siri" commands. This is sufficient because our goal is to reduce the accuracy of Siri by 45%, and the following range gives a 25% error rate to work with. We believe that this error bandwidth is necessary and sufficient for a semester long project.

2. *System Bounds:*

In order to determine these bounds, we had to first settle on getting an upper and a lower bound. Our upper bound is derived from the time it takes for a user to talk, with a little bit of slack. Our lower bound is derived from the time it takes for our system to run without implementing a speech processing system, which is the same as assuming that our speech processing system is idealized and takes no time to run. This was derived from our verification framework.

Our verification framework simulates the user as a computer to gather accurate statistics on how long our program actually takes to run. Since our program simulates an adversary, who will be using a computer to perform the attack, with a single computer we cannot accurately time how long our attack takes. With a single computer, we can at best gather information on the latency between when the adversarial computer receives the user's input and when it outputs this input. However, we need a second computer with more precise timing to simulate the user because we want to gather information on the latency between the actual time it takes from the user begins speaking to when the output is emitted from the adversarial computer.

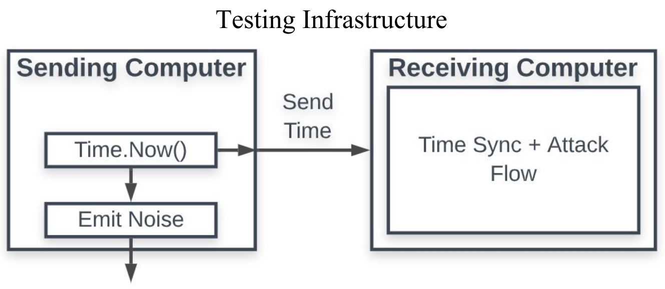


Figure 3: Testing Infrastructure Setup

C. *Accuracy/Performance Optimization*

We want to improve the performance of our system such that it will have a higher rate of "Hey Siri" reduction, while being more precise. We proposed several optimizations such that we could achieve better results within the limited timeframe of our semester long project.

In our initial report, we proposed a multithreaded program design, latency-hiding trick, and machine learning models to address the need for high performance. We intended to write our program in Python or C++ (if Python was too slow). We found substantial improvements in performance by using compiled Python over interpreted Python. By the end of the project, we instead came to appreciate the nature of digital signal processing. Because of MATLAB's mature support and transparent tooling for signal processing, we pivoted our program around a MATLAB implementation heavily reliant on signal manipulation.

The stretch goal we set out was to successfully use a model of adaptive adjustment to further tune our attack. Since people do not talk at exactly the same speed, we will be modeling our attack based on the average speed at which users say the phrase “Hey Siri”. We found after investigating a handful of machine learning models that a faster, more robust solution, again, relied heavily on signal processing fundamentals, specifically dynamic time warping (DTW) in conjunction with Mel-Frequency Cepstral Coefficients (MFCCs). Put simply, by matching the outstanding features of an audio signal (characterized by MFCCs) to a reference, our program is able to give a “path” that lets us warp the input signal to match the characteristics of the reference signal as closely as possible. Adaptive timing, then, is an obvious use case for dynamic time warping, allowing the program to more accurately identify the wake word “hey” regardless of the speed at which a user speaks.

III. Architecture and/or Principle of Operation

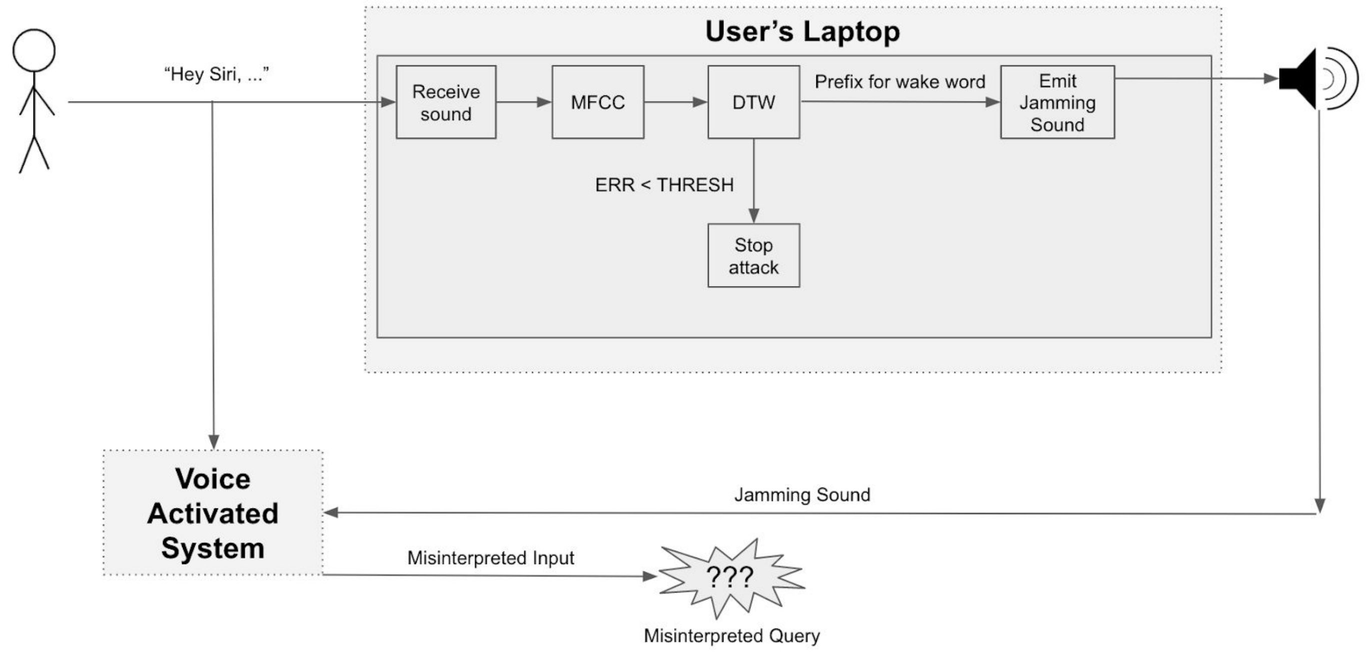
There are three major, relatively independent components for our project: timing architecture to more clearly understand the bounds, scope, and amount of time for each component of the project, signal processing to extract and compare the outstanding features of the microphone feed to our reference signals, and the final project deliverable that synthesizes this information (**Figure 10**, see below).

The first component of concern is the timing architecture designed as a multi-machine synchronization protocol. By taking advantage of the high-fidelity synchronization provided by Apple’s time servers, we can utilize two Macbooks with matching times and build a testing infrastructure to identify response times to events that happen on separate machines. This will be necessary to accurately record a program’s actual reaction time to an asynchronous event.

The second component of concern is the combination of MFCCs and dynamic time warping (DTW) to provide a basis for signal analysis and deciding whether a microphone feed registered the wake word.

The third component of concern is the exploit process. Though we initially envisioned the final product as a daemon process that runs in the background of a user computer, we found ourselves relying instead on executing a MATLAB program. This implementation fails to address the daemon nature of our initial design as well as making transparent the fact that this program runs in the first place; to run this exploit would require having access to MATLAB. However, we believe that because of the inherent difficulty in measuring the program’s performance and resource impact across a wide spectrum of devices, the focus of the project pivoted to demonstrating that this proof of concept was at least feasible in some fashion.

That being said, the third component will use the signal processing pipeline from the second component to accurately and quickly identify when the user has uttered the wake word in order to react and play a curated signal that will obfuscated the second part of a wake command. We have shown in our experiments that the entire wake command needs to surpass a likeliness threshold to trigger the smart speaker; thus, it is sufficient to spend the time of the first word to process and prepare the signal to output as the second word is said such that only the first half of the command is accurately understood.



IV. Design Trade Studies

A. Siri vs. other Smart Speakers

Consideration	Analysis
Wake word length	"Alexa" is the shortest length wake word, followed by "Hey Siri" and then "Okay Google". Jamming Alexa using the existing plan of attack proved to be very difficult due to the short wake word phrase. "Hey Siri" gives the system 0.3s on average to generate the jamming input, and "Okay Google" gives the system 0.4s on average for the same. This allowed us to eliminate Alexa from the list of the sound systems to consider.
Wake word detection accuracy by black-box system	We have already established earlier in the design document (Section I)[5] that Siri interprets the wake work correctly 90.25% of the time. A study on digital assistants[11] shows that Google Home has a better wake work accuracy than Siri.
Application	Smart speakers such as Google Home are only applicable when the user is at home. This limits the applicability of the attack, since most people are outside their homes for most of the day. Jamming a system such as Siri is more applicable, since a user's phone is likely to stay with him/her for the entire duration of the day.
Wake word distinction	A problem that came up late into our project was understanding the importance of acoustic distinction, and being able to recognize a word after filtering and signal processing. Quirky phrases like "Alexa" and "OK Google" benefit significantly because of the uncommon acoustic patterns that appear close together. Because we decided to work with "Hey Siri", more specifically the word "hey", filtering out false positives was a much more difficult process since many words and phrases sound remarkably similar to "hey", even to our own ears.

Figure 4: Comparison of smart speakers

The lower wake word detection accuracy and wider application initially made Siri a more attractive choice for a project spanning a semester. However, as the last row suggests, this introduced a new problem that we didn't anticipate: our difficulty in distinguishing the wake word from other common words and sounds made our false positive rate unsatisfactorily high, which we hope to address both for the demo and in future work.

B. Choice of language for implementation

At the beginning of our project, the two major languages we considered were Python and C given our collective skillset. We decided to implement initially implement this project in Python, but later switched to MATLAB due to the reasons below.

Consideration	Analysis
Readability/Writability	It's no secret that Python is an incredibly intuitive language to pick up. This saved a lot of development time in being able to rely on well-optimized abstractions such as lists, strings, and other readily available data types and paradigms. C++ does not offer this level of support for various programming abstractions, so we decided to not use C++ if we did not have to. As a domain specific language, MATLAB offers nearly identical capabilities to those that are provided in Python by the numpy/scipy libraries. However, because it is a commercial product, it has much better documentation and transparency, making it far easier to understand and debug our implementation. Because MATLAB is also optimized for engineering and scientific research, it is also better supported in some areas compared to Python.
Computing Speed	Because C++ operates much closer to the hardware level, directly compiling to machine code often begets greater performance benefits than executing Python scripts. There are certain workarounds such as Cython, which will directly compile Python code to binaries. However, even this will incur overhead because of safety considerations put in place by Python's language specification. This discussion rests almost entirely on computing speed. Because of the slower nature of I/O, any I/O bottlenecking should have no bearings on which language to consider for performance. MATLAB offers incredibly fast performance due to its domain specific nature and highly optimized support for signal processing.
Portability	All of our development is happening locally, which means portability is something that needs to be considered in the event that a laptop breaks or needs to be reset. In our collective experience, C++ can be very difficult to port from one system to another given inconsistencies in header files downloaded by XCode or installed by Brew. Meanwhile, Python can take advantage of virtual environments, which allow for an isolated testing silo that only has an understanding of virtually downloaded modules. This resolves dependencies by boiling down the issue to maintaining a requirements.txt file. As a result, Python is far easier to manage and port should the necessity arise. Similarly to Python, MATLAB is extremely portable. This is because most of the features we desired were already included in the Matlab 2018b release. We did not find ourselves relying on external MATLAB libraries other than those in the standard distribution.

Figure 5: Comparison of Programming Languages

C. *Commodity & Specialized Hardware/Environment*

When designing and internally testing our project, there was no special hardware being used. The design requires only a jammer (i.e. a laptop) and a "jammee" (i.e. a smart speaker, phone, etc). On reaching this decision, we considered the following:

Consideration	Analysis
Application	The object of our project is to demonstrate the feasibility of a jamming exploit. Part of this metric involves likelihood of replication, which informs our decision to defer to commodity hardware. Even if we can prove such an attack is feasible with special hardware such as advanced microphones, it's highly unlikely that this environment is commonplace whereas a laptop next to a phone is a much more realistic, replicable scenario.
Speed/Accuracy	Specialized hardware would be both faster at processing and more accurate in detecting a signal or filtering out noise in real-time. In contrast, our current design aims to filter and process in software, which will incur an execution overhead by nature. This is given the cumulative expertise of our team, which has informed out decision that it would be more productive to abstract into software. Even with that being said, there are tradeoffs to be had between latency and accuracy. For example, a more sensitive system might trigger more false positives and providing a faster response time while an emphasis on accuracy could require more processing time and thus increase response time. See our requirements for more details, but essentially, our project is currently at a compromise that allows for the use of commodity hardware with more sophisticated processing being done software-side.
Cost	Anyone hoping to replicate this attack would only need a laptop and a smart speaker. Specialized hardware would require buying more equipment or supplies to build such tools. Thus, our current approach is cheaper.

Figure 6: Comparison of Hardware Approaches

D. *Analysis of NLP Systems*

There are many NLP systems that we considered, but the two systems that were the most viable for this project were spaCy and CoreNLP. The final system we decided on was spaCy due to its speed over even CoreNLP, its closest competitor. We decided that an NLP system was not ideal for achieving the desired latency of our attack.

Consideration	Analysis
Feature Set	Both spaCy and CoreNLP have a robust feature set and are built for production environments. Other competing systems, such as NLTK based systems, have similar levels of feature sets.
Speed	Compared to other available NLP systems out there, spaCy and CoreNLP systems are the fastest as they are not built on the NLTK framework, which is significantly slower. The NLTK framework is meant for education, which makes it not ideal for latency-sensitive environment that we are using our NLP system for. We have to prioritize having a performant system, which generally means using NLP systems that are built and optimized for production usage. We also found that spaCy was faster than CoreNLP.[9][10]
Reliability	We wanted a system that was reliable while still being fast. This meant we seriously considered only the systems meant for industry over those meant for educational purposes (NLTK-based), as those are much less performant. CoreNLP is used in more production systems, so this translates to a more reliable system. On the other hand, spaCy is a newer system and consequently is not as well tested in various production systems. Its newness also means that the overall system is faster.

Figure 8: Comparison of NLP systems

Feasibility: We realized that our approach to using NLP systems that used entire words was not feasible due to timing constraints. The approach we experimented with consisted of a pipeline that first read in entire words or groups of words, which was bottlenecked by many factors, including waiting for the "end" of the words being spoken (which incurred a delay on the order of seconds), and processing whole words which cuts heavily into our total processing time. Additionally, this approach did not offer the granularity in timing that we could replicate using other (streaming-based) approaches.

E. *Volume Threshold to Lower Latency*

To lower the latency of our system, we established a lower noise threshold in response to its surroundings. This gave us an increased amount of processing time available to improve our program's performance and ensure we adhered to our latency bounds. This also increased the likelihood of false positives for our system.

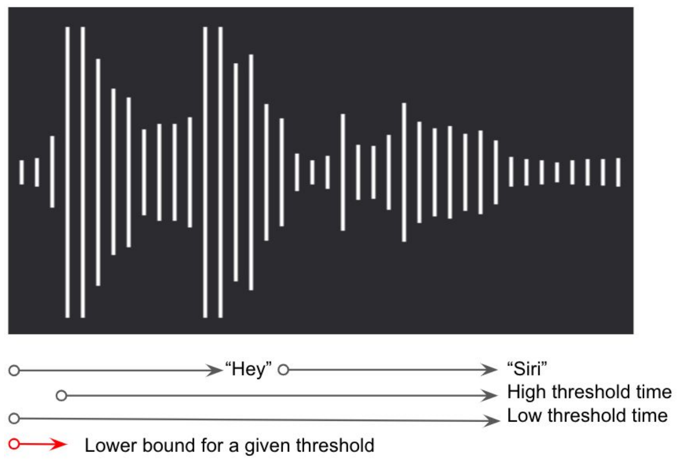


Figure 7: Latency Hiding Trick

We also experimented with constantly processing a stream of data rather than using a volume threshold. Since it is difficult to tune a volume threshold to be very precise, we wondered if constantly processing data would be more feasible. However, since we ignored the log energy as a factor when warping, our system would process data that should have been just meaningless noise as well. This led to a decrease in the overall performance of our system, so we decided to not go with this approach.

F. Stretch Goal: Adaptive Timing

Our initial vision for the attack was to have our program run in as little time as possible while still being sensitive to deviations in how people speak. We aimed to preserve accuracy by using simple machine learning models such as SVM to predict the time at which the "s" sound in "Siri" is said, given the input times for how long the user takes to say "hey" as well as the gap between how long they take to say "hey" and "Siri". Due to the extremely tight latency bounds we had established for our attack to be successful, we switched our approach to focus on dynamic time warping instead of machine learning. Ultimately, we achieved our goal for adaptive timing, but in a different fashion than we had originally anticipated.

V. System Description

As described in section III, the system architecture is composed of three key components as described below:

A. Timing Infrastructure

The timing infrastructure is critical for understanding the performance behavior and identifying timing bounds for each component of an attack. As a result, it is imperative that this infrastructure is both accurate and highly performant, for the time scale that many of these exploits happen occurs on the order of magnitude of milliseconds.

Consider, for example, one machine A that emits a signal, and another machine B that waits to receive the signal. Without any communication between these machines, machine B can record an absolute timestamp in step with its own system time as soon as it recognizes that the target signal has been emitted. Similarly, without any communication, machine A can timestamp the instant it fires off a signal, but there is no useful information present because machine A has no knowledge of machine B's reception. As this example demonstrates, we are less concerned with absolute time so much as we are at the mercy of relative times between the two machines on invocation and reception of the signal. Described in the context of our problem: there is a difference between timing machine B's reaction to the signal locally and timing machine B's reaction relative to when machine A sends out the signal. This reaction time is the value of concern because it will lower-bound our project's response time. Assuming optimization on the end of machine B's polling and I/O, whatever time recorded here will be the absolute lower-bound for reaction time, and any other time left will need to be used to process and prepare a response signal.

A second part of this timing infrastructure that has been as of yet unmentioned is the necessity of time synchronization. If machine A and machine B are off by even one second, this difference will entirely supersede the range of time during which we plan on acting. To nullify this issue, we relied on time synchronization through using Apple's time servers. By guaranteeing the machines are already clock-synchronized, we can then start a signal, timestamp its start, and have the receiver timestamp its response. The design for this testing component can be found in Figure 3.

The actual time synchronization was built as a bash script wrapping MATLAB interactions via the command line, which let us first synchronize system clocks and then run commands to output a signal and listen for one too. In our initial design report, we expressed interest in modularizing our timing framework. We approached this idea by wrapping a work script with the timing framework, which came in handy when we ultimately had to pivot away from Python to MATLAB.

B. Digital Signal Processing

Our baseline approach simply used a volume threshold to decide when to emit a jamming signal. The most apparent issue with this approach is that responding to any trigger would cause an unacceptably high false-positive rate. A second issue is that the program will not always consistently hear the command at the same volume if accounting for a user moving and speaking at variable volumes. While the threshold will be able to hear at or above the volume, we immediately fail to jam all queries that start below that query.

All things considered, the only constant in every voice-triggered query to Siri is the wake command to begin with: even more specifically the word "hey." Previous boundaries defined earlier motivate the use of an efficient yet fast algorithm that can intelligently parse a user's invocation of the wake word without the use of specialized hardware. Investigations and testing during the second half of the semester led us away from machine-learning solutions and more toward the direction of digital signal processing, which, given the right toolset, could quickly and accurately identify one single wake-word.

The two most important components of our project were the application of MFCCs and dynamic time warping:

1. MFCC

MFCCs have been used in speech recognition since the 1980s, and though they have fallen out of favor in robust black-box systems for deep neural networks, MFCCs offer a slightly less robust yet much faster and simpler solution. Decomposing a signal into overlapping, triangular windows and taking the discrete cosine transform of each section gives us a matrix of coefficients in which rows correspond to each window, (save for the first row, which represents the power of the signal on a log-log spectrum) and each column corresponds to a specific coefficient for the given sample. In terms of speech, almost all of the signal features are captured in the first 13 coefficients, which means these are the only ones we need to consider.

Because MFCCs are a way to identify features in a signal, we use a mean-squared error comparison between a test sound and our reference sound to classify what the program hears. Aside from giving us a fast and computationally cheap method of speech verification, MFCCs also benefit from being power-independent: theoretically, the same signal amplified should yield the same coefficients save for the power coefficients, which can be ignored in our case.

2. DTW

The second component we implemented was dynamic time warping. Dynamic time warping is a strategy that maps defining features of an audio signal against a reference[12]. The process outputs a "path" that can be used to shrink or expand a signal. In our case, the distribution of the length taken to say the word "hey" is normally distributed, but 30ms sample windows means that even 70% of our samples fall in a range of ±3 sample windows, motivating the use of DTW. Because each set of MFCCs per sample window is independent of others, we can take the vector of coefficients as a "sample" that is to be considered for DTW. By applying DTW to audio feed MFCCs and our reference MFCCs, we know how to warp the audio feed if necessary.

C. Exploit Program

The exploit program is the "user facing" component of this project that builds upon the work set forth by the aforementioned components. An audio stream takes in samples from the microphone in frames of 30ms (the minimum time required given our sampling rate of 16kHz). If the frames have energy above a certain threshold, MFCC coefficients are calculated on each sample for three frames, and these coefficients are warped using dynamic time warping against the MFCC coefficients of a reference sample of Spencer saying "hey". The mean square error is then calculated between the MFCCs of the warped sample and the MFCCs of the reference sample. If the mean square error is below the error threshold we set, the program will begin outputting a preselected signal (the word "Cyrus") designed to obfuscate the suffix of the wake word.

VI. Project Management

A. Schedule

See Figure 11 at the end of our report for a full schedule.

B. Team Member Responsibilities

All team members were responsible for ensuring that the components that they worked on were highly performant as to not impact the success of the timing-sensitive system. We all contributed to the design components of the system.

Because our project was so highly experimental, the actual design and implementation process was a mix of trying many different approaches to solving each component of our problem, and then refining the ones that got us closer to a working approach. Although we all worked together on integration fairly often, each group member had specialized expertise that made it easier for us to achieve our end goals.

Cyrus found performance bottlenecks that resulted from IO bound computation, which was crucial for deciding how we approached each component of our solution. Eugene would prevent our progress from stalling by creating fixes to make our system functional, allowing us to integrate each component into a testable system. Spencer focused on hacking together signal processing components for integration and understanding the relationship between the modules we used and performance of our system.

C. Budget

The actual hardware that we need for this project is very limited. In the beginning of our project, we assumed we would need a product for each type of commercial voice activated system on the market, and thus we bought an Echo Dot ("Alexa"), as well as a Google Home ("Ok Google"). We already have iPhones so we do not need to buy such systems. After we reduced the scope of our project, the only voice activated system hardware that we needed was just our iPhones. We will be generalizing relatively well across the space of different iPhones because we each own different models of iPhones. We also tested using the Apple HomePod to see how well our project generalizes to other Siri-equipped devices. The HomePod has a much more complex microphone array, and our system was unable to trick the HomePod into not activating. We believe this is the case because the HomePod is optimized for the home environment, not the low power environment that the iPhone is optimized for.

We also bought a decibel meter to ensure that our output falls within the 75dB constraint of medium to loud noise. This is to ensure we are outputting a reasonable type of output and not trivializing the problem with an arbitrary loud output, which goes against the goal of our project.

Material	Cost
BAFX Products - Decibel Meter / Sound Pressure Level Reader (SPL) / 30-130dBA Range - 1 Year Warranty	$18.99
Apple HomePod	$350.00

Figure 9: Bill of Materials

D. Risk Management

Our project has a lot of risk associated with the design process due to how much research is involved in the process. Our initial project design was overly broad and we did not realize how this was such an open problem. We scaled back from this approach after talking to various professors and realizing that we were trying to solve an open problem on a mature set of systems, which was way too much to tackle in one semester.

We scaled back to choosing a single system (Siri) and a single phrase to obfuscate ("Hey Siri") instead of attacking a combination of open source voice activated systems and black box systems within a three month period. By scaling back our project scope, it was actually possible for us to achieve our desired goals. In addition, it reduced the space of the problem significantly, as we are no longer trying to obfuscate an infinite set of commands.

We initially devised a series of fallback procedures to reduce the risk of our project not being completed. We knew that our project was possible by testing a preliminary jamming attack out using our voices, but the greatest risk for our project was in terms of latency bounds. If we did not meet these bounds, our project would not work because we will not be able to output the jamming output in time.

We spent a few weeks of our project running experiments and proof of concepts to make sure our project could be completed, given the technology that we had (in terms of the processing power of our computers, the speed of which memory could be accessed, and the sensitivity of our iPhones to the jamming signal).

Even after taking a more conservative approach while designing the system (one voice activation system, one word to jam) and having a few fallback procedures, we encountered numerous problems. We initially started off investigating NLP models in Python. If this system does not run quickly enough, we can switch to a compiled version in Python. Compiled instead of interpreted Python will run faster (nearly as fast as C), increasing the amount of time we can spend in our computational workflow. We are also using spaCy, a robust and fully featured NLP suite at the moment. If our system is not quick enough, we can strip the suite down to the core functionality that we are using. If our system still does not perform fast enough, we can consider switching it to be written in C++ for even greater speed boosts. The thing we didn't realise is that all these fallbacks would only help if our program was computationally bound. After trying NLP in Python and C++, we noticed that there was a minimal improvement in the response time while moving from Python to C++. This is when we decided to switch to a signal processing approach.

We also had backup plans in terms of the language choice we used, and ended up running compiled Python in order to improve the performance of our program. We did not anticipate the switch towards MATLAB, because we did not think that one of the limiting factors in our project would be more towards code transparency in Python modules.

E. Results

We measured the overall speed and performance of our program in terms of latency and overall system validation. We achieved an average response time of 462.5ms over 10 queries between the time the "Hey Siri" recording was played and the time the attacking device detected the "Hey" sound (right before it output the jamming signal). We also measured the overall system validation against Spencer's iPhone 7 Plus. The response rate of Siri dropped from 90% (with no jamming) to 46.67% over 30 queries. We also measured the volume of the output using a decibel meter placed next to the jamming device, and found that on average we could get Siri to not activate on Spencer's iPhone 7 Plus with an output volume of 78 dB, which was very close to our target goal of 75 dB..

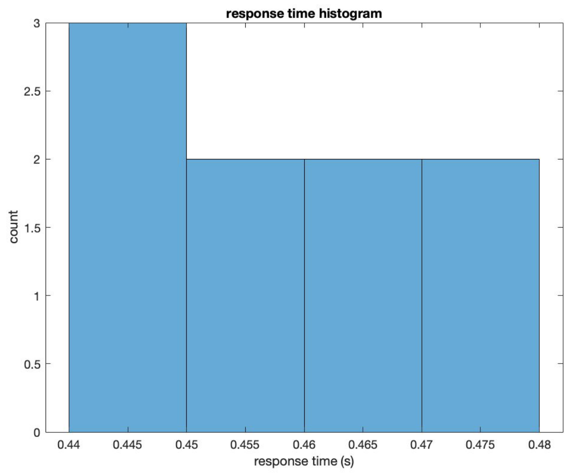
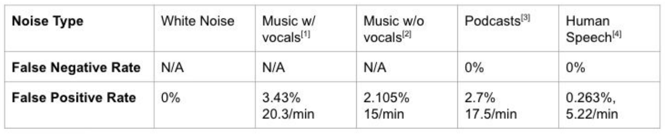


Figure 12: Response time histogram

We also looked at the results of our program in terms of how often we would trigger false positives and false negatives across various types of input noise. We tested with five different samples: white noise (the sound of a box fan), "Hey Jude" by the Beatles, "The Sleeping Beauty" by Tchaikovsky, "Great Bitter Lake Association" by Roman Mars, and words physically spoken by Spencer and Eugene.

We defined the false negative rate as the percent of time where the attack did not trigger when it should, and the false positive rate as the percent of time where the attack triggers when it should not. Because a possible opportunity to pass the volume threshold occurs every 30ms, 30s of loud-enough noise can result in 1000 samples, of which perhaps only 3 trigger a response. This proposed example demonstrates how to interpret the seemingly low false positive rate. Instead of trying to interpret the rates absolutely, understanding these results comparatively against other audio types is a sensible exercise.



Noise Type	White Noise	Music w/ vocals[1]	Music w/o vocals[2]	Podcasts[3]	Human Speech[4]
False Negative Rate	N/A	N/A	N/A	0%	0%
False Positive Rate	0%	3.43% 20.3/min	2.105% 15/min	2.7% 17.5/min	0.263%, 5.22/min

Figure 13: Error rates for different sound types

VII. Related Work

Our initial inspiration for our project was the attack on commodity hardware performed by last year's 18-500 group. Project LAKE[7] demonstrated an eavesdropping attack on keyboards using commodity hardware to log the keystrokes with relatively high accuracy. Previously, this had only been done in highly specialized environments (recording studios with studio-grade microphones), but this group extended the previously demonstrated exploit to work with a low powered wireless sensor package.

The actual attack we are performing was inspired in a similar manner. We saw many research papers that demonstrated the viability of using specialized attacks, such as ultrasound speakers or embedded noise, to hijack smart speaker systems such as Alexa. However, these attacks used environments that were difficult to replicate and did not provide source code or verifiable ways to reconstruct their proofs of concept attacks[1][3]. Additionally, some of these attacks, such as ultrasonic attacks, are trivial to block by adding a simple low-pass filter over the microphone input.

Our attack builds on the ideas provided by these attacks but introduces a problem space in which we aim to reduce accuracy in a manner that is more difficult for smart speaker manufacturers to block. There are no products on the market that currently seek to perform similar tasks to what we are doing. This is because most systems that people are building do not seek to decrease the performance of commercial voice activated systems.

VIII. Summary

Our system aims to reduce the accuracy that Siri can detect user input of the phrase "Hey Siri". It is limited by the fact that we are choosing to perform our attack using only commodity devices, limiting the overall speed and methods that we can use to reduce Siri's accuracy. It is also limited by how non-transparent commercial voice recognition systems are to fully understand. As these systems are extremely mature, it is difficult without very specialized attacks and specialized environments to create attacks that completely derail the intended functionality of these systems.

If we had more time, we could extend our system to run on specialized hardware, which could greatly improve the performance of our system and lead to more time for other potential optimizations and computations. Additionally, since the field that our project falls into is more of a research-driven space, we could attempt to perform precise attacks in various different manners. One such approach we could try if we had years to spare is optimizing generative adversarial networks (GANs) to create attacks to fool the DNN that Siri uses to decompose audio input. However, GANs would have a difficult time processing audio since it has a high sampling rate (44.1kHz is standard), and GANs are meant to work with input streams that sample far less frequently, such as images. Another approach would be using adversarial machine learning to fool the DNN that Siri uses. This kind of work is far more experimental, and would require more time to work as current GAN implementations are not meant to work with this specific use case.

IX. Future Work

Our metrics indicate that we are able to reduce the accuracy with which Siri detects the wake word, and at the same time maintain a low false negative rate. However, there is still a lot of scope for improvement. Two major improvements that we feel will greatly improve the success of our application are as follows:

1. *False Positive Rate*

Our application has a pretty low false positive rate for non-speech noises, but the false positive rate rises pretty quickly when speech that is similar sounding to hey is used as a test input. We think that this is due to the acoustical ambiguity of the word "Hey". To mitigate this effect, we think that trying this exact timing attack with a different jamming input on a different voice recognition system like Google that has a longer wake word ("Okay" gives us two syllabals to work with, whereas "Hey" only gives us one syllable to work with) would reduce the acoustical ambiguity, and in turn lead to a lower false positive rate.

2. *Volume*

Another important concern is the volume at which the jamming attack is delivered. A lower volume would make this DoS attack significantly less detectable, and hence much better suited for a real-world jamming attack. In our metrics, we focus on keeping the volume at 75 dB , which is comparable to the average volume at which humans speak. Our current attack outputs around 78 dB, and can still be detected and heard by a user in a quiet environment. Hence, experimenting with lower volumes would allow for better undetectability and in turn improve the applicability of the attack.

References

[1] Zhang, Guoming, et al. “DolphinAttack.” *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*, 2017, doi:10.1145/3133956.3134052.

[2] Carlini, Nicholas, et al. “Hidden Voice Commands.” *Proceedings of the 2005 Workshop on End-to-End, Sense-and-Respond Systems, Applications and Services*, USENIX Association, 2005, pp. 515–530.

[3] “CovertBand.” *CovertBand*, University of Washington, musicattacks.cs.washington.edu/.

[4] “Hey Siri: An On-Device DNN-Powered Voice Trigger for Apple's Personal Assistant - Apple.” *Apple Machine Learning Journal*, machinelearning.apple.com/2017/10/01/hey-siri.html.

[5] Novet, Jordan. “Apple Claims Siri's Speech Recognition Tech Is More Accurate than Google's.” *VentureBeat*, VentureBeat, 8 June 2015, venturebeat.com/2015/06/08/apple-claims-siris-speech-recognition-tech-is-more-accurate-than-googles/.

[6] Douglas, S C. “Introduction to Adaptive Filters.” *Introduction to Adaptive Filters*, pdfs.semanticscholar.org/aa48/98919244e59159bd276109b10dfbaa5dedf1.pdf.

[7] “Project LAKE.” *Team A2 Project LAKE Logging of Acoustic Keyboard Emanations*, course.ece.cmu.edu/~ece500/projects/s19-teama2/.

[8] “Global Interpreter Lock.” *GlobalInterpreterLock - Python Wiki*, wiki.python.org/moin/GlobalInterpreterLock.

[9] “5 Heroic Python NLP Libraries.” *EliteDataScience*, 8 Feb. 2018, elitedatascience.com/python-nlp-libraries.

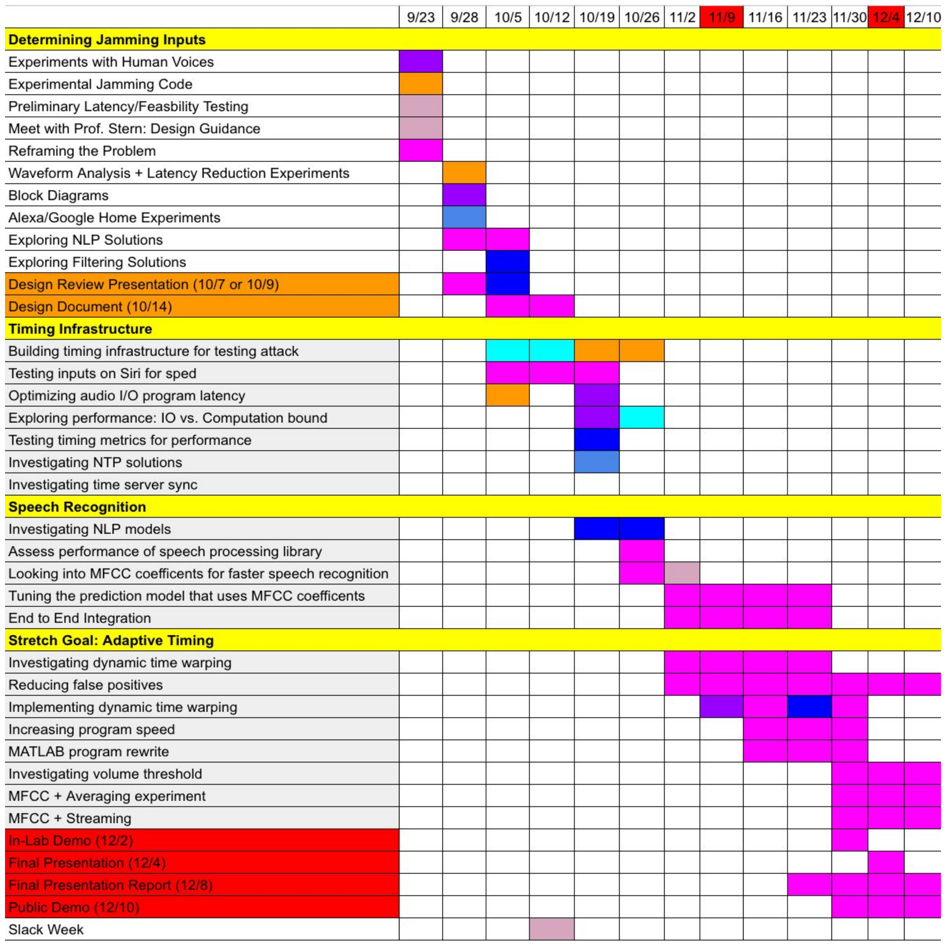
[10] BhavsarAn, Pratik. “NLTK Vs Spacy Vs Stanford CoreNLP - Ml-Dl.” *Ml*, 25 July 2018, ml-dl.com/nltk-vs-spacy/.

[11] “Digital Personal Assistants: Which Is the Smartest in 2018?: Perficient Digital.” *Digital Personal Assistants: Which Is the Smartest in 2018? | Perficient Digital | Perficient Digital Agency*, www.perficientdigital.com/insights/our-research/digital-personal-assistants-study.

[12] Rabiner, Lawrence. “Dynamic Time-Warping Solution.” *Fundamentals of Speech Recognition*, edited by Biing-Hwang Juang, Prentice Hall, 1993, pp. 221–229.

[13] “LibROSA.” *LibROSA*, librosa.github.io/librosa/feature.html#feature.

[14] “Fastdtw 0.3.4.” *PyPI*, Python, pypi.org/project/fastdtw/.



	9/23	9/28	10/5	10/12	10/19	10/26	11/2	11/9	11/16	11/23	11/30	12/4	12/10
Determining Jamming Inputs													
Experiments with Human Voices	🟪												
Experimental Jamming Code	🟧												
Preliminary Latency/Feasbility Testing	🟫												
Meet with Prof. Stern: Design Guidance	🟫												
Reframing the Problem	🟥												
Waveform Analysis + Latency Reduction Experiments		🟧											
Block Diagrams		🟪											
Alexa/Google Home Experiments		🔷											
Exploring NLP Solutions		🟥	🟥										
Exploring Filtering Solutions			🟦										
Design Review Presentation (10/7 or 10/9)		🟥	🟦										
Design Document (10/14)			🟥	🟥									
Timing Infrastructure													
Building timing infrastructure for testing attack			🟩	🟩	🟧	🟧							
Testing inputs on Siri for sped			🟥	🟥	🟥								
Optimizing audio I/O program latency			🟧		🟪								
Exploring performance: IO vs. Computation bound					🟪	🟩							
Testing timing metrics for performance					🟦								
Investigating NTP solutions					🔷								
Investigating time server sync													
Speech Recognition													
Investigating NLP models					🟦	🟦							
Assess performance of speech processing library						🟥							
Looking into MFCC coefficents for faster speech recognition						🟥	🟫						
Tuning the prediction model that uses MFCC coefficents							🟥	🟥	🟥	🟥			
End to End Integration							🟥	🟥	🟥	🟥			
Stretch Goal: Adaptive Timing													
Investigating dynamic time warping							🟥	🟥	🟥	🟥			
Reducing false positives							🟥	🟥	🟥	🟥	🟥	🟥	🟥
Implementing dynamic time warping								🟪	🟥	🟦	🟥		
Increasing program speed									🟥	🟥	🟥		
MATLAB program rewrite									🟥	🟥	🟥		
Investigating volume threshold											🟥	🟥	🟥
MFCC + Averaging experiment											🟥	🟥	🟥
MFCC + Streaming											🟥	🟥	🟥
In-Lab Demo (12/2)											🟥		
Final Presentation (12/4)												🟥	
Final Presentation Report (12/8)										🟥	🟥	🟥	🟥
Public Demo (12/10)											🟥	🟥	🟥
Slack Week				🟫									

🟩 Cyrus 🟦 Spencer 🔷 Eugene 🟥 All
🟧 Cyrus + Eugene 🟪 Cyrus + Spencer 🟫 Eugene + Spencer

Figure 11: Schedule