Jamming Attack on Voice Activated Systems

Author: Cyrus Daruwala, Eugene Luo, Spencer Yu

Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—We are developing a system capable of detecting wake words for voice activated systems. 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 (200-400ms after the user starts talking). Our solution uses commodity hardware to perform this attack, and will use time synchronization and the spaCy NLP framework.**

***Index Terms*—Activated, Alexa, Defence, DNN, Design, IOT, Jamming, 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 (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 also want 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.

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.

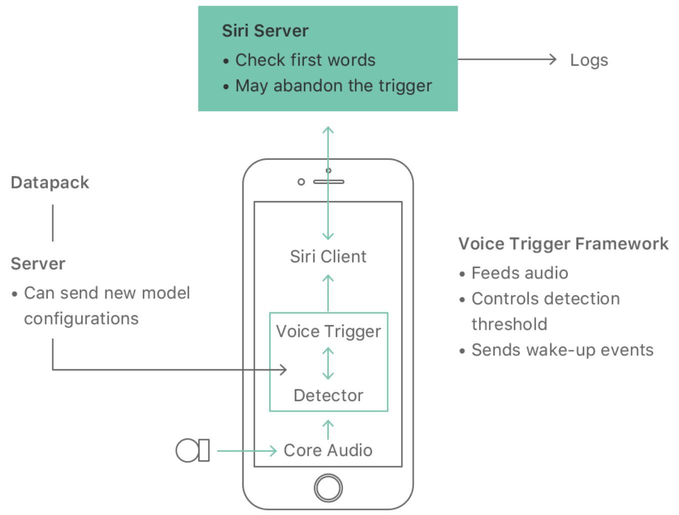

Figure 1: Siri's always-on "Hey Siri" detection system[4]

B. Meeting Latency Bounds

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":

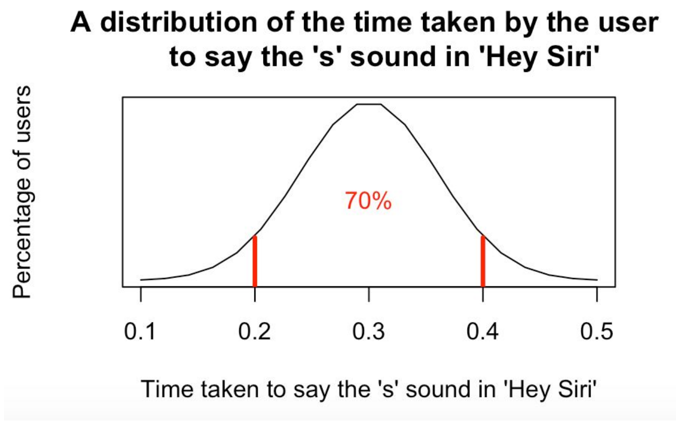

Figure 2: Distribution of time to say the "s" sound

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, which is necessary since this is a semester long project).

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 an NLP system, which is the same as assuming that our NLP system is idealized and takes no time to run. This time will be derived from our verification framework that we are building.

We will test this by building a verification framework that 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.

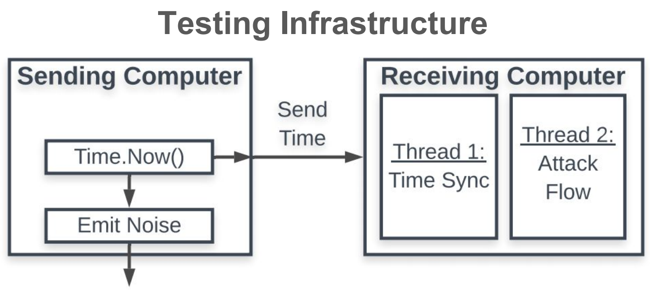

Figure 3: Testing Infrastructure Setup

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 have several proposed optimizations such that we can achieve better results within the limited timeframe of our semester long project.

To improve the runtime of our program and leave potentially more room for the actual computation, we will use parallel threads to improve the runtime of program. We also will try using a latency hiding trick, in which we decrease the threshold at which our system activates. This in turn makes the system more sensitive to noise, and we may have to add an adaptive filter to account for the external noise in the room if the system becomes too sensitive. We want our program to run within the 0.2 - 0.4 second range.

To improve the accuracy of our program and reduce the rate of false negatives, we will use our NLP model to detect the prefix of our wake word ("Hey Siri") and use adaptive filtering if there is available time in the overall runtime of the program. Adaptive filtering would greatly improve the performance by dynamically adjusting the filter's parameters[6] to better match the desired response signals.

Additionally, if we have time, we aim to 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". However, we can better predict the time at which the "s" sound will be said by the user if we model in terms of an input time it takes for the user to say "Hey", the time of the gap between the words "Hey" and "Siri", and predict the output time at which the user starts to say the word "Siri". We need to still prioritize speed in this problem, so we will test this using simple models such as SVM in order to achieve our performance goals while hopefully improving the accuracy of our program.

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, machine learning architecture to more quickly and effectively identify wake words, and the final project deliverable that synthesizes this information.

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 machine learning architecture. We plan on applying machine learning for natural language processing in order to more effectively identify when a user says the wake word instead of relying on a volume threshold.

The third component of concern is the exploit process. We are designing this component as a daemon process that runs in the background of a user computer. This component will use the trained language processing model 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.

This system design is unique in that much of the architecture and design challenges come in the form of support infrastructure for the process itself. Much of the training and time synchronization tweaking will happen outside of the third component but will be absolutely necessary in guaranteeing that the user-affecting portion will be able to accurately deliver on the benchmarks set forth.

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	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.

Figure 4: Comparison of smart speakers

The lower wake word detection accuracy and wider application make Siri a more attractive choice for a project spanning a semester.

B. *Choice of language for implementation*

The two major languages being considered are Python and C given our collective skillset. We have decided to implement this project in Python based on the following:

Consideration	Analysis
Readability/writability	It's no secret that Python is an incredibly intuitive language to pick up. This will help save development time in being able to rely on well-optimized abstractions such as lists, strings, and other readily available data types and paradigms.
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 to binary. 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.
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.

Figure 5: Comparison of Programming Languages

C. *Commodity vs. Specialized Hardware/Environment*

Currently, there is no specialized 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

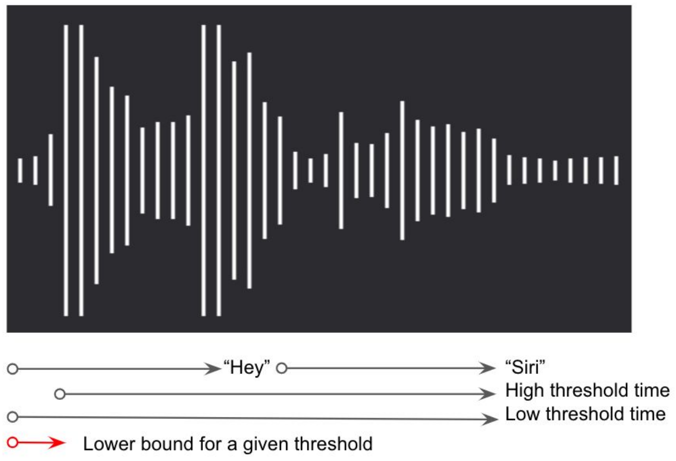

Figure 7: Latency Hiding Trick

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.

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, the 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 education 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

E. Latency vs. Success Rate

Filtering: Using an adaptive filter will lead to our system being able to better model its surroundings to detect signal vs. noise. However, this system will take up processing power and increase the overall latency of the system. A potential way that we can reduce the impact of the adaptive filter on latency is to use an additional thread to tune the parameters of the filter for subsequent iterations, so that this processing can occur in the background and not affect current usage of the filter.

Latency Hiding Trick: Using the latency hiding trick will lead to our system starting up at a lower noise threshold in response to its surroundings. This will lead to an increased amount of processing time available to improve our program's performance and ensure we adhere to our latency bounds. However, this may lead to our system becoming too sensitive to the noise in its surroundings and lead to false detection of the wake word and activation of the attack.

F. Stretch Goal: Adaptive Timing

For our adaptive timing model, we want to have our program run in as little time as possible while increasing the accuracy of our program. It is imperative that we preserve accuracy, so we use 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". If we get to this point we will have to test the accuracy we can achieve while still maintaining our program within the given latency bounds.

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 reaction 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 plan to rely on Apple's time-servers. By guaranteeing the machines are already clock-synchronized, we can then start a signal, timestamp its start, send the timestamp to the receiver, and have the receiver timestamp its reception without having to account for propagation delay in a packet. The design for this testing component can be found in Figure 1.

The actual time synchronization will be built in Python and wrapped with a shell script. The shell script will begin by updating the machine date and time with the information received from time.apple.com. Machine A will then load a stream for a file in Pyaudio and handle as much overhead as possible, such as initializing values and creating chunks to write to stream, before fetching the system time and playing audio. On machine B, the timestamp will be set as soon as Pyaudio recognizes the audio begin emitted from machine A. Machine B will next attempt to output its own audio after setting a third timestamp. We will explore various methods of simultaneous polling and outputting on machine B to determine which method is most effective: single process vs. multiprocessed vs. multithreaded (This is an especially important distinction for our Python script because of Python's use of the global interpreter lock). Machine B will then send the time info back to machine A to process. With these three timestamps, we can identify the time it takes for machine B to recognize the sound and the time it takes for machine B to generate a response, all relative to when machine A begins transmitting the signal.

One aim for this component is to modularize the timing harness in the event that we need to time other components of the project. For the sake of performance, it may not be possible to truly "modularize", but this requirement helps to inform a certain code quality standard that makes modification straightforward as our program develops.

B. Natural Language Processing Module

Thus far, a volume threshold has been able to successfully trigger when the wake word has been spoken, but there are issues with using a simple threshold.

For one, there is no distinction between a user speaking and any other event that passes this threshold. Although false positives are not as large of a concern for this project as are false negatives, for the sake of project integrity, the program should not trigger on every sound that is loud enough to be detected. Furthermore, using volume as a threshold can be incredibly difficult to consistently measure success; would we measure the output volume from the source, the destination, or somewhere in between? On top of that, volume output is measured in dB, but the volume threshold used in audio-based programs like Pyaudio have no clear conversion from its units to dB; as reference, we found that Pyaudio's threshold for slightly above a normal speaking volume is in the order of magnitude of thousands.

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. Bounding the requirements for triggering Siri motivates the use of an NLP module that can more intelligently parse a user's invocation of the wake word without the use of specialized hardware. Again, because the success of this project is so time-critical, it is essential to frontload work to the beginning of the process or offload it entirely; we defer to the latter case when training an NLP model to quickly recognize when a wake command is being said. By training the model beforehand, we can simply attach a pretrained model that processes audio streams, signaling as soon as it recognizes the beginning of a wake word.

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. At its core, the program runs as a daemon process (so as to avoid detection by the user in the way of a readily apparent GUI), which feeds audio into the pretrained NLP model. As soon as the NLP model recognizes the wake word is being said, it will begin outputting a preselected signal designed to obfuscate the suffix of the wake word.

VI. Project Management

A. Schedule

See Figure 6 at the end of our report for a full schedule.

B. Team Member Responsibilities

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

Cyrus will focus entirely on software, overseeing the time synchronization infrastructure setup, and optimizing code for latency. Spencer will focus on noise filtering and NLP models. Eugene will float between software architecture components and signals related work based on each task at hand.

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. If we have additional time at the end of our project, we may test using the Apple HomePod to see how well our project generalizes to other Siri-equipped devices.

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

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 is actually possible for us to finish our desired goals. In addition, it reduces the space of the problem significantly, as we are no longer trying to obfuscate an infinite set of commands.

We also have a series of fallback procedures to reduce the risk of our project not being completed. The greatest risk for our project is in terms of latency bounds. If we do not meet these bounds, our project will not work because we will not be able to output the jamming output in time. If our initial completed 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/C++ for even greater speed boosts.

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. 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 though, and would require more time to work as we are highly unsure about how this technology works.

A. Future Work

We do not plan on working beyond this semester on this project.

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.

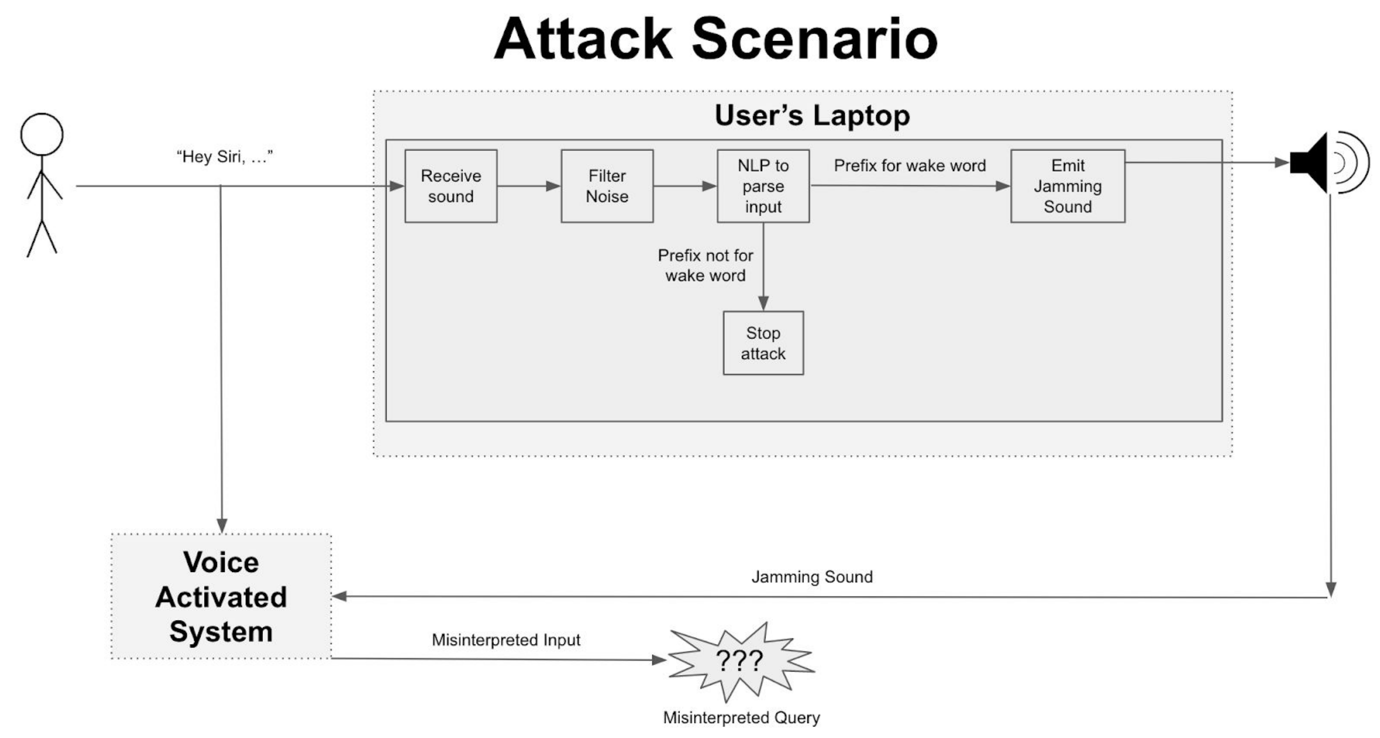

Figure 10: Attack Scenario Block Diagram

	Week of... (date corresponding to the start of the week)										
	10/7	10/14	10/21	10/28	11/4	11/11	11/18	11/25	12/2	12/4	TBD
Determining Jamming Inputs											
Design Review Presentation (10/7 or 10/9)	🟪										
Design Document (10/14)	🟪										
Timing Infrastructure											
Building timing infrastructure for testing attack	🩵	🩵									
Optimizing audio I/O program latency		🔷									
Testing timing metrics for performance		🟦	🔷								
Building program to listen for Siri wakeword		🩵	🔷								
Training model to recognize first half of wake command			🟦								
Performance											
Performance tuning to meet time bounds of command			🩵	🩵							
Generating noise after wake word detected			🔷	🔷							
Integration					🩵	🔷	🟦				
Extra performance tuning					🟦	🟦	🩵				
Adaptive timing							🔷	🩵			
In-Lab Demo (12/2)									🟪		
Final Presentation (12/4)										🟪	
Final Presentation Report (TBD)								🟪	🟪	🟪	🟪
Public Demo (TBD)									🟪	🟪	🟪
Slack	🟦🔷			🟦		🩵			🩵		

🩵 Cyrus 🟦 Spencer 🔷 Eugene 🟪 All

Figure 11: Schedule