Adversarial AI to Prevent Microarchitectural Website Detection Attacks

Sddec21-13
Client/Adviser: Berk Gulmezoglu

Ege Demir
Sean McClannahan
Aaron Anderson
Thane Storley
sddec21-13@iastate.edu
sddec21-13.sd.ece.iastate.edu/

Revised: 12/6/2021 Final Version

Table of Contents

1 Introduction

1.1 Acknowledgement

Berk Gulmezoglu: Significant technical consultation and contribution of equipment aiding in the advancement of our product.

1.2 Problem and Project Statement

-General problem statement:

Microarchitectural attacks are one way to exploit the low-level architectural problems of the hardware, letting attackers gain knowledge about the target user. One of the ways this is achieved is by exploiting the last level cache (LLC) on the computer. As the LLC is shared between all the processor cores, it exposes the memory access patterns of different processes. This vulnerability can be exploited by other processes to determine what the target user is working on.

We will focus specifically on website access detection. As many papers showed, the memory access pattern is often similar every time a website is being loaded on a specific browser. This pattern can be used to classify the websites that run in certain environments like a specific browser, operating system, or processor. As more fingerprints are recorded, the attacker will be able to better identify what websites the user is browsing. As of currently, there are no optimized defense mechanisms implemented in any of the browsers for this.

The detection algorithm can be highly enhanced using a neural network. As more data shows how different cache usage can be classified as certain websites, the accuracy of our neural network based classification algorithm should increase.

-General solution approach:

Initially, we implemented a version of this microarchitectural attack. We prepared the JavaScript program that captures the cache access using a method called Prime+Probe. Then, we classified the data we collected, using a machine learning neural network. Our attack needed to have a high success rate to be relevant to the research done before in this subject.

Our solution to this attack, guided by Prof. Gulmezoglu, was to find a mathematical solution to randomize the memory access pattern. This randomization process is naturally heavy on the resources. A solution already implemented by the Tor browser causes websites to load 500% slower. Our solution was to analyze the behavior of our trained Convolutional Neural Network(CNN) based model, and create distance metrics for each class that can be scalable. We edited a saliency map implementation library to achieve this.

2 Design

2.1 Functional and Non-functional Requirements

Functional Requirements:

- We will have a Javascript-based cache monitoring code
- We will have a Python-based AI model that identifies websites based on output from cache monitoring javascript code
- We will use Javascript code to introduce artificial noise in the cache more mathematically to reduce classification rate of our attacker code.

Non-Functional Requirements:

- Must not increase system overhead by more than 25%.
- Works with more than one browser.
- Attacker code must be able to identify with an accuracy rate of 80% for 50 websites.
- Defense code must lower accuracy of attacker code by at least 60%.

2.2 Previous Work And Literature

Shujian Yu, Kristoffer Wickstrøm, Robert Jenssen, Jose C. Principe. (2019) Understanding Convolutional Neural Networks with Information Theory: An Initial Exploration

Discusses convolutional neural networks (CNNs), and how measuring information flow using newly discovered estimators is much easier, with less approximation. Also talks about data processing inequalities and how that affects training CNNs.

Qiu, Shilin & Liu, Qihe & Zhou, Shijie & Wu, Chunjiang. (2019). Review of Artificial Intelligence Adversarial Attack and Defense Technologies. Applied Sciences. 9. 909. 10.3390/app9050909.

Explains recent developments in adversarial attacks and how those attacks have limited AI in security industries. Discusses how these attacks work, both in training and testing, and the defensive maneuvers that have or are being developed.

Oren, Y., Kemerlis, V., Sethumadhavan, S., & Keromytis, A. (2015). The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (pp. 1406–1418). Association for Computing Machinery.

Shows off an architecture-based attack that goes through the browser. Discusses that it isn't difficult to have the correct environment for the attack, and allows for easy information gathering. It discusses how this attack is preventable, but requires a lot of power from the browser.

Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, & Yuval Yarom (2019). Robust Website Fingerprinting Through the Cache Occupancy Channel. In *28th USENIX Security Symposium (USENIX Security 19)* (pp. 639–656). USENIX Association.

Discusses website fingerprinting attacks, how they work, and what defensive measures have and should be taken. They use machine learning techniques in conjunction with JavaScript to detect cache activity, and show it's accuracy in various environments. They then go into a potential solution, creating "artificial cache activity" to reduce the attack's effectiveness.

These papers explain how the attack works and give data that they acquired using code they wrote. They do not go in depth into potential solutions and did not attempt to write any code to prevent the attack. We are using the attack code they wrote to help us write code to prevent the attack they discuss in their papers. We are also using a neural network to analyze data which was not mentioned in these papers.

2.3 Initial Design

We've been working with, and understanding javascript code used to collect data from websites used in previous projects of this nature. It provided data analogous to cache usage.

The javascript program probes a given website, taking in cache activity for a specified time period, and then it creates a file containing the collected data, which can then be graphed.

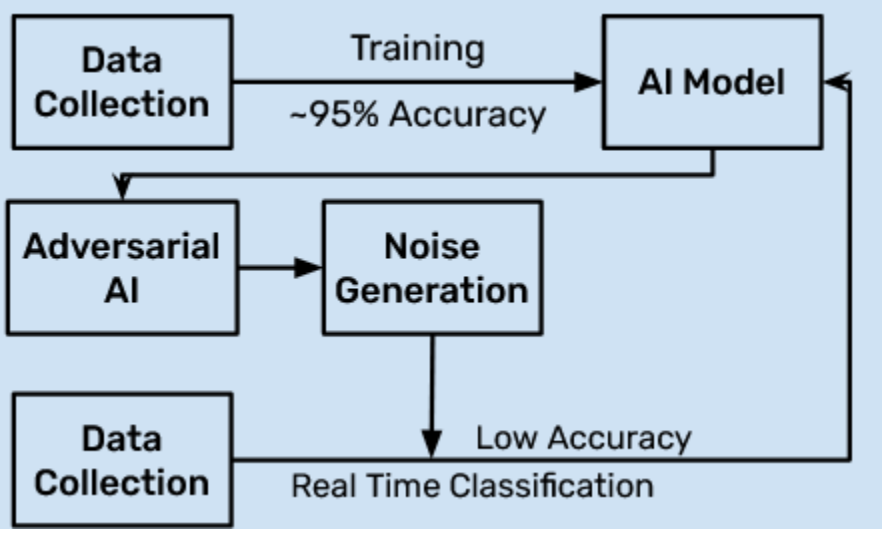
We'll want to feed this collected data through our deep learning model, and train it. We will then use adversarial AI tools to gain insight into our trained model. From there, another script will be created to create noise and fill the cache to try and prevent the attack.

- Data collection JS
- Model training
- Noise generation JS
- Trained model insight

2.4 Technical Constraints

- Architecture dependent
    - How caches are mapped and the size of those caches vary from architecture to architecture, so for that reason we are sticking with one architecture.
- Network latency might change, which might affect our classification performance
    - Specifically, our classification accuracy will be affected.
- The content of the website might change. This will cause the website to have a different cache fingerprint.
    - Social Media sites are the biggest example of this, such as Facebook and Twitter.
- As misclassification rate increases with our defense code, performance overhead is expected to increase with it.
    - Creating artificial noise in the cache could cause performance issues for other processes in the system.

2.5 Final Design Plan



Our final design consists of two different parts; Adversarial AI methods, and noise generation:

Adversarial AI Methods

We constructed a hypothesis that will be true for most AI models: If a website cache memorygram input is getting misclassified, it is getting classified as another website. Therefore, in low scale models, we can use performance intense($O(n^2)$) techniques to identify exploits in the model.

We used a highly edited version of a saliency map library to identify these exploits. Saliency maps infer importance metrics on specific inputs. It uses backpropagation just like how AI models are trained. But instead of changing weights of the model, it goes all the way back to the input layer to see how an input needs to be transformed to achieve a certain output with smaller loss. The absolute value is taken of the transformation vector and then the result gets normalized to produce the saliency map.

We have edited the saliency map library so that the results are not normalized, or taken absolute value of. Instead, it provides raw gradients for input transformation.

Further testing of how many times we need to update a specific input with these gradients to fool our model shows how far away each class is from any other class.

Noise Generation

We needed a method to generate high amounts of noise at specific times during the data collection phase of the attack code. This method needed to be performance intensive enough to disrupt the cache but not so intensive the overall system is impacted. Then using timings generated from the adversarial AI methods; determine the specific points at which to generate noise. The noise coupled

with the timings should cripple the AI.

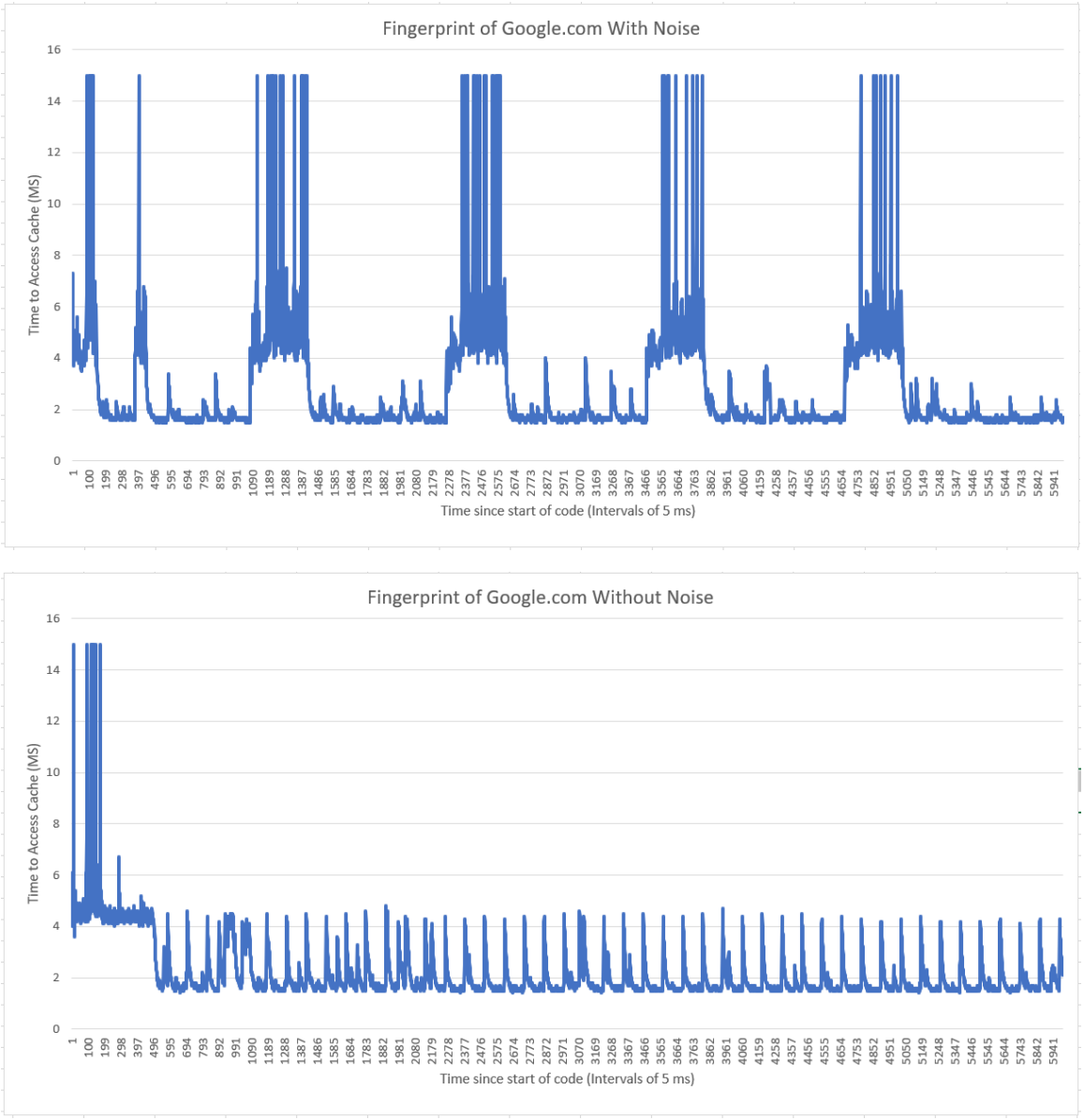


Image 1

2.6 Design Evolution

In the middle of this semester, our team split into two different groups. One group worked on noise generation and the other worked on further analysis of our AI model.

The AI team worked on exploration of our model, using loss distance metrics between every possible input-output class combination. That meant for every website, we would need to apply the gradients created to inputs, and observe how fast the accuracy of our model is changing. We did this with every input-output class combination, and found interesting results.

We have found duality in our model. This duality came from the fact that most websites were very easy to be misclassified as github.com. Further exploration showed that inputs taken from github were also extremely hard to misclassify. This duality gave us an idea we could exploit.

The noise generation team worked on trying to find methods to inject noise as precisely and as high as possible. We needed our code to cause cache misses on our Prime+Probe data collection code. We started with experimenting with different methods such as creating large arrays in a for loop, fetching multiple high resolution images from public websites and visiting various websites that needed high amounts of memory to load. The array generation was performance intensive but the javascript would hang. The image fetching method was lightweight and fast but did not generate enough noise. The method that worked consistently was visiting websites that needed high amounts of memory to load. Then both teams came together to work on the javascript code. The group that worked on the AI model helped the javascript group by providing their analysis of the AI. The AI model group helped finetune the javascript code by specifying the amount of noise that needed to be generated and the time intervals of noise injection that worked the best. The result of the two groups working together is what provided the final version of the Javascript code.

3 Implementation & Testing

3.1 Data Collection

Data collection for both creating a dataset to train our model, and as an attack code for individual website recognition, is done by an edited version of a Prime+Probe script from previous literature. When a specific website url is given to the script, the script will continue opening the website, wait for 30 seconds while constantly implementing the Prime+Probe algorithm to collect cache information, and download the collected data as a csv file. It will then close the website that was opened, open a new instance of itself, and close the previous instance. This automation is more useful for large scale data collection, as it lightens the burden of manually opening up a website constantly.

3.2 Noise Generation

Our aim with noise generation was to inject noise into the cache at certain intervals, hopefully throwing off the model. To achieve this, we took advantage of the noise generated by visiting another website. We developed a noise generator using javascript that would periodically visit a website, therefore injecting noise into the cache. We tested generating different levels of noise at different intervals, hopefully gathering a diverse dataset that would yield definite results.

While running our noise generator, we again collected data modeling cache usage. We used the same data collection script that we initially used to train our model, giving us an accurate idea of what the cache usage looked like while utilizing the intended website, *and* generating noise on top of this. This produced the dataset used to test how noise generation impacted website detection.

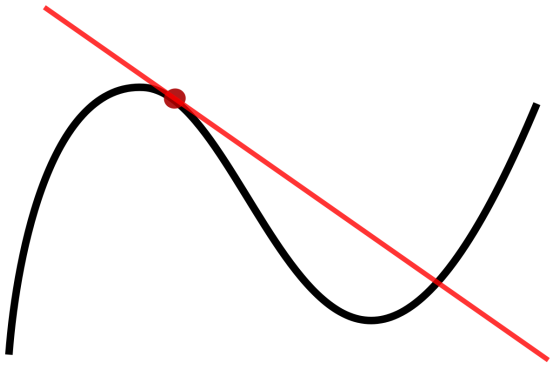
Calculation and Application of Gradients

Before understanding what gradients are, we need to understand the concept of **loss** from machine learning.

When training an AI model, we normally need a dataset to train our model on. This dataset includes **samples**, which are individually fed into our model. Our model returns confidence outputs, which describe how confident the model thinks the input sample belongs to each class (website). But, we already know what these numbers are supposed to be, because the input samples are labeled to describe which class they belong to. We therefore can create a distance function for each sample that describes how far our model's prediction is from the perfect prediction. We call this distance: **loss.** You might've already guessed, a lower loss refers to a better prediction. Therefore, during training, we are constantly trying to lower this value.

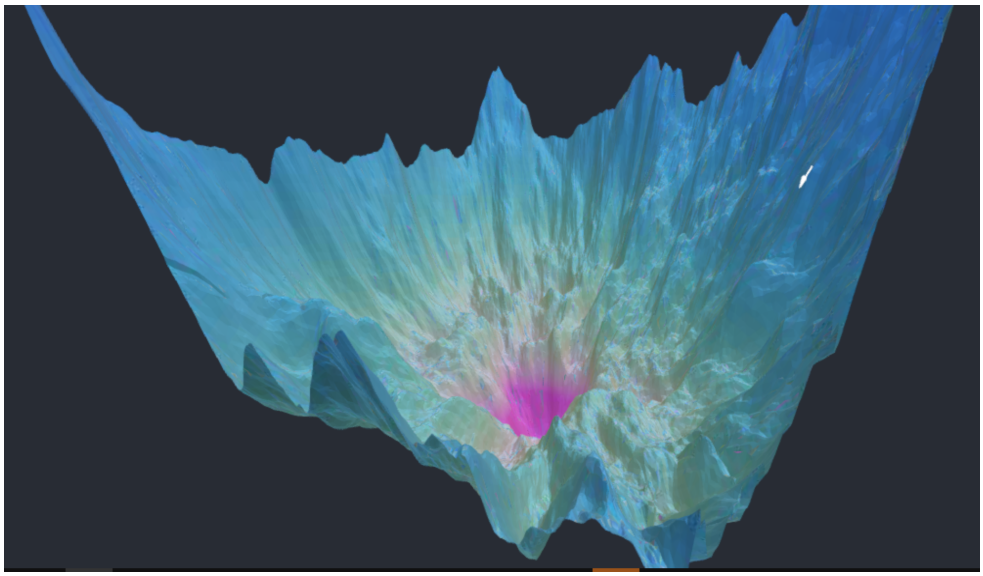
This loss value gives us some information about how we should change the weights of our model, so that we can improve our prediction.

Now comes a little bit of calculus:



The derivative of a function gives you some information about where the minimum (local or global) of that function is. In the figure above, you can see that if you were to take a step towards where the slope is decreasing, you also take a step towards the minimum.

In machine learning, imagine that this function describes the loss with respect to a specific weight. The derivative tells you how to change the weight to cause a decrease in loss value.



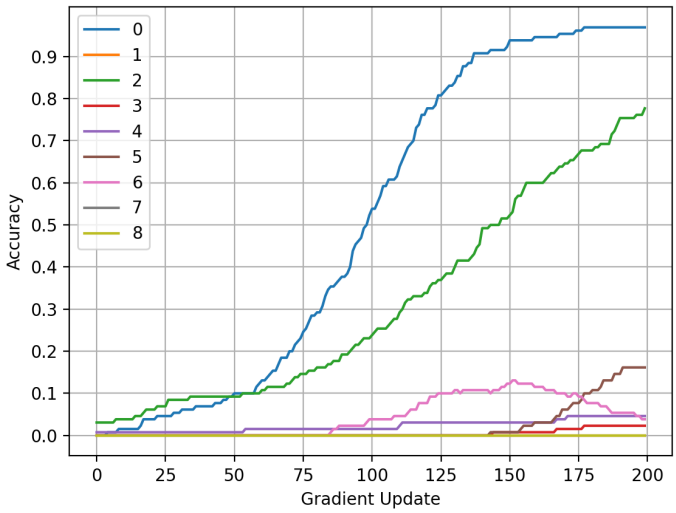
When we start describing loss value in higher dimensional space, we have to describe derivatives in a more multivariable context. That is what gradient is.

The figure above is a 2D representation of a loss function. The white arrow represents the gradient, which is also what we calculate. Be mindful that our actual loss function is more than 100,000 dimensional, which would be impossible to visualize.

What the saliency map algorithm does is, instead of loss value being calculated with respect to weights, it calculates with respect to the input. This gives you information about how to change the input instead.

Normally, for most saliency map algorithms, it also takes the absolute value of gradients, and normalizes the resulting values, in the hopes that the resulting vector gives insight into how much specific parts of an input matters.

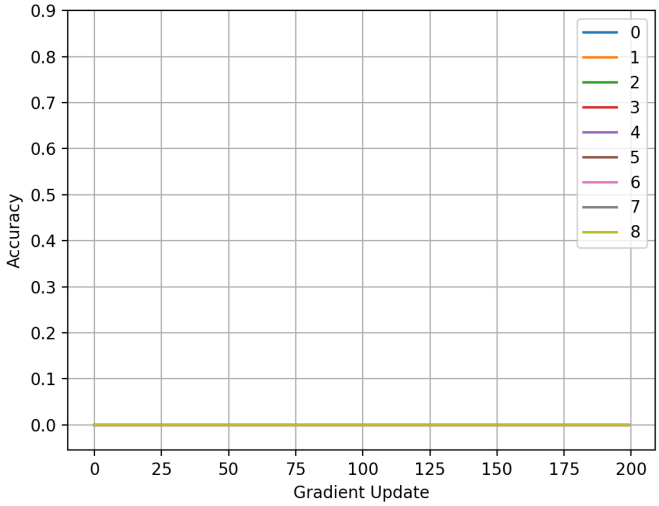
We modified a saliency map library, tf-keras-vis, so that the algorithm actually stops at calculating gradients. The calculated gradients would vary for every input-output class, which is also why this process has time time complexity of $O(n^2)$. Using these gradients, we updated cache memorygram inputs, and measured how many input updates it takes to fool our model. Remember that in this case, gradients are analogous to the noise that we are actually supposed to introduce.



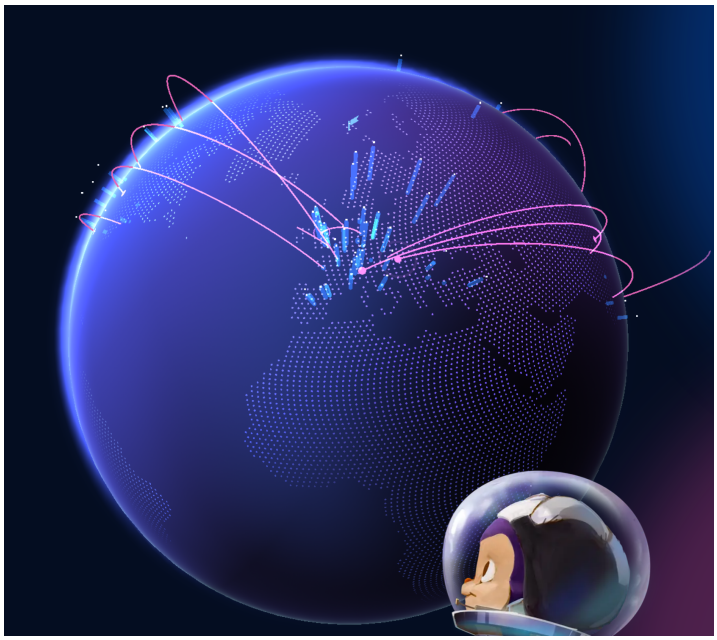
The figure above is the result of one of these calculations. It specifically belongs to apple.com website, which is labeled as 1 in this graph. You can think of the above figure as how many times we needed to update apple.com with it's respective gradient for every other website, so that our model would start to misclassify apple.com samples as other websites.

As we did this calculation for every class, we noticed something peculiar about class number 2, which is the website github.com. It seemed very easy for every website input to be converted to github input, but it was very hard for github input to be miscalculated.

The figure below shows the application of gradients specifically for github. Notice that after updating github inputs 200 times, the accuracy of our model on github inputs don't change at all. This means that it's harder to trick our model to recognize github cache data as any other website.



We believe this is due to the fact that Github has a performance intensive interactable globe animation in its front page, shown at the figure below.



3.4 Testing

To gather a benchmark for how much strain was caused on the system by our noise generation system, we ran the data collection code and monitored the system usage. Then, we ran our noise generation code alongside our data collection and gathered the system usage data for this case. We found that when 10 second intervals were used for noise generation, there roughly a 10-15% increase in cpu usage. When using 1 second intervals we found that there was nearly a 40-50% increase in cpu usage. Upon further testing, we found that these numbers were fairly inconsistent, but managed to stay within a reasonable range. This leads us to conclude that there is a linear progression from ~10% to ~50% increase in system usage when generating noise at 10 second and 1 second intervals respectively.

3.5 Results

Our initial testing has shown our data gathered by the “attack” script had several consistencies when gathering from the same website. Then after training the neural network for 9 different websites it was able to classify websites based on their fingerprint with around 90% accuracy.

4 Closing Material

4.1 Related Products and Literature

In terms of AI and noise, deep-fakes are very similar to microarchitectural attacks. They are generated by using neural networks to analyze video data to create fake videos of a subject. This is done by extracting thousands of faces from multiple videos or pictures of the subject and the target of the swap.. The AI is then trained to swap the faces of the target subject and source. One proposed solution to this problem is adding noise to the images or videos in order to confuse the AI. This subject is very heavily researched and documented as Deep-fakes have been a high priority problem for both the US Government and Tech Giants like Google.

4.2 Conclusion

This project demonstrates that these attacks can be very accurate without proper security measures. Our AI model, based on the data given to it, was very confident and correct in identifying websites. However, with proper measures taken, it is possible to throw off and confuse the model. Injecting noise into the cache using javascript, the model grew less confident with its now more incorrect answers. Finding where injecting the noise using saliency maps would matter most was also a key component.

There are some shortcomings with our research, however. Our model is on a smaller scale, despite what we set out to do. Our requirement to gather data on fifty websites wasn't met, only gathering enough for nine. Our ability to measure system performance and overhead while we measured noise injection was limited, and not one-hundred percent definitive. With more time, a deeper dive into fixing these issues would be easily possible.

As far as future work is concerned there is much to be expanded upon as microarchitectural attacks are a relatively new topic in the cyber security field. If we were to continue with this project our first step would be to collect more fingerprints of different websites in order to increase our sample size. This would allow us to retrain our model and would give us more data to analyze. On the topic of analysis we would also do further research into timings of noise generation. Specifically, seeing if our AI model can be tricked with less noise and more precise timing. This could potentially decrease the system overhead of our current noise generation code. In conclusion, we have succeeded in providing a proof of concept that shows that cache, when injected noise with consideration of exploits of the AI model, can be disrupted enough to lower the accuracy of said model greatly.

4.3 References

Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, & Yuval Yarom (2019). Robust Website Fingerprinting Through the Cache Occupancy Channel. In *28th USENIX Security Symposium (USENIX Security 19)* (pp. 639–656). USENIX Association

Oren, Y., Kemerlis, V., Sethumadhavan, S., & Keromytis, A. (2015). The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (pp. 1406–1418). Association for Computing Machinery.

Qiu, Shilin & Liu, Qihe & Zhou, Shijie & Wu, Chunjiang. (2019). Review of Artificial Intelligence Adversarial Attack and Defense Technologies. Applied Sciences. 9. 909. 10.3390/app9050909.

Shujian Yu, Kristoffer Wickstrøm, Robert Jenssen, Jose C. Principe. (2019) Understanding Convolutional Neural Networks with Information Theory: An Initial Exploration

Appendix I: Operation Manual

1. Prerequisites
    a. Operating System: Linux
    b. Browser: Chrome
    c. Languages: Python
2. Data Collection
    a. Run the javascript code to collect cache fingerprint data from the desired website. This should be specified in your javascript. The website can be changed in test2.html at line 265.
        i. `google-chrome test2.html`
    b. This code will output a csv file containing the fingerprint of the specified website
    c. Create a folder to contain all of the specific fingerprints for the specified website.
    d. For each website we recommend that around 100 fingerprints be generated so that the AI has enough data to train on.
    e. Repeat this process for each website one intends to train the AI on.
3. Model Training
    a. Put the collected csv data in a folder.
    b. Run dataPreprocessing.py in the folder/
        i. The python script will output 2 combined csv files(X values and corresponding classes)
    c. Run sddec_model.py to train the model using the newly generated dataset
        i. sddec_model.py outputs TrainedModel.h5 which refers to the newly created model for both testing and predictions.
4. Noise Generation and Data Collection
    a. Choose website that will be target of data collection (e.g. adobe.com)
        i. Be sure to choose a website that the neural network has trained for
        ii. Remember that the website can be chosen in line 265 of test2.html
    b. Run both the noise generation code and data collection code at the same time
        i. `google-chrome noiseGen.html & google-chrome test2.html`
    c. Check to see if the csv file has been downloaded
        i. The filename of the csv should start with ssdec21-13

Appendix II: Scrapped Designs

1. Originally planned to have an automatic tool that would detect abnormal cache usage and deploy noise generation code to counter a microarchitectural attack. This was determined to be out of the scope of the project and would take too much time to develop
2. We planned to have our project support multiple browsers but the training it would take to support it was not worth the effort. This is because we decided that proving that our noise generation code can thwart our AI is a higher priority than supporting multiple browsers.
3. Initially, we were going to use python libraries to implement adversarial AItechniques. However, we saw that these libraries were designed for use on images and would not translate well into cache fingerprints.
4. For noise generation we considered fetching high resolution images as a way to insert noise into the cache. We also tried creating large arrays. Upon further investigation we discovered that the noise generated using this method was not significant enough to cause the desired disturbances.