# Snoopy: Website Fingerprinting with JavaScript

Jack Cook — 6.888 final project

## Introduction

Side channels attacks are an elementary type of attack that can be exploited on basically any modern computer system. They rely on the fact that computers are required to share resources in order to be practical and economically viable. This shared resource can be anything from cache contention, to power usage, to network latency.

For this project, I originally took advantage of the cache contention side channel. I collected traces in the form of cache contention over time, known as a “memorygram,” and opened new tabs in my browser while these traces were being collected. After training a model on a large collection of memorygrams, I was able to predict new tabs that people opened in their browsers. When distinguishing between 10 websites, we were able to reach 97% accuracy, and when distinguishing between 100 websites, we reached 75% accuracy. This work demonstrates why side channel attacks are practical to exploit from JavaScript, and how we can defend against them.

## Background

This work is based in large part on findings from Shusterman et al., when they first demonstrated how this type of attack was possible. After collecting cache-contention-based memorygrams, the authors used MATLAB’s classification learner to train a model that accurately predicts websites opened in new windows. While the attacks are impressive (and scary), the paper left me with several unanswered questions. The authors demonstrated that these attacks worked with very high accuracy, but in my mind, did not prove the practicality of this attack.

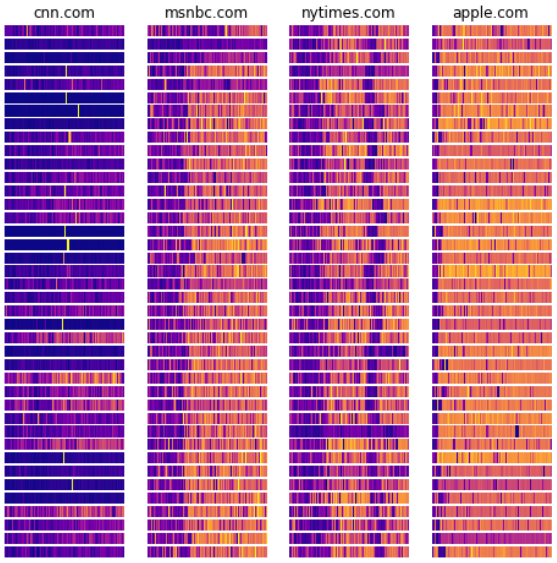
For example, the authors claimed that “If the target hardware configuration is known beforehand, [...] the attacker can customize the parameters of the JavaScript attack code to match the target PC’s parameters.” This implies that they believe their attack would carry over well to computers that were not present in the training data. However, the authors neglected to prove this.

Shusterman et al. also made assumptions that significantly limited the practicality of this attack. For example, each trace collected in their paper is 30 seconds long. If this attack were to be embedded in a webpage, many users will likely close the page before 30 seconds is up, and before opening any new tabs. The authors also didn’t experiment with added noise, and whether the results held up when other programs were running on the user’s computer in the background. For my projects, I wanted to push this type of attack to its limits. How practical is it to pull this attack off?

# Overview and Evaluation

## 1. Replicate Shusterman et al.

I spent much of the first half of this project replicating the original paper by Shusterman et al. I wouldn't be able to experiment with my questions and ideas if I didn't have a working version of this attack, and the authors didn't publish their source code. After a few days, I had my first working implementation of the attack. I wrote a Selenium script that would start data collection, open a new page, stop data collection, close the new page, and save the resulting memorygram. I collected all of my data in Chrome, and I was careful to match any details provided by the Shusterman et al. paper. For example, the size of the array that my JavaScript program allocated was also 20MB,




Memorygrams collected from four different websites over 15 seconds. Darker colors indicate more cache contention.

At the start, I was only distinguishing between four websites: cnn.com, msnbc.com, nytimes.com, and apple.com. I picked these because I figured CNN, MSNBC, and NYTimes are data-heavy websites that likely load multiple assets in a unique way, while Apple likely loaded fewer assets and completed loading quicker. I wanted to prove that this uniqueness would show up in a collection of memorygrams.

In the figure above, you can see that this is the case. Apple loads relatively few resources, and we can see that the cache frees up very quickly after a brief period of contention at the beginning. NYTimes loads a large asset around 10 seconds after the page loads, and CNN exhibits high cache contention the entire time that the page is open. The results are so apparent here that these websites could be classified correctly by simple inspection. It took longer than I expected to tune my parameters correctly, but I ultimately reached 100% accuracy through my 4-way classification problem by using Random Forest models with the scikit-learn Python package. When I expanded my dataset to include 10 different websites, my accuracy dropped to 90%, still well above the random-choice baseline of 10%. After further tuning my attack by adding new worker threads, my accuracy eventually reached 95%, at which point I moved on to other parts of my project.

## 2. Trace collection methodologies

I spent a large portion of this project investigating different ways to collect a memorygram. It quickly became clear to me that improving this step was paramount when it came to determining my attack's highest possible classification accuracy.

I started my project with a cache-contention-based attack such as the one employed by Shusterman et al. Their program allocated a 20MB array in memory and iterated over it in a variety of patterns that targeted various cache set-counts and associativities. After tweaking my code for a while, I decided to try switching to something simpler.

[REDACTED]

Algorithm for recording a single memorygram. 15000 is the length of each trace in milliseconds, and 5 is the duration of collection for each individual datapoint.

It turns out that this timing-based attack was not only simpler, but it also significantly increased my classification accuracy. After switching from my best cache-contention-based attack to my best timing-based attack, classification accuracy improved by about 4%. For this reason, I stuck with this timing attack for the remainder of my project.

## 3. Introducing WebAssembly

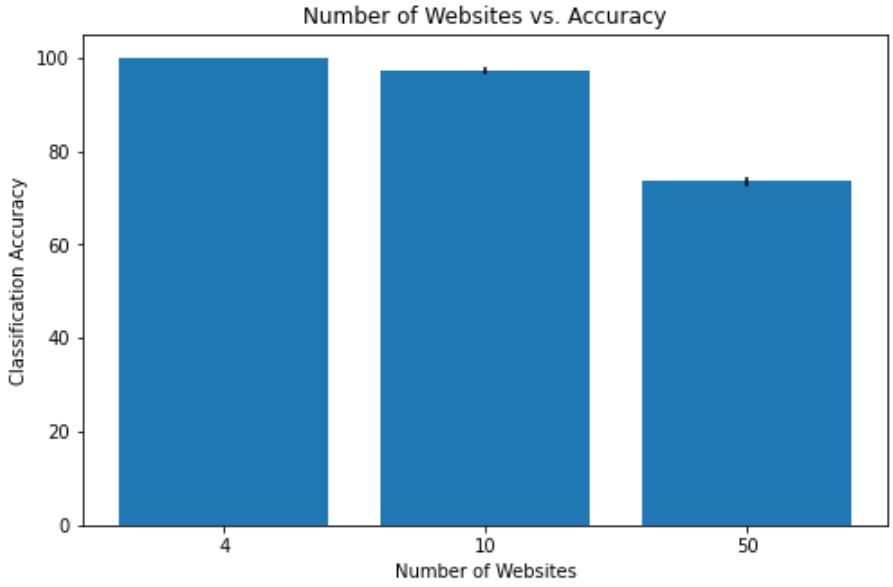
One question that was on my mind as I was reading the Shusterman et al. paper was whether their attack's accuracy could be improved if they compiled their code to WebAssembly. After all, the official WebAssembly homepage boasts how WebAssembly can run at "native speed," and many companies have seen success with WebAssembly when it comes to increasing speed. For our attack, the increased speed offered by WebAssembly could translate to making more memory accesses per second, resulting in more precise memorygrams.

I ported my memorygram collection code (pseudocode in section 2), from JavaScript to Go, which I then compiled to WebAssembly using the TinyGo compiler. All models trained on the WebAssembly version of our code were about 3% more accurate than the comparable JavaScript-based versions. This is what allowed me to reach my final accuracy of 97% for 10-way classification.

# Evaluation and Results

## 1. Increasing number of websites

Due to time constraints, I spent much of my time improving my 10-way classification accuracy, which is a relatively small number of websites compared to the 100-way classification explored by Shusterman et al. This allowed me to collect reasonably large training datasets within just a few hours. Near the end of the project, I left my laptop to collect a 50-website dataset over the course of 2 full days.

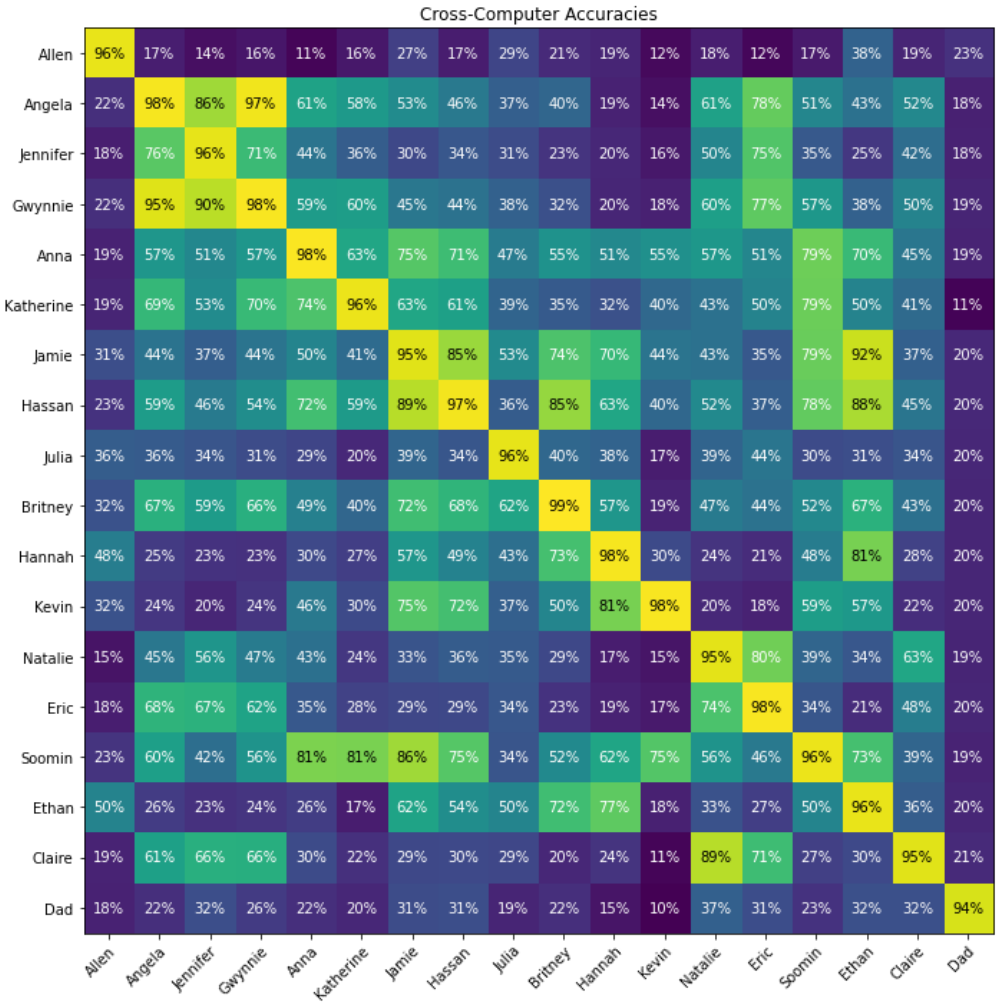



After training and evaluating on this dataset, my model was able to achieve 74.8% accuracy, which is competitive with the 72.5% 100-way classification accuracy achieved on macOS by Shusterman et al. More data collection needs to be done to see whether my attack holds up during 100-way classification.

## 2. Evaluating on novel computers

To me, the most interesting question was whether this attack would work on novel computers. Could we collect training data on one computer, and evaluation data on a brand new computer, while maintaining high classification accuracy? I ended up spending a large portion of my project answering this question.

After cleaning up my code and making it easy to install and run on new computers, I sent instructions to many of my MacBook-owning friends, hoping that at least a few of them would have computers with the exact same specifications. Each laptop collected data for at least six consecutive hours on the top 10 websites ranked by U.S. visitors. We can see the results of this data collection below.




Each cell depicts the accuracy of a model that was trained by the row-wise computer, and evaluated on the column-wise computer. All 18 computers are post-2015 MacBook Pros.

Angela, Jennifer and Gwynnie (group A) have 2017 MBPs with 2.3 GHz i5 processors. Anna and Katherine (group B) have 2018 MBPs with 2.3 GHz i5 processors. Jamie and Hassan (group C) have 2018 MBPs with 2.3 GHz i9 processors. Britney and Hannah (group D) have 2019 MBPs with 2.6 GHz i7 processors. Natalie and Eric (group E) have early 2015 MBPs with 2.7 GHz i5 processors.

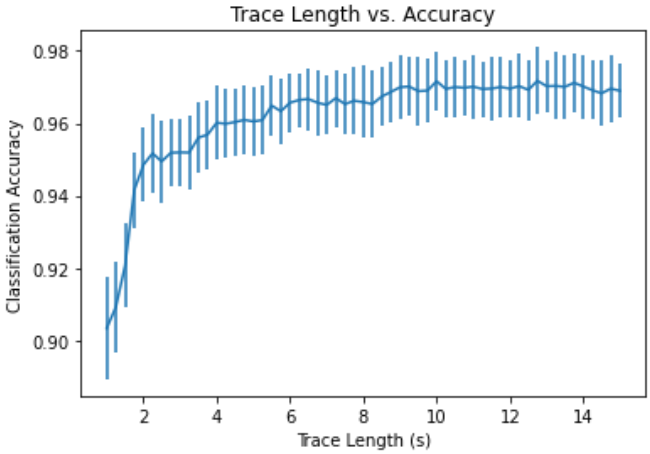
In the diagram above, we can see that groups A-E all exhibit increased accuracy when training and evaluating within the same group, rather than outside of their group. Interestingly, some models exhibited high accuracy on models that had almost nothing in common with the computer they were trained on. We can see this most clearly in the correlation between Jamie's laptop, a 2019 16" MacBook Pro with an i9 @ 2.3 GHz processor, and Ethan's laptop, a 2017 15" MacBook Pro with an i7 @ 2.9 GHz processor. The reason for this correlation is unclear, and more analysis needs to be done to identify what causes a model to carry over well between different computers.

However, the accuracy is almost always lower than when training and evaluation was done on the same computer. For this reason, I believe Shusterman et al. are misleading to ignore this

step of data collection and analysis. This step is crucial to proving that this attack is practical. In the real world, we will likely not have unfiltered access to the target computer for several hours in order to collect data.

### 3. Analyzing trace length

The final question I wanted to answer from the Shusterman et al. paper was how trace length affects accuracy. Requiring that traces are 30 seconds long makes this attack inherently less practical. In the real world, users click open and close pages within a matter of seconds. A recent analysis by Amazon showed that every 100ms of latency ultimately cost 1% in sales, which proves that many users are extremely quick to close webpages. A practical version of this attack would need shorter traces.




After reaching the final version of my attack, I went back and repeated my model training procedure, except I only included the first $n$ seconds of each trace. In the graph above, we can see that this attack holds up very well to shorter trace lengths, and after awhile, classification accuracy plateaus. After just 250 milliseconds, we still achieve 50% accuracy, well above our 10-way classification baseline of 10%. After 1 second, we reach 90% accuracy, and after 4 seconds, we reach 96% accuracy. The fact that we're able to retain most of our classification accuracy with trace lengths that mirror likely real-world user behavior significantly increases the practicality of this attack.

## Future Work

There are multiple directions that this work could be taken in the future. I believe more research needs to be done before we can definitively say whether this attack should be taken seriously.

1. Collecting data with different browsers

   For simplicity, I collected all of my data in Chrome, because I found that Chrome offered higher-precision timers than Firefox and Safari, which I believed would correlate with higher accuracy. Tor tries to blunt these types of attacks by offering timers with an extremely low precision of 100ms, however Shusterman et al. were able to work around this to achieve accuracies that were still well above chance.

2. Collecting noisy data

   I would have liked to collect data that was more noisy. During all of my data collection periods, and in the instructions I sent to my friends, I closed all programs before running the Selenium script. This inherently reduces the attack's practicality because most users browsing the web will have other programs open, such as a music player, an email client, or maybe even other browser tabs.

3. Further investigating memorygram collection methods

   There is no way to prove that my memorygram collection procedure, or any other memorygram collection procedure, is the best one. A new procedure with different ideas or different parameters could prove to collect better data about activity that is occurring on the same machine. During class, Thomas suggested combining my timing-based procedure with cache collection data, which was an idea I hadn't considered before. A better method could likely boost classification accuracy by several points.

4. Building defenses against this attack

   I would have liked to try to build a browser extension that defends against this type of attack, or at least considered what it would take to make this attack infeasible. One idea I had was adding random noise to timers offered in the browser, so that my code can't count on the fact that each datapoint in a memorygram is collected in exactly the same amount of time. Unfortunately, I didn't have time for this one, but it is important to think about how to make this attack less dangerous.

5. Investigating better models

   For this project, I stuck with Random Forest models and Extra Trees models from scikit-learn, because they gave me the best accuracy. However, Shusterman et al. decided to exclusively use CNN- and LSTM-based models. No matter what I tried, I couldn't get these types of models to work, but I'm sure more time could have allowed for the use of the models. And perhaps they would have improved my accuracy.

# Conclusions

After having completed this project, I'm glad to have pulled it off, and I'm glad I was able to learn so much along the way. At the start of this semester, I had never heard of a side channel attack, and I might not have believed that microarchitectural attacks could be pulled off from JavaScript. At a minimum, it helps me understand Richard Stallman's anti-JavaScript philosophy a little bit more than I previously did. Security vulnerabilities can really come from anywhere.

I additionally believe I achieved most of my goals when it comes to practicality. I still think that the Shusterman et al. paper makes many unrealistic assumptions when it comes to modeling real-world user behavior, and while this attack still isn't completely practical in its current form due to the noisy data limitation, I've removed some limitations around duration, improved accuracy with WebAssembly, and demonstrated that the authors' original assumption that this would carry over to new computers is realistic.

# Appendix

## 1. Intuitions with side-channel attacks

One lesson I learned during this project is that my intuitions will not always match what will achieve the best results when it comes to side channel attacks. For example, I decided that Chrome would be my browser of choice for data collection during this attack because I believed higher-precision timers would give me better accuracy. However, while doing this write-up and

reading over the Shusterman et al. paper again, I realized that the authors actually achieved their best accuracies in Firefox. It's not immediately clear to me why this is.

Another instance of this is in how I ended up switching to a timing-based attack. It's not immediately clear to me why a timing-based attack would outperform a cache contention-based attack, as differently websites load highly distinctive and unique assets while loading, which would theoretically affect the cache in predictable ways. However, I guess the same can be said for power consumption. The results don't lie.

## 2. Source code

If you're interested in exploring my project's code, you can check it out (mostly undocumented right now) at [REDACTED]. The `classifier` directory contains the Jupyter notebook I used to train my model, `wasm` contains the memorygram collection code and the compiled .wasm file, `src` contains the code for the data collection website, and `record_data.py` is used to perform data collection. The setup instructions I sent to my friends can be found here: https://www.notion.so/jackcook/Snoopy-Setup-Instructions-b8e2e3c14c6145d8b9186ece609568ad

## 3. Final note

I just wanted to say I've really enjoyed 6.888 this semester. I think I was the youngest and least experienced student in the class, and while I had a strong software background coming in, I had very little hardware background beyond 6.004. Despite this, I now feel like I have a high-level understanding of several concepts in hardware security, and this class has definitely made me reconsider what systems I consider to be secure or insecure. I hope our paths cross again in the future!