Spring 2021 ME/CS/ECE759 Final Project Report
University of Wisconsin-Madison

Cracking the Wifi PMKID on a GPU

Justin Hiemstra

May 7, 2021

Abstract

WPA2 Wifi cracking has been around for some time now, but prior to a new attack discovered in 2018 (the subject of this project), a successful attack required catching the initialization handshakes between a network and a supplicant. However, in 2018, the hash-cracking software producer HashCat realized that many Wifi stations enable roaming, and that on these networks a different type of identifier can be requested without the need for a supplicant directly from the router – the PMKID. Because the PMKID is generated through a series of cryptographic algorithms applied to the network password, BSSID, and other constant strings, it is possible to brute-force network passwords by using the network's BSSID and generating PMKIDs with a dictionary of passwords. If any generated PMKID matches the PMKID broadcast by the network, the correct password has been found.

This project has implemented a proof-of-concept PMKID brute-forcer that uses a dictionary of passwords (all of whichi must be 8 chars) and parallelizes PMKID generation by leveraging CUDA to run the cryptographic calculations on a GPU. Although the HashCat's implementation of PMKID cracking wasn't tested for speed, this implementation did not perform as well as hoped, completing only ~500 PMKID generations per second. While the initial project proposal had suggested that data transfer would the the bottleneck and that asynchromous memory transfers would be implemented, it was discovered that computation – and thread resources – is in fact the limit. As such, the program was only tested against relatively small password files that were randomly generated using python (with the correct password inserted upon generation).

Link to Final Project git repo: https://euler.wacc.wisc.edu/jhiemstra/me759-jhiemstra/-/tree/master/FinalProject

Table of Contents

1. General Information

1. Electrical and Computer Engineering
2. MS Professional
3. Justin Hiemstra (just me)
4. I am not *yet* interested in releasing my code as open source code – it isn't good enough.

2. Problem Statement

The Pwnagotchi (a portmanteau of "pwn" and "tomagotchi") is an open-source hardware and software package built on raspberry pi 0 that captures information from wifi networks that can be used for "cracking" those networks. When fully assembled with all its bells and whistles (ie an e-ink display, RTC, GPS shield and battery), the device costs ~$50, literally putting wifi hacking in anyone's pocket. At the time of writing, there are over 100,000 of these devices registered through the creator's statistics page, although this is likely a severe lower bound as many security-conscious people may opt not to register their Pwnagotchi. When the pwnagotchi is in range of a vulnerable Wifi network, it uses a trained machine learning model to optimize the speed with which it can capture crackable network parameters. Upon capture, these parameters are stored in the common Wireshark PCAP format, a filetype that can be fed to popular hash cracking programs like HashCat. If the attacker is lucky, and their password dictionary contains the correct password for the network, the password is guaranteed to be returned, giving the attacker future access to the network. It is important to note that this cracking is not done on the Pwnagotchi, but rather the information supplied by the Pwnagotchi is transferred to a much more capable machine.

The goal of this project was to understand one of the wifi attacks made possible by the Pwnagotchi and to leverage HPC to roll a custom brute-force dictionary attack against the supplied information. In particular, the PMKID attack[1] was chosen because it does not require the Pwnagotchi to force deauthenticate any supplicants from a network (something that may be illegal in certain areas[2] and was not done for this paper). Instead, the PMKID is supplied directly by routers that support roaming when queried by any device, connected or not. Because the PMKID is generated using a combination of crypto algorithms applied to the network's BSSID (ie its name), the network's password, and a fixed string, password's can be recovered by using a candidate password list and checking the generated PMKID against the PMKID supplied by the network. If the two match, the password has been found.

The goal of this paper was to implement the afore-explained PMKID attack as fast as possible by parallelizing PMKID generations through the use of a GPU.

3. Solution Description

Since the generation of a PMKID requires SHA1 hashing, the HMAC-SHA1-128 algorithm, and the PBKDF2 algorithm, I first found a cuda implementation of SHA1 – you can't reinvent the wheel, afterall! However, the implementation I found wasn't exactly what was needed, so I modified to to still run on cuda, but instead of taking a batch of things to hash, my modified code assumes that each thread will need to hash various things sequentially. This is the case with PMKID generation, as the PMKID is a truncated form of the output of the Pairwise-Based Key Derivation Function 2 (PBKDF2) that runs HMAC-SHA1-128 on an input 4096 times. While this is a simplification of the process, the takeaway should be that the generation of a single PMKID is a series of *sequential, non-parallelizable* computations, and as such, the parallelization component of

1 For the first documentation of this attack, see https://hashcat.net/forum/thread-7717.html

2 Although the network I attack in this paper is real, the owner of the network gave me their consent to test my program against their real-world password and wifi setup.

this attack is done by generating lots and lots of PMKIDs at once, using each thread to check a password. This is why the SHA1 implementation was changed to run on a single thread. A slightly more detailed description of the generation of a single PMKID is as follows:

Password → PMK = PBKDF2(Password, SSID, 4096 iterations)
→ PMKID = HMAC-SHA1-128(PMK, “PMK Name” | MAC_AP | MAC_STA)

After SHA1 was working, I implemented the other algorithms in CUDA, each running on a per-thread level. (This itself took much longer than I anticipated because I was not familiar with any of this crypto before proposing my project, and naively assumed PMKID’s were just a simple hash of some string). I did my best to store useful things in shared memory to reduce latency, but because each thread was running on a different input password, there was a limited amount of things that could reasonably stored in shared memory.

Once the algorithms were running correctly and outputting the expected PMKIDs, I generated a password file to test against (of the 40,000 passwords, I stuck the real password around line 35,000). Notably, my code assumes that every password in the file is 8 characters long, as this simplified the code and the network I tested against had a real-world 8-character password. While this prevented me from testing against well-known password files from Crackstation, I did verify that the target password did appear in many password files, and so this stands as a proof-of-concept implementation. However, because I initially anticipated running the code against large files, I broke up the data read/transfer from the host to the device into smaller pieces that are stored in a buffer, one buff_size at a time. This was helpful because it prevents trying to load a very large file into memory all at once, something that may be doable on Euler, but that would not work well on most machines – and even on Euler, the GTX 1080s we had access to only have 11GB of memory.

The basic program flow is as follows

1. Load some passwords from a file into a page-locked buffer.
2. Transfer those passwords to the device.
3. Launch many threads on the device.
3. For each password, a single thread on the device computes its PMKID.
4. Return a boolean if any of the PMKIDs match the PMKID obtained from the network.
5. If the returned boolean is true, terminate computation on the GPU and copy that password to from the device to the host.
6. Print the password and execution time.

In my proposal, I had suggested that I may try worksharing amongst GPUs and using asynchronous memory transfers. However, I did not implement these because of how slowly my program ended up executing. Although functional and correct, my early code was plagued with issues like eating too many registers per thread and spilling back to local memory, making it so I could only launch 256 lethargic threads per block. When I profiled my code, I was getting a theoretical maximum occupancy of 12% and and achieved occupancy of 5%. After timing memory transfers, it was apparent that memory transfer was not the bottleneck as I had expected – instead it was how resource intensive the crypto algorithms were. I will point out, too, that PBKDF2 is designed to be intensive and slow exactly to prevent brute-force attacks. Because I wasn’t hitting bandwidth limits, I decided to continue trying to slim my code, and was able to successfully cut the registers needed by each thread by nearly 50%.

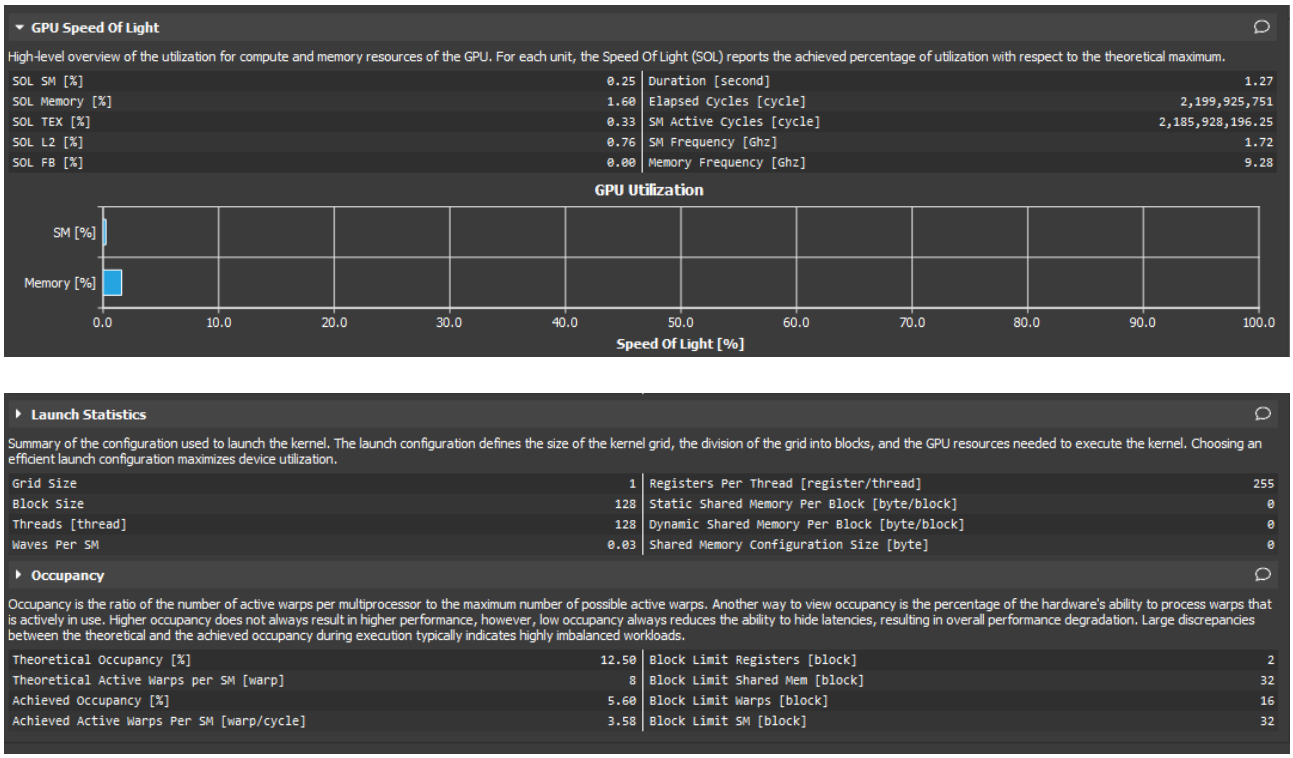
4. Overview of results. Demonstration of Project

The final result I obtained was a program that successfully parallelized a brute-force attack against a wifi PMKID, albeit not as quickly as I would have hoped. When the program was launched with a buffer containing the entire password set and 512 threads per block against a password file with 40,000 passwords, it took about 71.783 seconds of GPU computation time to find the password that was located at line 35,000. This equates to about 492 password attempts per second. When it comes to password guessing speeds, this is quite abysmal.

Initially, the code was even slower. When I first got things running, I had each thread allocating too much memory, and as a result I ran out of memory on the GPU. After quite a bit of debugging, I solved that issue but then was still running into another issue: when I compiled with the `–ptxas-options=-v` option, it was showing

```
976 bytes stack frame, 12 bytes spill stores, 12 bytes spill loads
ptxas info    : Used 255 registers, 384 bytes cmem[0]
```

The profile code looked like:



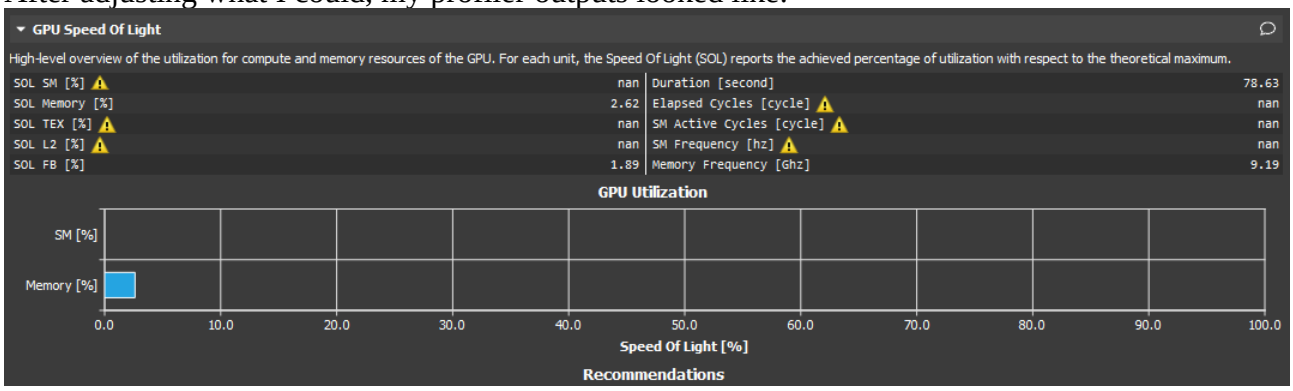
Not very good!

After looking up the GTX 1080's specifications, I realized that 256 threads each using 255 32-byte registers completely exhausted the architecture's resources. I spent the better part of another day combing through the code and slimming it to use even less memory. Finally, I got the registers per thread down to 119. and was able to launch 512 threads per block, although this didn't result in a very noticeable speed increase.

The biggest speed increase came from using as much shared memory as I could, and using `cudaMallocHost` to pin the host memory for the buffer of passwords. Initially I had done this

because I wanted to use asynchronous memory transfer, but eventually decided it wasn't worth implementing because of how slow the computations were happening.

After adjusting what I could, my profiler outputs looked like:



While the SM utilization looks lower, this is just because the values didn't compute correctly, and so weren't reported. Thankfully, the formerly awful occupancy was increased by a factor of nearly 5x, and while higher occupancy doesn't always imply faster code, the code itself saw about a 20% speed increase.

5. Deliverables

The following is a list of each file included in my git repo, along with a description of what each file is:

- PMKID.cu – The meat of the project, the code that you should compile & execute
- generated_small/big.txt – the password files I tested against.
- generate_passwords.py – the script I wrote to generate the password files.
- generate_real_PMKID.py – a script I borrowed from the web to check if my program was generating the correct PMKID
- Various profiles (*.nsight-cuprof-report)
- Various screenshots of the various profiles.
- run.sh – the Slurm script I used to run my code
- some_info.txt – some info I used to put things together
- threads_comparison.txt – checking to see how threads/block affected speed.

Here is the compile command to compile my PMKID.cu, assuming you're using whichever version of cuda is used on Euler when `module load cuda` is issued:

```
nvcc PMKID.cu -Xcompiler -O3 -Xcompiler -Wall -Xptxas -O3
-gencode=arch=compute_61,code=sm_61 --ptxas-options=-v -o
PMKID
```

To run the code:

```
./PMKID threads_per_block <generated_small.txt | generated_big.txt>
buff_size
```

For example:

```
./PMKID 512 generated_big.txt 460800
```

should run for about 1:10s.

There's a max of 512 threads per block or the code doesn't work, and I used `460800` for the buffer size. IT IS ESSENTIAL THAT THE BUFF SIZE BE A MULTIPLE OF 9. This is because each 8 character password is stored on its own line with a newline character (thus 9 bytes). If buff_size is not a multiple of 9, the program will not parse the passwords correctly past the first buffer and will fail to find the password unless everything is loaded into the first buffer.

The code structure has already been mostly explained, but to reiterate and give additional detail, the first chunk is a modified CUDA implementation of SHA1. After that there are a bunch of `__device__` functions that implement the crypto portion of finding a PMKID, and then finally there is the `main()`that reads in a chunk of passwords and launches the cuda kernel against those passwords.

6. Conclusions and Future Work

The hash algorithms used to generate keys are intentionally sequential, intentionally computation intensive, and intentionally memory intensive. This is done to slow brute-force attacks down, because there's a big difference between being able to perform a billion hashes per second and 500 hashes per second. While PMKID generation can be paralellized on a GPU, doing so still doesn't yield fantastic results (at least not my implementation) because the generation of each PMKID is a sequential, intensive process. That being said, I had a lot of fun learning some new crypto and implementing it on the GPU, as well as learning about how this attack works. Although my desktop has a GPU, I wasn't able to get HashCat working to compare its speed to my own code (there are some nvidia bugs on Ubuntu and messing with them destroyed my display, forcing me to reinstall my OS). However, in the future several things could be done to improve my own code. First, I could implement multiple GPU support, although this would be a relatively small speedup because its already so slow; doubling the pace of a snale doesn't make for a race car. This actually wouldn't be too hard, as it could be done simply by dividing the buffer of passwords in two and sending each chunk to its own device – the two GPUs don't need to communicate because their calculations are independent of one another. I would also like to keep looking through the code to see if I can reduce the intensity on each thread, because I think doing so would speed things up. If the compute bottleneck could be reduced sufficiently, it would be worth implementing asynchronous memory transfer, because then the attack could be run against larger password files more efficiently.

References

[1] *new attack on wpa/wpa2 using pmkid_2018,* Hashcat Advanced Password Recovery. URL: https://hashcat.net/forum/thread-7717.html.

[2] *Cuda hashing algos,* Mochimodev. URL: https://github.com/mochimodev/cuda-hashing-algos

[3] *PBKDF2,* Wikipedia. URL:https://en.m.wikipedia.org/wiki/PBKDF2

[4] *HMAC,* Wikipedia. URL:https://en.m.wikipedia.org/wiki/HMAC