Parallelized Reaver

Kabir Singh, Thomas Mason

May 2019

Abstract

In our final project, we explore parallelizing the main cracking procedure of the open-source WiFi hacking tool, Reaver. By exploiting vulnerabilities in the WiFi Protected Setup (WPS) protocol and utilizing OpenMP, we were able to see significant speedups in searching for and determining the correct WPS PIN to break into a vulnerable access point (AP). Due to recent hardware security improvements, however, this exercise was purely academic (and legal) as we simulated network exchanges with a "virtual" access point.

1 Introduction

1.1 WPA2

Every time a device communicates with a wireless access point (AP), it must first establish a reliable and secure connection through a pre-defined security protocol. Currently, the most common protocol is Wi-Fi Protected Access, or WPA, in its second iteration (WPA2). Initially proposed in 2003 as a replacement for the dated Wired Equivalent Privacy (WEP) protocol. WPA sought to modernize secure wireless security protocol by adopting updates in the IEEE 802.11i standard, specifically the Temporal Key Integrity Protocol (TKIP), which employs a per-packet key. In this way, additional packet integrity checks could be performed as each packet was bundled with a dynamically generated key. WEP used one statically generated key, which meant that any compromised network was permanently vulnerable [TGM13].

At a high-level, WPA2 works by establishing a four-way handshake between the communicating device and the AP:

1. The AP sends a probing message to the device to establish the key requirements needed for communication with the AP
2. The device sends its own message with integrity information, including authentication
3. The AP verifies the device message by checking against the key requirements and sends a temporary key
4. The device verifies that the temporary key sent by the AP is valid and finally sends a confirmation message to the AP

WPA2, although a marked improvement over its predecessors, is still vulnerable to a brute-forcing password crack attack as the 4-way handshake does not protect against an adversary snooping the network exchange and then brute-forcing password guesses. A common tool for this is the popular "Aircrack-ng". However, as with any application whose attack surface is mainly a password, choosing a strong password will often make wifi hacking by exploiting this vulnerability intractable. It is also worth noting that in late 2018 a new protocol WPA3 was announced which addresses these vulnerabilities, although it is not yet widely deployed.

1.2 WPS

WiFi Protected Setup (WPS) was introduced in 2006 as an alternative method for non-tech savvy users to connect to WiFi. Instead of entering a long passphrase, users could instead use a pre-programmed 8 digit PIN that came printed on routers enabled with WPS. In the early 2010s, it was discovered that this PIN verification procedure had a serious and exploitable vulnerability.

Ignoring the obvious issue of having it printed clearly on the router itself, the WPS PIN was exploitable via both "online" (sending active packet requests to AP) and "offline" (using returned hashing data to compute hack locally) methods [VP17]. Programs such as PixieWPS are able to perform an offline crack in as little as a few seconds (4s on average in our testing), whereas programs such as Reaver are advertised as taking as "little as four hours" for an online attack. Our project was to explore how to parallelize the hacking procedure Reaver utilized in its online attack [Vie12].

1.3 Reaver

The Reaver tool is straightforward in its implementation and how it exploits the WPS vulnerability; it simply pre-computes a complete set of valid WPS PINs and sends them sequentially over the network to the AP under the guise of legitimate WPS access requests. Reaver makes use of the User Datagram Protocol (UDP) protocol - a network communication protocol that is a handshake less and packet-based. Unlike Transmission Control Protocol (TCP), a UDP transmission does not throttle network traffic to one packet stream per port or enforce in-order transmission, both of which would be detrimental to this attack. In addition, Reaver includes an option to rate-limit the packet transmission, as many newer router architectures (wisely) only allow a certain number of WPS unsuccessful attempts per time interval. Under these conditions, Reaver essentially works by establishing a network connection, computing a table of PINs, and sequentially sending WPS-formatted packets to the target router with each of the PIN guesses, rate-limiting its traffic to avoid being banned from further attempts, until it finally finds the correct PIN.

2 Literature Survey

Interestingly, although there existed literature on parallelization of related and commonly bundled tools (in particular PixieWPS and Aircrack-ng, which used the Pthreads framework), very little discussion of parallelizing Reaver existed at the time of our project. To see why, we looked at the existing literature on designing other, similar tools such as Aircrack-ng and PixieWPS. Both Aircrack-ng and PixieWPS act as far more straightforward cracking tools, and both operate primarily (Aircrack-ng) or exclusively (PixieWPS) as offline cracking tools [Vie12]. As detailed in [D'O], Aircrack-ng works by gathering authentication packets containing the hashed key at the beginning, saving these to disc in a pcap file, and brute forcing guesses using this file in an offline fashion. This means that Aircrack-ng doesn't have to navigate network traffic and the associated delays and limits placed on speed for their primary functions, making it highly computation-intensive rather than I/O intensive. PixieWPS similarly works off of locally available cryptographic keys gathered upfront, resulting in a similar situation. As the approach taken by these designers shows, parallelization generally gives good performance results for Aircrack-ng and other offline attack tools.

Reaver is primarily an online tool, which is likely why it was not parallelized previously. However, in spite of this fact, if deployed against older router models that do not rate-limit the WPS PIN guesses of devices [Vie12], Reaver has the potential to behave more like an offline attack from the perspective of a parallel application. Although the PIN guesses would still be I/O bound in that they would need to transit a network, if no rate-limiting occurred, we believe a non-negligible performance gain could be achieved by parallelizing the CPU-intensive portions of the tool (such as PIN generation and the cracking loop). Given the potential gains from parallelizing this tool, and given that doing so makes for an interesting educational project in this space, we propose adding multicore functionality as our project.

3 Method

While analyzing, Reaver, we found two areas amenable parallelization: key generation and the main cracking loop (overall structure can be seen in Figure 1). These were the only two areas of the program that involved many repetitive and non-codependent steps over a large problem space - other functions in the program typically either had too many internal dependencies or operated on too small of a space to be worth the investment in communication and overhead.

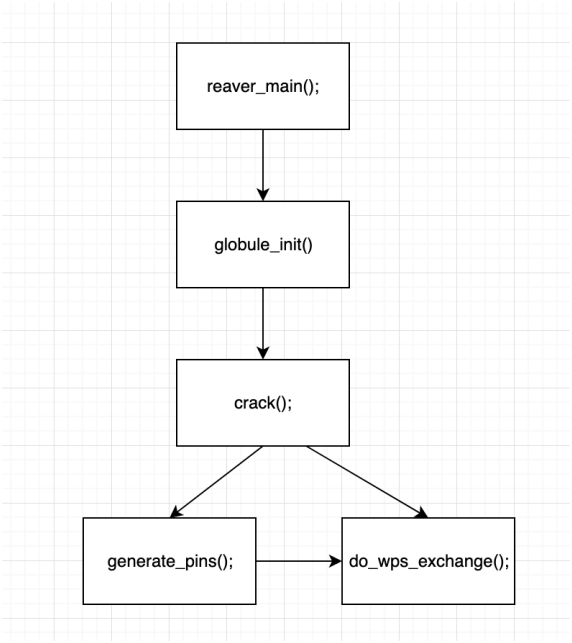

Figure 1: Code dependencies in reaver

3.1 Key Generation

One of the most challenging aspects of this project was to figure out a logical data structure such that each thread could access global variables as needed when making network requests, constructing keys for network requests, and keeping track of program status. Some of these variables were the same across processes, others not. We settled on creating an array of the `global` struct, with one entry for each thread. We then distributed process-specific information to each of these `global` structs. Then, for each of their (many) get/set methods, we indexed into the `global` struct array by the calling process' id within the function itself. In this way, function calls around the program would remain the same and the parallelism would be abstracted way for other components of the Reaver program.

3.2 Main Cracking Loop

The core of this project was parallelizing the main cracking loop. In this loop, the program establishes a connection with the AP it is trying to hack, and then sequentially sends PIN guesses until the correct one is found. We found this to be the most attractive part of the program to parallelize! Given our earlier work in distributing the `global` information to each thread, the OpenMP directive to parallelize the thread ended up becoming a seamless process. Each thread acted as its own, calling the same functions in the same manner, but iterating over and modifying internal data local to the process itself. Finally, we modified the state checking conditions in the loop to be parallel – this was our approach:

Every thread searched through the first keyspace – 0000 through 9999. Once a thread found the correct sub-key, the *chosen thread*, all other threads ceased to make requests and the *chosen thread* continued to search the second keyspace, 000 through 999. This was achieved through a somewhat straightforward arsenal of global state variables that each thread could see.

4 Experiments and Discussion

4.1 Experimental Setup

We had initially partially simulated our implementation of Parallelized Reaver, but eventually moved to a fully simulated environment so that a version could be compiled and run on the `crunchy` servers. For our benchmarking we used `crunchy1` which had the following specs:

- 64 Homogeneous Cores

- AMD Opteron Processors (6272)
- x86_64 Architecture
- 2 Threads per Core

Since we observed negligible speedup during th Key Generation process, we chose to focus on the performance improvements of the Main Cracking Loop. We used OpenMP's `omp_get_wtime()` to observe our runtime. We placed a single call to `omp_get_wtime()` before and after the call to the cracking procedure.

We tested with a variable number of threads, defined as the global constant `NUM_THREADS` in `src/params.h`. Each PIN used a randomly generated PIN (see section 4.2). We performed approximately 10 tests per each configuration and averaged the performance.

4.2 Simulation

In order to use Reaver as it exists, the machine it is running on have a network interface card (NIC) that supports "Monitor Mode" and has access to WiFi access point that it is legal to attempt this attack on. However, the CIMS machines at NYU do not meet these criteria, so we decided to simulate the network exchanges in order to be able to run and benchmark our code on the CIMS servers. To do this, we scrapped the actual exchange of packets over a live network and instead created a series of functions to give a locally generated set of values that would conform to what the function was expecting (target interface, target MAC address, etc), as most of these were simply used for ensuring that Reaver was communicating properly over the network. However, one central function that would be affected by this change was retrieving the hashed PIN value, which would now need to be simulated.

As the network packet exchange was now being simulated, we needed to simulate receiving the pin from the AP network interface. We did so by using a random number generator and having this pick randomly from the list of pre-generated PINs that Reaver uses as its search space. We then ran it through the same hash function that the guesses are put through and assigned the newly hashed "simulated" PIN to the object that would receive the PIN pulled from the network packet in a non-simulated situation.

4.3 Results

Threads	1	2	4	10	20
Avg Time (s)	0.45	0.28	0.26	0.32	0.37
Speedup	-	62%	71%	45%	22%

As the above table shows, the program was able to achieve a substantial level of speedup, with the best results occurring when using four threads. This is likely due to the size of the key search space upon which Reaver operates. The creation of each thread entails a certain amount of overhead and thus costs the application in terms of speed. However, when dealing with a limited search space (recall there are only 11,000 possible keys that can be tried in this application), each additional thread will divide the work it performs into smaller batches, meaning at a certain point the creation of additional threads will entail greater overhead costs than improvement gains from dividing up and already small pool of work. Based on the results we have obtained here, it appears that between 4 and 10 cores provide the optimal balance between overhead and division of labor. If this were a tool with a larger key space (such as Aircrack-ng) more cores would likely have continued to give performance gains, but in this situation the best performance is achieved with a smaller number of cores.

4.4 Theoretical Real-World Speedup

Since our program simulated network traffic, we were unable to glean what the exact real-world speedup would potentially look like. We were curious to know what sort of estimates we could compile, so we took a few measurements and assumptions and arrived at the following.
Assumptions:

- We are running an attack on an old router without WPS rate-limiting.

- We measured the min, max, avg, and std dev of network package response times from within our Reaver program. We found the following and assume it holds for our theoretical scenario.

Network Latency	Min	Max	Avg	Std. Dev
Time (ms)	5.180	14.592	9.867	3.793

From there:

- Given that the distribution of keys is uniform within the keyspace, the expected key value is 50005002 (5000 for expected key in uniform distribution between 0000 and 9999, 500 for expected key in uniform distribution between 000 and 999, and 2 for the security bit), or the 5,500th key. This example isn't really interesting, so let us only consider the worst case scenario: the correct PIN is at the very end of the keyspace.
- For the original version of Reaver (without rate limiting), at 11,000 sequential attempts at an average of 9.867ms each, that would put the expected runtime at approximately 108.54s.
- For our parallelized version of Reaver, since the keyspace is subsdivided by thread, each thrad only has to make an upper bound of 10,000/n + 1000 requests. Since the network latency would overcome the core distribution overhread we saw in the many-core results, the speedup would be more substantial per added thread. As the number of threads increased, the upper bound of requests made per thread would decrease. For example, with 4 cores, no core would be able to make more than 3500 requests, or 34.53s of runtime. In fact, we postulate the speedup would impressive until a large number of threads, at which the overhead would be ludicrous compared to the potential speedup, and we would run into the same issues as in the fully-simulated version.

5 Conclusion

In this project, we were able to identify two areas that we could apply the parallelization techniques we learned in class to. From our experiments in a simulated environment, we observed a significant speedup over the sequential version of the code. However, we found the speedup peaked at a set number of cores, leading us to believe there is a significant trade-off between the distributed workload and the actual distribution process. We also recognize that since this exercise is entirely simulated, it is difficult to extrapolate from this to its real-world performance (which is tricky to directly observe given the legal status of testing on WiFi systems one does not own).

References

[D'O] Thomas D'Otreppe. Aircrack-ng tool suite. https://www.aircrack-ng.org. Accessed: 2019-05-4.

[TGM13] Amin Babiker Tagwa Gali and Nabi Mustafa. A comparative study between wep, wpa and wpa2 security algorithms. In *International Journal of Science and Research*. IJSR, 2013.

[Vie12] Stefan Viehböck. Brute forcing wi-fi protected setup. In *Krebs on Security*. Wordpress, 2012.

[VP17] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in WPA2. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[VP17] [TGM13] [Vie12] [D'O]