Compression and Side Channel Analysis

Introduction

Hardware side channels have been a long standing problem for the security research community. Side channels are a class of security vulnerabilities where information is leaked through a side effect of the system, rather than through the program directly. For computer systems, an attacker might be able to monitor runtime, power consumption, memory usage, or contention in order to gain information about secret data in computation. For example, if an attacker can monitor the runtime of a program, and the runtime varies according to some secret data value, the attacker can gain insights about the secret value to make a more informed guess of the secret. In order to prevent leakage through side channels, the execution traces of secure hardware designs need to be independent of secret values.

On the other hand, data compression algorithms exploit the dependence of the input data, and use the probability distribution to do efficient compression. There are a variety of mechanisms that are used to improve compression based on the dependence between symbols to be encoded, including using seed files, modeling the data as a Markov Chain, and pretraining a model based on the expected distribution of the data. Therefore compression and security seem to have opposing goals: secure hardware should reduce the observability of data dependence to an attacker, and efficient compression should capitalize on the dependence within the source data distribution. For example, the optimal compression ratio of an encoding algorithm (and perhaps other metrics, such as runtime, power consumption, memory usage, etc.) is entirely dependent on the probability of the source data distribution. Therefore we want to explore how compression can induce side channel security vulnerabilities.

In this project, I first conducted a literature review of security and compression research to understand how this problem has been considered in the past. Over the past two decades, there has been significant work looking at how side channels could theoretically arise from data compression. Additionally, researchers have demonstrated a variety of side channel attacks which exploit data compression. After reviewing the literature, I ran experiments to show how the compression ratio can be used to leak information about a secret key in a block of text. For two different compression algorithms, LZ77 and Huffman coding, I measured the compression ratio while varying an attacker's “guess” of the secret, either in a seed file for LZ77 or colocated with the input data for Huffman. I measure the compression ratio while increasing the number of correct characters in the attacker's guess to see if an attacker can use the compression ratio in order to make a more informed guess of the secret key.

Background & Motivation

There has been significant research looking into hardware side channels from data compression, with the first paper dating back to 2002 [1]. For a more detailed overview on prior work, see the literature review section from my earlier milestone report (included in Appendix A). Rather than reviewing each paper, I will discuss interesting details, including the type of compression algorithm that was exploited, the type side channel that was monitored by the attacker, and the assumed attacker strategies.

Many of the initial compression-based side channel attacks, including CRIME [2], BREACH [3], and Heist [4], exploited Deflate, which is essentially LZ77 plus Huffman coding. However other compression algorithms have been shown to be vulnerable, including Base-Delta-Immediate (BDI) which has been proposed for on-chip caches [8] and other LZ-like codes, such as LZ4, PGLZ, zstd, [5] and Snappy, an open source compression algorithm by Google based on LZ77 [7]. Lastly, one paper looks at software-transparent texture compression algorithms embedded in GPUs. They are able to reverse engineer an unknown compression algorithm using a memory access pattern based side channel [6].

Researchers have been able to exploit various side channels, including compression ratio, timing, and memory access pattern. Looking at both demonstrated and proposed attacks, the far majority of these attacks exploit the compression ratio [1-4,7,8]. However, [5] considers leakage through variation in decompression time, and [6] considers leakage through variation in data-dependent DRAM traffic and

cache footprints. Therefore it seems that compression might leakage information through a variety of different practical side channels.

Lastly, prior work considered a few different attacker strategies. Many of the compression ratio side channel attacks assume that an attacker can co-locate some data with the victim data, including [2-5,7,8]. The GPU based attack [6], however, assumes a fairly weak attacker, where an attacker cannot affect the input to the compression algorithm and can only observe timing information of memory instructions. Another attacker strategy difference is between Heist [4] and its predecessors CRIME and BREACH [2,3]. While CRIME and BREACH require an attacker to be able to observe or manipulate network traffic, Heist does not. Obviously for all of the prior work, the attack requires a strategy where the attacker is able to observe the specified side channel in some way. Depending on the attack, this might be the compressed data size for compression ratio side channels, the latency of instructions for timing side channels, or cache hits and misses for memory access pattern side channels.

It is clear from this prior work that compression algorithms can induce hardware side channel vulnerabilities. Additionally, the variety of different attacks discussed above shows that it is quite difficult to ensure that compression is secure. As compression continues to be implemented throughout the hardware/software stack, it is likely that new side channels will be discovered. For example, compression has been proposed or is already used throughout the memory hierarchy, in large systems targeting machine learning, databases, and cloud computing applications, and embedded in hardware for video and image decoding. Therefore, it is important to understand the security vulnerabilities in order to ensure that compression used in hardware and software is secure.

Implementation

I explored and demonstrated how an attacker can gain information about a secret key embedded in compressed text by observing the compression ratio. I considered two different compression algorithms with some attacker controlled input, either a seed file or an attacker guess that was co-located with the source data. I indexed into random English text coming from a Sherlock Holmes novel to generate a data block of a specified length. Then I generated a random secret key of lower case letters, and inserted the key into the text at a randomly generated index. Therefore, both the source text and the location of the secret in the text were unknown to the attacker. In some experiments, I prepend the secret with an attacker-known prefix. Finally, to compute the compression ratio, I divided the length of the encoded data by the length of the original data.

I first considered the LZ77 compression algorithm. I started with LZ77 for a few reasons: first it was shown in prior work that LZ77 is vulnerable to compression ratio side channel attacks. Second, since LZ77 is looking for matches in the data, it intuitively makes sense that the compression ratio would decrease with a seed file that has more matching characters to the secret. This is because we expect that if you add the correct next character to the seed file, you would have to store one fewer literal in the compressed data, decreasing the size of the encoded data. Third, I chose to start with LZ77 because modifying the seed file was an interesting (and perhaps novel) way that the attacker could interfere with a victim's compression algorithm. In the literature I reviewed, seed files were not used to facilitate an attacker's guess of the secret. This could potentially extend to other attacker strategies where an attacker can affect or pretrain a model in order to reveal information about secret data.

In the LZ77 experiments, the attacker could modify a seed file to show that with more characters of the secret in the seed file, the compression ratio decreases. I ran experiments with and without an attacker-known prefix prepended to the secret in the text. In the experiment with an attacker-known prefix, the seed file contained this prefix followed by the attacker's guess of the secret. In the experiment without an attacker-known prefix, the seed file only contains the attacker's guess. Additionally, I ran experiments with 500 character source data and with 10,000 character source data to see how the length of the source data can affect the change in compression ratio.

In order to compare the effectiveness of a side channel attack across different compression algorithms, I also considered the leakage through the compression ratio of Huffman compression. In order

for the attacker to be able to affect the compression ratio, the attacker's guess was co-located at the beginning of a 500 character data block before encoding. Then I collected the compression ratio over multiple trials, each with different source data and a randomly generated key.

Lastly, I implemented an attacker strategy where an attacker incrementally guesses the secret key using the compression ratio. The text was 500 characters long, and the secret key was 16 characters and prepended by an attacker known prefix. I incrementally guessed a single character by adding it to the previous seed, and then I kept the character which resulted in the minimum compression ratio for the next iteration.

Results

Looking at the results from LZ77, Figure 1 shows that with more characters of the secret used in the seed file, the compression ratio will decrease, as shown with the blue data points. The red data points show the compression ratio remains approximately constant when a seed file consists of randomly generated characters. I reran the same experiment with source data of 10,000 characters. The results in Appendix B show with longer source data, the compression ratio still decreases with more characters of the secret in the seed file. However for longer source data, the change in compression ratio is relatively small for each character added to the seed file.

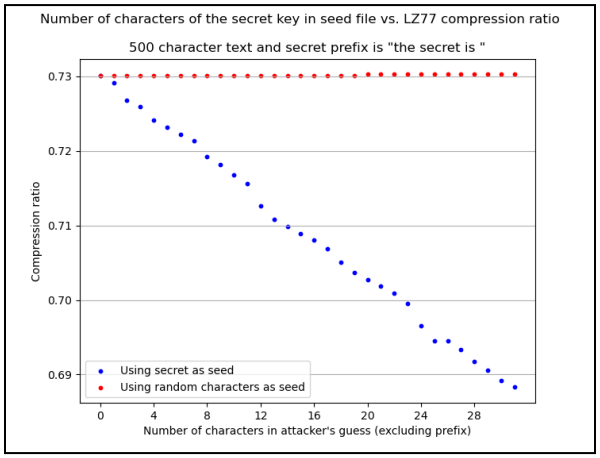

Figure 1: Varying seed files with 500 character text

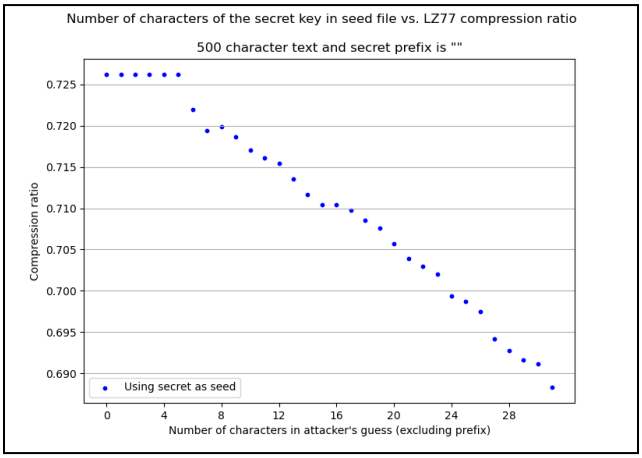

Figure 2: Varying seed files without a prefix to the secret

I then considered how the prefix prepended to the secret affects the compression ratio. Without an attacker-known prefix, there is no decrease in compression ratio until the seed file contains at least 7 secret characters, as shown in Figure 2. After the 7th character however, the compression ratio decreases, as we saw in the original experiment. Therefore, without an attacker-known prefix, the side channel does not seem to leak any information until there are at least 7 correct characters in the seed file.

Next I replaced the LZ77 encoder with a Huffman encoder to see if Huffman coding is also vulnerable to side channel leakage through the compression ratio. Instead of using an attacker controlled seed file, an attacker guess was co-located with the source data. Figure 3 shows the compression ratio with increasing numbers of characters of the secret co-located with the source data versus the compression ratio with increasing numbers of random characters co-located with the data. With more characters of the secret co-located with the source data, the compression ratio remains about constant. On the other hand, the compression ratio increases when random characters are co-located with the source data. Figure 3 shows a significant difference between the two data sets in the average compression ratio over many iterations. However, the ranges of values are quite large, as the two box plots overlap quite a bit. Therefore, we see that Huffman coding may also be vulnerable to side channel leakage through the compression ratio. However, it is required that the attacker's guess has many characters of the secret co-located with the source data, so it is not exactly clear how such an attack would be feasible.

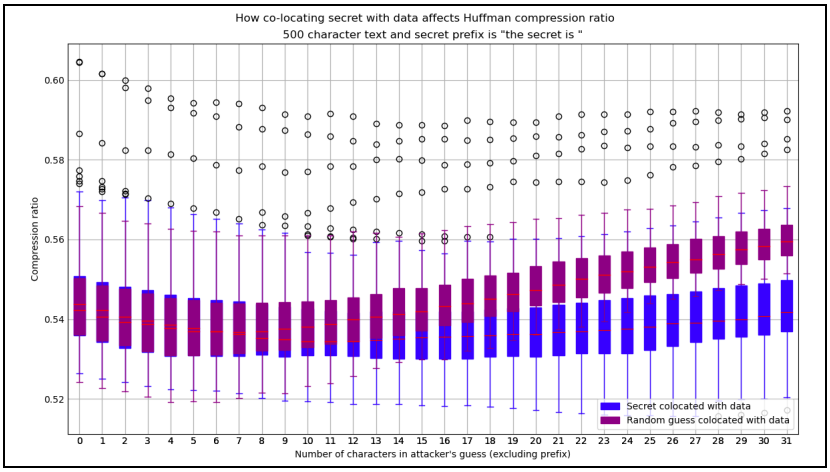

Figure 3: Huffman compression ratio with attacker guess colocated with victim data

Lastly, returning to LZ77, I tried to guess a 16 character secret key based on the compression ratio of 500 character text. I found out of 100 trials, I was able to guess the secret 83% of the time. Out of the 17% that was incorrect, for a small percentage the guess still matched over 50% of the secret. This data is shown Figure 4. Comparing this attack strategy to brute force, an attacker would have to make a linear number of guesses using this strategy versus an exponential number of guesses using brute force, with respect to the length of the secret.

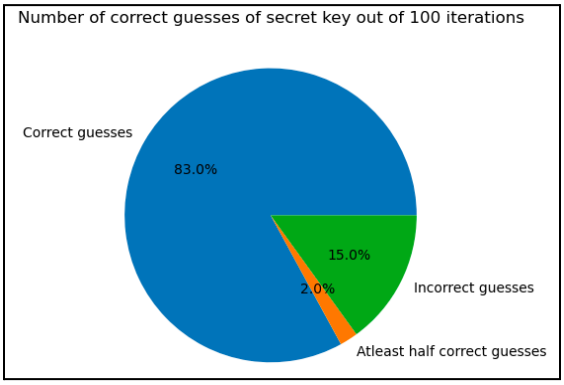

Figure 4: Attempts at guessing the 16 character secret in 500 character text based on LZ77 compression ratio

Discussion

Based on the results, it seems overwhelmingly possible that compression algorithms leak information through the compression ratio. If an attacker is able to see the file size and affect the compression algorithm with some guess of the secret (either through a seed file or another means), it is likely that they could learn something about the secret. Looking at longer versus shorter texts, a longer text still reduces in size with a better guess of the secret. However the leakage per character depends on the precision with which an attacker is able to view the file size. For example, if an attacker is only able to measure the file size in bytes or even kilobytes, this would make the side channel much less useful for an attacker, since they would not be able to observe the incremental changes with better guesses.

Second, in all of these experiments, I assume a fairly strong attacker who can both measure the output file size and affect the compression algorithm in some way. Many side channel attacks similarly require attacker interference in the algorithm, but others do not. Therefore, I am not sure if it would be possible (or computationally feasible) to learn something about secret data through the compression ratio if an attacker was unable to affect the compression algorithm. Second, I assume in all of my experiments

that an attacker is able to repeat experiments. Depending on the target of the attack, this is often a reasonable assumption. However, there are certain contexts where keys are only single-use, such that repeating the attack is ineffective at gaining more information about the secret data. Lastly, I was able to reason about how an attacker-known prefix would help an attacker. With the prefix, it seems very clear that the attacker can guess characters one at a time with pretty high accuracy, as I showed in my guessing strategy. However, without the prefix, it took several correct characters to observe a change in compression ratio. While having an attacker-known prefix may seem unrealistic, it is possible that the attacker knows something about the format of a dataset, even if the content is unknown. For example, an attacker may know the key in a JSON file for which they are trying to learn something about the value.

Lastly, I looked at two different compression algorithms, Huffman and LZ77, in order to compare how effective a compression ratio side channel attack would be against the two algorithms. As expected, the attack was very effective against LZ77, since the algorithm depends on matching sequences of symbols in the source data. Therefore if an attacker is able to guess the secret, a better match in a seed file would cause a lower compression ratio since there would be fewer literals to encode. However, the attack would be most effective on a greedy matching algorithm in LZ77 particularly with a secret prefix, because the seed would likely contain the best match. If the matching algorithm is lazy however, it is possible that the compression ratio does not change even when the seed file contains more characters of the secret.

Huffman coding was not as conclusive as LZ77. By co-locating a guess with the source text, we did see overall trends that if the guess was correct, the compression ratio was lower than if the guess was incorrect. However, it required significantly more characters in the guess to observe changes in the compression ratio, making the attack more computationally intensive. Additionally, Huffman showed a much wider range of compression ratio values, such that it would be hard to deterministically reason about the secret based on the compression ratio.

Conclusion

I investigated how side channels can arise from data compression both through experimentation and review of the literature. While I only considered compression ratio as the side channel in my experiments, with more time and resources it would be interesting to consider other side channels as well, such as timing or memory access pattern. Additionally, throughout my experiments, I assume a fairly strong attacker model: the attacker could either affect the seed of the LZ77 encoder or co-locate data with victim data for Huffman coding. In future work, it would be interesting to evaluate leakage under a weaker attack model.

As data usage globally is increasing exponentially, there is no doubt that compression will play an important role in accommodating this growth. Compression is already used throughout the hardware-software stack and will continue to be applied in new applications. However, applications of compression in hardware, particularly when it is unobservable to software, is uniquely dangerous as users cannot disable compression for secure execution. It has already shown that this software transparent compression exists in GPUs. As compression becomes more pervasive in cloud storage, on and off chip memory, and embedded in hardware for image and video processing, it is possible that many more devices are vulnerable to such attacks. Therefore, it is critical to consider the security requirements of systems such that sensitive data is not leaked unknowingly through compression.

References

[1] J. Kelsey, “Compression and information leakage of plaintext,” in FSE, 2002.

[2] J. Rizzo and T. Duong, “The CRIME attack,” Presented at Ekoparty 2012, Sep. 2012, slides online: https://docs.google.com/presentation /d/11eBmGiHbYcHR9gL5nDyZChu-lCa2GizeuOfaLU2HOU/edit.

[3] Gluck, N. Harris, and A. Prado, “BREACH: Reviving the CRIME attack,” Presented at Black Hat USA 2013, Aug. 2013, whitepaper online: https://breachattack.com/resources/BREACH%20-%20SSL,%20gone%20in%2030%20seconds.pdf.

[4] Mathy Vanhoef and Tom Van Goethem. Heist: Http encrypted information can be stolen through tcp-windows. In Black Hat US Briefings, Location: Las Vegas, USA, 2016.

[5] M. Schwarzl, P. Borrello, G. Saileshwar, H. Muller, M. Schwarz, and D. Gruss, “Practical timing side channel attacks on memory compression,” in S&P, 2023.

[6] Y. Wang, R. Paccagnella, Z. Gang, W. Vasquez, D. Kohlbrenner, H. Shacham, and C. Fletcher, “GPU.zip: On the Side Channel Implications of Hardware-Based Graphical Data Compression” in S&P, 2024.

[7] M. Hogan, Y. Michalevsky, and S. Eskandarian, “Dbreach: Stealing from databases using compression side channels,” in S&P, 2023.

[8] P.-A. Tsai, A. Sanchez, C. W. Fletcher, and D. Sanchez, “Safecracker: Leaking secrets through compressed caches,” in ASPLOS, 2020.

Appendix A - Literature Review from Milestone Report

There is a significant amount of research discussing side channels that arise from data compression. Kelsey had one of the first works which broadly analyzes the security implications of compression algorithms [2]. The paper discusses how lossless compression algorithms create side-channel security vulnerabilities, the practicality of such side-channel attacks, and how and when compression is often used on secret data before encryption.

Subsequently, two compression-based attacks, CRIME [3] and BREACH [4], were introduced in 2012 and 2013. Both of these two attacks relied on LZ77 and Huffman compression in order to steal private user information over the web. CRIME uses a side channel generated from the compression ratio of compressed TLS requests to recover the headers from HTTP requests. Since headers often contain cookies, an attacker would be able to steal a user's login. While this attack was quickly mitigated by disabling TLS compression, another attack, BREACH, uses the compression ratio of HTTP responses, which are commonly compressed using gzip. For a web application to be vulnerable to BREACH, its requests must be served with HTTP-level compressed responses and reflect both user-input and a secret in HTTP response bodies. User input is key, as it allows for an attacker to guess one character at a time and learn the secret by looking at the compression ratio of the response. Another attack, HEIST [5], builds on CRIME and BREACH to show that the length of HTTP responses can be leaked without an attacker having access to the victim's network.

While CRIME, BREACH, HEIST, and all other related attacks focus solely on the information leaked due to changes in compression ratio and look at HTTP traffic in particular, there are more recent works which look at compression side channels in other applications. For example Schwarzl et al. exploit timing side channels in compression algorithms [6]. Using decompression time, they show information about the source data, such as entropy and position of compressible strings, are leaked. Further this work looks into broader applications beyond HTTP traffic, including memory-compression, databases, file systems, and others.

GPU.zip [7] is another recent compression-based side channel attack, which is based on software transparent compression in both integrated and discrete GPUs. This means that there are compression algorithms running on the GPU which are not software visible, and therefore cannot be disabled by the programmer. The attack shows that the GPU compression schemes induce data-dependent DRAM traffic and cache footprints, which leaks information about the source data. In their case, an attacker is able to infer the values of individual pixels of a webpage that it does not have access to programmatically. Another recent and novel compression-based attack is Dbreach [8], which can extract information about plaintext in a compressed database with limited access to the compressed text. They introduce techniques to reduce noise and heuristics regarding how to generate a good guess of the secret, which, again, must be co-located with the source data.

Lastly, as more hardware optimizations are proposed in order to improve performance, there is a need to consider these optimizations' security implications before they are implemented in hardware. One example of this is SafeCracker [9], which evaluates the security vulnerabilities of compressed caches. While compressed caches are not yet deployed in hardware, more processors have recently included compression at some level of the memory hierarchy. This paper investigates the security properties of compressed caches, and in particular, how they reveal information of program data. In particular they propose an attack where there is attacker-controlled data co-located with secret data in a compressible cache, and using the compression ratio (they call it "compressibility"), they are able to learn about the content of the secret data.

Many of these attacks look similar to the original compression side channel attacks proposed over a decade ago: some attacker controlled data is compressed with victim data, and subsequently the attacker is able to learn about secret data by looking at the compression ratio. However, as compression continues to be implemented more widely in hardware systems, it is imperative to consider the security implications that come with novel applications of data compression in hardware optimizations.

There has also been some interesting work looking at mitigating compression side channel attacks. Two obvious mitigations include disabling compression entirely or isolating secret data and user controlled data to be compressed independently. Other proposed schemes involve simultaneous encryption and compression, including SecOComp's [cite] chaos-based encryption and compression scheme. However, it is unclear how efficient these schemes are in practice. Overall, the mitigations I found do not use novel and/or interesting compression algorithms in order to prevent compression-based side channel leakage, and therefore I will not focus on them for this project.

Appendix B - Additional results

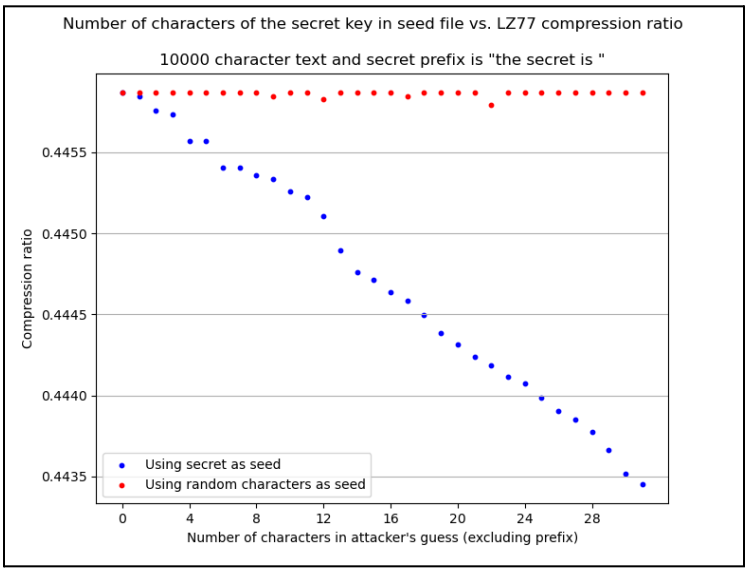

Figure 5: LZ77 compression ratio of varying seed files with 10,000 character source data