A Comparative Security Analysis of Compressed Caches

Matthew Musselman

Abstract—Compressed last-level caches have been proven to be vulnerable in the Safecracker paper, which demonstrated how a VSC compressed cache using the BDI algorithm can be manipulated into leaking secret data using only cache timing attacks and a heap spray inside the victim's process. This paper shows that this type of attack extends to other types of compressed caches (namely, the YACC architecture using the C-PACK algorithm) and analyzes how a cache's choice of architecture and algorithm influences its vulnerability to data leaks.

1 Introduction

Hardware caches are a vital part of any computer's architecture, improving the system's performance by eliminating reads to the main memory; as a result, there has been plenty of research on improving caches, targeting either the hit rate (chance to hold an entry) or hit time (speed at which an entry's data can be returned). One interesting avenue of research is cache compression, where a cache's data is compressed to reduce space; while this does slow down the hit time, it also allows the cache to hold more entries without changing the size of its data store, improving the hit rate significantly [3]. Many compressed cache designs exist: most are simply the combination of a compressed cache architecture (specifying how compressed lines are packed into the data store) and a compression algorithm (used to compress lines efficiently yet quickly).

Unfortunately, compressed caches have a unique security vulnerability, which was originally identified in the Safecracker paper [8]: if a cache's design allows an attacker process to compute the compressed size of another process's cache line, then it is possible to make inferences about what is contained in the line. Furthermore, if the attacker has a way to manipulate some the data inside the cache line, it may be possible to exactly deduce the remaining bits in the line, allowing secrets to indirectly read. Safecracker showed that this type of attack was possible on a Variable-Segment Cache (VSC) architecture [2] using the Base-Delta-Immediate compression algorithm [6], both common compressed cache components; additionally, the authors hypothesized that several other architectures and algorithms would be vulnerable in a similar way.

To test Safecracker's hypothesis, this paper replicates the Safecracker attack on an entirely different type of compressed cache: specifically, the Yet Another Compressed Cache (YACC) architecture [7] with the C-PACK compression algorithm [4]. Following that is an analysis of the various factors that make Safecracker's attack possible, comparing various architectures and algorithms to see which are the most and least exploitable. This paper then concludes by analyzing several design choices that prevent or reduce the effectiveness of compressed cache attacks.

2 Background

2.1 Safecracker's Attack on VSC+BDI

Safecracker's proof-of-concept attack targets the Variable-Segment Cache (VSC) [2], which is a simple architecture that uses tag over-provisioning and data compression in order to pack more lines into a cache set than what an uncompressed cache could normally handle. VSC breaks the data section of a cache's set into many small chunks (usually 8 bytes each) and allows entries in the tag section to point to a contiguous segment of several chunks. By using a compression algorithm to fit uncompressed lines into smaller segments, VSC can potentially fit more lines into a set than what an uncompressed cache can normally accommodate. (See Fig. 1 for an example of how VSC is structured.)

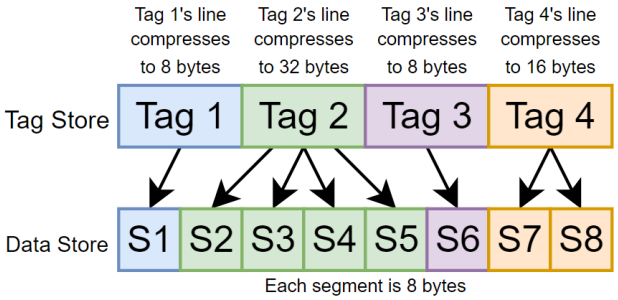

Fig. 1. Example of how VSC maps tags to segments of varying lengths.

The Base-Delta-Immediate (BDI) algorithm [6] is what is actually used to compress the data; it targets cache lines where each byte/short/word/etc can be represented as a base plus a small offset, with the offset requiring fewer bytes than the base. Cache lines that exhibit this quality can be compressed to fit into very few 8-byte VSC chunks.

What Safecracker found is that if a small secret (say, 4 bytes) is located at the end of a cache line in a VSC+BDI cache, then it is possible to replace the remaining bits with a pattern that causes the compression to fall below a certain threshold only if the attacker's secret matches a certain value [8]. For example, if the last 4 bytes of a cache line were 0x12345678, then filling the remaining 60 bytes with a repeated pattern of 0x39823201 would leave the line uncompressed, but repeating 0x12340000 would trigger BDI's "4-byte base, 2-byte offset" compression mode [6], signalling that the secret is close to 0x12340000. Safecracker showed that by using a fairly efficient search algorithm, it is possible to brute-force the secret by replacing the leading bytes to a secret in a strategic way and only observing how much the line compresses. (See Fig. 2 for an example of this attack.)

However, this attack on BDI cannot be used to discover a secret unless the compressibility of a cache line is observ-

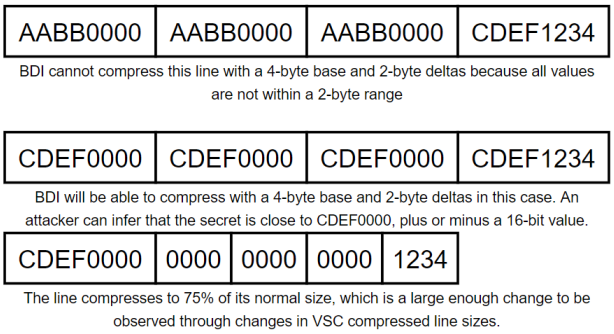

Fig. 2. Example of how BDI compresses data and leaks information about the secret through its compressed size

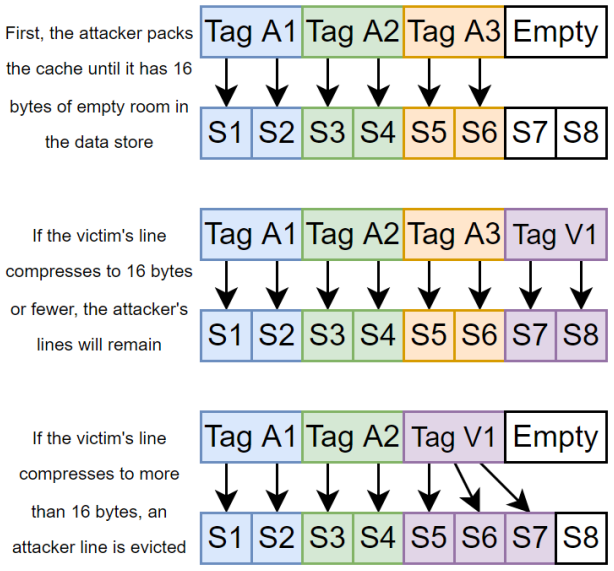

Fig. 3. Example of how Safecracker discovers the line size of a compressed victim line in VSC

able to an attacker. Safecracker showed that with the VSC architecture, it is possible to determine the size of a victim process's cache line using the "Pack+Probe" technique [8]. First, a memory bug such as a heap spray or a buffer overflow is exploited to all the attacker to write a string of bytes that triggers a conditional compression based on the secret data. Next, the attacker uses its own cache lines to fill up the entire cache set's data store, apart from N bytes. The attacker then makes the victim access its secret line; if the secret line compresses to more than N bytes, then one of the attacker's lines will be evicted. The attacker can then use a timing attack to check all its lines to see if any were evicted; if none were, then the attacker knows the victim's secret is no more than N bytes long when compressed. By carefully packing the cache set, it is possible to measure the compressibility of the victim's secret line, allowing the attack on BDI to proceed. (See Fig. 3 for an example of this attack.)

Safecracker showed that this attack on VSC+BDI was possible through a proof-of-concept simulation. Additionally, Safecracker states that its attacks can be generalized to other cache architectures and algorithms. To test this statement, this paper implements a similar attack on the YACC cache architecture and the C-PACK compression algorithm, both of which will be discussed shortly.

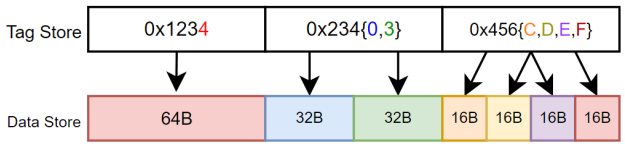

Fig. 4. Example of how YACC allows tags from the same superblock to co-locate in the same data line

2.2 YACC

Yet Another Compressed Cache (YACC) [7] is a compressed cache architecture that is much different from VSC; unlike VSC, YACC does not allow tags to map to arbitrary parts of a data store and instead adheres to a traditional tag-to-data mapping scheme (i.e., one tag maps to one 64-byte data block). YACC incorporates compression by allowing data from multiple lines in the same superblock (usually a 4-line or 256-byte chunk of the address space) to co-locate in the same 64-byte data line. Each valid data line in YACC can be in one of three states:

- Holding a single uncompressed line
- Holding two lines from the same superblock, each compressed to 32B or less
- Holding 3-4 lines from the same superblock, each compressed to 16B or less

YACC's design is much simpler to implement in hardware than VSC, and its superblock-based compression architecture is suited particularly well for applications with long arrays of compressible data. (See Fig. 4 for an example of how YACC is structured.)

2.3 C-PACK

C-PACK [4] is a compression algorithm that breaks up a 64-byte cache line into 16 32-bit words and replaces each word with a small prefix (either 2 or 4 bits long) followed by 0-32 bits of data needed to decompress the word. Each 32-bit word is compressed according to the following rules:

- All-zero words are compressed to 2 bits
- Words that are zero-extended bytes are compressed to 12 bits
- Words that are identical to a previous word in the line are compressed to 6 bits
- Words that match a previous word apart from the last byte or short are compressed to 16 or 24 bits
- All other words are left uncompressed, but with a 2-bit prefix marking them as such (totalling 34 bits)

Like BDI, C-PACK is designed to handle data that contains lots of repetition, apart from certain bytes at the end of words.

3 Attacking YACC

Compared to VSC, the YACC architecture is much better at hiding the sizes of its compressed lines from attackers. An attacker cannot use the compressibility of its own lines to directly measure the size of a victim's lines like in Safecracker's Pack+Probe technique [8], since YACC will not pack together lines belonging to different superblocks [7]. The only way for an attacker to get the timing information needed to perform Pack+Probe would be through the victim's interfaces that read or write to memory; even so, this is likely too noisy to be used. YACC also has the security benefit of only compressing to 16, 32, or 64 bytes, while VSC can do any multiple of 8 up to 64.

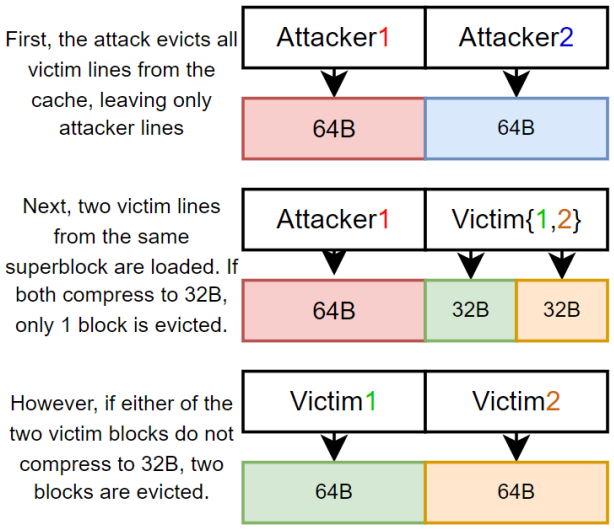

Fig. 5. Example of how compressed size information can be leaked through YACC

Despite these benefits, there is still a way to defeat YACC; however, it requires control over not just the free space in the victim's cache line containing the secret, but over the entire superblock. By controlling the entire superblock, it is possible to guarantee that the three superblocks without the victim's secret will compress as small as possible, meaning that the secret line can compress into the same data line as one of them–but only if the victim itself is 32B or smaller. This means that if the attacker can evict the entire set with its own lines and make the victim program reload the secret line and one of the other compressible lines in its superblock, then one of two scenarios will happen:

- If the secret line compresses to 32B or fewer, then reloading both the lines will only take up one line in the data store, evicting one of the attacker's lines.
- If the secret line compresses to 33B or more, then reloading both the lines will take up two lines in the data store, evicting two of the attacker's lines.

The attacker can then easily check which scenario happened by reloading all of its blocks and using the access times to count the number of misses. Therefore, if an attacker has access to an entire superblock of data apart from the victim's secret at the end of the last line (due to a heap spray, buffer overflow, or similar bug), then they can gather some information about the secret line's compressibility and gradually infer the victim's secret by exploiting the compression algorithm. (See Fig. 5 for an example of this attack.)

The last step of this attack can be sped up if the attacker knows the cache's replacement algorithm as well. For example, if LRU is used, then the attacker only needs to check the second of the attacker's cache lines that were loaded in before the victim's lines were reloaded, since that will be the second-least-recently-used line (the least recently used line will always be evicted). This strategy is used in the later experimental simulation to reduce the number of necessary cache reads.

4 Attacking C-PACK

Like BDI, C-PACK can be exploited to gradually gain information about a victim's secret by priming the non-secret bytes of a victim's cache line in a specific way. By assuming that the secret is 4 bytes, the cache line is 64 bytes, and the aforementioned YACC attack is available, the following algorithm can be used to exactly find the secret:

1) First, exploit the C-PACK rule that compresses a word if it matches a previous word in the line, apart from the last two bytes of the word. After placing several 4-byte "attack words" at the start of the 60-byte attack payload that have variations of the first two bytes, the secret word will compress from 34 bits to 24 bits only if its first 2 bytes match one of the variations. By carefully crafting the 60-byte payload so that the entire line compresses to 32 bytes only if the secret word compresses, it is possible to check whether the secret starts with one of the provided 2-byte variations by using the YACC attack to leak the size information.
2) After brute-forcing different groups of leading 16-byte shorts, the search algorithm will eventually find a small group of shorts that make the secret compress. The algorithm can then go through each short one-by-one and write a custom attack payload for each short, testing each until the exact short that causes the secret to compress is found. This short provides the first 2 bytes of the secret.
3) Next, repeat the same process, but instead using the C-PACK rule where a word is compressed to 16 bits if the word appears earlier in the line (apart from the last byte). All of the "attack words" in the 60-byte buffer will begin with the secret's leading two bytes found earlier, and will instead vary the second-to-last byte between words. A similar two-phase step will occur, finding a small list of potential bytes and then the exact byte that matches the secret's second-to-last byte.
4) Finally, repeat the same process with the rule where a word is compressed to 6 bits if it perfectly matches a previous word. Now, all the attack words in the payload will start with the 3 found bytes, varying only the last byte. The same two-phase search process occurs again until the byte is found. At this point, the secret is revealed.

This process is fairly quick; the slowest part is brute-forcing the leading 16-bit short, since only 6 attack words can be used at a time due to the compression needing to reach 32B, and also due to there being 2^{16} possible shorts. Brute-forcing the other bytes is much quicker, since there are only 256 possibilities for each, and more attack words are able to fit in a single payload thanks to C-PACK's compression rules. (See Fig. 6 for an example of this attack.)

For 8-byte secrets and larger, this algorithm can be used to find all of the individual words using the same algorithm, although the payloads may not be able to fit as many attack words. However, due to how C-PACK individually compresses words, it is impossible to determine the order of the secret's words using this algorithm alone. This may not be an issue if the secret has some known qualities (for example, if the words form a private key, then they only need to be shuffled until it matches up with the known public key). Also, if a buffer overflow attack is available, it may be possible to wipe some of the secret's words from the line; then, the order can be deduced by figuring out which word was wiped using the algorithm.

5 Proof-of-Concept YACC+C-PACK Attack

To prove that the previously mentioned attacks on the YACC architecture and C-PACK algorithm are viable, a simulation

0x43210000	0x43220000	0x43230000	0x43240000
0x43250000	0x43260000	0x000000FF	0x00000000
0x00000000	0x00000000	0x00000000	0x00000000
0x00000000	0x00000000	0x00000000	0x43251234

When C-PACK compresses this 64-byte string, each "attack word" (blue) compresses to 34 bits each and all the filler words (green) compress to 28 bits in total. Because the upper two bytes of the secret word (orange) match the upper two bytes of 0x43250000 (one of the earlier attack words), it will compress to 24 bits, making the entire line fit in 256 bits (exactly 32 bytes). However, if the upper two bits were different, the last word would compress to 34 bits, making the entire line no longer fit into 32 bytes.

Fig. 6. Example of an attack string that can be used to leak the first two byte of a secret (step 1)

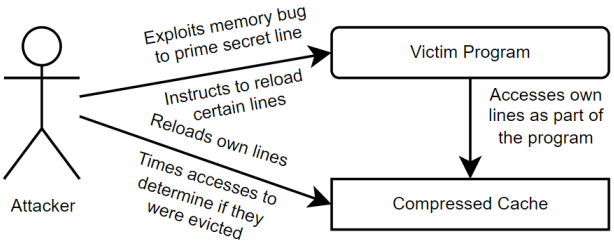

Fig. 7. Diagram of the threat model used by the proof-of-concept demonstrations in Safecracker and this paper

with the following configuration was constructed:

- A single set of a YACC+C-PACK cache with 8 64-byte lines is simulated using a data structure. This simulated cache allows for reading and writing to an underlying main memory structure. On reads, the cache returns whether or not the read was a hit (along with the data).
- The victim program is simulated as a data structure that keeps track of a 256-byte buffer in the simulated memory hierarchy; the buffer is aligned to a YACC superblock. At the end of this buffer is a secret, which is a sequence of 4 or 8 random bytes. (To avoid handling edge cases in the attacks, the secrets are forced to use unique, non-zero bytes.) The victim provides methods for reading and writing to anywhere in the buffer, except with the bytes containing the secret.
- The attacker is simulated as a function that interacts with the cache and victim structures. It may read and write bytes to the non-secret parts of the victim's buffer, and it also may access its own set of cache lines from the cache and gather timing data from the reads. It may not access the victim's lines directly, however.

See Fig. 7 for a diagram showing how these components interact.

With this setup, the attacker was always able to guess the 4-byte secret on the first try; for the 8-byte key, the attacker always found it within two tries (due to the inability to tell the words' order). On average, about 5450 iterations of priming the secret line and leaking the compressed size through YACC were needed to crack each 4-byte secret, and 13200 for each 8-byte secret. The full results for 10000 simulations of each secret length are shown in Table 1.

The code to run the attack can be found at: https://github.com/matthewm101/yacc_cpack_cache_attack.

Average Stat over 10000 Iterations	4 Bytes	8 Bytes
Guesses Needed	1	1.4978
Victim Buffer Bytes Modified	33588.6662	67516.6206
Victim Lines Reloaded	10904.0348	26391.4028
Attacker Lines Reloaded	49068.1566	118761.3126
Attack Iterations Required	5452.0174	13195.7014

TABLE 1
YACC+C-PACK Attack Simulation Results

6 Discussion

6.1 What Makes a Cache Architecture Exploitable?

Safecracker's authors already showed how VSC reveals the compressibility of its cache lines: because each line claims a variable amount of cache segments proportional to its compressed size, and because those segments are part of the same array of segments shared by all entries in a set, it is possible to measure line compressibility using the eviction-based Pack+Probe technique [8]. The attack on YACC described earlier is fairly different, but there is one important similarity: to detect whether or not YACC compresses a line, cross-process line evictions are utilized. The VSC and YACC attacks are primarily possible because both architectures allow one process to have its lines evicted by another process in a way dependent on the compressibility of each process's lines.

One possible hardware modification that the Safecracker authors suggested for VSC is cache partitioning: the act of separating the tag and data stores into partitions, where each partition is assigned to a process and no process can interact with another process's partitions [8]. This prevents cross-process evictions at a performance cost. The same modification can also be applied to YACC, making the attack described earlier impossible (since the attacker can no longer use the evictions of its lines to measure the victim's compressibility).

Even with cache partitioning preventing cross-process evictions, there is nothing preventing a process from observing its own evictions, which could become an issue if an attacker has both influence over enough lines in the victim's address space to evict an entire set partition, and also a way to time its interactions with the victim to determine cache hits or misses. Luckily, performing a timing attack through the victim's interface is much more challenging than getting timing info from the cache directly, and it is easy to fix a timing leak by adding random jitters or making all requests take the same amount of time.

6.2 What Makes a Program Exploitable?

Safecracker [8] and this paper both assume that the victim's program experiences some sort of memory bug, either a heap spray or a buffer overflow. Both also assume that the victim's secret is located all by itself in a cache line that is either mostly unallocated (in the case of a heap spray bug) or above an unbounded buffer (in the case of a buffer overflow bug). Removing either of these assumptions effectively defeats the attacks, since the ability to manipulate data next to the victim's secret is key to discovering details about it through the compression algorithm and architecture attacks.

In the case where a heap spray can enable the bug, an easy software mitigation is to simply fill all cache lines with secrets with uncompressible read-only bytes that cannot be allocated. If the free space next to the secret is absolutely necessary, YACC even offers a more subtle way to prevent compression: fill all the blocks in the same superblock with

uncompressible bytes, so that the line with the free space and secret cannot be compressed with anything else.

Buffer overflows are much more dangerous bugs, and there are plenty of pre-existing mitigations and techniques that either prevent them from happening or make them detectable.

6.3 What Makes a Compression Algorithm Exploitable?

If the compressed cache architecture allows a secret line's compressed size to be leaked, and the victim's program has a memory bug that lets data be written next to a victim's secret, then the last line of defense preventing the victim's data from leaking is the compression algorithm. As shown in Safecracker [8] and this paper, both BDI [6] and C-PACK [4] are particularly exploitable. While these algorithms have different approaches to compression, they share one key similarity: the compressibility of bytes later in the line can be controlled by bytes appearing earlier in the line. BDI allows certain levels of compression only if the last word in a line is within a small difference of the other words, and C-PACK will compress a word smaller if it can reference a similar word earlier in the line. This shared quality is what allows attackers to leak the secret in its entirety by trying different bytes until a compression occurs.

There are several other compression algorithms that exhibit this quality. One example is FPC-D [1], an algorithm that functions nearly identically to C-PACK; the main difference is that FPC-D only looks for repeated words that appear one or two words before a word being compressed, while C-PACK can look anywhere in the cache. This feature does slow down the attack used by C-PACK, but FPC-D's design still allows the entire 4 or 8-byte secret to be leaked.

Another interesting algorithm is Bit Plane Compression (BPC) [5], which arranges the bits of a 128-byte cache line into a 32-by-32-bit square and performs several operations that tend to result in an easily compressible line with lots of zeros. This algorithm heavily obfuscates the relationship between the input and the size of the compressed output, making it difficult to attack. However, it nonetheless exhibits the property where bits earlier in the line can influence the compressibility of bits later in the line, making it theoretically possible to crack secrets from leaked compression sizes.

Finally, it is worth discussing Frequent Pattern Compression (FPC), which is an algorithm that compresses each 4-byte word completely independently. FPC does not use a dictionary to compress repeated words or compute deltas between consecutive words like the previous algorithms; it simply looks at each 4-byte word in isolation and compresses it using a small set of patterns (most of which target small sign-extended numbers). This type of algorithm cannot be manipulated into leaking information about the secret via the line's compressed size, since it compresses the secret in the same way every single time. This type of algorithm is the most secure for a compressed cache to use; unfortunately, algorithms that compress words independently aren't the most effective, and their use will likely harm the performance of a compressed cache.

7 Conclusion

The susceptibility of YACC [7] and C-PACK [4] to the same type of attack as the one used in Safecracker [8] proves the Safecracker authors' claim that many different compressed cache architectures and algorithms have the potential to leak secret data through side channels. Luckily, there are plenty of alternative designs and algorithms that patch many of the issues faced by the most vulnerable caches, but these patches also tend to come with a performance cost. Further research is needed to determine which compressed cache architectures and algorithms still perform the best, even when modified to be Safecracker-proof.

References

[1] A. R. Alameldeen and R. Agarwal, "Opportunistic compression for direct-mapped dram caches," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 129–136. [Online]. Available: https://doi.org/10.1145/3240302.3240429

[2] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.* IEEE, 2004, pp. 212–223.

[3] D. R. Carvalho and A. Seznec, "Understanding cache compression," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 3, pp. 1–27, 2021.

[4] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas, "C-pack: A high-performance microprocessor cache compression algorithm," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 8, pp. 1196–1208, 2010.

[5] J. Kim, M. Sullivan, E. Choukse, and M. Erez, "Bit-plane compression: Transforming data for better compression in many-core architectures," *SIGARCH Comput. Archit. News*, vol. 44, no. 3, p. 329–340, Jun. 2016. [Online]. Available: https://doi.org/10.1145/3007787.3001172

[6] G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *2012 21st international conference on parallel architectures and compilation techniques (PACT)*. IEEE, 2012, pp. 377–388.

[7] S. Sardashti, A. Seznec, and D. A. Wood, "Yet another compressed cache: A low-cost yet effective compressed cache," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 3, Sep. 2016. [Online]. Available: https://doi.org/10.1145/2976740

[8] P.-A. Tsai, A. Sanchez, C. W. Fletcher, and D. Sanchez, "Safecracker: Leaking secrets through compressed caches," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1125–1140.