Compression: Impossible Securing Hardware-Level Memory Compression

Chenfei Liu*
ETH Zurich
Zurich, Switzerland
liuche@student.ethz.ch

Ali Hajiabadi*†
ETH Zurich
Zurich, Switzerland
ahajiabadi@ethz.ch

Kaveh Razavi
ETH Zurich
Zurich, Switzerland
kaveh@ethz.ch

Abstract

Transparent hardware-level memory compression has become increasingly attractive for commercial deployment, particularly in hyperscalers and cloud infrastructures. While it is well known that compression can introduce data-dependent leaks, as the compressibility of data inherently depends on its content, a common intuition is that such leaks can be mitigated using constant-time compression algorithms. Unfortunately, as we show in this paper, despite constant-time compression algorithms, all existing designs suffer from information leakage vulnerabilities due to inherently-insecure architectural design decisions. We develop several novel leakage primitives that exploit the fundamental design aspects shared across existing proposals. Relying on these leakage primitives, we show secret-dependent behaviors in widely-used cryptographic libraries and construct an end-to-end attack that leaks passwords from a popular web server. To secure memory compression against these serious vulnerabilities, we introduce a new *compression non-interference* property and prove that a compression system that adheres to this property does not introduce any leakage primitives. We then demonstrate that practical and secure memory compression is achievable by designing a new secure memory compression system called ZipLock. Our evaluation shows that compared to the insecure state-of-the-art baseline, ZipLock incurs only 3.5% performance overhead while providing a similar compression rate of 1.65×.

CCS Concepts: • **Security and privacy → Hardware attacks and countermeasures**; *Formal methods and theory of security*.

Keywords: Hardware-level memory compression, Software-transparent compression, compression non-interference

*Equal contribution joint first authors.
†Corresponding author.



This work is licensed under a Creative Commons Attribution 4.0 International License.

ASPLOS '27, Heraklion, Greece
© 2027 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3012-2/2027/04
https://doi.org/10.1145/3845814.3850069

ACM Reference Format:
Chenfei Liu, Ali Hajiabadi, and Kaveh Razavi. 2027. *Compression: Impossible* Securing Hardware-Level Memory Compression. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '27), April 11–15, 2027, Heraklion, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3845814.3850069

1 Introduction

Hardware-level memory compression is a promising technique to improve memory capacity and bandwidth utilization without requiring software changes [9, 27, 28, 36]. However, we find that recent proposals for memory compression rely on fundamentally insecure design choices as we demonstrate with a number of new leakage primitives that compromise common constant-time cryptographic libraries and web servers. To ensure that the introduction of hardware-level memory compression into cloud systems does not compromise security, we introduce the *compression non-interference* property that stops leakage through memory compression, followed by a new design, called ZipLock, that adheres to this property and achieves performance and compression ratio that are comparable to an insecure state-of-the-art baseline.

Memory compression. Hardware-level memory compression has long been studied in systems research to increase the effective capacity or bandwidth by packing data more densely [9, 11, 18, 19, 26–28, 30, 32, 34, 36, 37]. Early proposals required both hardware and OS support, but recent designs are software-transparent, making them practical for real-world deployment in cloud infrastructures and hyperscalers with an ever-increasing need for more memory capacity and bandwidth. However, compression can potentially introduce *data-dependent* timing side channels, which, similar to other data-dependent optimizations like DMP prefetchers [7] can break constant-time guarantees in cryptographic code. While a natural first solution is deploying constant-time (de)compression algorithms, a deeper question is *whether existing memory compression systems leak information through other fundamental design choices.*

System-level leakage primitives. Beyond the choice of compression algorithm, all memory compression systems must address several key design decisions: (1) the compression granularity, (2) the management of compressed pages, including their packing and memory allocation, and (3) the

handling of additional address translations, particularly for variable-sized compressed pages. We introduce three leakage primitives that exploit the key design choices: (1) the *Read Amplification* primitive leverages misaligned, variable-sized compressed data blocks (either line-level or page-level granularity depending on the design) that cause additional data-dependent memory accesses. (2) The *Write Expansion* primitive exploits how updates to compressed data blocks may increase their size, causing page overflows and triggering costly data-dependent relocation events. (3) The *Silent Compressed Load* primitive targets bandwidth-optimized compression systems that pack multiple cache lines into a single memory access, where the number of cache lines is data dependent. Our evaluation shows that all these leakage primitives introduce secret-dependent leaks in real-world cryptographic programs that are otherwise secure under standard constant-time guarantees. Additionally, we demonstrate practical exploitation of such leaks by disclosing arbitrary-length passwords from a widely used web server. These results prompt a critical question: *is it possible to design secure memory compression systems that are resilient against these leakage primitives?*

Secure design principles. We categorize our leakage primitives into two general classes: (1) instantaneous leaks, which stem from a single memory request and are self-contained, and (2) cascading leaks, which emerge from sequences of requests where earlier accesses influence subsequent behavior. We show that these two classes of leaks violate two key principles: (1) *uniform access*, which requires uniform behavior per memory request to prevent data-dependent leaks from individual accesses, and (2) *stateless footprint*, which requires absence of history-based and stateful operations; so a secure design should either eliminate the possibility of such events or trigger them in a stateless, data-independent fashion. We formally prove that a memory compression system that enforces these two design principles achieves a strong *compression non-interference* property; this security property ensures that memory compression introduces no additional data-dependent leaks beyond architectural execution, thereby preserving all architecture-level guarantees such as cross-privilege/VM isolation and constant-time cryptography. While these principles ensure strong security guarantees, applying them naively to existing memory compression designs may incur substantial overheads, or worse, void the benefits of memory compression entirely.

ZipLock. To show-case the possibility of secure memory compression, we present the design of ZipLock, which relies on secure mechanisms guided by our design principles. ZipLock builds on prior insights that coarse-grained page-level compression and keeping frequently accessed pages uncompressed improve compression and translation reach [27, 28], which we apply in a security-first approach. In particular, ZipLock enforces uniform page migrations between compressed and uncompressed regions and strictly separates these regions in a static manner to eliminate subtle side channels that measure the size of these regions to reveal page compressibility. Our evaluation shows that ZipLock achieves a compression ratio of 1.65× with a 1.37× slowdown compared to the uncompressed baseline, and only 3.5% overhead compared to the insecure state-of-the-art design at the same compression ratio [28]. We envision ZipLock as a secure baseline for future research on efficient and secure memory compression systems.

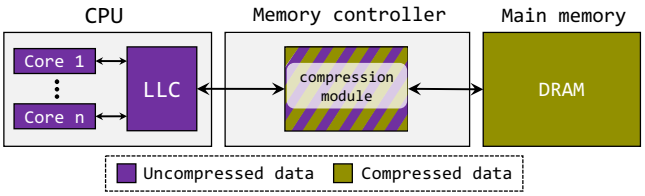


Figure 1. High-level design of a memory compression system, where the memory controller manages the compression.

Contributions. We make the following contributions:

- We provide a comprehensive security analysis of the core design aspects of system-level memory compression, leading to a number of novel leakage primitives.
- We show that these primitives introduce secret-dependent leaks in widely used cryptographic programs and can lead to password disclosure in web servers.
- We devise two design principles that together enforce the *compression non-interference* property, guiding the development of secure memory compression systems.
- We present ZipLock, a secure memory compression design that offers promising compression benefits for practical deployment.

You can find more information about Compression: Impossible, including the ZipLock implementation, using the following link: https://comsec.ethz.ch/compression-impossible.

2 Background on Memory Compression

Figure 1 shows a hardware-level memory compression system; CPU cores and caches operate on *uncompressed* data, while main memory stores data in *compressed* form. Compression is handled by the memory controller, which we refer to the responsible component as the *compression module*. In this paper, *memory compression system* specifically denotes OS-transparent, hardware-level designs. We next discuss their goals (Section 2.1) and key design choices (Section 2.2). Table 1 summarizes design choices and optimizations in proposed memory compression systems.

2.1 Goals of memory compression

Prior memory compression techniques aim to improve either *memory capacity* [9, 27, 28, 30] or *memory bandwidth* [36]. Capacity is improved by packing more data into the same

physical space, while bandwidth improves by transferring more data per memory access. Some prior work also enhances reliability by using compression to create redundancy for storing error-correcting code (ECC) metadata [26], but these designs still inherit the core design choices of capacity- and bandwidth-focused systems.

2.2 Key design choices for memory compression

The fundamental design choices that all memory compression systems must make, include (1) the selection of compression algorithms and compression granularity, (2) management of variable-sized compressed pages and their memory allocation, (3) packing of compressed data blocks within a page, and (4) address translation of compressed memory and its metadata management.

(1) Compression algorithm and granularity. The first step for a memory compression system is choosing a compression algorithm. While more complex algorithms yield higher compression ratios, hardware-level compression must carefully balance complexity and efficiency, as compression operations may lie on the critical path and their latency directly affects performance. Commonly used compression algorithms in the literature are Base Delta Immediate (BDI) [29], Frequent Pattern Compression (FPC) [2], Bit-Plane Compression [18], C-Pack [8], LZ [38], and Deflate [1, 10, 27]. In addition, the design must determine the granularity at which data is compressed and decompressed. Earlier proposals [9, 30] deployed block-level compression with a size of 64B, matching the cache line size as the amount of data typically read from or written to memory in a single access. However, more recent work [27, 28] has deployed more coarse-grained compression, matching the OS page sizes (e.g., 4KB), mainly because compressing at coarser granularity provides a higher compression ratio, although it incurs higher latency for (de)compression. We refer to the former as *line-level*, and the latter as *page-level* compression.

(2) Compressed pages. A key distinction between compressed and conventional (uncompressed) memory is the use of *variable-sized pages*. While conventional systems use fixed-size pages (e.g., 4KB), compressed memory supports multiple sizes to match data compressibility and improve capacity. For example, highly compressible pages may fit in 512B, while incompressible ones remain at 4KB. Although pages could be arbitrarily sized, this complicates allocation, address translation, and introduces fragmentation. Instead, most designs restrict pages to a small set of size buckets (e.g., 512B, 1KB, 2KB, 4KB), simplifying both memory management and translation (that we will discuss shortly).

As data changes over time, a compressed page may no longer fit its current size. A *page overflow* occurs when it becomes less compressible and must expand to a larger size (e.g., 512B to 1KB), while a *page underflow* occurs when increased compressibility allows shrinking to a smaller size.

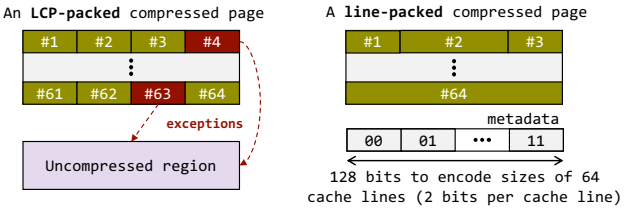


Figure 2. Data placement in line-level compressed pages: LCP-pack policy vs. line-pack policy.

(3) Data placement in compressed pages. While page-level compression treats an entire page as a single unit, line-level techniques must pack multiple variable-sized lines within a page. This requires a *data placement policy*; a compressed page contains 64 cache lines, similar to a 4KB uncompressed page, but each line may have different sizes. Prior work adopts two placement policies: *LCP-pack* [30] and *line-pack* [9]. Figure 2 illustrates these designs.

The LCP-pack policy enforces a uniform compressed size per page, where all 64 lines are compressed to a fixed target size (e.g., 16B). This uniformity simplifies address translation and offset calculation, as all lines occupy equal space. However, lines that cannot meet the target size become *exceptions* and are stored separately in uncompressed form, with their slots in the page marked invalid. While LCP-pack enables fast offset computation, it often suffers from poor compression ratios, especially for workloads with many exceptions.

Conversely, the line-pack policy allows cache lines to be compressed into variable, pre-defined size buckets (e.g., 8B, 16B, 32B, 64B). This enables denser packing and generally better compression grains. However, it requires additional metadata to determine each line's offset within the page. For example, supporting four size buckets requires 2 bits per line, totaling 64×2 bits (16B) per page. Overall, line-packing provides higher capacity gains than LCP-pack at the cost of additional metadata overhead.

Interestingly, page-level compression does not require per-line encoding, as it compresses the entire page as a single unit, and typically achieves higher compression ratios. Its main downside is higher (de)compression latency. However, recent designs introduce optimizations to mitigate these costs, particularly along the critical execution path [27].

(4) Compressed memory address translation. In conventional memory, address translation consists of a single step from virtual to physical addresses (Figure 3a); this translation is visible by the operating system, referred to as OS-visible physical addresses (OSPA). In contrast, OS-transparent memory compression introduces a second translation layer due to variable-sized pages. Specifically, addresses must be translated from OSPA to compressed physical addresses (CPA), where data resides in compressed form (Figure 3b). This requires per-page metadata describing the compressed layout

Table 1. Summary of prior work on hardware-level memory compression.

Proposal	RMC [11]	Buri[37]	LCP[30]	DMC [19]	Compresso [9]	PTMC[36]	COP[26]	MemZip[32]	TMCC [27]	DyLeCT[28]
Goal	Capacity	Capacity	Capacity	Capacity	Capacity	Bandwidth	Reliability	Bandwidth	Capacity	Capacity
Data placement policy										
Same as before						✓	✓	✓		
Line-level (line-packing)	✓				✓					
Line-level (LCP-packing)		✓	✓	✓						
Page-level									✓	✓
Metadata management										
Dedicated region	✓	✓	✓	✓	✓			✓	✓	✓
Inline metadata						✓	✓			
Performance optimizations										
Metadata cache	✓	✓	✓	✓	✓			✓	✓	✓
Zero-aware	✓	✓	✓	✓	✓				✓	✓
Profile-guided				✓					✓	✓
Prediction-based			✓			✓				
Skip-decompression		✓					✓			

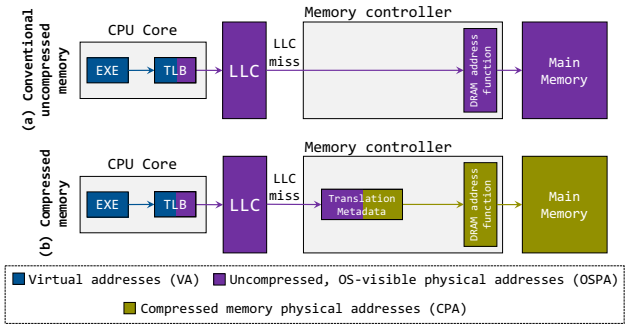


Figure 3. Address translation boundaries in (a) conventional uncompressed memory, and (b) compressed memory.

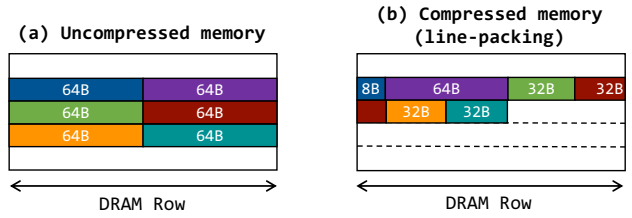


Figure 4. (a) An uncompressed memory. For simplicity, we assume a 128B DRAM row. (b) Compressed memory with line-packing and possible line sizes of 8, 32, and 64 bytes. The red compressed line crosses the row size boundary.

and location. Most systems store this metadata in a dedicated memory region, similar to page table entries. On each memory access, the metadata for the target OSPA page is fetched to perform the second-level translation.

Memory compressions that optimize bandwidth typically avoid additional address translation. Instead, they retain the compressed data in its original location and embed the required metadata alongside the data, also known as *inline metadata* management. Inline metadata includes details like compressed size bucket.

Performance optimizations. Memory compression inherently incurs slowdown due to (de)compression latency and additional address translation. Prior work mitigates these costs using optimizations such as metadata caching, profile-guided compression only for cold paths, and value-aware techniques for common patterns (e.g., zero pages).

3 System-Level Leakage Primitives

In this section, we present three novel leakage primitives that allow an attacker to infer information about compressed data. These primitives do not exist in conventional uncompressed memory. We assume constant (de)compression latency; all primitives instead arise from fundamental system-level properties of memory compression systems. As discussed in Section 2, such systems rely on variable-sized lines/pages or special handling of incompressible data, both introducing *data-dependent* behavior. In Section 4, we show that these primitives break constant-time guarantees in widely-used cryptographic libraries, and also enable end-to-end attacks with precise leakage of arbitrary-sized data. Throughout this section, we use the term *block* to denote the compression granularity (line or page), unless stated otherwise.

3.1 Primitive #1: *Read Amplification*

In traditional uncompressed memory systems, all blocks have fixed sizes, e.g., 64B lines and 64 lines per 4KB page. However, compressed memory systems compress blocks individually, resulting in variable sizes depending on data compressibility (Section 2.2). In this section, we show how different compression and placement policies lead to attacker-observable behavior during memory reads. We use the line-packing policy to illustrate data leaks in our primitive; however, the primitive is general, as all existing compression systems use variable-sized compressed blocks. We also discuss the vulnerabilities of LCP-packing and page-level compression.

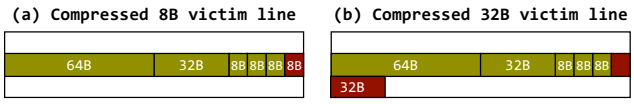


Figure 5. *Read Amplification* primitive that detects where the compressed size of a victim line is (a) 8B or (b) more than 8B, by pushing the less compressible lines to the next row.

Leaking data compressibility in line-packing. Based on the DRAM organization in commodity devices, a DRAM row size (e.g., 8KB) is divisible by the fixed cache line size (e.g., 64B). This configuration ensures that cache lines always reside within a single DRAM row, as illustrated in Figure 4a. However, in a line-packing placement, cache lines within a single page can have multiple target size buckets. Compressed memory systems select compressed size buckets in a way to optimize for higher compression rates. For example, prior work [11, 30] uses compressed line sizes of 0, 22, 44, and 64 bytes. Interestingly, even with row-size–divisible sizes, compressed cache lines may still span DRAM row boundaries, requiring two memory accesses to read a single line. For instance, Compresso [9] demonstrates that using compressed line sizes of 0, 8, 32, and 64 bytes significantly reduces DRAM row crossings, yet 3.2% of accesses still suffer from cache line splits. Figure 4b shows a simple example achieving a 1.9× compression ratio (384B to 200B). In this case, the compressed red cache line (32B) crosses a row boundary, with 24B in the first row and 8B in the second. *Loading this cache line results in two memory accesses.*

The possibility of crossing a DRAM row boundary is sufficient for an attacker to leak information and *amplify* the memory access timing in a data-dependent way, even in systems that are optimized to reduce such cases for normal applications. Hence, *crossing a DRAM row boundary is a direct consequence of variable-sized compressed blocks.*

***Read Amplification* primitive**. Figure 5 illustrates the *Read Amplification* leakage primitive, which enables an attacker to leak the compressed size of a victim cache line. For example, assume the attacker aims to determine whether the compressed size of a victim cache line is 8B in a line-packing placement with possible sizes of 0, 8, 32, and 64 bytes. The attacker first arranges compressed cache lines in a specific DRAM row such that only 8B of space remains at the end of the row. If the victim's compressed cache line is 8B, it fits within the same row (Figure 5a). Otherwise, it crosses the row boundary (Figure 5b). In the former case, loading the victim cache line requires only one memory access, whereas in the latter case, two memory accesses are observed.

Leaking data compressibility in page-level compression. Page-level compression techniques [27, 28] also support variable-sized pages, e.g., 1.5KB and 4KB compressed pages, and are therefore vulnerable to the *Read Amplification* primitive. For instance, a 1.5KB compressed page may cross a DRAM row boundary (e.g., 8KB), depending on its starting address offset. This behavior can reveal the compressibility of the page and its compressed size. However, as we will demonstrate in Section 4.2, simply reading variable-sized compressed pages is sufficient for the *Read Amplification* primitive to create the data-dependent timing difference required for an exploit.

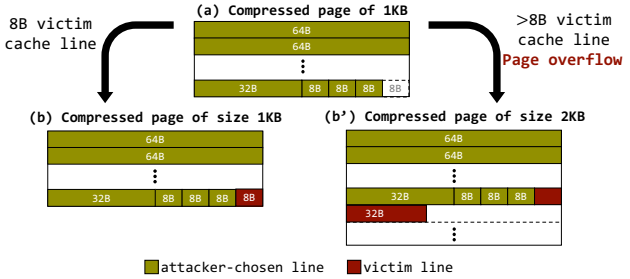


Figure 6. *Write Expansion* that detects where the compressed size of a victim cache line is (a) 8B or (b) more than 8B, for example 32B, through page overflow events.

Leaking data compressibility in LCP-packing. The LCP-packing mechanism assigns a fixed compressed cache line size to all lines within a page. However, it still permits multiple compressed line sizes across different pages, which results in variable-sized compressed pages. Furthermore, LCP-packing introduces *exceptions*, cache lines within a page that cannot be compressed to the target size. These exceptions are stored in a separate memory region, which their accesses can leak information about their compressibility.

3.2 Primitive #2: *Write Expansion*

Our next leakage primitive exploits data-dependent behavior during memory writes. As discussed in Section 2, memory compression systems manage variable-sized pages using mechanisms that maximize capacity. A *block inflation* occurs when a data block no longer fits in its allocated size. For example, in a line-packed page, a cache line may grow from 8B to 32B after a write, potentially requiring relocation of subsequent lines. Such inflations can lead to a *page overflow* in both line- and page-level schemes, where the page expands to a larger size bucket (e.g., 1KB to 2KB). Handling a page overflow usually involves relocating and recompressing the entire page, resulting in multiple memory reads and writes.

***Write Expansion* primitive**. Figure 6 illustrates the *Write Expansion* primitive, using a line-packed page as an example, where page overflows are exploited to leak the compressed size of a victim cache line. The attacker first primes the page with chosen blocks such that only the target victim size remains available. For example, in Figure 6a, only 8B of free space remains in a 1KB compressed page. When the victim block is written, a page overflow occurs if its compressed size exceeds 8B (Figure 6b'), whereas a size of 8B fits within the original 1KB page without overflow (Figure 6b). The

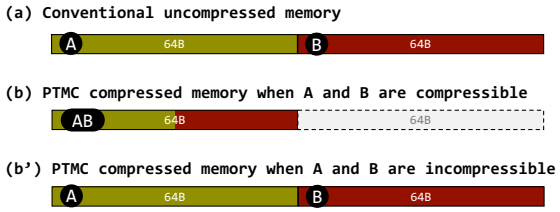


Figure 7. Bandwidth-optimized compression. (b) If lines A and B are compressible, one access brings both to the CPU; (b') otherwise, it is similar to (a) uncompressed memory.

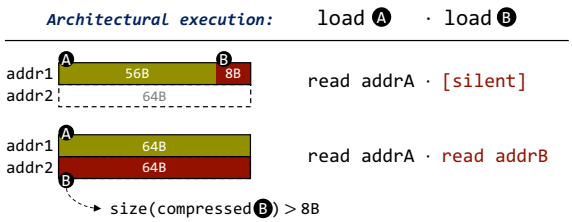


Figure 8. *Silent Compressed Load* primitive revealing the compressibility of victim cache line *B*, producing silent loads when lines *A* and *B* are compressible.

attacker can detect a page overflow even with user-level timers (e.g., distinguishing a normal memory read from a full page read+recompress+write sequence), or by accessing any line in the page, which is delayed until relocation completes.

A sibling primitive to *Write Expansion* is *Write Contraction*, which exploits page *underflows*. When a written block becomes more compressible, some systems shrink the page (e.g., from 2KB to 1KB) to improve capacity. *Write Contraction* introduces the same type of leakage as *Write Expansion*, and we use the terms interchangeably.

3.3 Primitive #3: *Silent Compressed Load*

Some memory compression systems improve bandwidth by enabling a single memory access to fetch more data than in conventional uncompressed memory. For example, PTMC [36] does not densely pack compressed lines in memory, and therefore does not improve capacity, but it enables fetching up to four lines with a single memory access. More specifically, if two lines Ⓐ and Ⓑ can be compressed into one 64B line (Figure 7b), accessing either Ⓐ or Ⓑ will bring both to the CPU. If they are not compressible (Figure 7b'), the behavior is the same as in uncompressed memory (Figure 7a), i.e., accessing lines Ⓐ and Ⓑ results in only one line being fetched. Figure 8 illustrates how the compressibility of a victim cache line Ⓑ can result in a silent load, that is, a memory read is skipped when line Ⓑ is compressible and can be packed with its adjacent line Ⓐ. Such *data-dependent silent loads* can leak the compressibility of data, leading to our *Silent Compressed Load* primitive.

From a practical point of view, an attacker can observe silent loads using established cache side channels, such as Prime+Probe [25]. For instance, the attacker can prime the cache with known data and later probe it to determine how many of its lines were evicted. This eviction pattern reveals whether the victim loaded one, two, or four cache lines, depending on the compressibility of the victim's data.

4 Security Implications of the Leakage Primitives

This section analyzes the security implications of our new leakage primitives. Given their data-dependent leaks, we first ask whether they can break constant-time guarantees in widely-used cryptographic libraries (Section 4.1). We then examine whether they can enable end-to-end attacks, where we disclose full passwords in web servers (Section 4.2).

4.1 Breaking constant-time cryptographic code

Prior work has shown that data-dependent microarchitectural optimizations in modern CPUs can break constant-time code [4, 7], and we explore whether future systems equipped with compressed memory might similarly compromise the security of constant-time implementations.

Constant-time programming. To ensure side-channel security in cryptographic code, developers adopt a constant-time programming discipline. This assumes a leakage model where an attacker can observe control flow, memory access patterns, and variable-latency instructions, known as the *constant-time leakage model (**CT**)*. Under this model, security is defined as non-interference [3]: a program is secure if its leakage observations remain identical for all inputs that differ only in secret values (formalized in Section 5).

We analyze widely used cryptographic libraries to determine whether leakage from our memory-compression primitives violates constant-time guarantees. To this end, we use LMTest [4], a state-of-the-art tool for evaluating side-channel security under various leakage models, and formally define our primitives using its domain-specific, Lisp-like language. As an example, we present the leakage model for the *Silent Compressed Load* primitive. Note that these primitives capture key design choices in prior compressed memory systems, with Table 1 indicating the affected designs.

***Silent Compressed Load* leakage model**. Listing 1 shows the leakage model for the *Silent Compressed Load* primitive, introduced by PTMC [36]. In LMTest, a leakage model consists of a *guard* that selects relevant events and a *body* that computes the observable leakage. Here, the guard captures memory load operations from address `addr` (line 2). The body models PTMC's leakage where even and odd addresses are packed if they are compressible. It first identifies the even–odd cache line pair containing the accessed address and retrieves both lines (lines 3–8). Each line is then compressed using PTMC's hybrid scheme, which selects the smaller output of FPC [2] and BDI [29] (lines 9–10). Finally, the model outputs a leakage bit (line 11): it returns 1 if the

```
(defleakage SilentCompressedRead []
(on [(load [addr]_sz)
  (let [even-line-id    (<< (>> addr 7) 1)
      odd-line-id       (+ even-id 1)
      even-addr         (<< even-line-id 6)
      odd-addr          (<< odd-line-id 6)
      even-data         (&mem.read-bytes even-addr 64)
      odd-data          (&mem.read-bytes odd-addr 64)
      even-comp-len     (hybrid-compress even-data)
      odd-comp-len      (hybrid-compress odd-data)]
  #("sl" addr (pack even-comp-len odd-comp-len)))]))
```

Listing 1. Leakage model of the *Silent Compressed Load* primitive. A cache line size 64B (with 6 bits in the address).

two compressed lines fit within 60B (with 4B reserved for metadata), and `0` otherwise. This captures the compressed-data-dependent leakage of the *Silent Compressed Load* primitive.

Testing results. We tested the leakage models of our system-level leakage primitives against 12 cryptographic programs from two widely-used libraries: libsodium [22] and libnettle [21]. Table 2 summarizes the results of our experiments. For each program and leakage model pair, we executed 200 test cases using the LMTest tool and report whether a secret-dependent leak was found.

As the results indicate, all programs are secure under the standard constant-time definition, i.e., they are non-interfering with respect to the constant-time leakage model (*CT*). However, almost all programs exhibit secret-dependent leakage for at least one of our memory compression attack primitives (except for x25519 in libnettle). In most cases, the programs leaked secrets under *multiple* leakage models.

To further analyze the root cause of the leaks, we closely examined the non-interference violation in libsodium x25519 under the *Silent Compressed Load* leakage. With the public input fixed, LMTest finds that flipping private-key bit 253 changes the compressibility of temporary stack data inside the field multiplication routine: two neighboring cache lines compress to 1+58=59 bytes in one trace, but require 1+60=61 bytes in another (incompressible in PTMC), producing secret-dependent memory traces.

4.2 Leaking HTTP passwords in web servers

To further demonstrate the practicality of exploiting compressed memory leakage primitives, we demonstrate an end-to-end proof-of-concept to use *Read Amplification* and *Write Expansion* primitives to *precisely* leak secrets in widely used web servers.

4.2.1 Threat model and assumptions.

We assume an attacker who aims to leak passwords from a victim nginx web server [24], with typical system defenses, such as ASLR, enabled. Note that ASLR randomizes addresses at page granularity. In the default nginx binary that we evaluate [24], the input and user passwords are allocated within the same page, which means that ASLR has no impact on our attack.

Table 2. Testing leakage model of our primitives against libsodium [22] and libnettle [21] cryptographic libraries. ‡ ***CT*** refers to standard constant-time leakage model.

	Program	*CT*‡	*Read Amplification* Line-pack	*Read Amplification* LCP	*Write Expansion* Line-pack	*Write Expansion* LCP	*Silent Compressed Load*
libsodium	ed25519	▼	■	■	■	■	■
	hmac	▼	■	■	■	■	▼
	poly1305	▼	■	▼	■	■	■
	salsa	▼	▼	▼	■	■	▼
	sha512	▼	▼	▼	■	■	▼
	stream_xor	▼	▼	▼	■	■	■
	x25519	▼	■	■	■	■	■
libnettle	ed25519	▼	■	■	■	■	■
	hmac	▼	▼	▼	■	■	▼
	salsa	▼	▼	▼	■	▼	▼
	sha512	▼	■	■	■	■	■
	x25519	▼	▼	▼	▼	▼	▼

■ Leak found ▼ Leak not found

Reverse engineering. The only information the attacker requires from the underlying compression design is the compression granularity and compressed page sizes. Given our proposed compression oracle, reverse engineering these details is straightforward: the attacker can build payload candidates under different assumptions (e.g., page-level vs. line-level granularity and different compression thresholds) and infer the correct parameters from timing differences in reads and writes.

Memory layout assumptions and capabilities. The *Write Expansion* primitive does not require any specific memory layout. It only requires the attacker input and secret to be allocated within the same page, which already occurs in the default nginx configuration. A page overflow naturally occurs when the compressibility of `input+secret` changes. In contrast, the *Read Amplification* primitive requires additional considerations, especially when targeting line-level compression techniques, where placing the victim page across a DRAM row boundary is strictly necessary. Since memory allocators in memory compression modules use free lists of pages, an attacker can perform memory massaging similar to those proposed for Rowhammer attacks [5]. First, the attacker performs a templating step to identify compressed physical pages, CPs, with the desired physical placements, e.g., one that crosses a DRAM row boundary and one that does not (Figure 5). Second, the attacker releases the CPs along with a sufficient number of other pages to reach the threshold that causes the target per-CPU pages (PCP) list to return the CPs to the buddy allocator [5]. On the victim side (nginx), the attacker sends a sufficient number of requests to exhaust nginx's PCP list and force it to allocate pages from the buddy allocator, causing it to reuse the CPs. Note that for page-level compression, such precise memory massaging is not strictly required, since these techniques already amplify reads by requiring the entire compressed page to be read for decompression. Reading variable-sized compressed pages

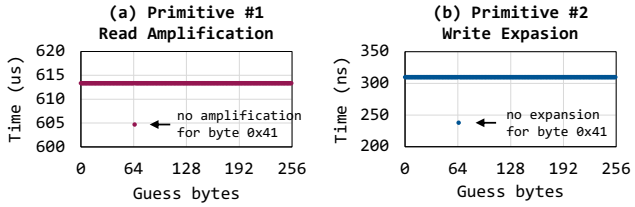


Figure 9. Leakage observations in gem5 using *Read Amplification* and *Write Expansion* primitives, for secret byte `0x41`.

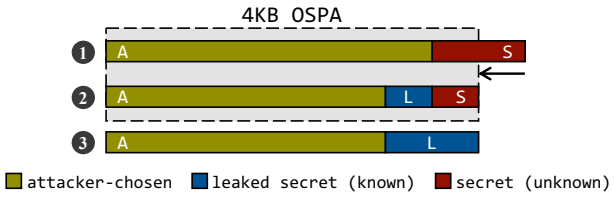


Figure 10. Byte-by-byte password leakage in nginx.

(e.g., 1.5KB vs. 4KB) produces a measurable timing difference, as used in our PoC.

Additionally, this massaging operates on physical-memory allocation and is orthogonal to ASLR: ASLR randomizes the victim's virtual address layout, but does not prevent the underlying compressed physical page from being selected through allocator manipulation.

4.2.2 Attack ingredients.

To achieve this attack, we require three main building blocks.

(1) Leakage observability. We first evaluate whether the primitives introduce observable timing differences using two microbenchmarks in gem5 with configurations detailed in Table 4. Depending on a target byte (`0x41`), the benchmarks trigger either read amplification or page expansion. Figure 9a shows that *Read Amplification* yields a measurable timing difference after 4K repetitions ($604\mu s$ vs. $614\mu s$). Figure 9b also shows that *Write Expansion* produces a clear gap ($238ns$ vs. $310ns$) even with a single execution, as it involves full page recompression and migration.

(2) Co-location and memory alignment. The attack also requires triggering the primitives by co-locating attacker-controlled data with the secret, enabling control over their combined compressibility, precise placement, and byte-wise leakage. Authentication in the nginx web server [24] provides this capability: it places the input before the target password hash, and later read both for comparison. As shown in Figure 10, this allows the attacker to (1) align chosen data with the secret and (2) leak it byte by byte.

(3) Compressibility oracle. Prior building blocks allow the attacker to place chosen/known data directly before a secret byte at each step. The remaining requirement is an oracle that lets the attacker test the correctness of 256 possible guesses. An uncompressed data block can be represented as $(A \cdot L \cdot s)$, where A denotes attacker-chosen bytes, L is the already leaked prefix, and s is the unknown target byte. The goal is to choose A such that, for each guess g, the 4KB page compresses into a smaller bucket when $g = s$ and a larger one when $g \neq s$ (e.g., 1.5 KB vs. 4KB [28]).

To achieve this, we devise a generic algorithm that constructs 256 candidates, denoted as $A_{L,g}$, for each $g \in \{0, \ldots, 255\}$ and leaked prefix L, given a compressor C and threshold T:

$$\forall g \in \{0,\ldots,255\} : C\left(A_{L,g} \cdot L \cdot s\right) = \begin{cases} C_1 \leq T & \text{if } g = s \\ C_2 > T & \text{if } g \neq s \end{cases}$$

To find suitable $A_{L,g}$ candidates, we assume a template $X_{L,g} = (U \cdot C)^*$, where U is high-entropy (incompressible) data of length u, and C is compressible data of length c formed by repeating the motif $M_{L,g} = L \cdot g$. We search over different (u, c) parameters and tune them in a way that small compressibility changes cross the threshold T. Our results confirm that given the state-of-the-art Deflate compressor [27, 28], and $T = 1.5$ KB, we can successfully build suitable candidates to be able to leak a 16B password in nginx.

Results. Putting these pieces together, we achieve a successful end-to-end exploit with 1.36 KB/s leakage rate using the *Read Amplification* primitive, leaking the password in less than a second.

Additionally, to evaluate robustness under system noise, we repeat the nginx attack in a three-core setup: one core runs the attack, while the others run `omnetpp` and `mcf` as memory intensive workloads from SPEC CPU2017 [33] to stress memory and caches. The attack achieves a leakage rate of 0.90 KB/s (compared to 1.36 KB/s without noise) and still leaking the full password, showing signals remain robust under noise.

These findings demonstrate that our proposed leakage primitives violate constant-time guarantees and lead to exploitable data-dependent leaks. This raises an important question: *can we define general design principles for memory compression that ensure side-channel security before such systems are deployed in real hardware?*

5 Secure Memory Compression Principles

A natural first instinct in designing secure memory compression is to ensure that (de)compression algorithms are constant time. However, as we have demonstrated, previous memory compression systems (Table 1) introduce data-dependent leaks, even if we enforce constant (de)compression latencies. The root cause of these leaks lies not in compression latency, but in fundamental design choices of compressed memory systems. In this section, we formalize these root causes and derive secure design principles for hardware-level memory compression.

5.1 Formal model and security property

Following prior established foundation [6, 13, 14], we model the security of memory compression using a *non-interference* property: compression is secure if it does not introduce additional observable behavior beyond the architectural level.
Observations. An attacker observes a trace of events $Obs := AObs \cup CObs_{\text{COMP}}$ for program p, where $AObs$ captures architectural events (e.g., `load` n), and $CObs_{\text{COMP}}$ captures events that are compression-induced (e.g., `POverflow` n) for a generic compression semantics, denoted as COMP.
States and execution. A machine state is $\alpha = \langle \sigma, \mu \rangle$, where σ denotes the architectural state (i.e., the contents of registers and memory), and μ denotes the state of the microarchitecture, including the memory compression structures (e.g., the tracked metadata). Executing program p under compression COMP from initial state α_0 produces a trace:

$$O_{\text{COMP}}(p, \alpha_0) = [\tau_1; \tau_2; \cdots; \tau_n] \quad \tau_i \in Obs$$

We write $\pi_{\text{ARCH}}(\cdot)$ to denote the projection of a trace that only includes the architectural events from $AObs$. We also define an architectural execution step:

$$\text{STEP}^1_{\text{COMP}}(p, \alpha) = (a, \overline{c}, \alpha')$$

which represents a single architectural execution step starting from state α, producing an architectural event $a \in AObs$ and a trace of triggered compression events $\overline{c} = [c_1; \cdots; c_n]$ after a (where $c_i \in CObs$) and resulting in a new state α'.
Security policy. A policy ϕ partitions memory into public and secret regions. Two states are *ϕ-low equivalent*, denoted $\alpha \approx_\phi \alpha'$, if they agree on all stored public data.

Definition 1 (Compression non-interference – CNI)**.**

$$p \vdash_{\text{COMP}} \text{CNI} \overset{\text{def}}{\iff} \forall \alpha, \alpha' : \sigma \approx_\phi \sigma' \ \wedge \ \mu \approx_\phi \mu' \wedge \pi_{\text{ARCH}}(O_{\text{COMP}}(p, \alpha)) = \pi_{\text{ARCH}}(O_{\text{COMP}}(p, \alpha')) \Rightarrow \ O_{\text{COMP}}(p, \alpha) = O_{\text{COMP}}(p, \alpha')$$

Informally, this property ensures that compression-induced events do not reveal additional information about secrets beyond architectural observations.

5.2 Secure design principles

The leakage primitives introduced in Section 3, all violate the CNI property in one of these two forms:

(1) Instantaneous leaks that arise from a single memory request (read or write) and can be observed in a repeatable and self-contained manner, without influencing or depending on subsequent memory operations.
(2) Cascading leaks that emerge as a result of a sequence of memory operations, where an initial access influences the behavior of subsequent memory requests, thereby triggering a context-dependent leak.

Read Amplification is an instantaneous leak, as each access independently produces the same observation (either crossing the DRAM row boundary or not) without affecting future behavior. In contrast, *Write Expansion* and *Silent Compressed Load* are cascading leaks: their observations depend on prior accesses that modify the compression state, leading to context-dependent behavior visible to the attacker. For example, in the *Write Expansion* primitive, a write operation first expands a compressed line, which then triggers relocation or recompression events, actions that can be observed by an attacker. Similarly, in *Silent Compressed Load* primitive, accessing even or odd lines influences access to the other.

Given these two high-level categories of compressed-data-dependent leaks, we propose two general design principles aimed at eliminating each class of leakage. We later show that enforcing these two design principles results in a memory compression systems that adheres to the CNI property.

5.2.1 Blocking instantaneous leaks.

To prevent instantaneous leaks, the memory controller must enforce uniform behavior across memory requests. For example, if a design may issue one or two memory accesses depending on its compression choices (e.g., due to variable-sized cache lines), it must either always issue two accesses or hide the additional one when it occurs. This requirement leads to our first principle: *uniform access*.

> ***Uniform access* (UA) principle**: All read and write requests are subject to a uniform procedure before returning to the processor.

Definition 2 formally specifies the *uniform access* principle for a memory compression design.

Definition 2 (Uniform Access)**.** A compression semantics COMP satisfies Uniform Access if:

$$\text{UA}_{\text{COMP}} \overset{\text{def}}{\iff} \forall p, \ \forall \sigma, \sigma', \ \forall \mu : \sigma \approx_\phi \sigma' \wedge \text{STEP}^1_{\text{COMP}}(p, \langle \sigma, \mu \rangle) = (a, \overline{c}, \langle \sigma_1, \mu_1 \rangle) \wedge \text{STEP}^1_{\text{COMP}}(p, \langle \sigma', \mu \rangle) = (a, \overline{c}', \langle \sigma'_1, \mu'_1 \rangle) \Rightarrow \overline{c} = \overline{c}'.$$

5.2.2 Blocking cascading leaks.

While *uniform access* prevents leaks from memory requests in isolation, cascading leaks can still persist in current designs. To eliminate them, the memory controller must make compression decisions in a stateless manner, independent of prior accesses. For example, operations such as page relocation or recompression should not depend on earlier block expansions or contractions. This requirement leads to our second principle.

> ***Stateless footprint* (SF) principle**: Any memory request has no data-dependent effects on subsequent accesses.

Definition 3 formally specifies this principle.

Definition 3 (Stateless Footprint)**.** A compression semantics COMP satisfies Stateless Footprint if:

$$\begin{aligned}\mathrm{SF}_{\mathrm{COMP}} \overset{\text{def}}{\iff} \forall p,\ \forall \sigma, \sigma',\ \forall \mu : \sigma \approx_\phi \sigma' \ \wedge \\ \textsc{Step}^1_{\mathrm{COMP}}(p, \langle \sigma, \mu \rangle) = (a, \overline{c}, \langle \sigma_1, \mu_1 \rangle) \ \wedge \\ \textsc{Step}^1_{\mathrm{COMP}}(p, \langle \sigma', \mu \rangle) = (a, \overline{c}', \langle \sigma_1', \mu_1' \rangle) \Rightarrow \mu_1 \approx_\phi \mu_1'.\end{aligned}$$

5.3 Compression non-interference theorem

Theorem 1 proves that memory compression COMP achieves CNI (Definition 1) by enforcing the UA and SF principles.

Theorem 1 (UA + SF imply CNI)**.** *If UA_{COMP} and SF_{COMP} hold, then for all programs p, COMP satisfies CNI:*

$$UA_{COMP} \ \wedge \ SF_{COMP} \Rightarrow p \vdash_{COMP} CNI.$$

Proof. Fix any p and any initial states $\alpha = \langle \sigma, \mu \rangle$ and $\alpha' = \langle \sigma', \mu \rangle$ such that $\sigma \approx_\phi \sigma'$ and

$$\pi_{\mathrm{ARCH}}(O_{\mathrm{COMP}}(p, \alpha)) = \pi_{\mathrm{ARCH}}(O_{\mathrm{COMP}}(p, \alpha')) = \Gamma.$$

We show $O_{\mathrm{COMP}}(p, \alpha) = O_{\mathrm{COMP}}(p, \alpha')$ by induction on the length of the common architectural trace Γ. If Γ is empty, both executions take no architectural steps, hence their full traces are equal. Otherwise, let the common architectural projection Γ begin with a (induction hypothesis). By the induction step discipline, we can write:

$$O_{\mathrm{COMP}}(p, \alpha) = [a] \cdot \overline{c} \cdot \tau \quad \text{and} \quad O_{\mathrm{COMP}}(p, \alpha') = [a] \cdot \overline{c}' \cdot \tau'$$

for some $\overline{c}, \overline{c}'$ and suffix traces τ, τ'. Equivalently, there exist states $\langle \sigma_1, \mu_1 \rangle$ and $\langle \sigma_1', \mu_1' \rangle$ such that:

$$\textsc{Step}^1_{\mathrm{COMP}}(p, \langle \sigma, \mu \rangle) = (a, \overline{c}, \langle \sigma_1, \mu_1 \rangle)$$
$$\textsc{Step}^1_{\mathrm{COMP}}(p, \langle \sigma', \mu \rangle) = (a, \overline{c}', \langle \sigma_1', \mu_1' \rangle).$$

By Definition 2 (*uniform access*), $\overline{c} = \overline{c}'$ and by Definition 3 (*stateless footprint*), $\mu_1 \approx_\phi \mu_1'$. Let $\alpha_1 = \langle \sigma_1, \mu_1 \rangle$ and $\alpha_1' = \langle \sigma_1', \mu_1' \rangle$. Since the architectural projections of the full executions were equal (i.e., Γ) and the first architectural event a matches, the remaining architectural projections from α_1 and α_1' are also equal: $\pi_{\mathrm{ARCH}}(\tau) = \pi_{\mathrm{ARCH}}(\tau')$. Moreover $\mu_1 \approx_\phi \mu_1'$ and (by construction of ϕ-low equivalence) $\sigma_1 \approx_\phi \sigma_1'$. Thus the induction hypothesis applies to the suffix executions, yielding $\tau = \tau'$, and therefore the full traces are equal. □

5.4 Applying the secure design principles

Applying our principles to existing memory compression systems is challenging, as they aggressively optimize for performance, capacity, and bandwidth, without flexibility for security enforcements. Naively patching them often undermines their core optimizations. For example, mitigating *Silent Compressed Load* would require redundant memory reads even when data is cached, wasting bandwidth, while preventing *Write Expansion* would require frequent recompression or relocation, incurring significant overheads.

These challenges highlight the need to rethink fundamental design choices in memory compression, such as data placement and compressibility decisions. Future work must balance efficiency with security under these principles. To support this direction, we propose a secure compression design as a baseline for future research.

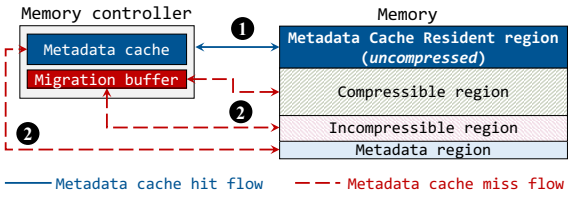


Figure 11. Overview of the ZIPLOCK design.

6 ZIPLOCK

We present ZIPLOCK, a memory compression system enforcing our secure principles. Its goal is twofold: (1) to demonstrate that a principled approach enables secure memory compression in practice, and (2) to show that such a design can still achieve promising compression without prohibitive slowdown compared to the state of the art. ZIPLOCK serves as a secure and practical baseline for future. Section 6.1 presents its design, Section 6.2 analyzes its adherence to the principles, and Section 6.3 evaluates compression and performance.

6.1 ZIPLOCK design

We build ZIPLOCK based on two key insights from recent work [27, 28]: (1) page-level compression achieves better compression ratio and translation reach than fine-grained, line-level compression, and (2) frequently accessed pages should remain uncompressed to avoid performance penalties of (de)compression on the critical path. However, ZIPLOCK leverages these insights to construct a security-first design, without aggressive and insecure optimizations.

Figure 11 shows an overview of ZIPLOCK; it dedicates a fixed memory region for metadata and includes a metadata cache in the memory controller, similar to prior work, which also determines which pages remain uncompressed. We introduce the *metadata cache resident region* (MCR), mirrored by the metadata cache, while the rest of memory is divided into compressible and incompressible regions. Pages whose metadata is not cached reside in either of these regions. On each memory access (i.e., an LLC miss), ZIPLOCK follows either the metadata cache hit flow ❶ or miss flow ❷.

❶**Metadata cache hit flow**. Whenever a metadata cache hit occurs, the corresponding page is, by design, located in the MCR region and can be accessed directly without any additional steps, as it is stored in uncompressed form.

❷**Metadata cache miss flow**. On a metadata cache miss, ZIPLOCK migrates the pages corresponding to both the evicted and newly fetched metadata entries, referred to as *evicted* and *incoming* pages. Evicted pages move from the MCR region to either compressible or incompressible, while incoming pages



Figure 12. A per OS page metadata entry in ZipLock.

are migrated from their original region (compressible or incompressible) into MCR. All migrations (MCR↔compressible and MCR↔incompressible) follow the same procedure: the full page is read into a buffer in memory controller (sized to match the compression granularity) and written back with any necessary modifications. Constant-time decompression is applied on reads and constant-time compression on writes, regardless of the source or destination (e.g., compression outcome may be discarded in the incompressible→MCR direction). Although some operations are functionally redundant, they are required to satisfy the *uniform access* principle.

Page-level compression. ZipLock performs page-level compression, compressing the entire page as a single block (i.e., 4KB granularity). To determine whether a page is compressible, ZipLock uses a fixed target compressed size for *all* pages, e.g., 2KB for 2:1 compression, aligned with the DRAM 8KB row size. This ensures that all blocks within their respective region, compressible or incompressible, have uniform sizes.

Metadata entries and region. ZipLock allocates a dedicated region in memory to store metadata for each page. Figure 12 illustrates the structure of a metadata entry. Each entry consists of: (1) *control bytes*, which encode the current state of the OSPA page, e.g., whether the page is compressible and its compressed size; and (2) *page address*, which specifies memory location of the compressible/incompressible page. Since we require 8B of metadata per page, the metadata region overhead is $\frac{8B}{4KB+8B} = 0.195\%$ of memory.

6.2 Security analysis

This section details the design choices in ZipLock that enforce secure memory compression; i.e., it strictly enforces the UA and SF principles, where Theorem 1 proves they imply non-interference. The goal is to prevent leakage that distinguishes between MCR↔compressible and MCR↔incompressible migrations that reveal data compressibility.

Uniform page migrations and compression. As discussed in Section 6.1, all memory accesses in the metadata cache hit flow are served entirely from the MCR region in uncompressed form, similar to conventional memory. Additionally, during the metadata cache miss flow, we apply the same procedure for *all* page migrations and (de)compression operations, regardless of the source and destination regions. Page migrations are performed solely through the migration buffer and do not update the LLC. As a result, an attacker cannot use cache-based side channels to observe the footprints or addresses of the migrated pages.

Static region partitioning. ZipLock partitions memory into four regions: MCR, compressible, incompressible, and metadata. While prior work [20] shows that dynamically resizing regions can improve capacity, ZipLock uses fixed, static partitioning to prevent cascading leaks from data-dependent resizing. The MCR size is determined by the metadata cache, while compressible and incompressible are set at design time based on expected workloads and advertised capacity to the OS, and remain fixed thereafter.

Aligned region partitioning. To further prevent attackers from identifying region boundaries, starting addresses must also satisfy three constraints: (1) align with the LLC size (e.g., 2MB), (2) align with DRAM row boundaries to avoid *Read Amplification* leaks from placing variable-sized pages within the same row, and (3) align with DRAM bank addresses to prevent distinguishing migrations via bank conflicts; misalignment can expose differences between MCR↔compressible and MCR↔incompressible migrations.

Metadata cache leaks. While ZipLock amplifies leakage from metadata cache misses, these remain address-dependent behaviors already leaked by other microarchitectural components (e.g., TLBs and caches). In Section 7, we discuss additional security considerations in practice.

Leakage testing. We implemented ZipLock model in LMTest, producing distinct observations for paths ❶ and ❷ in Figure 11, but indistinguishable events for MCR↔compressible and MCR↔incompressible paths. Table 3 presents the testing results, demonstrating that no leaks were found. We also tested ZipLock with a finer compression granularity (line-level) to demonstrate its robustness.

Table 3. Testing leakage of ZipLock against libsodium [22] and libnettle [21] cryptographic libraries.

Program	libsodium							libnettle				
	ed25519	hmac	poly1305	salsa	sha512	stream_xor	x25519	ed25519	hmac	salsa	sha512	x25519
CT	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼
ZipLock	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼
ZipLock (*line-level*)	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼

■ Leak found ▼ Leak not found

Table 4. Configuration parameters for simulation.

Core	3GHz OoO core, issue Width = 4, ROB = 192 entries 32KB L1I/D, 256KB L2, 2MB LLC, 64B cachelines
DRAM	8GB DDR4-2400MHz BL=8, tCL=14.16, tRCD=14.16, tRP=14.16
DyLeCT	Metadata cache: 128KB (LRU), (de)compressor: Deflate TMCC [27] (with partial-page decompression)
ZipLock	Metadata cache: 128KB (LRU), (de)compressor: Deflate TMCC [27] (always full-page decompression)

6.3 Evaluation

We evaluate compression and performance of ZipLock and two configurations of the state of the art, DyLeCT [28]:

- *DyLeCT-low-comp*: configured to the default 1.3× compression ratio in the original paper [28];

- *DyLeCT-iso-comp*: configured to the compression ratio similar to ZipLock.

Setup. We model ZipLock and DyLeCT [28] in the O3 CPU model of gem5 [23]. Table 4 shows the details of simulation parameters. We evaluated use the SPEC CPU2017 suite [33] and simulate 1B instructions from the representative region of workloads using the SimPoint methodology [15].

Compression ratio. Figure 13a shows the compression ratio of each design relative to the uncompressed baseline. ZipLock achieves a 1.65× ratio, while *DyLeCT-low-comp* achieves 1.25×, primarily due to its less aggressive compression configuration. Additionally, ZipLock performs more frequent page migrations (on each metadata cache miss), enabling more aggressive (re)compression of blocks as data becomes more compressible over time. However, *DyLeCT-iso-comp* has the same compression ratio as ZipLock, as we intentionally configured it to match ZipLock.

Performance. Figure 13b presents the execution time of ZipLock, *DyLeCT-low-comp*, and *DyLeCT-iso-comp*, normalized to the uncompressed baseline. While *DyLeCT-iso-comp* exhibits an average (geometric mean) slowdown of 1.32×, ZipLock incurs a slowdown of 1.37×. This means that ZipLock introduces only 3.5% overhead over *DyLeCT-iso-comp* (note that both designs achieve the same compression ratio). *DyLeCT-low-comp* incurs a slowdown of 1.11×, which is lower than ZipLock and *DyLeCT-iso-comp*, as it applies less aggressive compression. The performance overhead of ZipLock is higher for memory-intensive applications, particularly those suffering from TLB and metadata cache misses. For instance, ZipLock incurs a 2.7× slowdown on `omnetpp`, compared to 2.1× for *DyLeCT-iso-comp* and 1.3× for *DyLeCT-low-comp*. Overall, DyLeCT offers more flexibility in trading off compression ratio and performance, whereas ZipLock adopts strict design choices to uphold its security guarantees.

Memory traffic. Figure 14 shows page migration traffic in ZipLock and *DyLeCT-iso-comp*, normalized to total traffic in the respective design. Page migrations account for 35% of traffic in *DyLeCT-iso-comp*, and increased to 49% in ZipLock. However, increased traffic alone does not directly translate to slowdown. We analyze commit-cycle stalls next.

Stall reasons breakdown. Figure 15 shows commit-cycle breakdowns for the uncompressed baseline, ZipLock, and *DyLeCT-iso-comp* across SPEC CPU2017. `CommitSuccess` is the fraction of cycles committing instructions; remaining cycles are stalls (`LoadStall`, `StoreStall`, others such as misprediction squashes). ZipLock increases memory-stall time, especially loads that miss metadata and trigger page migrations: load stalls rise from 38% in baseline to 44% in ZipLock. *DyLeCT-iso-comp* shows a similar bottleneck, indicating main sources of overhead are compression related page migrations rather than ZipLock's security enforcements.

7 Discussion

7.1 Leakage upon running out of memory

OS-transparent, hardware-level memory compression schemes advertise a memory capacity larger than the physically available memory. This creates a situation where an attacker can fill memory with incompressible data, exhausting the physical memory while the OS continues to operate under the illusion that additional memory is still available. Prior work [9] suggests handling such cases using techniques similar to memory overcommitment in virtualized environments, such as memory ballooning. However, in our design, reaching this boundary can leak sensitive information, allowing an attacker to exploit the transition as a low bandwidth side channel. Since memory compression is mostly useful in cloud systems with a large number of virtual machines (VMs), a solution to this problem is ensuring that no single (malicious) VM can trigger memory compression and decompression on its own by reserving a certain amount of physical memory. Protecting against collusion by multiple malicious VMs requires linearly increasing the size of this reserved memory which reduces the benefits of memory compression for this strong threat model.

7.2 Leakage of complex bank addressing

In Section 6.2, we discussed that the compressible and incompressible regions in ZipLock must be aligned with respect to DRAM bank addresses to prevent leakage during MCR↔compressible/incompressible migrations via bank conflicts. While this alignment is straightforward under simple DRAM addressing functions (e.g., those used in gem5 and Intel CPUs), it becomes more challenging with complex functions involving multiple address bit XOR operations, such as those employed in AMD Zen CPUs [16]. In such cases, a conservative solution is to enforce a close page policy during page migrations to eliminate observable bank conflicts.

7.3 Primitive #4: *Fixed-Value Read*

Another optimization in memory compression systems that can introduce data-dependent leaks is zero-aware optimization, as we discussed in Section 2. We model this behavior as the *Zero Read* primitive and evaluate it against cryptographic libraries using LMTest. Interestingly, no secret-dependent leaks were observed in the tested programs from libsodium and libnettle. This outcome may be attributed to the granularity of the optimization where an entire block must be zero or non-zero, which appears too coarse to influence secret-dependent behavior of cryptographic code. Nonetheless, this optimization still introduces data-dependent leaks and remains insecure, violating the *uniform access* principle, which mandates disabling it in a secure memory compression system.

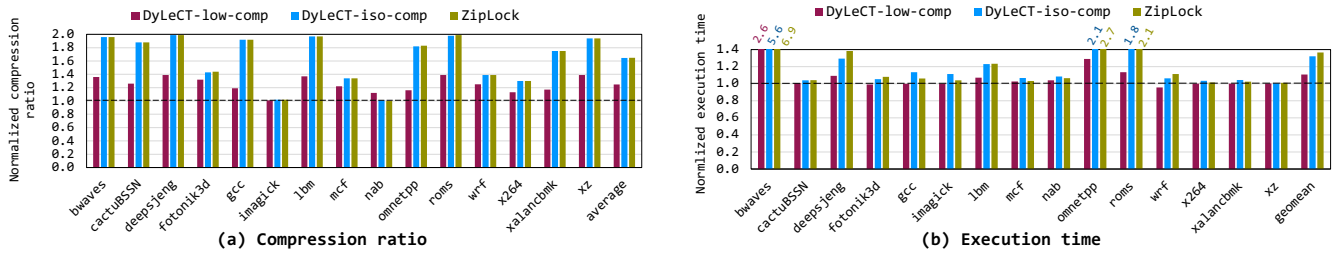


Figure 13. (a) Compression and (b) execution time of ZipLock and DyLeCT variants normalized to uncompressed baseline.

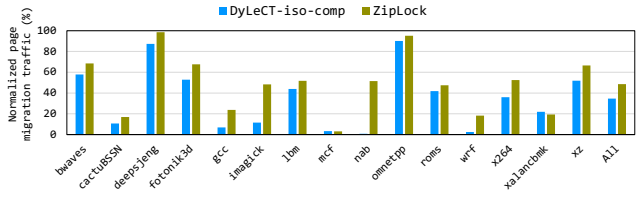


Figure 14. Memory traffic of page migrations in ZipLock and *DyLeCT-iso-comp*, normalized to their total traffic.

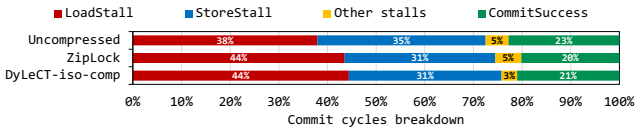


Figure 15. Commit cycles breakdown based on stall reasons in uncompressed baseline, ZipLock, and *DyLeCT-iso-comp*.

8 Related Work

We provide a comprehensive analysis of prior memory compression systems in Section 2 and Table 1.

Prior work has explored security vulnerabilities of memory compression on software level [12, 17, 31]. For example, Decomp+Time [31] exploits Linux memory compression. Such attacks are typically mitigated by constant-time programming. However, as we demonstrated in Section 4, even constant-time programs leak secrets under leakage primitives introduced by hardware-level memory compression; these leaks arise not from compression itself, but from the system designed for memory compression.

Safecracker [35] analyzes the security of cache compression, an orthogonal use case of compression; hardware-level memory compression, analyzed in this work, introduces vulnerabilities rooted in different hardware design choices, absent in cache compression. Our leakage primitives persist regardless of whether cache compression is deployed or not. In addition to identifying new vulnerabilities, this work is the first to provide formally proven design principles for secure memory compression.

9 Conclusion

We presented a rigorous security analysis of hardware-level memory compression systems, resulting in a number of novel system-level leakage primitives that expose data-dependent behavior to attackers. These primitives arise from fundamental design choices in memory compression and persist even when constant-time compression and decompression algorithms are used. To demonstrate their impact, we showed secret-dependent behaviors in widely-used cryptographic libraries that follow constant-time practices and constructed an end-to-end attack to leak passwords in a popular web server. To guide the design of secure memory compression systems, we introduced the new compression non-interference property where we formally prove it does not introduce any data-dependent leakage primitives. Leveraging this strong security property, we built ZipLock as a secure memory compression system. Based on our evaluation, ZipLock achieves a promising compression ratio with negligible performance overhead compared to the state of the art, and can serve as a secure baseline for future research toward secure and efficient memory compression designs.

Acknowledgments

We thank the anonymous reviewers and shepherd for their insightful feedback, which improved the final version of this paper. We also thank Quentin Bordier for his input on the exploits. This work has been supported by the Zurich Information Security and Privacy Center (ZISC) and the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE).

References

[1] Bulent Abali, Bart Blaner, John Reilly, Matthias Klein, Ashutosh Mishra, Craig B. Agricola, Bedri Sendir, Alper Buyuktosunoglu, Christian Jacobi, William J. Starke, Haren Myneni, and Charlie Wang. 2020. Data Compression Accelerator on IBM POWER9 and z15 Processors. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[2] Alaa Alameldeen and David Wood. 2004. *Frequent pattern compression: A significance-based compression scheme for L2 caches.* Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *USENIX Security Symposium*.

[4] Gilles Barthe, Marcel Böhme, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Marco Guarnieri, David Mateos Romero, Peter Schwabe, David Wu, and Yuval Yarom. 2024. Testing Side-Channel Security of Cryptographic Implementations against Future Microarchitectures. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[5] Matej Bolcskei, Patrick Jattke, Johannes Wikner, and Kaveh Razavi. 2025. Rubicon: Precise Microarchitectural Attacks with Page-Granular

Massaging. In *IEEE European Symposium on Security and Privacy (EuroS&P)*.

[6] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. SoK: Practical foundations for software Spectre defenses. In *IEEE Symposium on Security and Privacy (SP)*.

[7] Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher Fletcher, David Kohlbrenner, Riccardo Paccagnella, and Daniel Genkin. 2024. GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers. In *USENIX Security Symposium*.

[8] Xi Chen, Lei Yang, Robert P Dick, Li Shang, and Haris Lekatsas. 2009. C-pack: A High-Performance Microprocessor Cache Compression Algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 18, 8 (2009).

[9] Esha Choukse, Mattan Erez, and Alaa R. Alameldeen. 2018. Compresso: Pragmatic Main Memory Compression. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[10] Peter Deutsch. 1996. Rfc1951: Deflate compressed data format specification version 1.3.

[11] M. Ekman and P. Stenstrom. 2005. A Robust Main-Memory Compression Scheme. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[12] Yoel Gluck and Neal Harris. 2013. BREACH: Reviving the CRIME Attack. (2013).

[13] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *IEEE Symposium on Security and Privacy (SP)*.

[14] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-Software Contracts for Secure Speculation. In *IEEE Symposium on Security and Privacy (SP)*.

[15] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. Simpoint 3.0: Faster and More Flexible Program Phase Analysis. *Journal of Instruction Level Parallelism* 7, 4 (2005).

[16] Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Matej Bölcskei, and Kaveh Razavi. 2024. ZenHammer: Rowhammer Attacks on AMD Zen-based Platforms. In *USENIX Security Symposium*.

[17] John Kelsey. 2002. Compression and Information Leakage of Plaintext. In *Springer International Workshop on Fast Software Encryption*.

[18] Jungrae Kim, Michael Sullivan, Esha Choukse, and Mattan Erez. 2016. Bit-Plane Compression: Transforming Data for Better Compression in Many-Core Architectures. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[19] Seikwon Kim, Seonyoung Lee, Taehoon Kim, and Jaehyuk Huh. 2017. Transparent Dual Memory Compression Architecture. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[20] Muhammad Laghari, Yuqing Liu, Gagandeep Panwar, David Bears, Chandler Jearls, Raghavendra Srinivas, Esha Choukse, Kirk W Cameron, Ali R Butt, and Xun Jian. 2024. Memory Allocation Under Hardware Compression. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[21] libnettle 2025. Nettle: a low-level cryptographic library. https://www.lysator.liu.se/~nisse/nettle/nettle.html. Accessed 18-08-2025.

[22] libsodium 2025. Sodium Library. https://doc.libsodium.org/. Accessed 18-08-2025.

[23] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. 2020. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152* (2020).

[24] nginx 2026. nginx HTTP Web Server. https://nginx.org/en/. Accessed on 01-04-2026.

[25] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: the Case of AES. In *Springer Cryptographers' track at the RSA conference*.

[26] David J Palframan, Nam Sung Kim, and Mikko H Lipasti. 2015. COP: To Compress and Protect Main Memory. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[27] Gagandeep Panwar, Muhammad Laghari, David Bears, Yuqing Liu, Chandler Jearls, Esha Choukse, Kirk W. Cameron, Ali R. Butt, and Xun Jian. 2022. Translation-optimized Memory Compression for Capacity. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[28] Gagandeep Panwar, Muhammad Laghari, Esha Choukse, and Xun Jian. 2024. DyLeCT: Achieving Huge-page-like Translation Performance for Hardware-compressed Memory. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[29] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2012. Base-delta-immediate compression: practical data compression for on-chip caches. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[30] Gennady Pekhimnko, Vivek Seshadri, Yoonqu Kim, Hongyi Xin, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2013. Linearly Compressed Pages: A Low-complexity, Low-latency Main Memory Compression Framework. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[31] Martin Schwarzl, Pietro Borrello, Gururaj Saileshwar, Hanna Müller, Michael Schwarz, and Daniel Gruss. 2023. Practical Timing Side-Channel Attacks on Memory Compression. In *IEEE Symposium on Security and Privacy (SP)*.

[32] Ali Shafiee, Meysam Taassori, Rajeev Balasubramonian, and Al Davis. 2014. MemZip: Exploring Unconventional Benefits from Memory Compression. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[33] SPEC2017 2017. SPEC CPU2017 Benchmark Suite. Standard Performance Evaluation Corporation. http://www.spec.org/cpu2017/

[34] R.B. Tremaine, T.B. Smith, M. Wazlowski, D. Har, Kwok-Ken Mak, and S. Arramreddy. 2001. Pinnacle: IBM MXT in a memory controller chip. *IEEE Micro* 21, 2 (2001).

[35] Po-An Tsai, Andres Sanchez, Christopher W. Fletcher, and Daniel Sanchez. 2020. Safecracker: Leaking Secrets through Compressed Caches. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[36] Vinson Young, Sanjay Kariyappa, and Moinuddin K. Qureshi. 2019. Enabling Transparent Memory-Compression for Commodity Memory Systems. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[37] Jishen Zhao, Sheng Li, Jichuan Chang, John L. Byrne, Laura L. Ramirez, Kevin Lim, Yuan Xie, and Paolo Faraboschi. 2015. Buri: Scaling Big-Memory Computing with Hardware-Based Memory Expansion. *ACM Transactions on Architecture and Code Optimization* 12, 3 (2015).

[38] Jacob Ziv and Abraham Lempel. 2003. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23, 3 (2003).