Practical Timing Side-Channel Attacks on Memory Compression

Martin Schwarzl[1], Pietro Borrello[2], Gururaj Saileshwar[3], Hanna Müller[1], Michael Schwarz[4], Daniel Gruss[1]

[1] Graz University of Technology, [2] Sapienza University of Rome,
[3] NVIDIA Research, [4] CISPA Helmholtz Center for Information Security

2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179297

***Abstract*—Compression algorithms have side channels due to their data-dependent operations. So far, only the compression-ratio side channel was exploited, e.g., the compressed data size.**

In this paper, we present Decomp+Time, the first memory-compression attack exploiting a timing side channel in compression algorithms. While Decomp+Time affects a much broader set of applications than prior work. A key challenge is precisely crafting attacker-controlled compression payloads to enable the attack with sufficient resolution. Our evolutionary fuzzer, Comprezzor, finds effective Decomp+Time payloads that optimize latency differences such that decompression timing can even be exploited in remote attacks. Decomp+Time has a capacity of $9.73\,\mathrm{kB/s}$ locally, and $10.72\,\mathrm{bit/min}$ across the internet (14 hops). Using Comprezzor, we develop attacks that leak data bytewise in four different case studies: First, we leak $1.50\,\mathrm{bit/min}$ from Memcached on a remote PHP script. Second, we leak database records with $2.69\,\mathrm{bit/min}$, from PostgreSQL in a Python-Flask application, over the internet. Third, we leak secrets with $49.14\,\mathrm{bit/min}$ locally from ZRAM-compressed pages on Linux. Fourth, we leak internal heap pointers from the V8 engine within the Google Chrome browser on a system using ZRAM. Thus, it is important to re-evaluate the use of compression on sensitive data even if the application is only reachable via a remote interface.

I. Introduction

Data compression plays a vital role for reducing the memory and storage utilization and in file formats such as PDF, image, and video files. Similarly, operating systems (OSs) rely on memory compression [1], [2] to reduce system memory utilization. Memory compression is also used in databases [3] and key-value stores [4]. Compression can even increase performance and efficiency when storing or transferring data to slow storage devices or across networks. Hence, compression is widely used for HTTP traffic [5], [6] and file-system compression [7]. Recent trends include columnar (column-oriented) compression to reduce the disk utilization for databases [8], [9], [10], [11]. When compressing secret data, the compression ratio depends on the secret, introducing a *compression-ratio side channel*, often exploited in TLS-encrypted traffic [12], [13], [14], [15], [16], [17], [18]. All these attacks focused on web traffic and only exploited differences in the compressed size of data when compressed together with attacker-controlled data. The size of the compressed data is either accessed directly [12] or indirectly by observing the transmission time that linearly depends on the size of the compressed data [14] and, thus, the compression ratio.

Compression trades data size for computation time. However, so far, only the *result* of the compression, i.e., the compressed size, has been exploited to leak data but not the time consumed by the *process* of compression or decompression itself. First described by Kelsey et al. [19], most attacks focus on compressed web traffic. Surprisingly, security implications of compression in other settings, such as virtual memory or databases, have not been studied much. This raises two questions:

***Q1**: Are timing differences in compression and decompression exploitable if the compression ratio is unobservable?*

***Q2**: Can these timing differences be significant enough to exploit them in a fully remote setting?*

In this paper, we present Decomp+Time, the first memory-compression attack exploiting a *timing side channel in memory decompression*. We show that the *decompression time* directly leaks information about the compressed data. Our timing side channel exploits large timing differences for edge cases when decompressing nearly incompressible data. Since these edge cases require surgically crafted attacker-controlled payloads, we developed Comprezzor, an evolutionary fuzzer to generate memory layouts to trigger and amplify the edge cases. The techniques we present are generic and can be applied to various compression algorithms implementing sequence compression. We show that the Comprezzor-based payloads influence the decompression time so significantly that they can be observed remotely when the compressed data never leaves the victim system, i.e., the compression-ratio side channel is not exploitable.

We compare latency differences induced by Comprezzor-generated algorithm-specific payloads and manually crafted ones and find that Comprezzor-generated attacker payloads have latency differences up to three orders of magnitude above manually crafted layouts. We evaluate four realistic secret-leakage scenarios by generating these precise high-latency-inducing payloads. We even demonstrate remote attacks on an in-memory database system without executing code on the victim machine and without observing the victim's network traffic. Hence, our case studies show that compressing sensitive data poses a security risk in any scenario using compression and not just for web traffic.

We systematically analyze six compression algorithms, including widely-used algorithms such as DEFLATE (in zlib), PGLZ (in PostgreSQL), and zstd (by Facebook). Comprezzor

DOI 10.1109/SP46215.2023.00073

is easy to extend to new compression algorithms, and it already fully supports all of these compression algorithms. Our findings show that the decompression time not only correlates with the entropy of the uncompressed data but also with various other aspects, such as the relative position of secret data or alignment of compressible data. In general, these timing differences arise due to the design of the compression algorithm and, *importantly, also its implementation*. Our results show that all analyzed compression algorithms are susceptible to timing side channels when observing data-compression and -decompression times.

We evaluate Decomp+Time in scenarios where secret data is compressed alongside attacker-controlled data. This is a common scenario in virtual memory and also in databases where victim data and attacker-controlled data may be placed in a single cell, e.g., when storing structured data like JSON documents.[1] The attacker guesses the secret bytes while measuring the decompression time e.g., via a web request that on the server-side performs a simple read access to the data in compressed memory. We evaluate the capacity of Decomp+Time in a covert channel abusing the memory compression of Memcached, an in-memory object caching system. We can, on average, transmit $9.73\,\mathrm{kB/s}$ locally and $10.72\,\mathrm{bit/min}$ across the internet (14 hops).

We present 4 case studies leaking compressed data byte by byte: First, we attack an internet-facing PHP application using Memcached internally to leak a secret in $5.32\,\mathrm{min}$ per byte over the internet, i.e., $1.50\,\mathrm{bit/min}$. Second, we leak database records from a remote PostgreSQL instance with transparent database compression at $2.97\,\mathrm{min}$ per byte, i.e., $2.69\,\mathrm{bit/min}$. Third, we exploit ZRAM, the Linux memory compression module, transparently introducing timing side channels regardless of the security needs of the application.[2] In this setting, we leak a secret locally in $0.16\,\mathrm{min}$ per byte, i.e., $49.14\,\mathrm{bit/min}$. Fourth, we demonstrate an end-to-end exploit leaking internal heap pointers from sandboxed JavaScript inside the Chrome browser.

Our work highlights the importance of re-evaluating the use of compression on sensitive data on any layer, even if the application is only reachable via a remote interface.

Contributions. The main contributions of this work are:

1) We present a systematic analysis of timing leakage for several lossless data-compression algorithms.
2) We develop an evolutional fuzzer to find surgically precise attacker payloads to trigger extremely slow edge cases in memory decompression algorithms.
3) We demonstrate the possible leakage rate with a remote covert channel leaking $9.73\,\mathrm{kB/s}$ locally, and $10.72\,\mathrm{bit/min}$ across the internet (14 hops).
4) We leak secrets byte by byte using Memcached, PostgreSQL, and ZRAM, with leakage rates between $1.5\,\mathrm{bit/min}$ to $2.69\,\mathrm{bit/min}$ in the remote setting and $1.5\,\mathrm{kB/s}$ to $9.73\,\mathrm{kB/s}$ in the local setting.

Disclosure. We responsibly disclosed our findings to the developers, and the issues were assigned `CVE-2022-0925`. Comprezzor and parts of the attacks, as well as a demo are openly accessible[3].

II. Background and Related Work

A. Data Compression Algorithms

Lossless compression reduces the size of data without losing information. One of the most popular algorithms is the DEFLATE compression algorithm [20], which is used in gzip (zlib). The DEFLATE compression algorithm consists of two main parts, LZ77 followed by Huffman encoding. The `Lempel-Ziv` (LZ77) part scans for the longest repeating sequence within a sliding window and replaces repeated sequences with a reference to the first occurrence [21]. This reference stores distance and length of the occurrence. The `Huffman`-coding part tries to reduce the redundancy of symbols. When compressing data, DEFLATE first performs LZ77 encoding and Huffman encoding [21]. When decompressing data (inflate), they are performed in reverse order. The algorithm provides different compression levels to optimize for compression speed or compression ratio. The smallest possible sequence has a length of $3\,\mathrm{B}$ [20]. Other algorithms provide different design points for compressibility and speed. Zstd, designed by Facebook [22] for modern CPUs, improves both compression ratio and speed, and is used for compression in file systems (e.g., btrfs, squashfs) and databases (e.g., AWS Redshift, RocksDB). LZ4 and LZO are optimized for compression and decompression speed. Especially LZ4 gains its performance by using a sequence compression stage (LZ77) without the symbol encoding stage (Huffman) like in DEFLATE. FastLZ, similar to LZ4, is a fast compression algorithm implementing LZ77. PGLZ is a fast LZ-family compression algorithm used in PostgreSQL for varying-length data in the database [3].

B. Prior Data Compression Attacks

In 2002, Kelsey [19] first showed that any compression algorithm is susceptible to information leakage based on the compression-ratio side channel. Duong and Rizzo [12] applied this idea to steal web cookies with the CRIME attack by exploiting TLS compression. In the CRIME attack, the attacker adds additional sequences in the HTTP request, which act as guesses for possible cookies values, and observes the request packet length, i.e., the compression ratio of the HTTP header injected by the browser. If the guess is correct, the LZ77-part in gzip compresses the sequence, making the compression ratio higher, thus allowing the secret to be discovered. For CRIME, the attacker needs to spy on the packet length, and the secret needs a known prefix such as `cookie=`. To mitigate CRIME, TLS-level compression was disabled for requests [14], [13].

[1] Importantly, there is no indication or recommendation to not place victim and attacker-controlled data in one cell.

[2] This is particularly dangerous as users are not informed about this behavior of the OS, that introduces leakage in their applications.

[3] See https://github.com/IAIK/Memory-Compression-Attacks and https://streamable.com/qxr9h4.

The BREACH attack [13] revived the CRIME attack by attacking HTTP responses instead of requests and leaking secrets in the HTTP responses such as cross-site-request-forgery tokens. The TIME attack [14] uses the time of a response as a proxy for the compression ratio, as it can be measured even via JavaScript. To reliably amplify the signal, the attacker chooses the size of the payload such that additional bytes, due to changes in compressibility, cross a boundary and cause significantly higher delays in the round-trip time (RTT). TIME exploits the compression ratio to amplify timing differences via TCP windows and does not exploit timing differences in the underlying compression algorithm itself. Vanhoef and Van Goethem [15] showed with HEIST that HTTP/2 features can also be used to determine the size of cross-origin responses and to exploit BREACH using the information. Van Goethem et al. [17] similarly showed that compression can be exploited to determine the size of any resource in browsers. Karaskostas and Zindros [16] presented Rupture, extending BREACH attacks to web apps using block ciphers. Voracle [18] exploits compression in VPNs using similar techniques as CRIME. Tsai et al. [23] demonstrated cache timing attacks on compressed caches, leaking a secret key in under $10\,\mathrm{ms}$. **Common Theme.** Prior attacks primarily exploit the compression-ratio side channel. However, the time taken by the underlying compression algorithm is not analyzed or exploited as side channels. Additionally, these attacks largely target the HTTP traffic and website content, and do not focus on the broader use of compression such as memory compression, databases, and others, that we target in this paper.

C. Fuzzing to Discover Side Channels

Historically, fuzzing has been used to discover memory corruption bugs in applications [24], [25], [26], [27]. Typically, it involves feedback based on novelty search, executing inputs, and collecting ones that cover new program paths with the goal of triggering bugs. Some fuzzers use genetic algorithms to improve the coverage [28], [29]. Directed fuzzing guides the exploration towards specific program points that are identified as interesting [30], [31], [32]. Recently, fuzzing has also been used to discover side channels both in software and in the microarchitecture [33], [34], [35], [36]. `ct-fuzz` [37] used fuzzing to discover timing side channels in cryptographic implementations. Nilizadeh et al. [38] used differential fuzzing to detect compression-ratio side channels that enable the CRIME attack. Bang et al. [39] used symbolic execution to discover side-channel leakage for compression-ratio attacks.

III. High-level Overview

In this section, we discuss the high-level overview of memory compression attacks and the attack model.

A. Attack Model & Attack Overview

Attack Model. Most prior attacks discussed in Section II-B focus on the compression ratio side channel. Observing the compression ratio over the network requires a strong attacker, monitoring network traffic. Additionally, this information must be exposed by the system, which is typically not the case if compressed data is only handled on the remote system and not transferred to the attacker. Even the TIME attack and its variants [15], [17] only exploit timing differences due to the TCP protocol. None of these exploited or analyzed timing differences due to the compression algorithm itself, i.e., the focus of our attack. Once attacker data is compressed with the secret, the attacker measures the latency of a subsequent read access to the attacker-controlled data. As we expect system noise, when performing the experiment, we assume that the attacker can repeat the measurement multiple times. Furthermore, we assume no software vulnerabilities in the application itself. A public API provides an interface to upload, read and modify data, which is compressed and stored either in main memory or on the disk. The threat model is similar to fully remote attacks, as, presented by Schwarzl et al. [40].

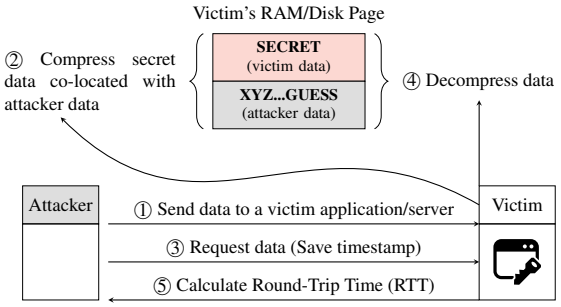

Fig. 1: Overview of a memory compression attack exploiting a timing side channel.

Data Co-location. We assume that the attacker can co-locate data with secret data. This assumption is in line with all previous memory compression attacks [12], [13], [14], [15], [17], [16]. For HTTP requests/responses, the attacker was able to arbitrarily co-locate guesses of the cookie value, e.g., `Known Data (Prefix) || Secret Data || Attacker-controlled data`. Moreover, co-location can also occur in the other direction with a known suffix or direct co-location of attacker-controlled and secret data e.g., `Secret Data || Known Data (Suffix) || Attacker-controlled data`.

In applications, co-location is possible not only in HTTP requests, but also via a memory storage API like Memcached, with a shared database between attacker and victim that compresses multiple rows or columns. For cellular compression, co-location might occur in JSON fields storing data from different origins. Moreover, co-location can occur directly in virtual memory. For instance, pointers can be co-located with other attacker-controlled data structures (on the heap) and compressed by the operating system. In such a setting, potential targets are internal malloc pointers to libc functions for breaking ASLR or internal pointers to metadata in JavaScript engines. We present four case studies, where co-location leads to data leakage in commonly used software in Section VI.

Attack Overview. Figure 1 illustrates an overview of a memory compression attack in five steps. The victim application

can be a web server with a database or software cache, or a filesystem that compresses stored files. **First**, the attacker sends its data to be stored to the victim's application. **Second**, the victim application compresses the attacker-controlled data, together with some co-located secret data, and stores the compressed data. The attacker-controlled data contains a partial guess of the co-located victim's data `SECRET` or, in the case where a prefix or suffix is known, `prefix=SECRET`. The guess can be performed bytewise to reduce the guessing entropy. If the partial guess (e.g., SECR) is correct, the compressed data not only has a higher compression ratio, but it also influences the decompression time. **Third**, after the compression happened, the attacker requests the content of the stored data again and takes a timestamp. **Fourth**, the victim application decompresses the attacker-controlled input together with the secret data and acknowledges the request. **Fifth**, the attacker takes another timestamp when the application responds and computes the RTT as the difference between the two timestamps. Based on the RTT, which depends on the decompression latency of the algorithm, the attacker infers the correct guess and leaks the secret data. Thus, the attack relies on the *timing differences* of the compression algorithm itself, which we characterize next.

IV. Systematic Study: Compression Algorithms

In this section, we provide a systematic analysis of timing leakage in compression algorithms. We choose six popular compression algorithms (zlib, zstd, LZ4, LZO, PGLZ, and FastLZ), and evaluate compression and decompression times based on the input data entropy. Zlib, implementing the DEFLATE algorithm, is used, e.g., for compressing files and in gzip. Zstd is Facebook's alternative to Zlib. PGLZ is used in PostgreSQL. LZ4, FastLZ, and LZO were built to increase compression speeds. For each algorithm, we see timing differences in the range of hundreds to thousands of nanoseconds depending on the input data.

A. Experimental Setup

We conducted the experiments on an Intel i7-6700K (Ubuntu 20.04, kernel 5.4.0) with a fixed frequency of $4\,\mathrm{GHz}$. We evaluate the latency of each compression algorithm with three different input values, each $4\,\mathrm{kB}$ in size. The first input is the same byte repeated 4096 times, which should be *fully compressible*. The second input is *partly compressible* and a hybrid of two other inputs: half random bytes and half compressible repeated bytes. The third input consists of random bytes which are theoretically *incompressible*. With these, we show that compression algorithms have different timings depending on the compressibility of the input.

B. Timing Differences for Different Inputs

For each algorithm and input, we measure the decompression and compression time of a $4\,\mathrm{kB}$ data blob over $100\,000$ repetitions and compute the mean values and standard deviations.

Decompression. Table I lists the decompression latencies for all evaluated compression algorithms. Depending on the entropy of the input data, there is considerable variation in the decompression time. All algorithms incur a higher latency for decompressing a fully compressible page compared to an incompressible page, leading to a timing difference of few hundred to few thousand nanoseconds for different algorithms. This is because, for incompressible data, algorithms can augment the raw data with additional metadata to identify such cases and perform simple memory copy operations to "decompress" the data, as is the case for zlib where the decompression for an incompressible page is $5375.93\,\mathrm{ns}$ faster than for a fully-compressible page. For decompression of partially-compressible pages, some algorithms (FastLZ, LZ4, LZO, zstd) lead to lower latencies compared to fully-compressible pages. Zlib and PGLZ lead to a higher decompression latency for partially-compressible pages compared to fully-compressible pages. This shows the existence of even algorithm-specific variations in timings. PGLZ does not create compressible memory in the case of an incompressible input, and hence we do not measure its latency for this input.

TABLE I: Different compression algorithms yield distinguishable timing differences when decompressing $4\,\mathrm{kB}$ content with a different entropy ($n = 100000$).

Algorithm	Fully Compressible (ns)	Partially Compressible (ns)	Incompressible (ns)
FastLZ	7257.88 (±0.23%)	4264.56 (±2.27%)	1155.57 (±0.92%)
LZ4	605.79 (±1.02%)	218.68 (±1.76%)	107.90 (±2.49%)
LZO	2115.65 (±2.05%)	1220.07 (±3.64%)	309.44 (±6.27%)
PGLZ	813.75 (±0.71%)	5340.47 (±0.38%)	-
zlib	7016.02 (±0.33%)	13 212.53 (±0.35%)	1640.09 (±1.51%)
zstd	941.05 (±0.94%)	772.55 (±0.77%)	370.59 (±2.87%)

Compression. For compression, we observed a trend in the other direction (Table IV in Appendix B lists compression latencies for different algorithms). For different levels of compressibility, there are also latencies between the three different inputs, which are clearly distinguishable in the order of multiple hundreds to thousands of nanoseconds. Thus, timing side channels from compression might also be used to exploit compression of attacker-controlled memory co-located with secret memory. However, attacks using the compression side channel might be harder to perform in practice as the compression of data might be performed in a separate task (in the background), and the latency is, therefore, not easily observable for an attacker. Hence, our work focuses on attacks exploiting the decompression timing side channel.

Handling of Corner Cases. For incompressible pages, the "compressed" data can be larger than the original size with the additional compression metadata. Additionally, it is slower to access after compression than raw uncompressed data. Hence, this corner-case with incompressible data may be handled in an implementation-specific manner, which can itself lead to additional side channels. For example, a threshold for the compression ratio can decide when a page is stored in a raw format or in a compressed state, like in Memcached-PHP [4]. PGLZ, the algorithm used in PostgreSQL database, which computes the maximum acceptable output size for input by

checking the input size and the strategy compression rate, could fail to compress inputs in such corner cases.

In Section VI, we show how real-world applications like Memcached, PostgreSQL, and ZRAM deal with such corner cases and demonstrate attacks on each of them.

C. Leaking Secrets via Timing Side Channels

Thus far, we analyzed timing differences for decompressing different inputs, which in itself is not a security issue. In this section, we demonstrate Decomp+Time to leak secrets from compressed pages using these timing differences. We focus on sequence compression, i.e., LZ77 in DEFLATE.

1) Building Blocks for Decomp+Time

Decomp+Time has 3 building blocks: *sequence compression* to modulate the compressibility of an input, *co-location* of attacker data and secrets, and *timing variation for decompression* depending on the change in compressibility of the input.

Sequence compression: Sequence compression i.e., LZ77 tries to reduce the redundancy of repeated sequences in an input by replacing each occurrence with a pointer to the first occurrence. This results in a higher compression ratio if redundant sequences are present in the input and a lower ratio if no such sequences are present. This compressibility side channel can leak information about the compressed data.

Co-location of attacker data and secrets: If the attacker can control a part of data that is compressed with a secret, as described in Figure 1, then the attacker can place a *guess* about the secret and place it co-located with the secret to exploit sequence compression. If the compression ratio increases, the attacker can infer if the guess matches the secret or not. While the CRIME attack [12] previously used a similar set up and observed the compressed size of HTTP requests to steal secrets like HTTP cookies, we introduce a more general attack that does not require observability of compressed sizes.

Timing Variation in Decompression: We infer the change in compressibility via its influence on the decompression timing. We observe that even sequence compression can cause variation in the decompression timing based on compressibility of inputs (for all algorithms in Section IV-B). If the sequence compression reduces redundant symbols in the input and increases the compression ratio, we observe faster decompression due to fewer symbols. Otherwise, with a lower compression ratio and more symbols, decompression is slower. Hence, the attacker can infer the compressibility changes for different guesses by observing differences in decompression time. For a correct guess, the guess and the secret are compressed together and the decompression is faster due to fewer symbols. For incorrect guesses with more symbols it is slower.

2) Launching Decomp+Time

Using the building blocks described above, we set up the attack with an artificial victim program that has a 6 B secret string (`SECRET`) embedded into a 4 kB page. The page also contains attacker-controlled data that is compressed together with the secret, like the scenario shown in Figure 1. The attacker can update its own data in place to make multiple guesses. The attacker can also read this data, which triggers a decompression of the page and allows the attacker to measure the decompression time. A correct guess that matches the secret results in faster decompression.

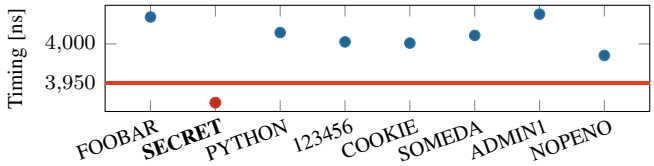

Fig. 2: Decompression time with Decomp+Time for different guesses of the secret value. A threshold (line) separating the correct from wrong secrets.

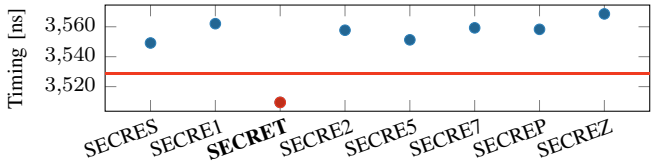

Fig. 3: Bytewise-leakage of the secret's last byte. A threshold (line) separates the correct from the wrong guess.

We perform the attack on the zlib library (1.2.11) and use 8 different guesses, including the correct guess. For each guess, a single string is placed 512 B away from the secret value; Note that this offset is arbitrarily chosen, and other offsets also work. Other data in the page is initialized with dummy values (repeated number sequence from 0 to 16). To measure the execution time, we use the `rdtsc` instruction.

Evaluation. Our evaluation was performed on an Intel i7-6700K (Ubuntu 20.04, kernel 5.4.0) with a fixed frequency of 4 GHz. To get stable results, we repeat the decompression step with each guess 10 000 times and repeat the entire attack 100 times. For each guess, we take the minimum timing difference per guess and choose the global minimum timing difference to determine the correct guess. Figure 2 illustrates the minimum decompression times. With zlib, we see that the correct guess is faster on average by 71.5 ns ($n = 100, \sigma_{\bar{\mu}} = 199.55\%$) compared to the second-fastest guess. Our attack correctly guessed the secret in all 100 repetitions of the attack. While we used a 6 B secret, our experiment also works for smaller secrets down to a length of 4 B.

Bytewise Leakage. If the attacker manages to guess or know the first three bytes of the secret, the subsequent bytes can even be leaked bytewise using our attack. Both CRIME and BREACH assume a known prefix such as `cookie=`. Similar to CRIME and BREACH [12], [13], [16], we try to perform a bytewise attack by modifying our simple layout. We use the first 5 characters of $SECRET$ as a prefix (`"SECRE"`) and guess the last byte with 7 different guesses. On average, the latency is 28.37 ns ($n = 100, \sigma_{\bar{\mu}} = 186.61\%$), between the secret and second fastest guess. Figure 3 illustrates the minimum decompression times for the different guesses. However, we

observe an error rate of 8 % for this experiment, which might be caused by the Huffmann-decoding part in DEFLATE.

While techniques like the Two-Tries method [12], [13], [16] have been proposed to overcome the effects of Huffman-coding in DEFLATE to improve the fidelity of bytewise attacks exploiting compression ratio, we seek to explore whether bytewise leakage can be reliably performed via the timing only by amplifying the timing differences.

3) Challenge of Amplifying Timing

While the decompression timing side channels can be used in attacks, the timing differences are quite small for practical exploits on real-world applications. For example, the timing differences we observe for the correct guess are in tens of nanoseconds, while most practical use cases of compression, like a Memcached server accessed over the network or PostgreSQL database accessed from a disk, could have access latencies of milliseconds.

Amplification. To enable memory compression attacks even via the network, we need to amplify the timing difference between correct and incorrect guesses. However, it is impractical to manually identify inputs that could amplify the timing differences, as each compression algorithm has a different implementation that is often highly optimized. Moreover, various input parameters could influence the timing of decompression, such as frequency of sequences, alignments of the secret and attack-controlled data, size of the input, entropy of the input, and different compression levels provided by algorithms. We develop an evolutionary fuzzer, Comprezzor, to automatically find inputs that amplify the timing difference between correct and incorrect guesses for compression algorithms.

V. Evolutionary Compression-Time Fuzzer

Compression algorithms are highly optimized and complex. Hence, we introduce Comprezzor, an evolutionary fuzzer to discover attacker-controlled inputs for compression algorithms that maximize differences in decompression times for certain guesses enabling bytewise leakage.The motivation for this automated tool is that there are too many possibilities for crafting efficient payloads manually. Our manual attempts only result in minimal timing differences that are difficult to exploit.

Comprezzor empowers genetic algorithms to amplify decompression side channels. It treats the decompression process of a compression algorithm as an opaque box and mutates inputs to the compression while trying to maximize the output, i.e., timing differences for decompression with different guesses. The mutation process in Comprezzor focuses on the entropy of data and memory layout and alignment that end up triggering optimizations and slow paths. Figure 4 illustrates a high-level overview of the steps Comprezzor performs.

While previous approaches used fuzzing to detect timing side channels [38], [37], Comprezzor can dramatically amplify timing differences by being specialized for compression algorithms by varying parameters like the input size, layout, and entropy that affect the decompression time. The inputs discovered by Comprezzor can amplify timing differences to such an extent that they are even observable remotely.

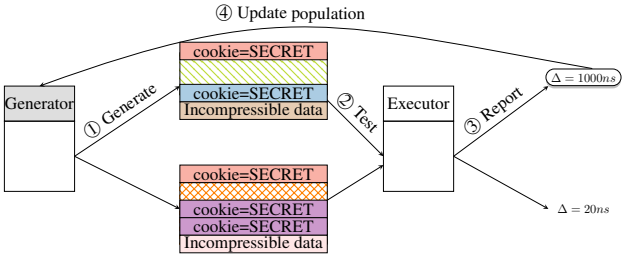

Fig. 4: Comprezzor generates memory layouts with different entropy, input size, and secret alignment, leading to high-latency decompression times. Every iteration, samples with the highest latency difference are used as input to generate newer layouts.

A. Design of Comprezzor

In this section, we describe the key parts of our fuzzer: Input Generation, Fitness Function, Input Mutation and Evolution.

Input Generation. Comprezzor generates memory layouts for Decomp+Time by maximizing the timing differences on decompression of the correct guess compared to incorrect ones. Comprezzor creates layouts with sizes in the range of 1 kB to 64 kB. It uses a helper program that takes the memory layout configuration as input, builds the requested memory layout for each guess, compresses them using the target compression algorithm, and reports the observed timing differences in the decompression times among the guesses. A memory layout configuration is composed of a file to start from, the offset of the secret in the file, the offset of guesses, how often the guesses are repeated in the layout, the compression level (i.e., between 1 and 9 for zlib), and a modulus for entropy reduction that reduces the range of the random values. The fuzzer can be used in cases where a prefix or suffix is known and unknown.

Fitness Function. The evolutionary algorithm of Comprezzor starts from a random population of candidate layouts (samples) and takes as feedback the difference in time between decompression of the generated memory containing the correct guess and the incorrect ones. Comprezzor uses the timing difference between the correct guess and the second-fastest guess as the fitness score for a candidate. The fitness function is evaluated using a helper program performing an attack on the same setup as in Section IV-A. The program performs 100 iterations per guess and reports the minimum decompression time per guess to reduce the impact of noise. This minimum decompression time is the output of the fitness function for Comprezzor.

Input Mutation. Comprezzor is able to amplify timing differences thanks to its set of mutations over the samples space specifically designed for data compression algorithms. Data compression algorithms leverage input patterns and entropy to shrink the input into a compressed form. For performance reasons, their ability to search for patterns in the input is limited by different internal parameters, like lookback windows, look-ahead buffers, and history table sizes [3], [21]. We designed the mutations that affect the sample generation process to

focus on input characteristics that directly impact compression algorithm strategies and limitations towards corner cases.

Comprezzor mutations randomize the entropy and size of the samples that are generated. This has an effect on the overall compressibility of sequences and literals in the sample [21]. Moreover, the mutator varies the number of repeated guesses and their position in the resulting sample, stressing the capability of the compression algorithm to find redundant sequences over different parts of the input. This affects the sequence compression and triggers corner cases, e.g., subsequent blocks to be compressed are directly marked as incompressible (cf. Section V-B). All these factors contribute to Comprezzor's ability to amplify timing differences.

Input Evolution. Comprezzor follows an evolutionary approach to generate inputs that maximize timing differences. It generates and mutates candidate layout configurations for the attack. Each configuration is forwarded to the helper program that builds the requested layout, inserts the candidate guess, compresses the memory, and returns the decompression time.

Comprezzor iterates through different generations, with each sample having a probability of survival to the new generation that depends on its fitness score. The fitness score is the time difference between the correct guess and the nearest incorrect one. Comprezzor discards all the samples where the correct guess is not the fastest or slowest. A retention factor decides the percentage of samples selected to survive among the best ones in the old generation ($5\,\%$ by default). The population for each new generation is initialized with the samples that survived the selection and enhanced by random mutations. By default, $70\,\%$ of the new population is generated by mutating the best samples from the previous generation. To avoid locally optimal solutions, a percentage of completely random new samples is injected in each new generation. Comprezzor runs until the maximum number of generations is evaluated, and returns the best candidate layouts.

B. Results: Fuzzing Compression Algorithms

Evaluation. Our test system has an Intel i7-6700K (Ubuntu 20.04, kernel 5.4.0) with a fixed frequency of $4\,\mathrm{GHz}$. We run Comprezzor on four compression algorithms: zlib (1.2.11), Facebook's Zstd (1.5.0), LZ4 (v1.9.3), and PGLZ in PostgreSQL (v12.7). Comprezzor can support new algorithms by just adding compression and decompression functions.

We run Comprezzor with 50 epochs, 1000 samples each, and a retention factor of $5\,\%$, selecting the best 50 samples in each generation. We randomly mutate the selected samples to generate $70\,\%$ of the children and add $25\,\%$ of randomly generated layouts to the new generation. The overall runtime of Comprezzor was $2.46\,\mathrm{h}$ for zlib, $1.73\,\mathrm{h}$ for zstd, $1.64\,\mathrm{h}$ for LZ4, and $2.09\,\mathrm{h}$ for PGLZ. Table III (Appendix A) lists the maximum timing differences found for the four compression algorithms. Particularly, for zlib and PGLZ, the fuzzer discovers cases with timing differences of multiple microseconds between correct and incorrect guesses.

Zlib. Comprezzor discovers a corner case in zlib where all incorrect guesses lead to a slow code path, and the correct guess leads to a significantly faster execution time. Using Comprezzor with a known prefix, we observe a high timing difference of $71\,514.75\,\mathrm{ns}$, which is **3** orders of magnitude larger than the manually-discovered latency difference (cf. Section IV-C). This memory layout also leads to similarly high timing differences across all compression levels of zlib. To rule out microarchitectural effects, we confirm the experiment on different systems with an Intel i5-8250U, AMD Ryzen Threadripper 1920X, and Intel Xeon Silver 4208.

On further analysis, we observe that the corner case identified by the fuzzer is due to incompressible data. The initial data in the page, from a uniform distribution, is primarily incompressible. For such incompressible blocks, DEFLATE can store them as raw data blocks, called *stored blocks* [20]. Such blocks have fast decompression times as only a single `memcpy` operation is needed on decompression instead of the actual DEFLATE process. In this particular corner case, the correct guess results in such an incompressible *stored block* which is faster, while an incorrect guess results in a partly-compressible input which is slower.

Correct Guess. In the case where the guess matches the secret, the entire guess string, i.e., `cookie=SECRET`, is compressed with the secret string. All subsequent data in the input is incompressible and treated as a stored block and decompressed with a single `memcpy` operation, which is significantly faster than Huffman and LZ77 decoding.

Incorrect Guess. In the compression case where the guess does not match the secret, only the prefix of the guess, i.e., `cookie=`, is compressed with the prefix of the secret, while another longer sequence, i.e., `cookie=FOOBAR` leads to forming a new block. Therefore, when decompressing, this block must now undergo the Huffman decoding (and LZ77), which results in several table lookups, memory accesses, and higher latency. Thus, the timing differences for the correct and incorrect guesses are amplified by the layout that Comprezzor discovered. We provide more details about this layout in Figure 10 in the Appendix D and also provide listings of the debug trace from zlib for the decompression with the correct and incorrect guesses, to illustrate the root-cause of the amplified timing differences with this layout.

Larger Secret Sizes. Evaluating a larger secret size ($1\,\mathrm{kB}$ random string) with Comprezzor on zlib, results in similar high timing differences in the range of tens of microseconds for the correct guess using a byte-by-byte attack.

Takeaway We showed that it is possible to amplify timing differences for decompression timing attacks (answers **Q1**). With Comprezzor, we presented an approach to automatically find high timing differences in compression algorithms.

VI. Case Studies

In this section, we present case studies showing the security impact of the timing side channel. We present a local covert channel leveraging the high-latency scenarios found by Comprezzor. Furthermore, we present a remote covert channel that exploits the decompression of memory objects in Memcached.

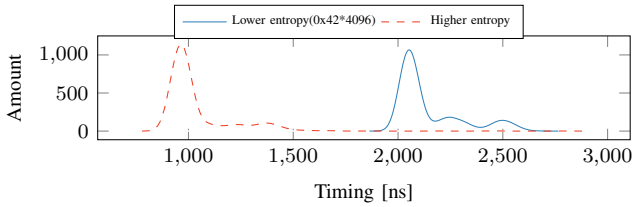

Fig. 5: Timing when decompressing a zlib-compressed 4 kB page with high entropy compared to a page with low entropy.

We demonstrate Decomp+Time on a PHP application that compresses secret data together with attacker data to leak the secret bytewise. We leak inaccessible values from a database, exploiting PostgreSQL's internal compression. We show that OS-based memory compression (ZRAM) also has timing side channels that can leak secrets. In these case studies, we do not artificially restrict the possible layouts of the pages, since these are possible ways in which those systems may be used, as confirmed by their developers during responsible disclosure. In our fourth study, on Chrome, we create co-location between attacker-controlled data and internal heap pointers and exploit ZRAM compression to leak the internal pointer.

A. Covert channel

To evaluate the transmission capacity of memory-compression attacks, we evaluate the transmission rate for a covert channel, where the attacker controls the sending and receiving end. Similar to previous works [41], [42], [43], [44], [45], we evaluate a cross-core covert channel using shared memory. The maximum capacity poses a leakage rate limit for our other attacks. Our local covert channel achieves a capacity of 9.73 kB/s ($n = 100$, $\sigma_{\bar{\mu}} = 0.00097\%$).

Setup. We create a simple key-value store that communicates via UNIX sockets. The store takes input from a client and stores it on a 4 kB-aligned page. The sender inserts a key and value into the first page to communicate with the server. The receiver inserts a small key and value as well, placed on the same 4 kB page. If the 4 kB-page is full, the key-value store compresses the whole page. Compressing full 4 kB-page separately also occurs on filesystems like BTRFS [7].

Sender and receiver agree on a time frame to send and read content. The basic idea is to communicate via the observation on zlib that memory with low entropy, e.g., 4096 times the same value, requires more time when decompressing compared to pages with a higher entropy, e.g., repeating sequence number from 0 to 255. Note that the content of the page controlled by the receiver is co-located to the senders controlled part. Figure 5 shows the decompression latency histogram for both cases for the key-value on an Intel i7-6700K running at 4 GHz. On average, we observe a timing difference of 3566.22 ns (14 264.88 cycles, $n = 100000$).

Transmission. We evaluate our cross-core covert channel by generating and sending random content from `/dev/urandom` through the memory compression timing side channel. The sender controls 4095 B of a 4 kB page. The sender transmits a '1'-bit by performing a store with high-entropy data. Conversely, to transmit a '0'-bit, the sender stores a low-entropy data. To trigger the compression, the receiver also stores data in the store which fills a full 4 kB page, which the key-value store then compresses. The receiver performs a fetch request from the key-value store, which triggers a decompression of the full 4 kB page. To distinguish bits, the receiver measures the mean RTT of the fetch request.

Evaluation. Our test machine has an Intel Core i7-6700K (Ubuntu 20.04, kernel 5.4.0) with all cores running at 4 GHz. We repeat the transmission 50 times and send 640 B per run. To reduce the error rate, the receiver fetches the receiver-controlled data 50 times and compares the average response time against the threshold. Our cross-core covert channel achieves an average transmission rate of 9.73 kB/s ($n = 100$, $\sigma_{\bar{\mu}} = 0.0068\%$) with an error rate of 0.082 % ($n = 100$, $\sigma_{\bar{\mu}} = 0.023\%$). The capacity of the unoptimized covert channel is in line with other state-of-the-art microarchitectural cross-core covert channels that do not rely on shared memory [46], [47], [48], [49], [50], [51], [52], [33].

B. Remote Covert Channel

We extend the scope of our covert channel to a remote covert channel. In the remote scenario, we rely on Memcached on a web server for memory compression and decompression.

Memcached is a simple key-value store, widely used for web site caching [53]. Internally, it uses a slab allocator with a fixed unit of contiguous physical memory assigned to a certain slab class which is typically a 1 MB region [54]. PHP offers the possibility to use Memcached for caching, and memory compression is enabled by default if Memcached is used [4]. PHP-Memcached has a threshold that decides at which size data is compressed, with the default value being 2000 B. Furthermore, PHP-Memcached compares the compression ratio to a compression factor and decides whether it stores the data compressed or uncompressed in Memcached. By default, the compression factor is 1.3, i.e., it is only compressed if the size would be reduced by 23 % or more [4].

Bypassing the Compression Factor. While the compression factor already introduces a timing side channel, we focus on scenarios where data is always compressed. This is used in Section VI-C useful for leaking co-located data. Intuitively, it should suffice to prepend highly-compressible data to enforce compression. However, we found that only prepending and adopting the offsets for secret repetitions, as for zlib, also influenced the corner case we found and the large timing difference. We integrate prepending of compressible pages to Comprezzor and also add the compression factor constraint to automatically discover inputs that fulfills the constraint and leads to large latencies between a correct and incorrect guess.

Transmission. We use the page found by Comprezzor that triggers a significantly lower decompression time to encode a '1'-bit. For a '0'-bit, we choose content that triggers a significantly higher decompression time. The sender places a key-value pair for each bit index at once into PHP-Memcached. The receiver sends GET requests to the resource, causing

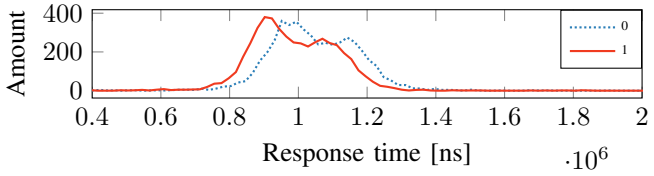

Fig. 6: Distribution of HTTP response times for zlib-decompressed pages stored in Memcached on memory compression encoding a '1' and a '0'-bit.

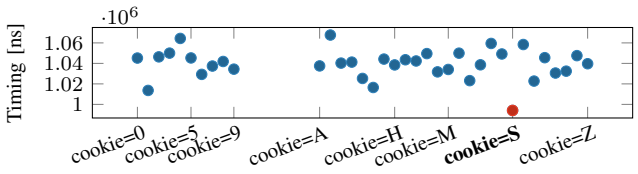

Fig. 7: Response times for the correct byte guess (S) and the incorrect guesses (0-9, A-R, T-Z) leaked from PHP-Memcached. Subsequent bytes are similar (cf. Appendix E). Standard error margins are below 1 % of the value for all guesses.

decompression of the data containing the sender content. The timing difference of the decompression is reflected in the RTT of the HTTP request. Hence, we measure the timing difference between the sent HTTP request and the first received response.

Evaluation. Our sender and receiver use an Intel i7-6700K (Ubuntu 20.04, kernel 5.4.0) and connect to the internet with a 10 Gbit/s connection. For the web server, we use a dedicated server in the Equinix [55] cloud, 14 hops away from our network (over 700 miles physical distance) with a 10 Gbit/s connection. The victim server uses an Intel Xeon E3-1240 v5 (Ubuntu 20.04, kernel 5.4.0). Our server runs Nginx 1.18.0, with a PHP (version 7.4, FPM enabled) website that allows storing and retrieving data, backed by Memcached 1.5.22, the default version on Ubuntu 20.04. We perform a simple test where we perform 5000 HTTP requests to a PHP site that stores zlib-compressed memory in Memcached. Figure 6 illustrates the timing difference between a '0'-bit and a '1'-bit. The timing difference between the mean values for a '0'- and '1'-bit is 61 622.042 ns. We transmit a series of random messages of 8 B over the internet. Our simple remote covert channel achieves an average transmission rate of 10.72 bit/min ($n = 20$, $\sigma_{\bar{\mu}} = 15.96\%$) at an average error rate of 0.93 %. We achieve a similar transmission rate as Schwarzl et al. [40] with remote memory-deduplication attacks. Our covert channel outperforms the one by Schwarz et al. [56] and Gruss et al. [57], even though our attack works with HTTP instead of the more lightweight UDP sockets. Other remote timing attacks usually do not evaluate their capacity with a remote covert channel [58], [59], [60], [61], [62], [60], [63]. Note that our numbers to mount a successful covert channel over such a distance is way below the numbers reported by Van Goethem et al. [63, Table 1].

C. Remote Attack on PHP-Memcached

Using our building blocks to perform Decomp+Time and the remote covert channel, we perform a remote attack on PHP-Memcached to leak secret data from a server over the internet. We assume a memory layout where secret memory is co-located to attacker-controlled memory, and the overall memory region is compressed. As mentioned in Figure 1, we assume that the attacks can arbitrarily co-locate arbitrary data to secret data. This is reasonable as the developer might store additional metadata, e.g., API keys co-located to the attacker-controlled data, or might make some modifications. Also, structured document data might be cached within the same memory slab as the API of Memcached does not prevent an user from merging data of multiple users together. The compressed data is never visible to the user, and the compression ratio is not exposed, relying on the compression ratio is not possible.

Attack Setup. We use the same setup as in Section VI-B and run the attack using the same server setup as used for the remote covert channel. We define a 6 B long secret (`SECRET`) with a 7 B long prefix (`cookie=`) and prepend it to the stored data of users. PHP-Memcached compresses the data before storing it in Memcached and decompress it when accessing it again. For each guess, the PHP application stores the uploaded data to a certain location in Memcached. On each data fetch, the PHP application decompresses the secret data together with the co-located attacker-controlled data and then responds only the attacker-controlled data. The attacker measures the RTTs and discerns the timing differences between the guesses.

Evaluation. For the bytewise attack, we assume each byte of the secret is uppercase alphanumeric (36 different options). For each of the bytes to be leaked, we generate separate memory layouts using Comprezzor that maximize the latency between guesses. We repeat the experiment 20 times. On average, our attack leaks the entire secret string in 31.95 min ($n = 20$, $\sigma_{\bar{\mu}} = 60.58\%$) i.e., 5.32 minutes per byte or 1.5 bit/min. Since the latencies between a correct and incorrect guess are in the microseconds range, we do not observe false positives with our approach. Figure 7 shows the median response time for each guess in the first iteration as a representative example. It can be seen that the response time for the correct guess is significantly faster than the incorrect guesses.

Takeaway: We show that a PHP application using Memcached to cache blobs hosted on Nginx enables covert communication with a transmission rate of 10.72 bit/min (answers **Q2**). Moreover, we demonstrate a remote memory-compression attack on Memcached leaking 1.5 bit/min.

D. Leaking Data from Compressed Databases

In this section, we show that an attacker can exploit compression in databases to leak inaccessible information from the internal database compression of PostgreSQL. In this setting, the compression ratio is not visible to the attacker, only the timing can be observed. A potential attack scenario for structured text in a cell is where JSON documents are

stored and compressed within a single cell, and the attacker controls a specific field within the document. While our focus is restricted to cell-level compression, compressed columnar storage [8], [9], [10], [11] or columnar databases, may also be vulnerable to decompression timing attacks. The attacker controls data in the same cell or the content of a cell in the same column as the target data.

PostgreSQL Data Compression. PostgreSQL is a widespread open-source relational database system using the SQL standard. PostgreSQL maintains tuples saved on disk using a fixed page size of commonly 8 kB, storing larger fields compressed and possibly split into multiple pages. By default, variable-length fields that may produce large values, e.g., `TEXT` fields, are stored compressed. PostgreSQL's transparent compression is known as `TOAST` (The Oversized-Attribute Storage Technique) and uses a fast LZ-family compression, `PGLZ` [3]. Data in a cell is stored compressed if such a form saves at least 25 % of the uncompressed size to avoid wasting decompression time. Data stored uncompressed is accessed faster than data stored compressed as the decompression algorithm is not executed.

Attack Setup. To assess the feasibility of an attack, we use a local database server with the database stored on an SSD and access two differently compressed rows with a Python wrapper using the `psycopg2` library. The first row contains 8192 characters of highly compressible data, while the second one 8192 characters of random incompressible data. Both rows are stored in a table as `TEXT` data and accessed 1000 times. The median for the number of clock cycles required to access the compressible row is 249 031, while for the uncompressed one is 221 000, which makes the two accesses distinguishable. On our 4 GHz CPUs, this is a timing difference of 7007.75 ns. We use Comprezzor to amplify these timing differences and demonstrate bytewise leakage.

Leaking First Byte. For the bytewise leakage of the secret, we first create a memory layout to leak the first byte using Comprezzor against a standalone version of PostgreSQL's compression library, using a similar setup as the previous Memcached attack. A key difference in the use of Comprezzor with PostgreSQL is that the helper program measuring the decompression time returns a time of 0 when the input is not compressed, i.e., the data compressed with PGLZ does not save at least 25 % of the original size. Comprezzor found a layout that sits exactly at the corner case where a correct guess in the secret results in a compressed size that saves 25 % of the original size. Hence, a correct guess is saved compressed, while for any wrong guess, the data is saved uncompressed.

Leaking Subsequent Bytes with Secret Shifting. We observed that one good layout can be reused for bytewise leakage in PGLZ. The prefix can be shifted by one character to the left by a single character, i.e., from "cookie=S" to "ookie=SE", to accommodate an additional byte for the guess. Shifting allows bytewise leakage with the same memory layout. Note that we could not mount this shifting approach on DEFLATE.

Evaluation. We perform a remote decompression timing attack against a Flask [64] web server that uses a PostgreSQL database to store user-provided data. We used the same Equinix cloud-server setup as used for the Memcached remote attack (cf. Section VI-C). The server runs Python 3.8.5 with Flask 2.0.1 and PostgreSQL 12.7. We use a similar setup as in Section VI-C with the difference that attacker-controlled data is co-located to a secret in a database cell. The secret is never shown to the user. Using the layout found by Comprezzor, the entry in the database is stored compressed only when the secret matches the provided data. A second endpoint in the server accesses the database to read the data without returning the secret to the attacker. The attack leaks bytewise over the internet by guessing again uppercase alphanumeric characters (36 possibilities per character), including the correct one. We repeat the attack 20 times. The average time for the attack, i.e., the time required to determine the guess with the highest latency that the server had to decompress before returning, is 17.84 min (2.97 min/B) ($n = 20$, $\sigma_{\bar{\mu}} = 0.33\%$) i.e., 2.69 bit/min. Figure 8 illustrates the median response times showing how the correct guess results in a slower response. Without fixing the CPU frequency, twice as many requests are required to clearly determine the correct secret. However, keeping the server busy automatically leads to an almost constant frequency. We guess over the set of all printable characters and observe one secret byte in 7.83 min.

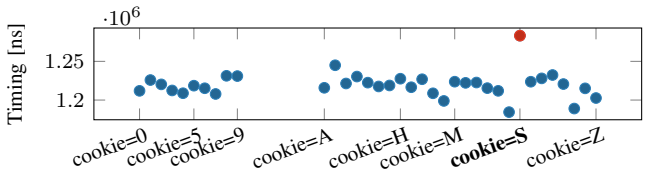

Fig. 8: Distribution of response times for the bytewise leakage in the remote PostgreSQL attack, with the correct guess (S) and incorrect guesses (0-9, A-R, T-Z). This is similar for subsequent bytes leaked (cf. Appendix E). The standard error is below 1 % for all guesses.

Takeaway: Secrets can be leaked from databases due to timing differences caused by PostgreSQL's transparent compression, if applications store untrusted data with secrets in the same cell. Our decompression timing attack on PostgreSQL leaks a byte across the internet with 2.69 bit/min.

E. Attacking OS Memory Compression

In this section, we show how memory compression in modern OSs can introduce exploitable timing differences. We demonstrate bytewise leakage of secrets from compressed pages in ZRAM, the Linux implementation of memory compression. Co-location can be achieved here if virtual memory is compressed together with mostly attacker-controlled data. As we show in Section VI-F, co-location can occur for V8-internal pointers, together with attacker-controlled data. The compression ratio is not observable for the attacker since the attacker cannot read the compressed memory from ZRAM.

Background. Memory compression is a technique used in many modern OSs, e.g., Linux [65], Windows [66], or MacOS [67]. Similar to traditional swapping, memory compression increases the effective memory capacity of a system. When processes require more memory than available, the OS can transparently compress unused pages in DRAM to ensure they occupy a smaller footprint in DRAM rather than swapping them to disk. This frees up memory while still allowing the compressed pages to be accessed from DRAM. Compared to disk I/O, DRAM access is an order of magnitude faster, and even with the additional decompression overhead, memory compression is significantly faster than swapping. Hence, memory compression can improve the performance despite the additional CPU cycles required for compression and decompression. The Linux kernel implements ZRAM [68], enabled by default on Fedora [65] and Chrome OS [67].

1) Characterizing Timing Differences in ZRAM

To understand how memory compression can be exploited, we characterize its behavior in ZRAM. On Linux systems, ZRAM appears as a DRAM-backed block device. When pages need to be swapped to free up memory, they are instead compressed and moved to ZRAM. Subsequent accesses to data in ZRAM result in a page fault, and the page is decompressed from ZRAM and copied to a regular DRAM page for use again. We show that the time to access data from a ZRAM page depends on its compressibility and thus the data values. According to the previous experiments, we characterize the latency of accessing data from ZRAM pages with different entropy levels: pages that are *incompressible* (with random bytes), *partially-compressible* (random values for 2048 bytes and a fixed value repeated for the remaining 2048 bytes), and *fully-compressible* (a fixed value in each of the 4096 bytes). We ensure a page is moved to ZRAM by accessing more memory than the memory limit allows. To ensure fast run times for the proof of concept, we allocate the process to a *cgroup* with a memory limit of a few megabytes. We measure the latency for accessing a 8-byte word from the page in ZRAM, and repeat this process 500 times. Table II shows the mean latency of ZRAM accesses for different ZRAM compression algorithms on an Intel i7-6700K (Ubuntu 20.04, kernel 5.4.0). The latency for accesses to ZRAM is much higher for partially-compressible pages (with lower entropy) compared to incompressible pages (with higher entropy) for all compression algorithms. This is because the process of moving compressed ZRAM pages to regular memory on an access requires additional calls to functions that decompress the page. ZRAM pages that are stored uncompressed do not require these function calls (cf. Appendix C). We observe the largest timing difference for the `deflate` algorithm (close to 10 000 ns) and `842` algorithm (close to 7000 ns); we observe moderate timing differences for `lzo` and `lzo-rle` (close to 1000 ns), and `zstd` (close to 750 ns); the smallest timing difference are for `lz4` and `lz4hc` (close to 250 ns). These timing differences largely correspond with the algorithm's raw decompression latency (cf. Table I). Accesses to a fully-compressible page in ZRAM, i.e., a page containing the same byte repeatedly, are faster (by 200 ns) than accesses to an incompressible page for all the compression algorithms. This is because ZRAM stores such pages with a special encoding as a single-byte (independent of the compression algorithm) that only requires reading a single byte from ZRAM on an access to such a page.

TABLE II: Mean latency of accesses to ZRAM. Distinguishable timing differences exist based on data compressibility in the pages ($n = 500$ and 6 %of samples removed as outliers with more than an order of magnitude higher latency).

Algorithm	Incompressible (ns)	Partly Compressible (ns)	Fully Compressible (ns)
deflate	1763 (±12%)	12 208 (±2%)	1551 (±12%)
842	1789 (±11%)	8785 (±2%)	1556 (±10%)
lzo	1684 (± 9%)	4866 (±4%)	1479 (±12%)
lzo-rle	1647 (± 9%)	4751 (±4%)	1453 (±12%)
zstd	1857 (±10%)	2612 (±9%)	1674 (±11%)
lz4	1710 (±11%)	1990 (±7%)	1470 (±10%)
lz4hc	1746 (± 9%)	2091 (±9%)	1504 (±11%)

2) Leaking Secrets via ZRAM Decompression Timings

In this section, we exploit timing differences between accesses to a partially-compressible and an incompressible page in ZRAM (using `deflate` algorithm).

Attack Setup. We demonstrate bytewise leakage attack on a program with a 4 kB page stored in ZRAM containing both a secret value and attacker-controlled data, as is common in many applications like databases. To determine optimal data layouts an attacker might use, we combine this program with Comprezzor. With a known secret value, Comprezzor runs the program with the attacker guessing each byte position successively. For each byte position, Comprezzor generates the optimal memory layouts. In such an optimal layout, when the attacker's guess matches the secret-byte, the page entropy reduces (page is partially compressible), and ZRAM decompression takes longer; and for all other guesses, the entropy is high, and ZRAM decompression is fast. We repeat this process to generate optimal data layouts for each byte position. Note that this optimal data layout only relies on the number of repetitions of the guess and the relative position of the guessed data and the secret (a property of the compression algorithm), and is applicable with any data values. Using these attacker data layouts, we perform bytewise leakage of an unknown secret. At each step, the attacker guesses one byte (0-9, A-Z) and denotes the guess with the highest latency as correct. We repeat this attack for 100 random secrets.

Evaluation. Figure 9 shows the bytewise leakage for a secret value (`cookie=SECRET`), with the decompression times for guesses of the first four bytes depicted in each of the graphs. For each byte, among guesses of (0-9, A-Z), the highest decompression time successfully leaks the secret byte value (shown in red). For example, for byte 0, the highest time is for `cookie=S`. Similar trends are observed for the remaining bytes, as shown in the Figure 13 in Appendix E for byte 1, the highest latency is observed for `cookie=SE`. For byte 2, we observe a false positive,

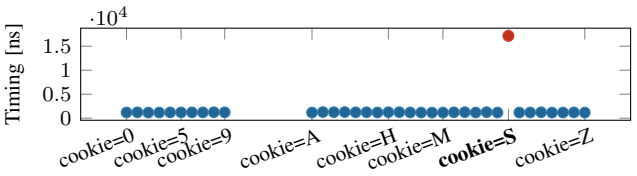

Fig. 9: Times for guesses (0-9, A-Z) for the first byte (S) of the secret leaked bytewise from ZRAM. The highest times correspond to the secret-byte value (shown in red). The standard error is below 1 % for all guesses.

`cookie=SE8`, which also has a high latency, along with the correct guess `cookie=SEC`. But in the subsequent byte 3, when both these strings are used as prefixes for the guesses, the false positives are eliminated, and `cookie=SECR` is obtained as the correct guess. The next two bytes are also successfully leaked to fully obtain `cookie=SECRET`, as shown in the Figure 13 in Appendix E. Repeating the experiment with 100 randomly generated secrets, we observe that in 90 out of 100 cases, the secret is leaked successfully. Our attack successfully completes in 58.6 s ($n = 90, \sigma_{\bar{\mu}} = 642.22\%$) on average, i.e., 49.14 bit/min. In 9 out of the 10 remaining cases, we narrow down the secret to within four candidates (due to false positives for the last-byte guess), and in the last case, we recover 4 bytes out of the 6-byte secret (the false positives grow for the 5th byte and beyond in this case). The false positives in our ZRAM PoC are caused by the Comprezzor-generated data layouts that are not as robust as in previous PoCs. Comprezzor with ZRAM is a few orders of magnitude slower (almost 0.03x the speed) compared to iterations with raw algorithms studied in Section V-B. Moving a page to ZRAM and compressing it requires accessing sufficient memory to swap the page out, which is much slower than executing just the compression algorithm. Consequently, the explored search space is smaller. Such false positives can be addressed by using multiple strong data layouts, or by fuzzing for a longer duration.

F. Leaking Heap Pointers from Google Chrome

For exploits in Javascript environments like V8 in Chrome, breaking memory randomization is often the first step. Such ASLR breaks usually rely on information leaks to disclose pointers from V8 isolates. However, vulnerabilities like out-of-bound reads that allow information leaks are promptly patched when discovered. In this section, we use our side channel to disclose a heap pointer and thus break heap-memory randomization. We assume a Chrome browser running on a device with ZRAM enabled. We run our experiments on a notebook equipped with an Intel i5-8250U CPU and 16 GB DDR4 RAM running Ubuntu 20.04 (kernel 5.4.0-124-lowlatency) and Google Chrome 90.0.4430.72. We setup a 4 GB ZRAM device as swap partition with the deflate algorithm. As a **timer**, we use a JavaScript counting thread [69], [70], [71], [72].

Co-Location. The first major requirement to leak a pointer is co-location between attacker-controlled data and a heap pointer (secret) on a 4 kB page. In JavaScript, elements of `TypedArrays` (e.g., `Uint8Array`) are memory-backed by an `ArrayBuffer` object. A backing heap pointer points to the location where the `ArrayBuffer` stores the data in memory [73]. All such 64-bit values in JavaScript (including pointers) are encoded using the IEE754 floating-point representation. Hence, to store a pointer in memory, non-typed arrays of numbers encoding 64-bit pointers are used. Thus, attacker-controlled numbers and secret heap-pointers (to `TypedArray`) can be co-located in a non-`TypedArray`, leading to the desired co-location of attacker data and the target pointer. We massage the allocations such that the target pointer is at the beginning of a 4 kB page. Listing 3 illustrates a snippet that co-locates the backing pointer of a `TypedArray` with a mostly attacker-controlled 4 kB region. Listing 4 is a memory dump showing the resultant co-location within the memory of a Chrome process. The resultant layout is indeed dependant on Chrome's allocator and may not always be the same. So, we measure which offsets we get with repeated runs of the same code snippet. For the same pointer, we observe that offsets of 0x0 or 0xc0 within the page occur in 84 % of the allocations (cf. Appendix E). Using Comprezzor, we can generate memory layouts for different byte offsets of the pointers with our setup. Note that the 32-bit compressed-pointers V8 uses to refer to objects in the same isolate are not randomized for each execution and do not affect our attack.

Trigger Swapping from JavaScript. The default `swapiness` value for Ubuntu 20.04 is 60. This means that if 80 % of memory is used, the kernel starts to perform swapping.Therefore, to swap the target data from RAM to the ZRAM swap device area, the attacker has to create high memory pressure. As the heap size per process is limited to 4 GB, this can be challenging. One approach an attacker can adopt is to spawn multiple processes to achieve the desired memory pressure. We observe that every iframe gets a separate renderer process, therefore, a higher memory pressure can be achieved. However, iframes from same domains (including subdomains) might get merged back into a similar process [74]. This could lead to the main tab crashing as the memory limit per tab is exceeded. Therefore, the attacker requires to use multiple iframes embedding content from different web servers to trigger swapping reliably. As `COEP` is set to `cross-domain`, the server under the attackers control requires to set the `CORP` to `cross-origin` [75]. Each of the domains can allocate about 4 GB. Therefore, to trigger swapping frequently, 4 additional remote servers from different domains are required. To ensure that the target is always evicted, we delete the iframes and repeat the allocation a second time. Note that this is the worst case scenario, assuming an idle system. Other system activity can only increase the probability of swapping out target pages. We run an experiment which constantly loads an iframe and tries to evict a certain target page. To successfully evict the target page from memory, the attacker requires, on average, 14.91 s ($n = 100, \sigma_{\bar{x}} = 7.59\%$).

Total Attack Runtime. The attacker has to guess all 256 possibilities per byte to leak the correct pointer. We use

Comprezzor to generate layouts for the 6 byte offsets. For stable results, 20 measurements per guess are required. Thus, leaking a single byte requires about 20 s for the swapping part in JavaScript and about one second to evaluate the memory layout, i.e., 21 s per guess. This leads to a total runtime of 29.8 h (21 s * 20 (tries per guess) * 256 = 107520/3600 = 29.8h) per byte. An attacker can invest additional engineering effort to perform multiple guesses in one iteration. Our theoretical runtime is only 7 min/B (21 s * 20 / 60 = 7m).

Takeaway: We show that even if an application does not explicitly use compression, its data may still get compressed by the OS due to memory compression. We demonstrate a local attack leaking 49.14 bit/min and port the attack to JavaScript leaking heap pointers in Google Chrome.

VII. Mitigations

Taint tracking. The best strategy to mitigate compression side channels is to avoid sensitive data being with potential attacker-controlled data. Mutexion [76] enforces mutually exclusive compression between attacker-controlled data and secret data in HTTP. This approach uses automated annotations of secret and attacker-controlled data. However, finding all the sources and sinks can be a complex problem for software developers, especially in large and complex software projects. Taint tracking tries to trace the data flow and mark input sources and their sinks. Paulsen et al. [77] use taint analysis to track the flow of secret data before feeding data into the compression algorithm. Their tool, Debreach, is about 2-5 times faster than SafeDeflate [77]. However, it is only compatible with PHP, and the developer needs to flag the sensitive input which is being tracked.

Disabling LZ77. A naive solution is to disable compression or at least disable the LZ77 part. Karakostas et al. [78] showed for web pages that, this adds an overhead between 91 % and 500 %. Furthermore, attacks on symbol compression have not been studied well enough to provide security guarantees.

Mitigating the Timing Side Channel. Constant time implementations might remove the timing side channel [79], [80], [81]. There are several solutions to automatically transform code into a constant-time version to mitigate side channels [82], [83], [84], [85], [86]. However, such solutions usually rely on code linearization, executing both the taken and not-taken path of branches. While this is feasible for cryptographic implementations with a limited number of branches, compression algorithms have too many input-dependent branches. Moreover, the overhead might get considerably worse without the memcpy optimization.

Masking. Karakostas et al. [78] presented a generic defense technique called Context Transformation Extension (CTX). The general idea is to use context-hiding to protect secrets from being compressed with attacker-controlled data. Data is permuted on the server side using a mask, and on the client side, an inverse permutation is performed (JavaScript library). The overhead compared to the original algorithms decrease with the number of compressed data [78].

Duplicating secrets. As Decomp+Time uses the generated layouts by Comprezzor, the guess is placed multiple times to trigger edge cases. Placing the secret multiple times might already be effective enough to mitigate Decomp+Time. We leave it as future work to evaluate the effectiveness.

Randomization. Yang et al. [87] showed an approach with randomized input to mitigate compression side-channel attacks. The service would require adding an additional amount of random data to hide the size of the compressed memory. However, as the authors also show, randomization-based approaches can be defeated at the expense of a higher execution time. Also, Karaskostas et al. [78] showed that size randomization is ineffective against memory compression attacks. It is also unclear if size randomization mitigates the timing-based side channel of the memory decompression.

Keyword protection. Zieliński presented an implementation of DEFLATE called SafeDeflate [88]. SafeDeflate mitigates memory compression attacks by splitting the set of keywords into sensitive and non-sensitive subsets. Depending on the completeness of the sensitive keyword list, this approach is considered secure. As Paulsen et al. [77] mention, it is easy to overlook a corner case. Furthermore, this approach leads to a loss of compression ratio of about 200 % to 400 % [78].

The aforementioned mitigations focus on mitigating compression-ratio side channels. As the compression and decompression timings are not constant, a timing side channel is harder to mitigate. Since the latency for a correct guess is in the region of microseconds, not many requests (≤ 200) are required per guess to distinguish the latency. Therefore, in a remote setting, a simple DDoS detection might detect an attack but only after a certain amount of data being leaked.

VIII. Conclusion

In this paper, we presented Decomp+Time, a timing side-channel attack on several memory-compression algorithms. We developed Comprezzor, an evolutionary fuzzer to amplify timing latencies when performing attacks on different compression algorithms. Our remote covert channel achieves a transmission rate of 9.73 kB/s locally and 10.72 bit/min over the internet (14 hops). We showed bytewise leakage with a leakage rate of 1.50 bit/min across the internet from a server using Memcached hosting, a PHP application. We leaked database records from PostgreSQL with 2.69 bit/min. We showed that we can locally attack ZRAM on Linux and leak heap pointers from Google Chrome. Our results show that compression of sensitive data can be dangerous even if the compressed data is not directly observable.

Acknowledgments

We thank our anonymous reviewers for their valuable feedback and comments on the paper. Furthermore, we want to thank Moritz Lipp, Jonas Juffinger and Claudio Canella for feedback on this work. This work was supported by generous funding and gifts from Red Hat. Any opinions or recommendations expressed in this work are those of the authors and do not necessarily reflect the views of the funding parties.

REFERENCES

[1] P. Yosifovich, A. Ionescu, M. E. Russinovich, and D. A. Solomon, *Windows Internals Part 1*, 7th ed. Microsoft Press, 2017.

[2] Apple Insider, 2013. [Online]. Available: https://appleinsider.com/articles/13/06/13/compressed-memory-in-os-x-109-mavericks-aims-to-free-ram-extend-battery-life

[3] PostgreSQL, "TOAST Compression," 2021. [Online]. Available: https://www.postgresql.org/docs/current/storage-toast.html

[4] php.net, "memcached.constants.php," 2021. [Online]. Available: https://www.php.net/manual/en/memcached.constants.php

[5] Mozilla, "Compression in HTTP," 2021. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Compression

[6] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Rfc 2616, hypertext transfer protocol – http/1.1," 1999. [Online]. Available: http://www.rfc.net/rfc2616.html

[7] BTRFS, "Compression," 2021. [Online]. Available: https://btrfs.wiki.kernel.org/index.php/Compression#Why_does_not_du_report_the_compressed_size.3F

[8] Amazon, 2012. [Online]. Available: https://docs.aws.amazon.com/redshift/latest/dg/t_Compressing_data_on_disk.html

[9] Citus, "Columnar: Distributed PostgreSQL as an extension." 2021. [Online]. Available: https://github.com/citusdata/citus/blob/master/src/backend/columnar/README.md

[10] Microsoft, "concepts-hyperscale-columnar," 2021. [Online]. Available: https://docs.microsoft.com/en-us/azure/postgresql/concepts-hyperscale-columnar

[11] Oracle, 2022. [Online]. Available: https://www.oracle.com/a/ocom/docs/database/hybrid-columnar-compression-brief.pdf

[12] J. Rizzo and T. Duong, "The CRIME attack," in *ekoparty security conference*, vol. 2012, 2012.

[13] Y. Gluck, N. Harris, and A. Prado, "BREACH: reviving the CRIME attack," *Unpublished manuscript*, 2013.

[14] T. Be'ery and A. Shulman, "A Perfect CRIME? Only TIME Will Tell," *Black Hat Europe*, 2013.

[15] M. Vanhoef and T. Van Goethem, "HEIST: HTTP Encrypted Information can be Stolen through TCP-windows," in *Black Hat US Briefings, Location: Las Vegas, USA*, 2016.

[16] D. Karakostas and D. Zindros, "Practical new developments on breach," *Black Hat Asia*, 2016.

[17] T. Van Goethem, M. Vanhoef, F. Piessens, and W. Joosen, "Request and conquer: Exposing cross-origin resource size," in *USENIX Security Symposium*, 2016.

[18] A. Nafeez, "Compression oracle attacks on vpn networks," *Blackhat, USA*, 2018.

[19] J. Kelsey, "Compression and information leakage of plaintext," in *Fast Software Encryption*, 2002.

[20] P. Deutsch, "Rfc1951: Deflate compressed data format specification version 1.3," USA, 1996.

[21] E. Chen, "Understanding zlib," 2021. [Online]. Available: https://www.euccas.me/zlib/#deflate

[22] Y. Collett, "Smaller and faster data compression with Zstandard," 2016. [Online]. Available: https://engineering.fb.com/2016/08/31/core-data/smaller-and-faster-data-compression-with-zstandard/

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

[24] M. Zalewski, "American Fuzzy Lop," 2021. [Online]. Available: https://github.com/Google/AFL

[25] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *USENIX Workshop on Offensive Technologies (WOOT)*, August 2020.

[26] LLVM Project, "libFuzzer – a library for coverage-guided fuzz testing." 2018. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[27] M. Payer, "The fuzzing hype-train: How random testing triggers thousands of crashes," *IEEE Security and Privacy*, 2019.

[28] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *NDSS*, 2017.

[29] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program smoothing," in *IEEE Symposium on Security and Privacy*, 2019.

[30] M. Bohme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *CCS*, 2017.

[31] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *ACM Sigplan Notices*, vol. 40, no. 6, 2005, pp. 213–223.

[32] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *31st International Conference on Software Engineering*, 2009.

[33] D. Weber, A. Ibrahim, H. Nemati, M. Schwarz, and C. Rossow, "Osiris: Automated Discovery Of Microarchitectural Side Channels," in *USENIX Security Symposium*, 2021.

[34] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, "ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures," in *NDSS*, 2020.

[35] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, "Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis," in *USENIX Security Symposium*, 2020.

[36] A. Fogh, "Covert Shotgun: automatically finding SMT covert channels," 2016. [Online]. Available: https://cyber.wtf/2016/09/27/covert-shotgun/

[37] S. He, M. Emmi, and G. Ciocarlie, "ct-fuzz: Fuzzing for Timing Leaks," in *International Conference on Software Testing, Validation and Verification (ICST)*, 2020.

[38] S. Nilizadeh, Y. Noller, and C. S. Pasareanu, "Diffuzz: differential fuzzing for side-channel analysis," in *International Conference on Software Engineering (ICSE)*, 2019.

[39] L. Bang, A. Aydin, Q.-S. Phan, C. S. Păsăreanu, and T. Bultan, "String analysis for side channels with segmented oracles," in *SIGSOFT*, 2016, pp. 193–204.

[40] M. Schwarzl, E. Kraft, M. Lipp, and D. Gruss, "Remote Page Deduplication Attacks," in *NDSS*, 2022.

[41] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches," in *USENIX Security Symposium*, 2015.

[42] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache Attacks on Mobile Devices," in *USENIX Security Symposium*, 2016.

[43] Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security Symposium*, 2014.

[44] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," in *DIMVA*, 2016.

[45] B. Gülmezoğlu, M. S. Inci, T. Eisenbarth, and B. Sunar, "A Faster and More Realistic Flush+Reload Attack on AES," in *COSADE*, 2015.

[46] Z. Wu, Z. Xu, and H. Wang, "Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud," in *USENIX Security Symposium*, 2012.

[47] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *S&P*, 2015.

[48] D. Evtyushkin and D. Ponomarev, "Covert Channels Through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations," in *CCS*, 2016.

[49] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *USENIX Security Symposium*, 2016.

[50] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. Alberto Boano, S. Mangard, and K. Römer, "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud," in *NDSS*, 2017.

[51] B. Semal, K. Markantonakis, K. Mayes, and J. Kalbantner, "One Covert Channel to Rule Them All: A Practical Approach to Data Exfiltration in the Cloud," in *TrustCom*, 2020.

[52] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "CrossTalk: Speculative Data Leaks Across Cores Are Real," in *S&P*, 2021.

[53] Memcached, "memcached - a distributed memory object caching system," 2020. [Online]. Available: https://memcached.org/

[54] holmeshe.me, "Understanding the memcached source code - slab ii," 2020. [Online]. Available: https://holmeshe.me/understanding-memcached-source-code-II/

[55] Equinix, 2021. [Online]. Available: https://www.equinix.com

[56] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "NetSpectre: Read Arbitrary Memory over Network," in *ESORICS*, 2019.

[57] D. Gruss, E. Kraft, T. Tiwari, M. Schwarz, A. Trachtenberg, J. Hennessey, A. Ionescu, and A. Fogh, "Page Cache Attacks," in *CCS*, 2019.

[58] X.-j. Zhao, T. Wang, and Y. Zheng, "Cache Timing Attacks on Camellia Block Cipher," *Cryptology ePrint Archive, Report 2009/354*, 2009.

[59] D. Jayasinghe, J. Fernando, R. Herath, and R. Ragel, "Remote cache timing attack on advanced encryption standard and countermeasures," in *ICIAFs*, 2010.
[60] H. Aly and M. ElGayyar, "Attacking aes using bernstein's attack on modern processors," in *International Conference on Cryptology in Africa*, 2013.
[61] V. Saraswat, D. Feldman, D. F. Kune, and S. Das, "Remote Cache-timing Attacks Against AES," in *Workshop on Cryptography and Security in Computing Systems*, 2014.
[62] O. Acıiçmez, W. Schindler, and C. K. Koc, "Cache based remote timing attack on the aes," in *CT-RSA*, 2006.
[63] T. Van Goethem, C. Pöpper, W. Joosen, and M. Vanhoef, "Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections," in *USENIX Security Symposium*, 2020.
[64] Flask, "Flask," 2021. [Online]. Available: https://flask.palletsprojects.com/en/2.0.x/
[65] M. Larabel, "What Is Memory Compression in Windows 10?" 2020. [Online]. Available: https://www.phoronix.com/scan.php?page=news_item&px=Fedora-33-Swap-On-zRAM-Proposal
[66] C. Hoffman, "What Is Memory Compression in Windows 10?" 2017. [Online]. Available: https://www.howtogeek.com/319933/what-is-memory-compression-in-windows-10/
[67] DennyL, "Enabling ZRAM in Chrome OS," 2020. [Online]. Available: https://support.google.com/chromebook/thread/75256850/enabeling-zram-doesn-t-work-in-crosh?hl=en&msgid=75453860
[68] N. Gupta, "zram: Compressed RAM-based block devices," 2021. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/blockdev/zram.html
[69] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript," in *FC*, 2017.
[70] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the Line: Practical Cache Attacks on the MMU." in *NDSS*, 2017.
[71] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *S&P*, 2019.
[72] P. Vila, B. Köpf, and J. Morales, "Theory and Practice of Finding Eviction Sets," in *S&P*, 2019.
[73] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector," in *S&P*, 2016.
[74] A. Agarwal, S. O'Connell, J. Kim, S. Yehezkel, D. Genkin, E. Ronen, and Y. Yarom, "Spook.js: Attacking Chrome Strict Site Isolation via Speculative Execution," in *S&P*, 2022.
[75] WebDev, 2022. [Online]. Available: https://web.dev/coop-coep/
[76] T. Moon, H. Kim, and S. Hyun, "Mutexion: Mutually exclusive compression system for mitigating compression side-channel attacks," *ACM Transactions on the Web (TWEB)*, 2022.
[77] B. Paulsen, C. Sung, P. A. Peterson, and C. Wang, "Debreach: mitigating compression side channels via static analysis and transformation," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019.
[78] D. Karakostas, A. Kiayias, E. Sarafianou, and D. Zindros, "Ctx: Eliminating breach with context hiding," *Black Hat EU*, 2016.
[79] J. Agat, "Transforming out timing leaks," 2000.
[80] H. Mantel and A. Starostin, "Transforming out timing leaks, more or less," 2015.
[81] C. Pereida García, B. B. Brumley, and Y. Yarom, "Make Sure DSA Signing Exponentiations Really Are Constant-Time," in *CCS*, 2016.
[82] P. Borrello, D. C. D'Elia, L. Querzoni, and C. Giuffrida, "Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization," in *CCS*, 2021.
[83] S. Cauligi, G. Soeller, B. Johannesmeyer, F. Brown, R. S. Wahby, J. Renner, B. Grégoire, G. Barthe, R. Jhala, and D. Stefan, "Fact: A dsl for timing-sensitive computation," in *PLDI*, 2019.
[84] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi, "Ghostrider: A hardware-software system for memory trace oblivious computation," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.
[85] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *ISSTA*, 2018.
[86] L. Soares and F. M. Q. Pereira, "Memory-safe elimination of side channels," in *CGO*, 2021.
[87] M. Yang and G. Gong, "Lempel-ziv compression with randomized input-output for anti-compression side-channel attacks under https/tls," in *International Symposium on Foundations and Practice of Security*. Springer, 2019.
[88] M. Zielinski, "Safedeflate: compression without leaking secrets," Cryptology ePrint Archive, Report 2016/958. 2016, Tech. Rep., 2016.
[89] S. Rostedt, "ftrace - Function Tracer," 2017. [Online]. Available: https://www.kernel.org/doc/Documentation/trace/ftrace.txt

APPENDIX A
COMPREZZOR-DISCOVERED TIMING DIFFERENCES

Table III shows the timing differences discovered by Comprezzor for different compression algorithms. We run Comprezzor with 50 epochs and a population of 1000 samples per epoch. We set the retention factor to $5\,\%$, selecting the best 50 samples in each epoch. We randomly mutate the selected samples to generate $70\,\%$ of the children and add $25\,\%$ of completely randomly generated layouts in the new generation.

APPENDIX B
TIMING DIFFERENCE FOR COMPRESSION

Table IV shows the timing differences when compressing incompressible, partially incompressible, and fully-compressible memory. The execution time and, thus, latency depends on the level of compressibility. For compressible memory, the timing is lower, which may appear counter-intuitive, but it means higher redundancy in the data and, thus, e.g., smaller Huffman trees and fewer sequences. For incompressible memory, the opposite case occurs, with more sequences and a larger tree, consuming more computation time. Additionally, when the compression ratio for a block is then too low, the compression is discarded, and additional `memcpy` operations are performed instead. All of this consumes computation time, leading to the timing differences we see in Table IV. For the intermediate case of partially incompressible data, both cases occur for part of the blocks leading to an intermediate timing. However, observing the compression time can be difficult, as no request or operation latency observable by the attacker depends on the compression time. Hence, we we focused on attacks exploiting the decompression timing side channel.

APPENDIX C
KERNEL TRACE FOR ZRAM DECOMPRESSION

To highlight the root cause of the timing differences in ZRAM accesses, we trace the kernel functions called on accesses to ZRAM pages using the `ftrace` [89] utility. Listing 1 shows the trace of kernel functions called on an access to an incompressible and compressible page in ZRAM. The incompressible page contains random bytes, while the compressible page contains 2048 bytes of the same value and 2048 random bytes, similar to the partially-compressible setting in Section VI-E1. We only list the functions which are called when the ZRAM page is swapped in to regular memory. The main difference between the two cases (colored in red) is that the functions performing decompression of the ZRAM page are only called when a compressible page is swapped-in, while

TABLE III: Timing differences between correct and incorrect guesses found by Comprezzor and the corresponding runtime.

Algorithm	Max difference for correct guess (ns)	Runtime (h)
PGLZ	109233.25	2.09
zlib	71514.75	2.46
zstd	4239.25	1.73
LZ4	2530.50	1.64

TABLE IV: Different compression algorithms yield distinguishable timing differences when compressing content with a different entropy. ($n = 100000$

Algorithm	Fully Compressible (ns)	Partially Compressible (ns)	Incompressible (ns)
FastLZ	38 619.07 (±0.74%)	58 887.40 (±0.50%)	79 384.89 (±0.40%)
LZ4	44 748.02 (±0.15%)	47 731.08 (±0.16%)	47 316.56 (±0.16%)
LZO	5645.86 (±2.18%)	5915.28 (±2.78%)	7928.21 (±3.91%)
PGLZ	44 275.84 (±0.13%)	65 752.55 (±0.12%)	-
zlib	38 479.53 (±0.22%)	80 284.72 (±0.23%)	76 973.82 (±0.20%)
zstd	3596.41 (±0.42%)	22 288.14 (±0.52%)	29 284.77 (±0.34%)

these functions are skipped for the page stored uncompressed. As the operating system knows from the stored meta-data whether the page is stored compressed or uncompressed and can skip the corresponding functions for uncompressed pages. Of these additional function calls, the main driver of the timing difference is the `__deflate_decompress` function in Listing 1 which consumes 12555 ns. This ties in with the characterization study in Section VI-E1 which showed the average timing difference between accesses to compressible and incompressible pages to be close to 10000 ns for ZRAM with `deflate` algorithm. These timings are for the `deflate` implementation in the Linux kernel, a modified version of `zlib v1.1.3`; hence these timings differ from the `zlib` timings in Table I for the more recent `zlib v1.2.11`.

```
1  Incomp.(ns) Comp.(ns) Function
2  Incomp.(ns) Comp.(ns) Function
3            0          0 swap_readpage
4           62         61 page_swap_info
5          126        123 __frontswap_load
6          195        188 __page_file_index
7          254        248 bdev_read_page
8          326        310 blk_queue_enter
9          395        379 zram_rw_page
10         460        442 zram_bvec_rw.isra.0
11         527        505 generic_start_io_acct
12         590        575 update_io_ticks
13         661        634 part_inc_in_flight
14         729        755 __zram_bvec_read.constprop.
15         813        838 zs_map_object
16         967       1040 _raw_read_lock
17           -       1229 zcomp_stream_get
18           -       1306 zcomp_decompress
19           -       1373 crypto_decompress
20           -       1433 deflate_decompress
21           -       1499 __deflate_decompress
22           -      14053 zcomp_stream_put
```

Listing 1: Kernel function trace for ZRAM access to an incompressible and compressible page.

Appendix D
Layout Discovered by Comprezzor

Figure 10 shows the layout discovered by the Comprezzor that amplifies the timing difference for decompression of the correct and incorrect guesses in Zlib dictionary attack. Note that for correct guesses, the entire guess string, i.e., `cookie=SECRET`, is compressed with the secret string. And as discussed in Section V-B, the subsequent data is incompressible and invokes only a single `memcpy` operation, which faster than Huffman or LZ77 decoding. For wrong guesses, only the prefix is compressed, introducing the timing difference we exploit. Listing 2a and Listing 2b show the debug trace from the Zlib code for decompression with the correct and incorrect guesses to illustrate the root cause of the timing differences. On a decompression, this block must now undergo the Huffman decoding (and LZ77), which results in several table lookups, memory accesses, and higher latency.

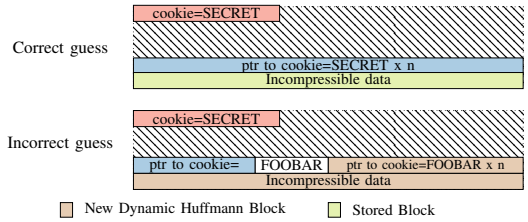

Fig. 10: Incorrect guesses with the corner case discovered by Comprezzor lead to a dynamic Huffman block creation for the partially-compressible data that is slow to decompress.

```
1        length 12
2        distance 16484
3        literal 0x17
4        length 13
5        distance 14
6        literal 0xb3
7        length 13
8        distance 14
9        literal 'x'
10       length 13
11       distance 14
12       literal 0x05
13       length 13
14       distance 14
15       literal 0xa9
16       length 13
17       distance 14
18       literal 0x81
19       length 13
20       distance 14
21       literal '['
22    stored block (last)
23      stored length 16186
24      stored end
25   check matches trailer
26 end
```

(a) Trace for correct guess in zlib. Here the entire guess string is compressed, and the remainder is incompressible (decompressed fast as a stored block).

```
1        length 6
2        distance 16484
3        literal 'F'
4        literal 'O'
5        literal 'O'
6        literal 'B'
7        literal 'A'
8        literal 'R'
9        literal 0x17
10       length 13
11       distance 14
12       literal 0xb3
13       length 13
14       distance 14
15       literal 'x'
16       length 13
17       distance 14
18       literal 0x05
19    dynamic codes block (last)
20      table sizes ok
21      code lengths ok
22      codes ok
```

(b) Trace for incorrect guess in zlib. Here only part of the guess string (`cookie=`) is compressed, and the remainder `cookie=FOOBAR` is separately compressed (decompression requires a slower code block for Huffman tree decoding).

Listing 2: Zlib traces for compression

Appendix E
Bytewise Leakage

Figure 11 illustrates the bytewise leakage of the secret (SECRET) for a PHP application using PHP-Memcached. Figure 12 shows the bytewise leakage of data stored in PostgreSQL. The prefix value can be shifted bytewise, which allows reusing the same memory layouts found by Comprezzor. Figure 13 shows the last two bytes leaked from a secret (SECRET) in a ZRAM page. This is a continuation of Figure 9

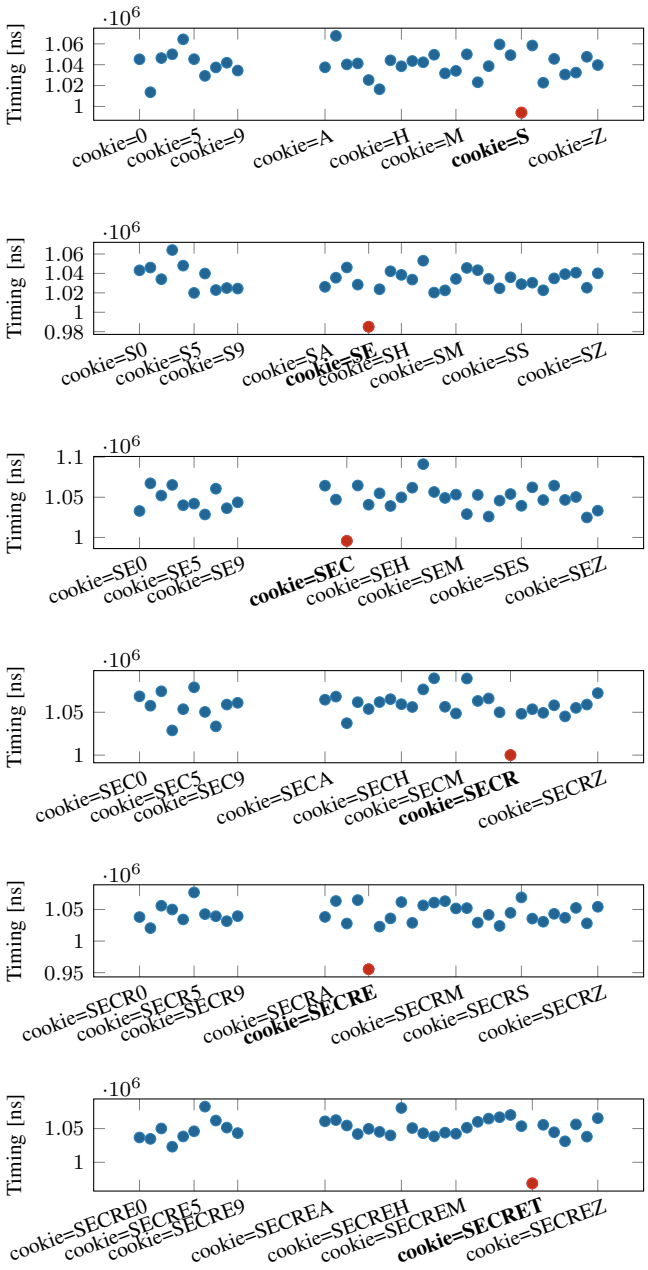

Fig. 11: Bytewise leakage of the secret (S,E,C,R,E,T) from PHP-Memcached. In each plot, the lowest timing (shown in red) indicates the correct guess. The standard error is below 1 % for all guesses.

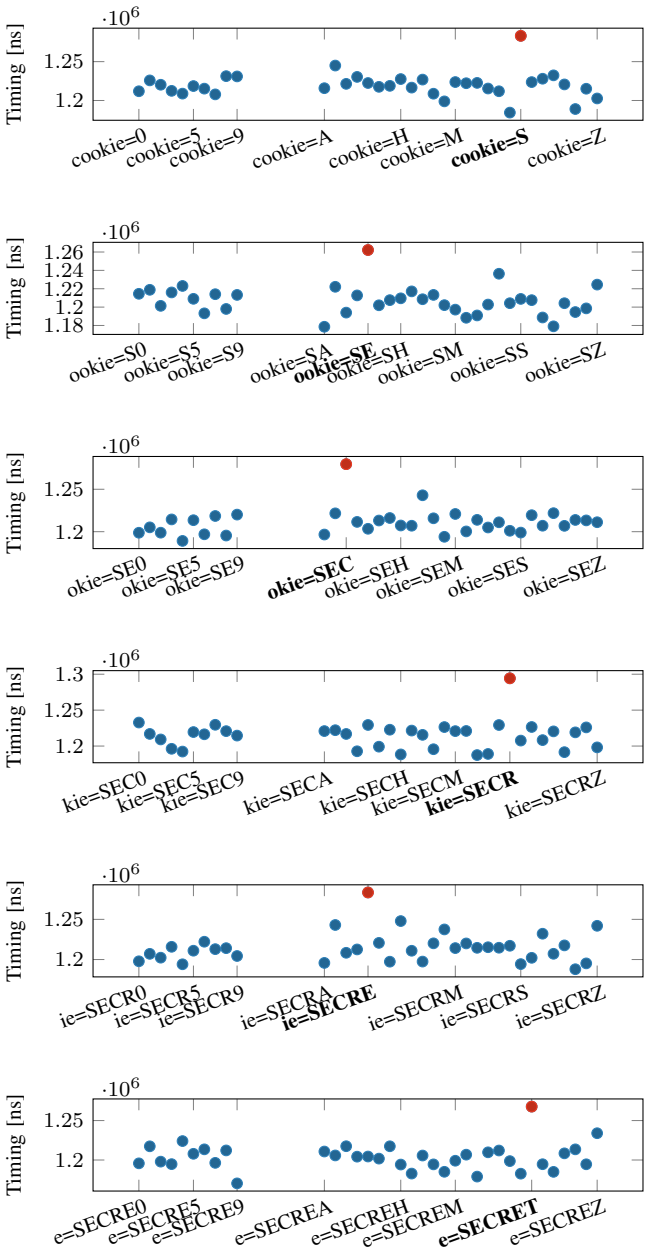

Fig. 12: Bytewise leakage of the secret (S,E,C,R,E,T) from PostgreSQL. The known prefix (cookie=) is shifted left by 1 character in each step, allowing the same memory layout to be reused. In each plot, the highest timing (shown in red) indicates the correct guess. The standard error is below 1 % for all guesses.

which showed the leakage of the first four bytes. All three cases expose extremely high timing differences with an orders-of-magnitude gap between the correct and wrong guesses. The standard error is below 1 % for all guessess.

JavaScript Memory Layout

Our allocation script cf. Listing 3 creates a memory layout such that a 64-bit heap pointer pointing to the backing store is stored into a regular JavaScript array (non_typed_arrays).

This can be achieved by first inserting a TypedArray with the full length of a page in the first slot of the list. To co-locate attacker-controlled data, the script inserts 4096 64-bit numbers into a regular JavaScript Array (`colocate_data`). This array will then be inserted into the array containing the 64-bit pointers (non_typed_arrays) and the garbage collection will be triggered. As 64-bit values in JavaScript are represented using

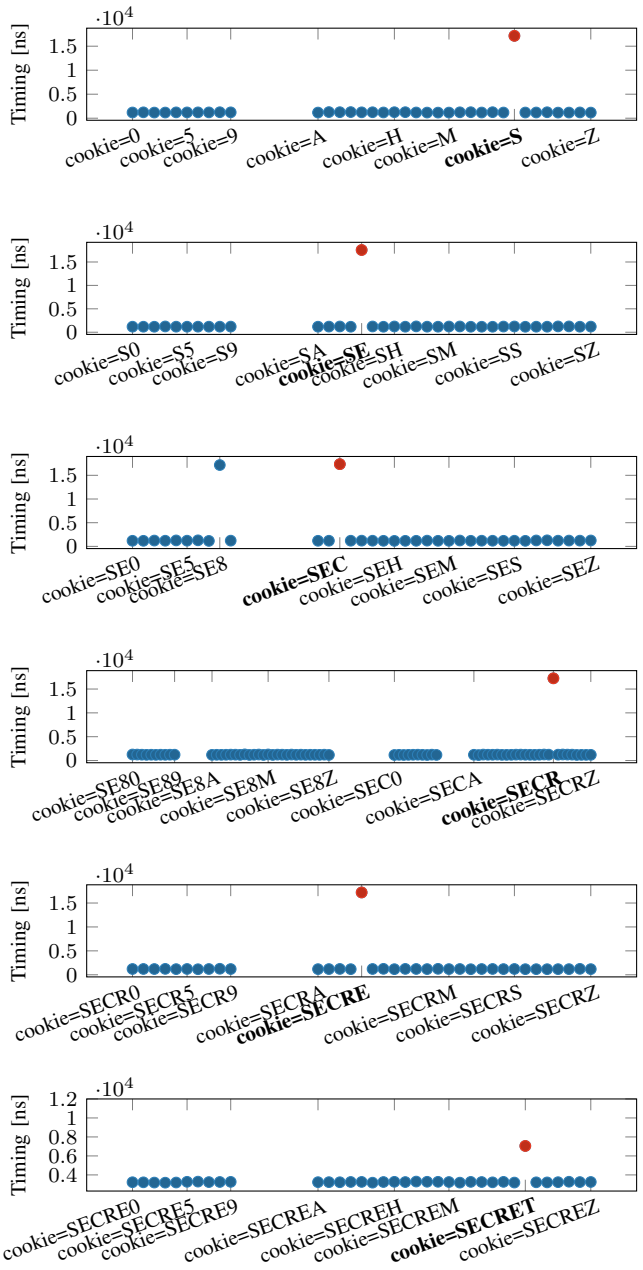

Fig. 13: Bytewise leakage of the secret (S,E,C,R,E,T) from ZRAM. Times for guesses (0-9, A-Z) for each of the bytes are shown. The highest value in each plot (shown in red) indicates the correct secret value for the byte. The standard error is below 1 % for all guesses.

the IEE754 floating-point representation, we use a conversion function `itof` to encode a 64-bit hexadecimal pointer to IEE 754 floating-point number. This function takes the BigInt and stores it into an `Float64Array`. By dumping the V8 memory we found that a number of 203 elements in the list to a memory layout, where the attacker controls most of the page and the only data that varies are the two heap pointers at offset 0. Listing 4 illustrates the generated memory layout

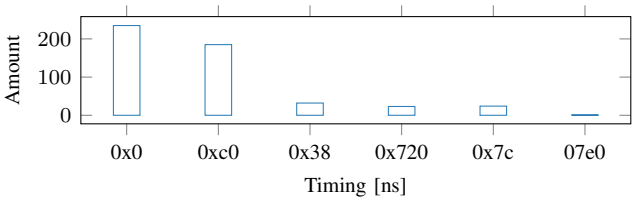

Fig. 14: Pointer offset distribution in Chrome.

after running the script from Listing 3. The first line (00) shows the 64-bit heap pointers aligned to page offset 0. We observe that the data between offset (10:) and offset (7c:) is constant and contains some compressed pointers to JavaScript objects. The data from offset `7c` to `0xfff` is fully attacker-controlled indicated in Listing 4 by the value (0xcafebabe). Using a fixed suffix, the attacker can use Decomp+Time to leak the correct byte values of the pointer. However, we do not observe that the heap pointer is always placed at page offset 0. We repeat our experiment 500 times to see the offset distribution. Moreover, we observe that if only the attacker-controlled data is modified, the garbage collection does not reorganize the heap. Figure 14 shows the distributions in 47 % of the cases the pointer is at offset 0. In 37 % of the cases the pointer is at offset 0xc0. The remaining 16 % of the positions the pointer was at 4 other offsets.

```
const NUM_VALS = 203;
var non_typed_arrays = new Array(NUM_VALS);
non_typed_arrays.fill(Object);
// inserts target TypedArray including the 64-bit
    pointer
non_typed_arrays[0] = allocTypedArray(4096,0x31);
for(var k = 1; k < NUM_VALS; k++) {
 let colocate_data = [];
 for (let i = 0; i < 4096; i++) {
  // itof converts a BigInt to IEE 754 floating-
     point representation
  colocate_data[i] = itof(0xcafebabecafebaben);
 }
 non_typed_arrays[k] = colocate_data;
 triggerGC(); // trigger garbage collection
}
```

Listing 3: Co-locate V8 heap pointers with attacker-controlled data.

```
00: 00 00 01 01 58 3c 00 00  30 72 9c 00 58 3c 00 00
10: 02 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
20: 00 00 00 00 05 22 04 08  10 00 00 00 b5 23 04 08
30: b5 23 04 08 b5 23 04 08  b5 23 04 08 b5 23 04 08
40: b5 23 04 08 b5 23 04 08  b5 23 04 08 05 22 04 08
50: 10 00 00 00 c1 46 24 08  5d 44 24 08 c1 e7 24 08
60: dd 30 25 08 71 c7 24 08  25 f3 24 08 59 d7 24 08
70: b1 2a 25 08 99 2a 04 08  a2 22 00 00 ca fe ba be
80: ca fe ba be ca fe ba be  ca fe ba be ca fe ba be
*
1000
Legend: 00-0f:|64-bit pointers at offset|
10-7b:|Static data|, 7c-fff:|Attacker-controlled
    data|
```

Listing 4: Memory dump of the target page from Chrome after allocating non-typed and typed arrays and adding them to a list. The attacker can co-locate two V8 64-bit heap pointers and attacker-controlled data.