AMPhitryon: Efficient small data compression for low-bandwidth covert channels

Steffen Wendzel[1 [0000−0002−1913−5912]], Sebastian Zillien[1 [0000−0003−3360−1251]], and Sebastian Zander[2 [0000−0002−2084−7204]]

[1] Ulm University, Germany
{steffen.wendzel,sebastian.zillien}@uni-ulm.de
https://www.uni-ulm.de/en/in/omi/
[2] Murdoch University, Perth, Australia
s.zander@murdoch.edu.au

Abstract. A plethora of known methods describe how policy-breaking transmissions can be conducted, e.g., through network censorship circumvention channels or through local covert channels. While some of these channels have very low capacities, only a few studies have been published on improving the content-compressibility for such channels. However, these methods are either not useful for compressing *small* data, require to crawl third-party content (e.g., websites), or can only transfer data with a small bitrate. Nevertheless, in reality typical payloads are often small (e.g., password exfiltration, status updates, or chat messages) and usually remain uncompressed. In this paper, we introduce AMPhitryon, a novel approach to compress *small* payloads <100 B (especially 4-20 B). It can be used with existing tools to compress small payloads, while larger payloads can be compressed with arbitrary compressors. AMPhitryon employs a local dictionary that is compiled incrementally in a context-specific setting (e.g., the context could be a network flow or a chat). Our evaluation shows that AMPhitryon achieves higher compression ratios for short messages than existing techniques. It reaches the lowest (best) compressibility ratios (down to 0.30) with the smallest payloads (<20 B).

Keywords: Data Compression · Internet Censorship · Network Security · Covert Channels · Network Traffic · Information Hiding.

1 Introduction

Covert channels are policy-breaking and usually stealthy communication channels [21]. Although covert channels are often used for illegitimate purposes, they also have legitimate applications, such as hidden remote system administration, hidden sensor data collection, or censorship circumvention. The latter case of censorship circumvention becomes increasingly important as censorship is now present in several countries around the world [3,18].

Typically, the core goal of censors is to block unwanted flows to sites serving undesired content, such as websites featuring critical opinions or chat platforms

where undesired opinions are exchanged. Censorship can be circumvented using tools such as Tor, Psiphone, VPN setups, network steganography tools, obfuscation tools, or a combination of these (e.g., Tor obfuscation extensions). However, the use of some of these tools, such as Tor or VPNs, can be easily detected by censors. Hence, in this paper, we focus on tools that establish different forms of policy-breaking communication channels, referred to as *covert channels*, which are challenging to detect.

Successful detection of covert channels often depends on the amount of traffic that can be observed. In addition, more stealthy channels also often have very low bandwidths, such as covert channels in games providing up to 13 bit/s [20] or temperature-based covert channels providing up to 12.5 bit/s [9]. Typically, the less traffic is transmitted, the harder it is to identify covert channels [15]. Consequently, users will aim to compress the data before transmitting.

However, most existing compressors, such as GZip or BZip2, are not optimized for small messages that are typically exchanged on covert channels. In many scenarios, covert channels transfer many small payloads (i.e., <100 B), such as chat messages, username/password strings, or small fractions of website content (small images, scripts, or .css files). Existing compression algorithms have difficulties compressing such small data chunks, as these may contain only a few redundancies or the overhead of the internal dictionary is large.

In this paper, we address this compression problem by introducing a novel compression approach tailored to small payloads <100 B, with a focus on 4-20 B. Our approach is loosely based on the concept of *covert channel amplification* [17], i.e., transferring only small "pointers" to larger chunks of existing data accessible to the communicating peers (Alice and Bob). In our scenario, Alice and Bob iteratively build a dictionary that they both use to replace parts of messages with smaller pointers to the dictionary. For instance, Alice might send chat messages or part of a website to Bob. Every new byte string can be added to the dictionary used by Alice and Bob. Once a part of the string occurs again, Alice (and Bob) can simply refer to the particular byte string in the dictionary and decrease the size of even small payloads. Rather than building a dictionary on-the-fly, a pre-built dictionary optimized for certain types of messages could be used by Alice and Bob.

In particular, our key contributions are the following:

1. A novel lossless method to compress small messages dubbed AMPhitryon[3], which can be used to minimize the "footprint" of censorship circumvention. It can be used with arbitrary circumvention tools and covert channels to compress the data prior to encryption or embedding.

[3] We use "AMP" to refer to *amplification*. Moreover, reflected by the prefix *amphi-* (Ancient Greek ἀμφί), which can be translated as *on both sides*, our amplification relies on the combination of *both* components: the transferred (compressed) data channel and the dictionary component being pointed to by the compressed data. The term AMPhitryon links to *Amphitryon*—the husband of *Alcmene* in Greek mythology. *Amphitryon* itself can be translated as *harassing both sides* [19].

2. An in-depth evaluation of AMPhitryon's compression performance, including a comparison with traditional compression approaches.

The key results of our evaluation show that AMPhitryon provides better compression performance than all state-of-the-art compressors when it comes to small messages. To aid replication studies, the AMPhitryon implementation is available at `https://anonymous.4open.science/r/AMPhitryon-8AF5/`.

The remainder of this paper is structured as follows. Section 2 discusses related work. Section 3 defines the censorship circumvention scenario and describes the design and implementation of AMPhitryon. Section 4 evaluates the proposed approach and compares it with traditional compression techniques. Section 5 concludes and outlines future work.

2 Related Work

Dictionary-based compression techniques emerged early, including those used by the GIF file format and the UNIX `compress` command [14]. These techniques rely on the presence of regularly occurring patterns that are used to form *dictionaries*. The core idea of these methods is to refer to dictionary elements in compressed data with references smaller than the data they refer to, leading to a reduction in size compared to uncompressed data [14].

Most common compression schemes, such as ZIP, GZIP, LZMA/7z, utilize internal dictionaries stored in the data (i.e. file), which makes them less effective for applications focused on short messages. Two recently proposed compression schemes, *Brotli* [1] and zstd [10] are dictionary-based too, but provide support for external dictionaries. We compare AMPhitryon with both.

Recently, Wendzel *et al.* introduced the idea of *covert channel amplification* in [17]. The amplification is achieved by transferring small pointers that refer to data chunks that are larger than the pointer itself. In that sense, fewer data must be sent than is actually transmitted. Such an amplification can be achieved in multiple ways. In case of DYST (*Did You See That?*) [17], a sender points to previously seen broadcast packets which hash values match the secret messages to be transferred. While DYST was the first attempt towards realizing covert channel amplification, its typical transmission rate is < 1 bit/s (< 59 bit/s when multiple pointers are used in parallel while reducing stealthiness). The transmission rate of DYST was increased through improved coding and reference strategies of SHP (*Silent History Protocol*) by Weissenborn and Wendzel [16]. SHP improved DYST's bitrate between 136 times (121.276 bit/s instead of 0.887 bit/s) and 595 times (2.383 bit/s instead of 0.004 bit/s) compared to DYST's standard configuration [16].

Another attempt to improve DYST was made by Zillien *et al.* with OPPRESSION (*Open Knowledge Compression*) [22]. OPPRESSION is an addition to censorship circumvention tools and requires the sender and receiver to crawl websites, such as Wikipedia or ProjectGutenberg to build dictionaries in the form of Patricia Tries (a variant of tree). To transfer a secret message, the sender must transfer a pointer to a tree node.

Keller and Litzinger [6] published a method to compress a packet's content and embed a secret message in the gained space. In comparison to earlier works, they focus on short packets ($30 - 250$ bytes) and apply a Huffman-inspired and run length-inspired coding on non-equidistributed payloads.

A related field is steganography by *cover selection*, where an image database is created and the values of the images are hashed. If the secret message matches a hash, the image is transferred instead. Several methods have been proposed to conduct or optimize the cover selection process, e.g., [2,4,7].

Recently, specialized compression techniques have also been proposed in other research domains. For example, Roy et al. proposed a novel lossless compression algorithm for Deoxyribonucleic acid (DNA) or ribonucleic acid (RNA) sequences that outperforms standard techniques, such as zstd or LZMA/7z [12].

In contrast to the vast majority of traditional compression schemes, AMPhitryon separates the dictionary from the data to be transmitted. The only similar compression techniques are Brotli and zstd, which also use external dictionaries and have shown good compression performance on small data chunks. Furthermore, AMPhitryon is not limited to (rather) low transmission rates, such as SHP and DYST, nor does it depend on downloading public websites (OPPRESSION) or image databases (cover selection steganography), which introduces synchronization issues and due to the size and public nature of these datasets might cause the attention of adversaries. Moreover, unlike OPPRESSION, AMPhitryon does not focus solely on textual messages, but can deal with binary data. Compared to the work by Keller and Litzinger, AMPhitryon can compress even smaller payloads (below 30 bytes). Finally, AMPhitryon is the only method that has been designed to be part of covert channel tools in practical settings, as it can be incorporated by such tools before secret data is embedded or encrypted.

3 Scenario, Design & Implementation

3.1 Threat Scenario

We consider a scenario in which Alice and Bob want to remain stealthy in order to prevent being detected by an observer, for example, the case of censorship circumvention. We assume that in a scenario like censorship circumvention, often only small messages are exchanged, such as chat messages. To evade detection, Alice and Bob aim to use stealthy covert channels that typically have low bandwidths. In addition, they aim to minimize the amount of data exchanged, since the more data is being exchanged, the higher the likelihood that their communication will be detected. Hence, Alice and Bob employ AMPhitryon to compress all their messages.

For instance, Alice may use an online chat tool that allows her to interact with other dissidents or remote human rights activists, including Bob. Alice is free to select any censorship circumvention tool in this setting as AMPhitryon can serve as an extension for arbitrary tools. Rather than a censorship

circumvention tool, Alice may use a stealthy, low-bandwidth covert channel to communicate with Bob and others. For example, covert channels in game traffic [20] or temperature-based covert channels [9] have very low bandwidths and would greatly benefit from using AMPhitryon to compress the data *before* encryption.

3.2 Design and Implementation

As AMPhitryon is a compression system, it is independent of the particular censorship circumvention tool. A typical circumvention tool is provided with payload chunks, e.g., content of a chat or website. However, as any dictionary-based compression scheme [14], AMPhitryon relies on the redundancy of patterns in the data to be compressed. AMPhitryon cannot handle randomized content where byte values follow a discrete uniform distribution, i.e., a succeeding bit is set with approx. $p = 0.5$. Hence, the compression must be done on non-encrypted payload chunks *before* they are being encrypted or steganographically embedded. AMPhitryon supports payload chunks with text or binary encoding. The compressed data can then be encrypted before it is steganographically embedded.

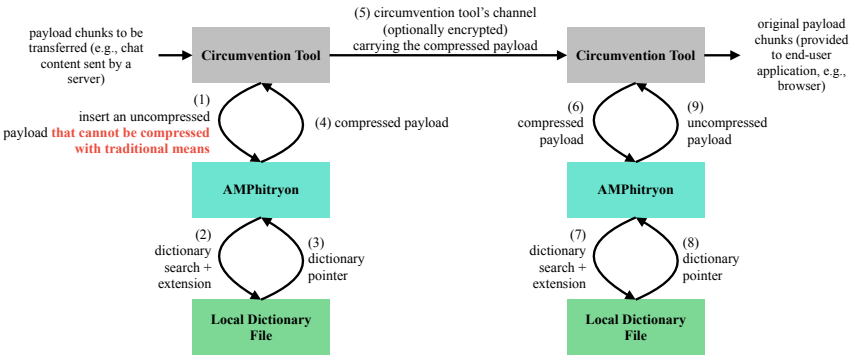


Fig. 1: Functionality and interaction of AMPhitryon with censorship circumvention tool

High-level Functionality Fig. 1 provides an overview of AMPhitryon's functionality. Initially, payload data chunks are provided to the circumvention tool. In step 1, the circumvention tool provides small payload chunks to AMPhitryon.[4] AMPhitryon then compresses the data. AMPhitryon scans the payload chunks for content that matches bit strings in the local dictionary (step 2).

[4] In case of large data, traditional compressors could be used instead of AMPhitryon.

If a chunk is found in the directory, the original payload chunk is replaced with a pointer/reference to the dictionary (step 3). Otherwise, the chunk is encoded in its original form and added to the local dictionary so that it can be referenced in the future.

In other words, AMPhitryon's dictionary is *incremental*, i.e., every original message chunk that was not found in the dictionary will be added to it so that it can be used for compression should the message appear again.

Once all possible pointers have been set, the compressed data is fed back to the circumvention tool (step 4). The tool can then transfer the payload to a peer (step 5) and optionally encrypt it.

In step 6, the receiving end forwards the decompressed data to its local AMPhitryon. The data is decompressed using the local dictionary (steps 7 and 8). Note that original chunks are added to the local dictionary of the receiver so that they can be referenced by future pointers in the same way as during compression at the sender. Finally, the de-compressed payload is fed back to the circumvention tool (step 9), which provides it to the end-user's application.

Flow-dependency of Dictionaries AMPhitryon generates a novel dictionary for every communication so that the content of the dictionary is optimized for the specific use-case. For example, a chat in Italian builds a different dictionary than a chat in German, French, or English or a transfer of HTML content. However, by simply not deleting (but keeping) the dictionary of a previous flow, the dictionary can be enhanced by a (similar) communication, which leads to improved performance due to a higher chance of finding matches for chunks that are being pointed to. Instead of building a dictionary from scratch, a dictionary built for a similar communication could be used from the start to achieve higher compression rates.

Implementation We implemented AMPhitryon in C and evaluated it under Ubuntu Linux 22.04 and macOS 15.7. The code does not rely on additional libraries. We evaluated our implementation by compressing and subsequently decompressing millions of data chunks (we evaluated cryptographic checksums of all decompressed results to detect errors, which did not occur). Including compressor and decompressor, our implementation contains around 500 lines of code. Our implementation, sample data, and hands-on documentation are available here: `https://anonymous.4open.science/r/AMPhitryon-8AF5/`.

Compression Format AMPhitryon compresses the input message using the data format as illustrated in Fig. 2 with a chunk size of $N = 12$ bytes. A magic bit shown as a red block indicates whether a pointer value (bit=1) or a number of original plaintext blocks follow (bit=0). To distinguish original plaintext chunks (including binary data) from pointers, plaintext blocks are preceded by length bytes in which the magic bit is not set. The following describes the control data added during the compression:

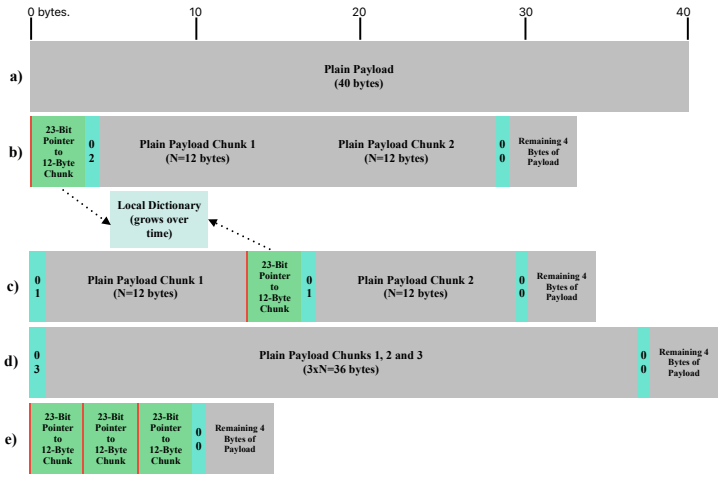


Fig. 2: AMPhitryon's message format illustrated using a chunk size of $N = 12$. (a) plain payload, (b) likely scenario in which one pointer was found (red colored: MSB indicating that a 23-bit pointer follows), (c) scenario in which one pointer was found but more metadata is needed, (d) scenario in which no pointer was found and the message size *increases*, and (e) optimal case

0x80–0xff The first bit is set. The following seven bits of this byte and the next 16 bits jointly represent a 23-bit pointer to a dictionary entry (the data being pointed to has the size N, called *chunk size*).

0x01–0x7f The first bit is not set. The following seven bits indicate how many original chunks follow.

0x00 The first bit is not set and no full message chunk follows. The following bytes are original plaintext bytes that are the remaining bytes of size $\leq N$.

AMPhitryon's compression relies solely on the pointers that refer to dictionary content that is larger than the pointers. The larger the chunk being pointed to, the higher the compression of the chunk, but the more challenging it is to find a matching dictionary entry. Since the magic bit requires one bit and the pointer size is 23 bits (3 bytes combined), chunk sizes of $N \geq 4$ bytes are a reasonable choice for our evaluation.

In our example of Fig. 2b, the original payload size of 40 bytes was reduced to a size of 33 bytes using only one pointer. Fig. 2c shows a scenario where more fragmentation leads to a slightly smaller compression (34 bytes). A worst-case example is shown in Fig. 2d: no pointers have been used as the dictionary did not contain any matching bit strings, but the control information must be embedded nonetheless, i.e., the overall message size increases to 42 bytes. In such a case where the compressed data is larger than the uncompressed data, one should transfer the uncompressed data instead (e.g., indicated by a header bit that tells

the receiver if the data is compressed, or alternatively, one could sacrifice the magic byte value 0xfe for this purpose). An optimal case is shown in Fig. 2e, where a maximum number of pointers is used and the resulting message size is only 14 bytes.

In general, we can assume the compressed size C of an original uncompressed message O to be within

$$3P \leq C \leq 3P + (O - NP) \cdot ((N + 1)/N),$$

with P being the number of 3-byte pointers that are set and N being the chunk size. In the ideal case, the original payload is completely replaced by pointers ($C = 3P$). In the typical case, the pointers replace NP bytes of uncompressed data with $3P$ bytes. For the remaining uncompressed $O - NP$ bytes, in the worst case the number of bytes increases by a factor of $(N+1)/N$ per chunk due to the necessary magic bytes. However, C will be smaller in many cases as multiple uncompressed chunks (up to 127) can be chained together with a single magic byte.

4 Evaluation

In this section, we evaluate the performance of AMPhitryon by comparing the compression with well-known off-the-shelf compression algorithms. We also investigate the dictionary sizes and computation times of AMPhitryon in different settings.

4.1 Dataset

We evaluated AMPhitryon using plausible and publicly available data. In particular, we have chosen IRC chat log files from the irc-disentanglement dataset provided by [8,5] and the Chromium Conversations Corpus [11] provided via the ConvoKit Python package[5]. Tab. 1 shows more information about the datasets.

Dataset	Time-Frame	Messages	Log Files
Ubuntu	2005-2018	220,46	153
Chromium	2008-2016	2,853,885	1,357

Table 1: Chat message datasets

The Ubuntu dataset contains chat messages from the `#ubuntu` IRC channel[6]. The Chromium dataset contains messages from developers commenting on

[5] `https://convokit.cornell.edu`
[6] `https://irclogs.ubuntu.com`

changes to the Chromium project. For the chromium dataset, we combined multiple days of messages, so each log file has at least 1,500 lines. We did not cut days; therefore, all messages from a single day are found in a single log file. We chose these datasets since chats would be one of the likely use cases of AMPhitryon. A chat server sends and receives many small messages, which can be challenging to compress for regular compression algorithms. With the incremental dictionary of AMPhitryon, the algorithm learns often-used words or phrases, which increases compression performance over time.

We filtered the datasets for outliers and cut all messages to a maximum length of 50 bytes, as longer messages are not the target for AMPhitryon and could be effectively compressed with off-the-shelf algorithms. We identified lines as outliers that contained more than 15 consecutive instances of the same symbol. This removes extreme examples, but still leaves messages with repeated words in the dataset.

4.2 Compression Performance

We compared AMPhitryon against four off-the-shelf compression algorithms: Gzip, Zstandard(zstd), LZMA2 and Brotli. For the four reference algorithms, we used the strongest compression settings available and compressed the log files line by line, simulating a message-by-message compression. To measure compression performance, we chose the compression ratio (R) between the message size before (M) and after compressing (C) [13]: $R = \frac{C}{M}$. We started with an empty dictionary for each log file and also compressed the files line by line, i.e., per chat message. We evaluated chunk sizes from 4 up to 20 bytes to find the optimal chunk size.

Fig. 3 shows the compression performance of AMPhitryon by chunk size. For this plot, we averaged the compression ratio over all lines for each file. Each line in the plot represents a single input file of the two datasets. We can see a generally optimal compression performance for AMPhitryon with a chunk size between 7 and 12 bytes. Depending on the dataset, the optimal chunk size varies between 9 and 10 bytes.

Fig. 4 shows a ridge-line plot of the compression performance of the four selected algorithms. For AMPhitryon we used a chunk size of 10. The plot shows the distribution of compression ratios for each algorithm. We can see that AMPhitryon generally performed better than the other compression algorithms, with most of the ratios below 1.0. The off-the-shelf algorithms all perform significantly worse, with a majority of compression ratios above 1.0, which means sizes actually increased in the majority of cases.

We can see some outliers in both directions. The low outliers for the reference algorithms stem from repetitive lines that were not removed by our filters. Those lines include repeated short words, but not consecutive repeated characters. The high outliers can be explained by very short messages, as the reference algorithms always add a minimum overhead.

Tab. 2 shows more statistics for the compression performance of each algorithm. We can see that AMPhitryon performs very consistently without strong

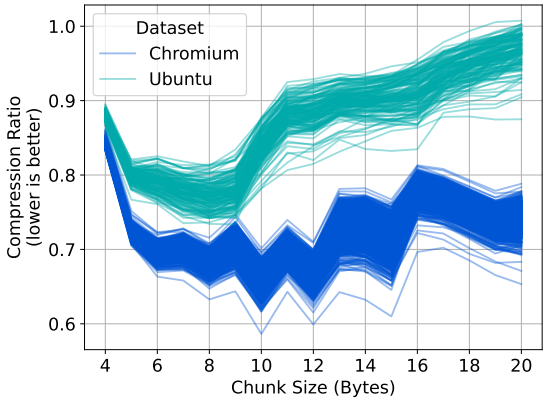


Fig. 3: AMPhitryon compression performance by chunk size

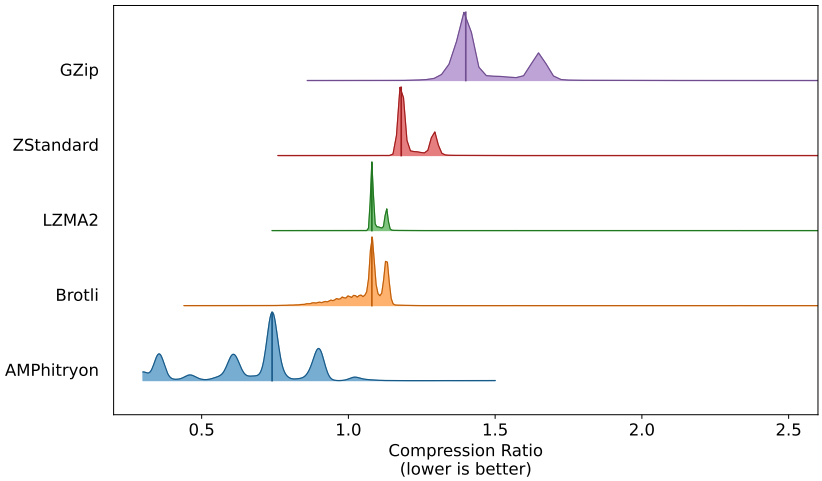


Fig. 4: Comparison of compression ratios of AMPhitryon (chunk size 10) with existing compression standards

Algorithm	Min	Max	Mean	Median
AMPhitryon	0.30	1.50	0.67	0.70
Gzip	0.86	11.0	1.47	1.40
LZMA2	0.74	3.0	1.10	1.08
Brotli	0.44	3.0	1.06	1.08
ZStandard	0.76	5.5	1.22	1.18

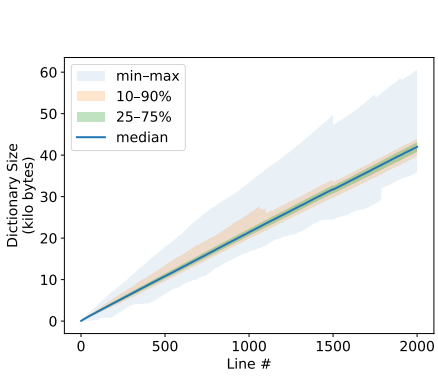
Table 2: Minimum, maximum, mean and median compression ratios for different compression techniques (AMPhitryon chunk sizes 4-20)

outliers. The mean compression ratio of AMPhitryon is even better than the best performances of most other algorithms.

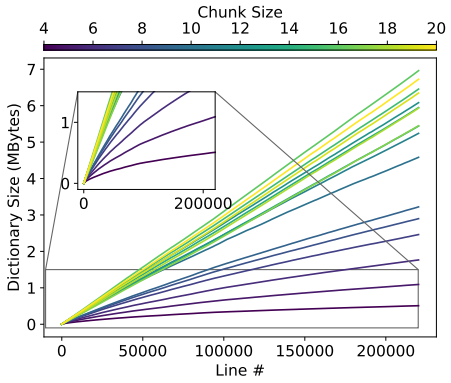
4.3 Dictionary Size

For AMPhitryon we also investigated the growth of the dictionary size after each chat line. Fig. 5a shows a plot of this with minimum, maximum, median, and quartiles. The ridges in the plot stem from files in the dataset with less than 2,000 lines (mostly the Ubuntu dataset).

We can see that the dictionary size was still growing almost linearly for all files, which shows that the dictionary was not "saturated", and there would still be potential to gain by learning new chunks.



(a) *Average* AMPhitryon dictionary size by line number (chunk size 10)



(b) AMPhitryon dictionary sizes by line number for a large file

Fig. 5: Dictionary Sizes

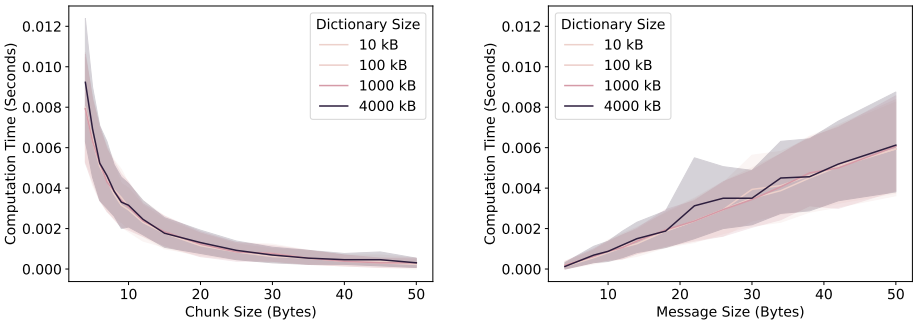
There is an inbuilt limit on the dictionary size at 8 MB (due to the 23-bit pointer). In our tests, as shown in Fig. 5a we did not encounter this limit, but it would be easy to implement a rotation where the oldest chunks are replaced in favor of newer ones.

We also tested the dictionary size with a single large file by combining all logs of the Ubuntu dataset into a single file. From Fig. 5b we can see that the dictionary size is growing roughly linearly for larger chunk sizes. Smaller chunk sizes do show a clear "saturation" effect as the dictionaries grow slower and slower. This is highlighted in the zoomed-in section of the plot.

4.4 Computation Time

We also evaluated the computation times of AMPhitryon when performing compressions with various configurations. We compressed messages with lengths

from 4 to 50 B and chunk sizes ranging from 4 to 50 B. For these compressions, we used predefined dictionaries of different sizes, from 10,000 B to 4,000,000 B. The results presented in Fig. 6 show the actual computation time, including filesystem access (writing the compressed output to file) and other overhead.[7] The experiments were conducted on a MacBook Pro (M3 chip) running macOS 15.7.



(a) Computation time vs. chunk size (b) Computation time vs. message size

Fig. 6: Computation times for AMPhitryon compression

Fig. 6a shows that the computation time decreases rapidly with increasing chunk size. This is expected, as with larger chunks, AMPhitryon has to perform fewer dictionary searches. Fig. 6b reveals that the computation time increases linearly with the message size. This shows the computational efficiency of AMPhitryon. The results also show that the dictionary size does not have a significant impact on computation times.

4.5 Limitations

A limitation of AMPhitryon is its dependence on redundancy in the payload. If payload is encrypted *before* being provided to AMPhitryon, such redundancies cannot be found efficiently and the approach becomes ineffective. As mentioned, AMPhitryon is tailored to process data *before* encryption at the sender side and *after* decryption at the receiver side.

5 Conclusions

Limiting the footprints of censorship circumvention channels and other types of covert channels is crucial to prevent these channels from being detected and protect the users from possible ramifications, such as legal action risks. In this

[7] If compression outputs were written to memory, performance would improve further.

paper, we introduced AMPhitryon, a novel compression approach for small data chunks typically transferred through covert channels in censorship circumvention scenarios. The core idea of our method is to refer to data of a given context (e.g. chat messages) while building a compression dictionary on the fly.

Using datasets of real chat messages, our evaluation shows that AMPhitryon significantly outperforms the classic compression methods `gzip`, `zstd`, and `brotli` with small chat messages. AMPhitryon achieves a respectable mean compression ratio of 0.67, while the mean compression ratios of all other techniques exceed 1.0 (which means on average message sizes *increased*).

Our approach is generic because it does not rely on specific channel types, i.e., it does not matter whether the circumvention channel is a cryptographic or steganographic channel as long as the provided payload features some form of redundancy *before* being processed by AMPhitryon and encrypted afterwards.

While we focus on censorship/covert circumvention as a scenario, one could apply AMPhitryon to other scenarios as well. Potentially AMPhitryon could even be applied outside of the security context. For example, it may be used for the compression of sensor data in low-bandwidth environments before data transmission.

In future work, we plan to integrate and evaluate our approach with popular censorship circumvention tools. We also aim to investigate whether even higher compression rates can be achieved with a flexible pointer size.

References

1. Alakuijala, J., Farruggia, A., Ferragina, P., Kliuchnikov, E., Obryk, R., Szabadka, Z., Vandevenne, L.: Brotli: A general-purpose data compressor. ACM Transactions on Information Systems (TOIS) **37**(1), 1–30 (2018)
2. Evsutin, O., Kokurina, A., Meshcheryakov, R.V.: Approach to the selection of the best cover image for information embedding in JPEG images based on the principles of the optimality. J. Decis. Syst. **27**(Supp), 256–264 (2018)
3. Freedom House: Freedom of the net report (2025), `https://freedomhouse.org/sites/default/files/2025-11/Freedom_on_the_Net_2025_Digital.pdf`
4. Fridrich, J.: Steganography in Digital Media: Principles, Algorithms, and Applications. Cambridge University Press (2009)
5. Gouravajhala, S.R., Vernier, A.M., Shi, Y., Li, Z., Ackerman, M., Kummerfeld, J.K.: Chat disentanglement: Data for new domains and methods for more accurate annotation. In: Proc. 21st Annual Workshop of the Australasian Language Technology Association (2023)
6. Keller, J., Litzinger, S.: Hiding information in short network packets. In: Proc. 9th European Interdisciplinary Cybersecurity Conference (EICC 2025). p. 133–150 (2025). `https://doi.org/10.1007/978-3-031-94855-8_9`
7. Kharrazi, M., Sencar, H.T., Memon, N.D.: Cover selection for steganographic embedding. In: Proc. ICIP. pp. 117–120. IEEE (2006)
8. Kummerfeld, J.K., Gouravajhala, S.R., Peper, J., Athreya, V., Gunasekara, C., Ganhotra, J., Patel, S.S., Polymenakos, L., Lasecki, W.S.: A large-scale corpus for conversation disentanglement. In: Proc. 57th Annual Meeting of the Assoc. for Comput. Ling. pp. 3846–3856 (2019). `https://doi.org/10.18653/v1/P19-1374`

9. Masti, R.J., Rai, D., Ranganathan, A., Müller, C., Thiele, L., Capkun, S.: Thermal covert channels on multi-core platforms. In: 24th USENIX Security Symposium. pp. 865–880. USENIX Association, Washington, D.C. (Aug 2015)
10. Meta Platforms, Inc.: Zstandard, https://facebook.github.io/zstd/
11. Meyers, B.S., Munaiah, N., Prud'hommeaux, E., Meneely, A., Alm, C.O., Wolff, J., Murukannaiah, P.K.: Chromium conversations (Jul 2018). https://doi.org/10.5281/zenodo.2590548, https://doi.org/10.5281/zenodo.2590548
12. Roy, S., Maity, D.K., Mukhopadhyay, A.: A lossless reference-free sequence compression algorithm leveraging grammatical, statistical, and substitution rules. Briefings Funct. Genomics **24** (2025). https://doi.org/10.1093/bfgp/elae050
13. Salomon, D.: Data compression: the complete reference. Springer, 4 edn. (2007)
14. Sayood, K.: Introduction to Data Compression. Morgan Kaufmann Publishers, 2 edn. (2000)
15. Wails, R., Sullivan, G.A., Sherr, M., Jansen, R.: On precisely detecting censorship circumvention in real-world networks. In: Annual Network and Distributed System Security Symposium, NDSS 2024. The Internet Society (2024). https://doi.org/10.14722/ndss.2024.23394
16. Weissenborn, C., Wendzel, S.: Silence speaks volumes: A new paradigm for covert communication via history timing patterns. arXiv (2025). https://doi.org/10.48550/arXiv.2511.22259
17. Wendzel, S., Schmidbauer, T., Zillien, S., Keller, J.: DYST (Did you see that?): An amplified covert channel that points to previously seen data. IEEE Trans. Dependable and Secure Comput. (TDSC) (2024). https://doi.org/10.1109/TDSC.2024.3410679
18. Wendzel, S., Volpert, S., Zillien, S., Lenz, J., Rünz, P., Caviglione, L.: A survey of internet censorship and its measurement: Methodology, trends, and challenges. Computers & Security (2025). https://doi.org/10.1016/j.cose.2025.104732
19. Wikipedia: Amphitryon (2025), https://en.wikipedia.org/wiki/Amphitryon
20. Zander, S., Armitage, G., Branch, P.: Reliable Transmission Over Covert Channels in First Person Shooter Multiplayer Games. In: Proceedings of 34th Annual IEEE Conference on Local Computer Networks (LCN) (October 2009)
21. Zander, S., Armitage, G., Branch, P.: A survey of covert channels and countermeasures in computer network protocols. EEE Commun. Surv. Tutorials **9**(3) (2007)
22. Zillien, S., Schmidbauer, T., Kubek, M., Keller, J., Wendzel, S.: Look what's there! utilizing the internet's existing data for censorship circumvention with OPPRESSION. In: Proc. 19th ACM ASIA Conf. Computer and Communications Security (AsiaCCS). ACM (2024). https://doi.org/10.1145/3634737.3637676