December 2022

REAL-TIME COMPRESSION OF SOFTWARE TRACES

Clinton Grant

REAL-TIME COMPRESSION OF SOFTWARE TRACES

AUTHOR:
Clinton Grant

ABSTRACT

Techniques are presented herein that support the compression of software-generated traces as a stream, in real time, with reduced central processing unit (CPU) overhead. Such an approach may reduce cloud hosting bandwidth charges and is relevant when moving troubleshooting information from a device into the cloud for analysis. Additionally, such an approach eliminates the bursty nature of file-based compression that is typically achieved using legacy compression utilities. As a result, the presented techniques are more amenable to small CPU footprints such as, for example, a cloud-based router having just a single CPU. Aspects of the presented techniques have a broad scope and may be applied to any software system that generates traces, which is typically all modern software systems. Further aspects of the presented techniques may potentially be applied to industry technologies (such as OpenTelemetry) that support the distributed tracing of cloud hosted applications.

DETAILED DESCRIPTION

Techniques are presented herein that support the real-time compression of software traces in a distributed software system with a focus on the compression of such traces in a streaming manner. For simplicity of exposition the presented techniques will be described and illustrated in the following narrative in connection with networking devices. However, it is important to note that the approach of the presented techniques applies broadly to all software systems, such as cloud-hosted software, and not specifically just to networking devices.

Within networking devices, device vendors aim to support a high rate of tracing to capture very granular troubleshooting information, frequently in excess of one million traces per second. With that perspective, the primary focus is on performance in the compute tradeoff with higher compression ratios. The techniques presented herein may be applied to traces that are written to persistent storage or that are being streamed over a network. Additionally, the presented techniques can achieve a trace compression ratio of two to four times, yielding 1/2 to 1/4 the size of the original data. The ability to stream traces off of a network device becomes important in retaining a larger troubleshooting history. Since the export of traces is a real-time stream, the compression must also operate in a streaming manner. Many of the existing compression libraries and utilities cannot support streaming since they operate on large windows of data that are typically presented as files.

An additional problem that is addressed by the presented techniques is the reduction in bandwidth that is achieved for cloud hosting. It is well known that cloud providers enforce egress bandwidth charges. As vendors look to host more cloud software, and even looking to the decomposition of existing systems for hosting in the cloud, bandwidth preservation will be important. In particular, exporting traces to cloud hosted analysis engines will confront this challenge. An analogy would be to the reason Internet Protocol (IP) header compression (IPHC) exists, where in this case tracing-specific compression may be employed to exploit the unique knowledge that is had regarding the format and the nature of the metadata that is to be captured along with the trace point code specific information.

Recent network equipment vendor operating system (OS) releases have introduced unified tracing (UT) as an improvement over the existing binary tracing (Btrace) distributed tracing architecture. UT provides a trace streaming model where all of the traces that are generated at the different processes (running on all of the field-replaceable units (FRUs)) are streamed and temporally ordered at a main route processor card. Additionally, such an aggregate streaming model enables the export of the UT messages (UTMs) off of a network device and on to an external collector.

Currently, file-based compression, employing zlib in default compression mode, is employed to compress a UT file (UTF) tracelog file once the file is filled to the nominal 20 megabyte (MB) limit. The compression overhead is incurred in a Btrace Manager (BTMAN) daemon (as will be described and illustrated below in connection with Figure 2). A thread is spawned for that workload, and approximately a factor of 10 compression is achieved. However, there is no concept of stream compression.

There are a number of reasons to pursue the application-specific compression of UTM traces, several of which are described below.

A first reason encompasses stream compression. Since all of the UTMs that are exported from a device will be streamed, there is no file-by-file transfer where existing compression utilities such as zlib may be used. A collection facility may be designed to receive a stream of UTMs from multiple connected devices. Application-specific stream compression can significantly reduce the ingress bandwidth at the collection facility, which for a cloud hosted collection facility may translate into dollar savings.

A second reason encompasses compression CPU overhead. Typically, the network device hardware platforms are constrained by CPU resources when compared with, for example, disk resources. At the high end, certain wireless hardware is equipped with 240 gigabytes (GB) of disk of which only approximately 10GB is used for tracelogs. Adding in the extra processing that is required from UTM and UTF is a challenge at wireless scale, so evolving to an application-specific compression scheme frees up CPU resources at the system level.

A third reason encompasses faster decompression. A trace-specific decompression integrates natively into the Btrace decoder with minimal overhead, whereas the current gzip file-based decompression consumes up to 30% of the decoder before the traces can be filtered and rendered.

The benefits of application-specific compression stem from the fact that the most expensive part of any compression algorithm is the need to find the redundancy in some window in the realm of a Lempel-Ziv (LZ) style compressor. The efficiency of this search differs and represents a tradeoff between how hard to look for redundancy, which results in better compression ratios, versus the computing that is required to perform the more extensive searching.

Application-specific compression is the technique that lies behind Joint Photographic Experts Group (JPEG) image and Moving Picture Experts Group (MPEG) video compression where domain-specific knowledge yields results that are better than that which are possible with a non-domain specific generic compressor.

A key to the techniques presented herein (as introduced above and as will be described and illustrated below) is the exploitation of redundancy in the trace generation that occurs at the execution context using a delta encoding strategy. There is a great deal of redundancy which only increases with the density of tracing. Expanding on the analogy of IP header compression (as described above), UTMs that are produced by an OS may be treated as 'packets' with the header format that is presented in Table 1, below.

Table 1: Header Format

UTM Header	Trace Header	Trace Payload

An entire UTM may be up to eight kilobytes (kBs) in length.

A UTM header may carry different common parameters such as, for example, a timestamp, a length, etc.

A trace header may describe the various message types, including:

- A binary encoded message header. This type captures software trace point information and the trace payload field holds the arguments that were supplied to printf() such as format specifiers (e.g., %s, %d %u, etc.).
- A packet capture (PCAP) message header. This type enables the trace payload to carry a network packet, which may be interleaved with the traces. An OS-based trace decoder may include a packet dissection engine.
- A structured data header. This type allows software objects to be captured as the trace payload using an interface definition language (IDL) after serialization.

For purposes of illustration, a portion of an exemplary trace from a network switch is provided below:

```
2022/08/21 11:31:41.352784018 {fed_F0-0}{1}: [xcvr] [13228:29670]: UUID: 0, ra: 0
(debug): xcvr poll:Able to get Active PHY device lpn
2022/08/21 11:31:41.352785838 {fed_F0-0}{1}: [devobj] [13228:29670]: UUID: 0, ra: 0
(debug): DOM: READ_EPHY_REG COPPER_MII_STATUS
2022/08/21 11:31:41.352786143 {fed_F0-0}{1}: [devobj] [13228:29670]: UUID: 0, ra: 0
(debug): 54182:Rd-Mode:0, Page:0, Reg:1
2022/08/21 11:31:41.352786687 {fed_F0-0}{1}: [xcvr] [13228:29670]: UUID: 0, ra: 0
(debug): Read ephy hw reg: Entered
```

Modern tracing systems often capture a range of information. A first type of such information encompasses trace header particulars and may consist of, for example a timestamp; a trace length (in bytes); a trace identifier; metadata for a source file and line number that is producing the trace; a process name (e.g., in the above exemplary trace - "fed"); a process ID; a thread ID (e.g., in the above exemplary trace – “PID:ThreadID = 13228:29670”); a location such as an FRU, slot, bay, and chassis number (e.g., in the above exemplary trace - the FRU is F0, slot is 0, bay is 0, and chassis number is 1); a software module name or identifier (e.g., in the above exemplary trace the modules include 'xcvr' and 'devobj'); and a trace level such as ERROR, WARN, INFO, and DEBUG (e.g., in the above exemplary trace the level is DEBUG).

It is important to note that the above-described density is typically defined by the level that is set by some configuration. As the level of trace verbosity increases, such as in a DEBUG mode, the volume of traces increases dramatically and, as has been observed, so too does the redundancy across the traces. As expected, each process, software module, and even function will emit more traces.

A second type of information often captured by modern tracing systems encompasses a trace message's payload and may include a trace point specific message. This is typically string data and/or numeric data such as integers and floats for binary encoded messages.

As background, the binary tracing architecture that has been implemented in some OSs captures the arguments that were supplied to printf() like format specifiers (such as %s, %d %u, etc.) along with proprietary specifiers for media access control (MAC) and IP addresses. The integer arguments are limited in size, though not in number. Hence it can be expected that strings that are captured using %s arguments will be present in larger traces.

The UTM compression strategy that is supported by the techniques presented herein encompasses a multistage approach. A first stage, employing a delta-encoding strategy, focuses on the compression of the fixed UTM header and the fixed headers for trace encoded messages and application context messages. For the trace encoded messages, a second stage may be applied to the payload or the trace arguments of the binary form, considering strategies that check for duplicate arguments, or applying Huffman or Lempel-Ziv-Welch (LZW) encoding to achieve further compression. To emphasize, in the tradeoff between CPU consumption versus compression ratio the intention is to favor CPU performance.

To help illustrate the issue of ingress bandwidth, Figure 1, below, presents elements of an arrangement under which a collection facility is collecting traces from multiple devices.

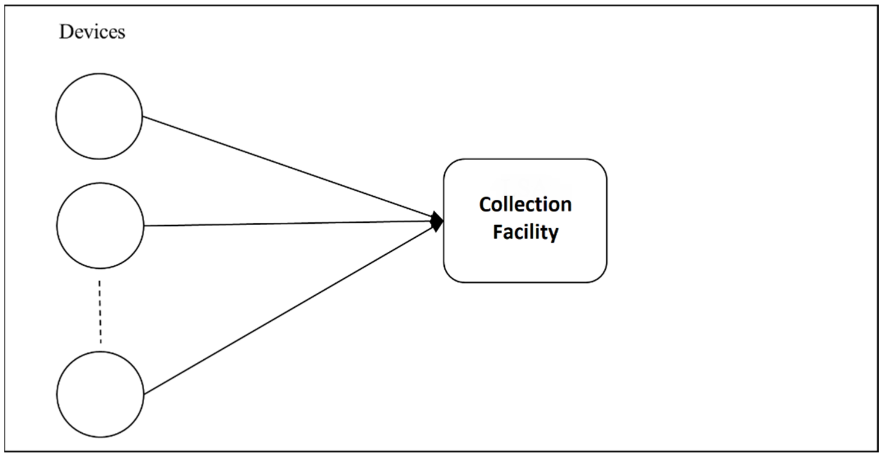

Figure 1: Devices Connected to Collection Facility

As depicted in Figure 1, above, the accumulated bandwidth at the collection facility grows rapidly. For example, if 50 devices are exporting traces each having a size of 200 bytes, an estimate of the aggregate bandwidth at the collection facility is shown in the fourth column of Table 2, below.

Table 2: Aggregate Bandwidth at Collection Facility for Various Device Trace Rates

Device Trace Rate (traces/sec)	Device Egress Bandwidth (Mbits/sec)	Aggregate Rate @ Collection Facility (traces/sec) for 50 devices	Aggregate Bandwidth @ Collection Facility	History @ 24hrs (Tbytes)
1k	1.6	50k	81.9 Mbits/sec	0.885
10k	16	500k	819 Mbits/sec	8.85
100k	160	5M	8.19 Gbits/sec	88.5

To help illustrate an architecture that is possible according to the techniques presented herein, Figure 2, below, presents elements of one exemplary arrangement.

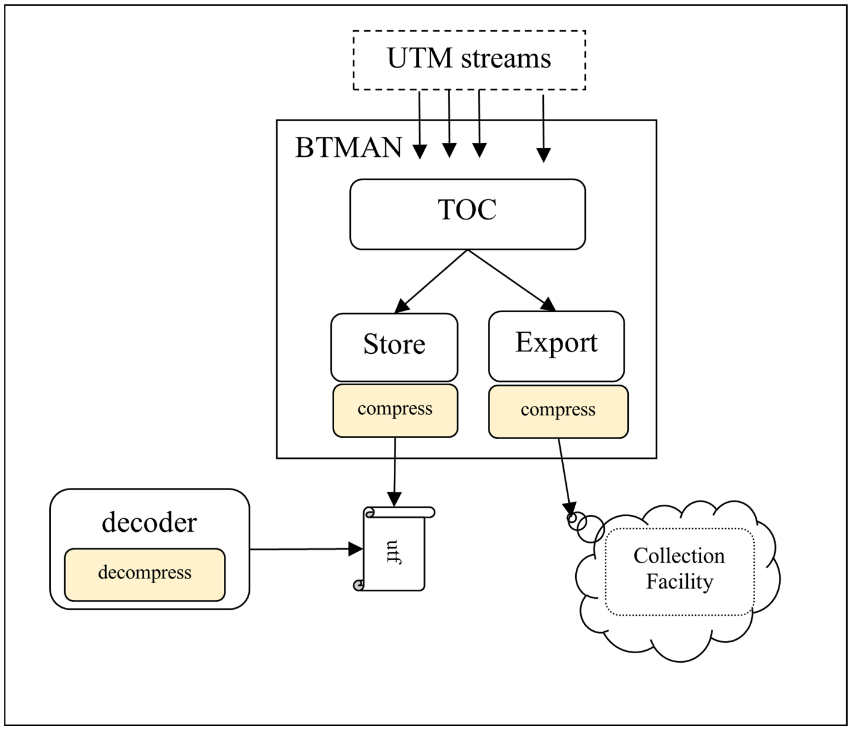

Figure 2. UT System Architecture with Compression Blocks

As depicted in Figure 2, above, UTM compression (according to aspects of the techniques presented herein) may be in placed within a UT architecture, in relation to a BTMAN temporal ordering and coalescing (TOC) module. In the context of the UTM stream consumers, UTM compression may be applied by a UTF library, when writing the traces to files, and it may be applied by an export UTM consumer that sends the UTM stream off-box to a collection facility.

In the case of compressing UTMs for streaming export off of a device, periodically an uncompressed UTM may be emitted as a beacon into the stream to improve reliability, with the expectation that the underlying transport is Transmission Control Protocol (TCP). Of course, when writing compressed UTMs to UTF files the first message that is written to each file must be uncompressed to bootstrap the delta decoding.

The narrative that follows explores a UTM trace compression algorithm and examines the compression ratio versus the performance trade-offs that may be achieved through the use of aspects of the techniques presented herein. Among other things, tabulated results will be presented from several data sets covering routing and wireless use cases. The wireless data sets include active and standby devices. Within the below results, a compression ratio is defined as:

Compression ratio = raw bytes / compressed bytes

for all UTM message types and for trace header and payload data.

Table 3, below, presents the results of UTM header compression (not including LZW payload compression which will be covered in more detail in Table9, below).

Table 3: Exemplary Compression Results

No. of Traces / No. of Files	Compression Ratio
34,919,910 / 173	**4.2** (0.238)
26,668,163 / 173	**2.48** (0.403)
24,880,771 / 166	**2.51** (0.398)
1,251,315 / 12	**1.88** (0.532)
17,072,083 / 173	**1.81** (0.553)
29,899,842 / 173	**2.53** (0.396)

The next section of the instant narrative examines the BTMAN thread level performance. The involved UT implementation employs three threads – a 'main' thread (the main thread of the evLoop), a 'demux' thread (the TOC thread), and a 'utf' thread (the UTF thread that currently performs gzip compression and UTF quota management).

The involved test case includes the installation of 20 thousand (K) Open Shortest Path First (OSPF) routes on a switch from a peer router. The peer interface is shut and then not shut four times, with a 30 second (s) delay after the shut action and a 60s delay after the not shut action. The switch has the following debug elements enabled – OSPF, express forwarding, and all binary level traces set to DEBUG. For example:

```
debug ip ospf spf
debug ip ospf rib
debug ip ospf flood

debug cef all

set platform software trace all debug
```

Before UTM stream compression is enabled, Figure 3, below, demonstrates distinct spikes in the UTF thread as the zlib compression processing is performed.

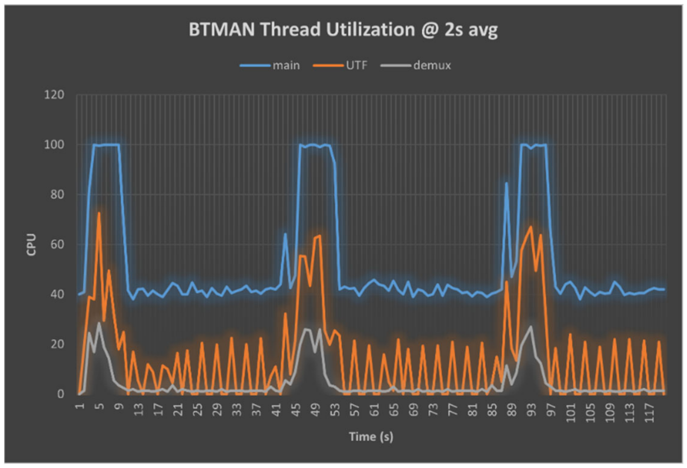

Figure 3. BTMAN Thread Utilization Using gzip Compression

Figure 3, above, illustrates the impact of the gzip compression when generating traces at scale. The orange line for the UTF thread shows the impact, here it peaks above 70% of CPU of a single core but it can get closer to 100% as more traces are offered

When zlib compression is replaced with native UTM stream compression, Figure 4, below, illustrates that the overhead of the UTF thread is vastly reduced and the total overhead of BTMAN is reduced.

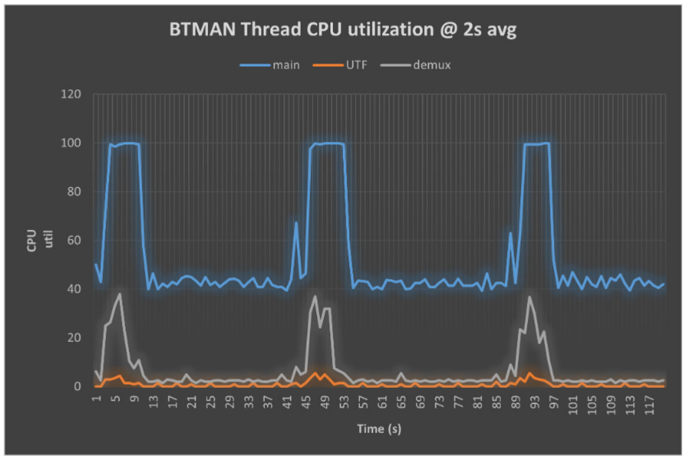

Figure 4. BTMAN Thread Utilization Using Native UTM Compression

As depicted in Figure 4, above, running the same test as above (in connection with Figure 3), but this time with UTM stream compression enabled including LZW payload compression, it can be seen that the overhead of the UTF thread is significantly reduced and the bursty nature of the compression is all but eliminated.

Figure 5, below, depicts a situation consisting of duplicate trace argument and LZW compression enabled on the trace payload data.

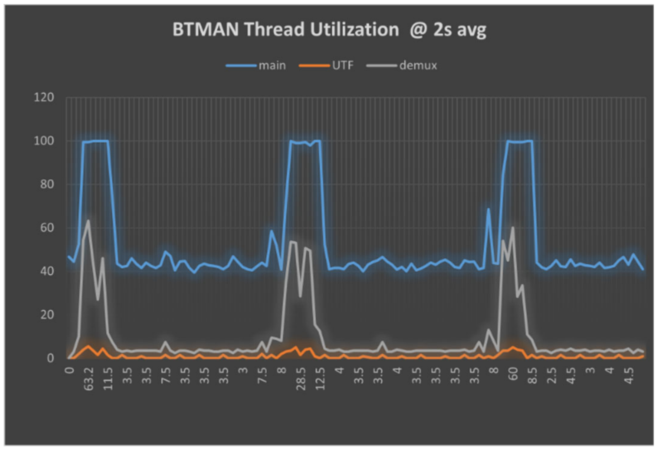

Figure 5: UTM Stream Compression with LZW

As indicated in Figure 5, above, the utilization of the demux thread increased slightly, compared with Figure 4, above, in order to achieve a better compression ratio.

A key method that is employed by the techniques presented herein is delta encoding, where a current message is compared with the global delta state that is maintained, field-by-field, for the UTM header. The premise is that a significant portion of the adjacent traces repeat common fields. For example, a burst of traces for a given process will have the same process name, the same process trace key fields, the same process identifier (PID), and for single-threaded OS daemons the same thread identifier (TID). This pattern is leveraged in the first stage of the compression that operates on the headers, collapsing repeated fields to a single bit in the UTM compressed header. The bitmap maintains a bit for each of the header fields, and whether it may be duplicated from the previous message.

Figure 6, below, depicts elements of the approach that was described above.

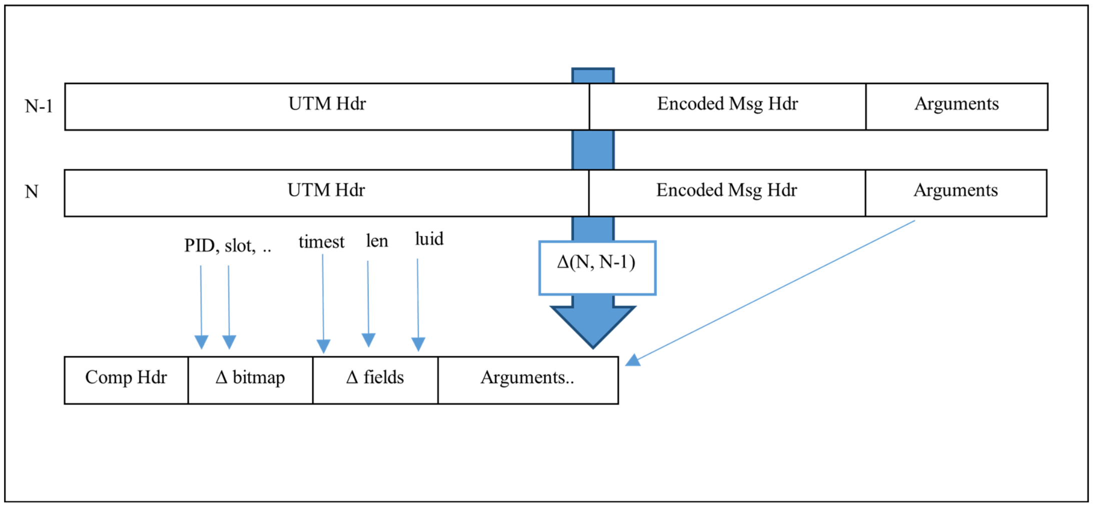

Figure 6: Delta Encoding of Fixed Headers

The delta across each of the header fields may be calculated using a variety of strategies. For all of the header fields, typically the compression strategy involves a direct comparison with the corresponding field of the previous trace. This is a given for all of the fields that are a single byte. When a delta between the fields is detected, there may be further scoping to see if the delta can be compressed. Every byte that is saved in a UTM compression header (as described below) for frequently changing fields can improve the compression ratio.

Aspects of the techniques presented herein introduce a UTM compression header. Such a header is six bytes in length, with the goal of keeping the length as short as possible so as not to rob from the compression ratio. Table 4, below, presents the fields of a UTM compression header.

Table 4: UTM Compression Header

id	0	1	2
length (bytes)	1	2	3
field	Magic/ver	Flags/length	bitmap

As indicated in Table 4, above, a one-byte version (ver) field is of the same type as the UTM version/magic field. The UTM version/magic value is currently 0xA2, where the UTM header version is 2. For the compressed header, it may be set to 0xB2 where the magic increments to 0xB for the header version 2.

As further indicated in Table 4, above, a two-byte flags/length field may encompass (1) a three bit flag value (e.g., FLAG_DUP_ARG, FLAG_ARG_HUFFMAN, and FLAG_ARG_LZW) and (2) a 13 bit length value (supporting a maximum size of 8K). The size of the length value may be reduced to 12 bits if a 4K compressed message size can be achieved.

A three-byte bitmap field, as indicated in Table 4, above, records the UTM header fields which have a delta relative to the previous message, where the bit corresponds to the field identifier. If the bit is 0, then the field in the current message is identical to the previous message so it can be reconstructed by copying the previous message value. If the bit is 1, then it means that the field value is different and only the delta has been recorded. To reconstruct the value, the delta may be applied to the previous UTM message field.

According to the techniques presented herein, different encoding strategies may be employed for each UTM field based on a field's characteristics. Several possible strategies are described below.

Under a pure delta strategy, the delta for UTM field F is the difference between the current value and the value in the previous UTM message. That is, the delta is F(current) – F(previous). This strategy only works for monotonically increasing fields such as a timestamp. Otherwise, if the delta is a negative value no bytes will be saved.

A zigzag encoding strategy allows for the encoding of negative delta values, in a positive integer, yet still reduces the underlying number of bytes that are required to store the information. Without such a scheme, a negative delta of a 64-bit quantity would result in a negative 64-bit value with the sign extension to the upper most bits, achieving no byte savings. Zigzag encoding may be applied to a message length to reduce such a value from two bytes to one byte of storage. Such an approach may also be applied to a four byte PID field, reducing it to two bytes.

A comparison strategy takes into account the fact that there are many fields which cannot be compressed using delta encoding. Under such an approach, if a difference is detected between the current value and a previous UTM value then compression may be abandoned and the new value may be retained and allowed to flow into the stream. Similarly, this strategy may be applied to most of the remaining fields which are already one byte in size. This strategy is possible since a trace-by-trace compression scheme is employed where compressed and uncompressed traces may be interleaved. For example, if the delta between adjacent trace timestamps exceeds four bytes in resolution then the compression may be abandoned and the original trace may pass into the stream. Compression is an optimization game, and it is not worth sacrificing bytes in the compression header or common fields to capture a compression of 100% of traces.

Aspects of the above discussion may be further explicated with reference to an example encompassing a delta compression of a timestamp value. Under the example, it may be expected that an 8-byte timestamp field will be monotonically increasing. The compression algorithm should maintain the delta state for the last field recorded, so for the timestamp field it is only necessary to encode the delta since the last timestamp. This provides the possibility of reducing the timestamp information that is recorded to just four bytes per message. It is important to note that there is the possibility that the delta may be more than four bytes. In such a case the original eight byte field of the timestamp may be preserved.

In the following snippet that is taken from a switch:

```
2022/05/02 03:09:58.821845210 {cmand_R0-0}{1}: [poe] [6127:16561]: UUID: 0, ra: 0
(ERR): tdl handle is null in add port health record
2022/05/02 03:09:58.841494934 {cmand_R0-0}{1}: [poe] [6127:16561]: UUID: 0, ra: 0
(ERR): tdl handle is null in add port health record
2022/05/02 03:09:58.861132409 {cmand_R0-0}{1}: [poe] [6127:16561]: UUID: 0, ra: 0
(ERR): tdl handle is null in add port health record
```

repeated logs from the cmand process are seen which are identical and differ only in their timestamp value. In the above, a first UTM timestamp (e.g., UTM(i)) encompasses 0x da 18 30 53 07 2b eb 16 (or u64 0x16eb2b07533018da) and a second UTM timestamp (e.g., UTM(i+1)) encompasses 0x 96 ed 5b 54 07 2b eb 16 (or u64 0x16eb2b07545bed96),
yielding a delta value of 0x12BD4BC.

Such a delta value may be encoded as a UTM compressed (UTMc) message of nine bytes in size:

0xb2 0x09 0x10 0x00 0xbc 0xd4 0x2b 0x01

In the instant example, if the message was a repeat, with the only difference being in the timestamp, the compression reduces the original message of 114 bytes down to nine bytes. Since the compression is message-by-message, the compression is indicated by changing the first nibble ('magic') of the message from A to B, retaining the version nibble (which is currently version 2). This is an example of best-case compression.

As may be expected, delta decompression is an inverse operation, based on the previous message. Continuing with the above example, all of the fields may be copied over from the previous message based on the bit field. Every message would need its timestamp adjusted. After the message copy, the timestamp delta may be applied by adding 0x12BD4BC as an unsigned 64-bit (u64) operation.

When attempting to store a delta field in the compression field, there is a chance that it may overflow the predetermined compressed size. In the case of timestamp delta compression, this can occur if the 32-bit size is exceeded. It may also occur with a PID value and a message length. As long as the proportion of messages that are affected is very small (i.e., less than one percent), then the original UTM trace with a magic/version byte of 0xA2 may be written in uncompressed form. A UTM decompressor may handle both compressed and uncompressed UTMs, behaving as something of a reset.

Currently, UTM traces carry a process name in clear text consuming a maximum of 16 bytes. With the delta compression technique (as described above), it is possible to compress repeated names to a single bit. In a case where it is not possible to reuse the process name from a previous UTM, there are a number of options. First, all 16 bytes of the new process name may be captured. Second, a null terminated process name may be captured as a variable length field in a compressed message (which may benefit from the fact that process names tend to be short – dbm, sman, wncd, etc.). Third, a captured name may be compressed using Huffman encoding and preserved as a variable length field in a compressed message.

Based on sample data from an exemplary network, it is possible to see the proportion of traces where process name changes are small, in this case approximately eight percent of the traces:

```
field 10 (locn_luid_proc_name), strategy (compare), delta_count: 1343278,
proportion: 0.078
```

When the average trace size is approximately 212 bytes, if it were possible to save eight bytes on this proportion the impact on the compression ratio would be minimal.

After handling the compression of a UTM and any encoded message (as described above), the next frontier is a compression of the trace arguments that are associated with the encoded message format strings. In the first example:

```
" Failed to open directory: %s err: %s"

" New active fru: %d, slot: %d bay: %d chassis: %d"
```

the two string specifiers used in the format string will be concatenated in the trace arguments, while the second will have the four integers concatenated. The trace arguments will comprise a concatenation of numbers of various sizes (8, 16, 32, and 64 bits and strings) and structure argument types for MAC and IP addresses. There are a number of strategies that may be investigated for the compression of the such UTM traces arguments and several candidate strategies will be described below.

A first strategy encompasses Huffman encoding. Such an approach is effective when there are frequently used characters. For example, it works well for English text. A message may be thought of as a sequence of symbols, with each symbol drawn independently from some known distribution.

A second strategy encompasses Run Length Encoding (RLE). Like Huffman encoding, RLE mostly take advantages of frequent or repeated single characters. Elias gamma codes are an example.

A third strategy encompasses LZW. LZ comprises a family of adaptive compression algorithms where repeated parts of the input are stored by reference. Each code word may refer to a single character in the input or to a substring. The algorithm does not assume any a priori knowledge of the symbol probabilities. For historical context, LZ77 and LZ78 are original versions (based on the concept of a sliding window providing the DEFLATE derivative) and are used in gzip, (pk)zip, and Portable Network Graphics (PNG) files while LZW derivatives are used in the Unix compress utility and graphics interchange format (GIF) image encoding.

A fourth strategy encompasses transforms. These can help organize the data in a way that improves the compression. Two that have been considered include Move-to-Front (MTF) and Burrows-Wheeler-Transform (BWT), the latter being utilized to great effect in bzip2 compression.

The next section of the instant narrative summarizes different run analyses that were completed in connection with validating aspects of the techniques presented herein.

A first run analysis considered runs of zeros and was based on a suspicion that the arguments contain many zero bytes. The traces and bug information that were generated from the OS emit all integers as uint64_t data, regardless of the original precision. For runs of zero length, data from an example wireless network, listed in Table 5, below, show that most of the runs are less than eight bytes, although the largest run of zeros that was detected was 48 bytes.

Table 5: Exemplary Run Lengths

Run Length (bytes)	Proportion
1-7	97.8%
8-15	1.8%
16-31	0.3%

A second run analysis considered American Standard Code for Information Interchange (ASCII) printable characters. The premise in looking for runs of printable characters in the argument strings was to see if Huffman encoding could be applied to reduce the size of these arguments. The letter frequency of ASCII printable characters may be used to construct a Huffman encoding dictionary. Huffman compression might then yield a 60% to 70% compression for just those runs.

Searching for the largest run-length of printable characters is an O(n) operation, though not particularly general. It would also require new headers to be embedded in the argument payload that is captured in the UTM compressed message form to mark the start and length of the compressed runs. That is, it would need be managed within the space of all of the arguments that are being traced, and any additional headers would reduce the compression efficiency.

An observation is that OS-only bug information and debug artifacts are very string dependent. This can be seen in the argument lengths of printable characters. It is a common pattern in an OS to use snprintf() before tracing or logging data. Consider an additional debug dataset:

```
        Processed 1251315 messages!
…
        Printable Arg Max Run Length Histogram:
        bin[     0-7] : 517348, proportion: 0.413443
        bin[    8-15] : 596005, proportion: 0.476303
        bin[   16-31] : 81402, proportion: 0.065053
        bin[   32-47] : 21275, proportion: 0.017002
        bin[   48-63] : 733, proportion: 0.000586
        bin[   64-79] : 104, proportion: 0.000083
        bin[   80-95] : 20, proportion: 0.000016
        bin[  96-111] : 7, proportion: 0.000006
        bin[ 112-127] : 34143, proportion: 0.027286
        bin[ 128-159] : 0, proportion: 0.000000
        bin[ 160-191] : 36, proportion: 0.000029
        bin[ 192-223] : 2, proportion: 0.000002
        bin[ 224-255] : 4, proportion: 0.000003
        bin[ 256+   ] : 0, proportion: 0.000000
```

16 6815

Looking at the maximum run length of trace arguments for encoded messages where there are printable characters (i.e., the trace logging a string via a %s format specifier) it can be seen that approximately 47% of the runs are in the range of 8 to 15 characters.

Regarding Huffman encoding (the first strategy that was noted above), various Huffman dictionaries have been tried, including those constructed from the New York Times (NYT) corpus giving plain letter frequencies, though it must be recognized that the strings being logged are not NYT English. Other Huffman dictionaries were specifically built from the data sets of trace arguments being compressed.

In summary, the results were not so promising. A general purpose Huffman dictionary that was constructed form the NYT corpus yielded low compression efficiency. A Huffman dictionary that was built for all of the trace arguments having a maximum run length of printable ASCII characters greater than or equal to 16 yielded negligible results for four data sets

In summary, the strength of Huffman encoding arises when it is combined with other strategies. It can achieve a 60% compression on plain English text, but in practice it is rarely used directly.

Regarding RLE (the second strategy that was noted above), it is simple and fast. It can compress N bits to O(log N) in a best case. No compression takes place until a run length is greater than six, and expansion is possible for a run of length two or six. Such an approach may be bad for text, but is good on long runs (e.g., pictures).

Regarding LZW (the third strategy that was noted above), Huffman encoding and RLE take advantage of frequent or repeated single characters. In contrast, LZ refers to a family of adaptive compression algorithms. The premise is that certain substrings are more frequent than others, so the approach is to store repeated parts by reference. Variants of LZ compression are employed in LZ77 (an original version) and DEFLATE is used in, for example, (pk)zip, gzip, and PNG.

LZW offers relatively fast encoding and decoding, and importantly can work in a streaming model. Compared with Huffman encoding it requires no prior information about the input data stream, meaning it can compress the input stream in one single pass. In the case of trace argument compression, it needs to work with a small input set since it is applied to one message at a time. If an average message is 212 bytes, then the arguments are on average 100 bytes.

The LZW dictionary size needs to be considered – typically this can be 12-bit entries, prefilled for the first 256 byte values. However, this may be excessive for UTM argument values. Consideration is therefore given to 8-bit or 9-bit entries, with the dictionary preloaded with 7-bit values prefilled from 0x0 to 0x7F. In theory this provides for 0x00 – 0x7f covering all of the printable ASCII characters, 0x80 – 0xff covering ASCII substrings,

An 8-bit entry allows for 127 substrings or unseen bytes. A 9-bit entry allows for 256 such substrings, but runs the risk of failing to compress well, if at all. To guide this strategy, the byte distribution in the UTM encoded arguments may be examined. For a number of sample images, Table 6, below, demonstrates that the majority of the byte values are less than 128.

Table 6: Byte Distribution for UTM Encoded Message Payload

Byte range (inclusive)	Proportion
0..63	0.388356
0..127	0.989477
128..256	0.010523

The distribution that was indicated in Table 6, above, may be attributed to several different factors, including the larger argument entries existing as ASCII strings, compared with the 64/32/16-bit arguments. Additionally, the values typically traced do not use the full precision of the integer type. Further, the OS bi-endian interface treats all integers as 64 bits, so there can be some degree of 0x00 padding.

It is compelling to think that 7-bit code words may be used due to the special nature of the input data. It is important to note that LZW also needs to encode substrings. If the input stream raw values are reserved to the range [0..127], then the control values and substring codes may exist from [128..255] and still fit within a byte. Typical LZW implementations choose code word sizes of 9-12 bits. Since tracing data has short argument lengths, averaging around 100 bytes if the average UTM is 212 bytes, the compression ratio would be impacted. Also, LZW dictionaries cannot grow unbounded and are reset or cleared as they grow to a limit. If 8-bit code words are maintained then data and argument bytes must be dealt with having a bit 7 set.

There are a number of options for the above, two of which will be described below.

Under a first option, as the arguments are scanned the encoding may be abandoned if a byte is encountered with bit 7 set. However, this might be one or two bytes per argument payload which would defeat too much of the compression. Analysis may be performed to see how many argument payloads are free of such bytes.

A second option introduces control words to delimit such argument bytes with a bit 7 set. A value is needed to indicate that the dictionary is reset. An End-of-Data (EOD) code and a Reset-Dictionary (RES) code may be employed, as follows:

Code, code, EOD, 0xFF, 0xFF…, RES, code..

It is conceivable that the output code words can be further Huffman encoded if there is some knowledge of the frequency distribution of the code words.

Transforms (the fourth strategy that was noted above) may help organize the data in a way that improves compression. Two approaches that have been considered include Move-to-Front (MTF), and Burrows-Wheeler-Transform (BWT), the latter being utilized to great effect in bzip2 compression. Given the performance requirements of the UTM compression, it is unlikely a transform can be employed since it will add another pass over the data and thus does not support streaming. BWT followed by MTF, RLE, and Huffman encoding is the algorithm that is used by the bzip2 program which achieves the best compression on English text.

Concerning a LZW design, performance is critical for any UTM argument compression. Since the compression must run for each UTM message, the implementation must avoid system(), malloc(), and free() calls. Further, it needs to run close to the BTMAN TOC offered rates. Classical implementations favor a trie approach with an LZW encoder determining the longest-prefix-match within the stream of bytes. A challenge with the trie approach is the memory management and cleanup that are required at each UTM. Other alternatives look to hash table implementations, which if implemented using a probing technique (linear or quadratic) may avoid the memory management overhead but there still remains the issue of cleanup and resetting the dictionary which is a necessity for LZW.

Considerations regarding the above include an array-based symbol table to avoid malloc() calls and the implementation of a longest prefix match with an array symbol table (where a hash table based on probing may employ quadratic probing (since linear probing is not optimal due to the primary clustering) and a trie may be employed to get the longest prefix match).

Given the points that were made above, the favored approach is a two-dimensional (2D) lookup table at the LZW encoder. The lookup key comes from the symbols that are taken from the offered alphabet with the generated code values representing prefixes. Naturally, each value from the alphabet is also a prefix.

The 2D table size is determined by the code word size. An initial implementation may start with the alphabet of 7-bit values (0-127) and 8-bit code words. Eventually it will be extended to the typical offering of 8-bit alphabet values and support for 9-,10-, and 11-bit code words, etc. With 8-bit code words, new prefix codes may be allocated from the range 128-255 making allowances for special codes to indicate DICTIONARY RESET and END of DATA/NULL values.

The above approach should be adequate given that the average UTM message size is approximately 212 bytes, putting the average argument payload around 100 bytes. Since the LZW algorithm is adaptive, the encoding dictionary can grow unbounded, so when no more code words can be allocated it is necessary to RESET, or flush, the dictionary. This action needs to be signaled to the LZW decoder so that it can maintain synchronization.

It is important to note that it is not necessary to maintain each longest matching prefix string, or byte, sequence. Each prefix only needs to record the last character (or byte) that was appended to form the new prefix.

Aspects of the techniques presented herein, as described and illustrated in the above narrative, may be further explicated through an encoding example. For the input string “**YO!_YOU!_YOUR_YOYO!**” Figure 7, below presents elements of an exemplary 2D encoding table.

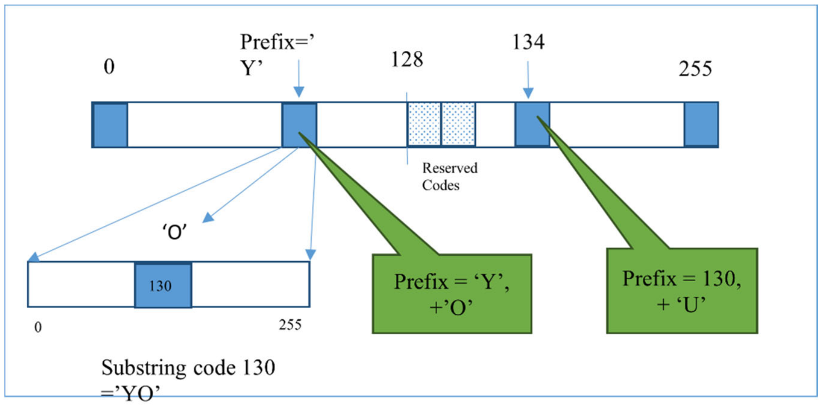

Figure 7: Exemplary 2D Encoding Table

The above allows the code table that is presented in Table 7, below, to be constructed.

Table 7: Exemplary Encoding Table

Code (ASCII)	**Character**	**Code (Generated)**	**Substring**
32	_	130	YO
33	!	131	O!
79	O	132	!_
82	R	133	_Y
85	U	134	YOU
89	Y	135	!_Y

Using the code words that are presented in Table 7, above, the compressed string becomes **89 79 33 32 130 85 132 134 82 133 79 130 33**.

Similar to the LZW encoding step (as described above), a decoder must build the dictionary in the same fashion, but in this case, it is a single dimension table that is indexed by the code word values. There is a corner case where the decoder is expected to decode a code word that is not present in the dictionary but which is in the process of being constructed since the decoder is one step behind. This case is well documented in the literature, and the solution is to take the first alphabet value of the previous code word in the stream and treat it as the next alphabet value to be appended to the previous code word to form the new code word.

For example, for the input string "**ABABABA**" and the code sequence **65 66 130 132**, the decoder dictionary may construct a table as shown in Table 8, below.

Table 8: Exemplary Encoding Table

Code (key)	**Substring (value)**
65 (ASCII)	A
66 (ASCII)	B
..	
130	AB
131	BA
132	ABA

As depicted in Table 8, above, a dictionary lookup for the code 132 fails. To build the entry for code 132, the first value of code 130 is appended to the substring for 130 to form "ABA," the new substring.

The other function that is required of the decoder is to flush the dictionary when it receives in the stream a reset code word.

The LZW design that was described above can achieve a fast reset of the encoding dictionary by having a shadow bit array, or bitmap, to represent the cells in the 2D table. A single bit may be used to mark a table entry that is in use, indicating that the code word is valid. This allows memset() calls to be avoided by clearing u64 datatypes to represent the dictionary entries.

Extending the compression results that were presented in Table 3, above, tests were run on a platform:

```
processor       : 39
vendor_id       : GenuineIntel
cpu family      : 6
model           : 62
model name      : Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz
stepping        : 4
microcode       : 0x42e
cpu MHz         : 3340.771
cache size      : 25600 KB
physical id     : 1
siblings        : 20
core id         : 12
cpu cores       : 10
apicid          : 57
initial apicid  : 57
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
```

to identify the new compression ratio that is possible with LZW enabled on the encode message argument payloads greater than or equal to 16 bytes. The results of those tests are presented in the third column of Table 9, below.

Table 9: Exemplary LZW Compression Results

No. of Traces	Compression Ratio	Compression Ratio w/ LZW	Performance
34,919,910	**4.43**	**4.58**	214.1Mbytes/sec
26,668,163	**2.50**	**2.59**	137.6 Mbytes/sec
24,880,771	**2.49**	**2.62**	127.2 Mbyte/sec
1,251,315	**1.90**	**2.00**	95.0 Mbyte/sec
17,072,083	1.81	**1.96**	80.0 Mbytes/sec 96.1 Mbytes/sec – after lzw 9b tuneup 105.6 Mbyes/sec – after moving DEBUG traces-to-NOISE
29,899,842	**2.53**	**2.56**	199 Mbytes/sec (18.22 s)

Note that for Table 9, above, duplicate argument checking was enabled for all of the test cases and run before LZW payload compression in priority.

A key observation from all of the above is that all of the code words have a fixed width, in the instant case eight bits. A knowledge of the alphabet that is to be compressed can allow for a second stage of compression where the LZW encoding is followed by a Huffman-based compression stage that maps the generated codes to Huffman codes in order to gain better bitwise efficiency.

Based on a frequency analysis of the bytes that are to be compressed, the probability distribution is similar to the plot that is presented in Figure 8, below, for the UTM encoded message payloads (arguments).

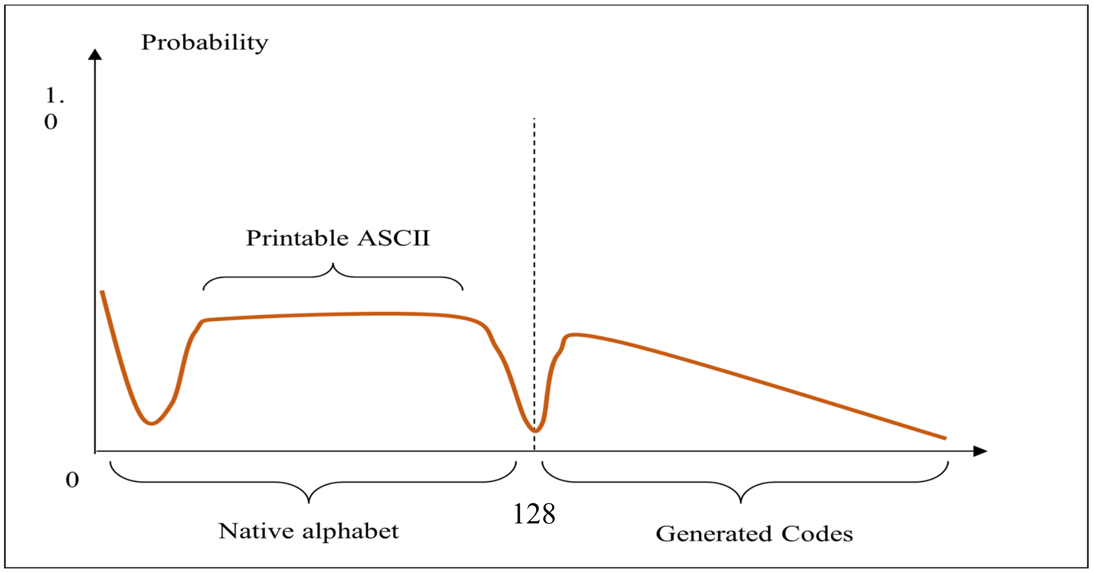

Figure 8: Probability Distribution Function for UTM Argument Code Generation

The premise is that the more frequent alphabet values (e.g., 0 through 127) may be Huffman encoded for fewer bits. That is, low number values are frequent (especially 0); printable ASCII characters from decimal 32 to 126 are more frequent, and within that set particularly SPACE (32) and the number 0 (48), and lower case 'a-z' characters are more frequent than upper case 'A-Z' characters; and the codewords from 128+ are generated in sequential order, meaning that a higher weight may be applied to those closer to 128 (i.e., the generated codewords represent the substrings already seen and hence likely to be used again).

Supporting a larger LZW dictionary of 512 entries and the generation of 9-bit codes brings the challenge of packing the 9-bit code values efficiently when compared with the 8-bit code generation.

For x86_64 platforms it is possible to leverage the Intel and AMD intrinsic instruction sets to perform the integer packing operations into the encoded stream buffer for greater performance. For example, the Advanced Vector Extensions 2 (AVX2) support 256 and 128 vector widths, in principle allowing operations on up to 16 x int16 LZW codes. To keep the single instruction, multiple data (SIMD) output to the encoded buffer stream simple, it is possible to look to common multiples of 9-bit codes that end on a byte boundary such as 72-bits (8 codes) for 128-bit operations or 144-bits (16 codes) for 256-bit operations.

As described and illustrated in the above narrative, the techniques presented herein encompass all aspects of UTM stream compression. A key point of novelty in the presented techniques is the delta encoding of trace header information. Part of the solution employs LZW and Huffman compression, which are well known components of many compression solutions.

Use of the techniques presented herein offers a number of advantages. First, compression is applied at the unit of a single trace, across the trace headers and separate from the trace payload, with different strategies and algorithms applied to the trace headers versus the trace payload. Additionally, compressed traces may be interleaved with non-compressed traces. Second, trace compression is stream based. As equipment vendors look to cloud hosting of their software, bandwidth charges that are applied by the different cloud hosting providers can be egregious. Stream compression, as supported by the techniques presented herein, reduces the tracing bandwidth between nodes.

Third, general purpose compression utilities (such as gzip, zlib, bzip, etc.), which are computationally expensive compared with trace specific compression, may be avoided. Those utilities are expensive since they must make multiple passes through the data in a given window (e.g., 32kB) to transform and then search for redundancy. By contrast, the techniques presented herein make only a single pass over the tracing data. Fourth, faster decompression performance is achieved compared with general purpose tools.

In summary, techniques have been presented herein that support the compression of software-generated traces as a stream, in real time, with reduced CPU overhead. Such an approach may reduce cloud hosting bandwidth charges and is relevant when moving troubleshooting information from a device into the cloud for analysis. Additionally, such an approach eliminates the bursty nature of file-based compression that is typically achieved using legacy compression utilities. As a result, the presented techniques are more amenable to small CPU footprints such as, for example, a cloud-based router having just a single CPU. Aspects of the presented techniques have a broad scope and may be applied to any software system that generates traces, which is typically all modern software systems. Further aspects of the presented techniques may potentially be applied to industry technologies (such as OpenTelemetry) that support the distributed tracing of cloud hosted applications.