computers

*Article*

# The Use of Template Miners and Encryption in Log Message Compression

**Péter Marjai [1], Péter Lehotay-Kéry [1] and Attila Kiss [1,2,*]**

[1] Department of Information Systems, ELTE Eötvös Loránd University, 1117 Budapest, Hungary; g7tzap@inf.elte.hu (P.M.); lkp@caesar.elte.hu (P.L.-K.)
[2] Department of Informatics, J. Selye University, 94501 Komárno, Slovakia
* Correspondence: kiss@inf.elte.hu

**Abstract:** Presently, almost every computer software produces many log messages based on events and activities during the usage of the software. These files contain valuable runtime information that can be used in a variety of applications such as anomaly detection, error prediction, template mining, and so on. Usually, the generated log messages are raw, which means they have an unstructured format. This indicates that these messages have to be parsed before data mining models can be applied. After parsing, template miners can be applied on the data to retrieve the events occurring in the log file. These events are made from two parts, the template, which is the fixed part and is the same for all instances of the same event type, and the parameter part, which varies for all the instances. To decrease the size of the log messages, we use the mined templates to build a dictionary for the events, and only store the dictionary, the event ID, and the parameter list. We use six template miners to acquire the templates namely IPLoM, LenMa, LogMine, Spell, Drain, and MoLFI. In this paper, we evaluate the compression capacity of our dictionary method with the use of these algorithms. Since parameters could be sensitive information, we also encrypt the files after compression and measure the changes in file size. We also examine the speed of the log miner algorithms. Based on our experiments, LenMa has the best compression rate with an average of 67.4%; however, because of its high runtime, we would suggest the combination of our dictionary method with IPLoM and FFX, since it is the fastest of all methods, and it has a 57.7% compression rate.

**Keywords:** log file processing; template mining; compression; encryption

check for updates

**Citation:** Marjai, P.; Lehotay-Kéry, P.; Kiss, A. The Use of Template Miners and Encryption in Log Message Compression. *Computers* **2021**, *10*, 83. https://doi.org/10.3390/computers10070083

Academic Editor: George Angelos Papadopoulos

Received: 31 May 2021
Accepted: 20 June 2021
Published: 23 June 2021

## 1. Introduction

Creating logs is a common practice in programming, which is used to store runtime information of a software system. It is carried out by the developers who insert logging statements into the source code of the applications. Since log files contain all the important information, they can be used for numerous purposes, such as outlier detection [1,2], performance monitoring [3,4], fault localization [5], office tracking [6], business model mining [7], or reliability engineering [8].

Outlier detection (also known as anomaly detection) is done by detecting unusual log messages that differ significantly from the rest of the messages, thus raising suspicion. These messages can be used to pinpoint the cause of the problem such as errors in a text, structural defects, or network intrusion. For example, a log message with high temperature values could indicate a misfunctioning ventilator. The authors of "Anomaly Detection from Log Files Using Data Mining Techniques" [1] proposed an anomaly-based approach using data mining of logs, and the overall error rates of their method were below 10%. There are three main types of anomaly detection methods, such as K-Means+ID3 (supervised) [9], DeepAnT (unsupervised) [10] or GANomally (semi-supervised) [11]. Supervised techniques work based on data sets that have been labeled "normal" and "abnormal". Unsupervised algorithms use unlabeled datasets. Semi-supervised detection creates a model that represents normal behavior [2].

With the appearance of large-scale systems, performance monitoring has become a vital task. Since these systems produce numerous log messages every minute, it is impossible to monitor the data by hand. For these reasons, various tools have been created to aid the developer tasked with the monitoring of the performance. One such tool was proposed by the authors of "Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems" [3] that uses machine learning to find the relations between the performance and different modules of the system. Another such tool has been proposed in "Personalized Ranking for Digital Libraries Based on Log Analysis" [4], where the authors analyzed user feedback from access logs and predicted user preferences.

The authors of "Scalable Offline Monitoring" [6] proposed an approach to monitor IT systems offline, where system actions are logged in a distributed file system and subsequently checked for compliance against policies formulated in an expressive temporal logic.

Log files also contain information of the preference of the users, which can be used to create business models. Such models can be later used for example to provide better deals for the customers of a webshop that generated the log files. For example, authors of "An e-customer behavior model with online analytical mining for internet marketing planning" [7] developed constructs for measuring the online movement of e-customers to identify important dimensions of their behavior and abstract changes in their behavior by developing an e-customer behavioral graph.

Automated log analysis can also be used for reliability engineering. In "A Survey on Automated Log Analysis for Reliability Engineering" [8], the authors provided a survey on this topic.

The volume of the software logs has escalated rapidly due to the evolution of computers and the wide usage of different types of software. More and more storage space is needed to store these files. The authors of "Loghub: a large collection of system log datasets towards automated log analytics" [12] provide a large collection of log datasets. To handle such datasets, compression of these files is necessary. A comprehensive comparison between the general compressing algorithms can be seen in "A study of the performance of general compressors on log files" [13].

Numerous techniques that take different aspects of log files into account have been developed to decrease the size of these files. A compression algorithm for CAN traffic log files is proposed by the authors of "Efficient lossless compression of CAN traffic logs" [14]. It uses semantic compression to retain precise information, which is needed for forensic purposes. Operational profiles, containing the most common usage scenarios of different softwares, are used to compress log files in "An industrial case study of customizing operational profiles using log compression [15]. Since creating a profile could be time- and resource-consuming, they use an approach that converts log files into log signals, which are then used to easily create profiles.

In "SEAL: Storage-efficient Causality Analysis on Enterprise Logs with Query-friendly Compression" [16], the authors created a dependency graph from the log files, which was then compressed. Hidden structures can also be used to compress log files. In "Logzip: Extracting Hidden Structures via Iterative Clustering for Log Compression" [17] the authors provide a tool that uses such items to compress log files. In "Improving State-of-the-art Compression Techniques for Log Management Tools" [18], a method is proposed by the authors that pre-processes small log blocks. After the pre-processing, a general compressor is applied on the data.

Many times, choosing the best data compressor of a given set is done by the empirical comparison of the compressors. One of the goals of "Time-universal data compression and prediction" [19] was to turn this process into a part of the compression method, automating and optimizing it.

In "Lossless Compression of Time Series Data with Generalized Deduplication" [20], the authors presented a new strategy for data deduplication to provide compressed data storage for large amounts of time series data. In their approach, they split each data chunk into a part that must be stored directly and a part worth deduplicating. This enables greater compression than the classic deduplication based on their analysis.

Shared dictionary compression has been shown to reduce data usage in pub/sub networks, but it requires manual configuration. A new dictionary maintenance algorithm by the authors of "PreDict: Predictive Dictionary Maintenance for Message Compression in Publish/Subscribe" [21] adjusts its operation over time by adapting its parameters to the message stream and enabling high compression ratios.

Learning categorical features with large vocabularies has become a challenge for machine learning. In "Categorical Feature Compression via Submodular Optimization" [22], the authors designed a vocabulary compression algorithm, a novel parametrization of mutual information objective, a data structure to query submodular functions and a distributed implementation. They also provided an analysis of simple alternative heuristic compression methods.

The authors of "Integer Discrete Flows and Lossless Compression" [23] introduced Integer Discrete Flow (IDF), a flow-based generative model, an integer map that can learn transformation on high-dimensional discrete data. They also demonstrate that IDF-based compression achieves state-of-the-art lossless compression rates.

In "On the Feasibility of Parser-based Log Compression in Large-Scale Cloud Systems" [24] the authors built LogReducer based on three techniques to compress numerical values in system logs: delta timestamps, correlation identification, and elastic encoding. Their evaluation showed that it achieved high compression ratio on large logs, with comparable speed to the general-purpose compression algorithm.

The authors of "Pattern-Guided File Compression with User-Experience Enhancement for Log-Structured File System on Mobile Devices" [25] introduced file access pattern guided compression (FPC). It is optimized for the random-writes and fragmented-reads of mobile applications, featuring dual-mode compression: foreground and background compression to reduce write stress on write-mostly files and to pack random-reading file blocks.

The raw log messages are usually unstructured, but all the previously mentioned techniques require a structured input. Because of that, the use of log parsing is necessary. It is used to transform the raw messages into structured events. These events can be later used to encode the log messages and decrease the size of the log files. Several algorithms have been introduced to address this problem [26–28]. Since log files can contain sensitive user data, it is important to encrypt the files. Up to this date, various encrypting methods have been proposed [29,30].

## 2. Materials and Methods

In this paper, we propose a method that uses a dictionary and employs different template miners to extract message types from raw log lines. The utilized algorithms are IPLoM, LenMa, Spell, Drain, and MoLFI. We then use these message types to build a dictionary where each ID represents a template/message type. This dictionary is used to encode the raw lines into entries that consist of the corresponding ID and the parameters of the specific line.

We compare the size of the encoded messages and their corresponding dictionary with the size of the raw messages. We do not evaluate the performance or accuracy of the algorithms. Since parameters could be sensitive user data, we use encryption methods on the encoded messages. Specifically, these methods are AES, Blowfish, and FFX. The size of the encrypted messages is also compared to the size of the original raw messages. Lastly, we study the speed of the template miners.

The structure of the paper is as follows. Section 3 gives a brief overview of log parsing and how the individual template miners work. A high-level definition of the encryption methods is explained in Section 4. Section 5 contains the description of the performed experiments. Numerical examples based on log lines generated by real-life networking devices are represented to evaluate the compression capacity of the different algorithms.

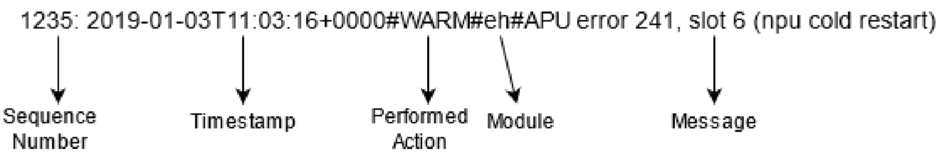
The possible future work about the issue in question and the conclusion we draw can be read in Section 6.

## 3. Concepts and Problems

### *3.1. Log Parsing*

To gather run-time information of a software, logging is used as a programming routine. This is carried out by programmers by inserting commands into the source code to print the desired information into log file entries. One line of the log file is referred to as a log record, which is created by a log print statement.

Since developers are allowed to write free-text messages, the entries are usually raw, unstructured messages. They typically describe an event that occurs in the form of raw messages (free text explanation) such as restarts, system updates, or flash messages. These unstructured entries usually contain more than just a message, but also include other information, like the *Timestamp* (containing the time of the event occurrence), the *Module* (which generated the message), or the *Performed action*. An example from our log lines can be seen in Figure 1.



```
1235: 2019-01-03T11:03:16+0000#WARM#eh#APU error 241, slot 6 (npu cold restart)
```

**Figure 1.** An example of a log event.

A word delimited by a space in the message part of the entry is called a *Token*. For example, in Figure 1, the word "APU" is a token. A log message is always made up of two parts. The first is the fixed part, which is called *Template* and is the same for all appearances. The template contains the *Constant tokens*. These are the words that cannot be expressed by a wildcard value in its associated message type; for example, the "restart" token in the example is a constant. The other part is the variable part which may alter at different occurrences. This part incorporates the *variable tokens*, which can be represented by a wildcard value in its associated message type. The "241" and "6" and "cold" values in Figure 1 are variable tokens.

Parsing each log record $r$ into a set of message types (and the belonging parameter values) is the objective of log parsing. More formally, in the case of an ordered list of log records, $\text{log} = r_1, r_2, \ldots, r_N$ containing $M$ different message types generated by $P$ different processes, where these values are not known, a *Structured log parser* parses the log entries and returns all $M$ different message types. Using such a parser is necessary for almost any log analysis technique. Log parsers are powerful tools; however, they do not apply to all cases, which means pre- and post-processing are also necessary. There are guidelines for pre-processing in " An Evaluation Study on Log Parsing and Its Use in Log Mining" [31] like the use of regex to identify trivial constant parts or deleting duplicate lines.

### *3.2. Iplom*

IPLoM is a log data clustering algorithm introduced in " Lightweight Algorithm for Message Type Extraction in System Application Logs" [32] that iteratively partitions a set of log messages that are used as training examples. The algorithm is divided into four steps.

The first step partitions the messages by *event size* (the number of tokens in the message part). The algorithm assumes that log messages belonging to the same message type have equal event sizes. It uses this heuristic to partition the messages into nonoverlapping batches of messages.

The second step is to partition by *token position*. Since all messages have the same length, they can be viewed as n-tuples where n is the length of the messages. This step also uses a heuristic, which is that the column with the minimum number of unique words is likely to contain constant words at that position in the message type. The messages are again partitioned by these unique words in a way that the result partitions will only contain one of the previously discovered unique values at that position.

In the third step, new partitions are created by *bijection*. To determine the most frequently appearing token count amidst the positions, the number of unique tokens at each position is calculated. This implies the number of message types in the partition. After this, the first two positions with the same number of unique tokens as the most frequent token count are chosen to partition again the log messages containing these tokens at the given positions.

Finally, in the fourth step, message types are created for each cluster by counting the unique tokens in all positions. If a position has multiple values, then it is treated as a variable, otherwise, it is considered to be a constant.

## 3.3. Lenma

LenMa was proposed in "Clustering system log messages using length of words" [28] and uses the assumption that messages belonging to the same message type have words of equal length in the same positions. First, a word length vector $Vm$ is created from the message. For example, the word length vector of "APU error, slot 6 (npu cold restart)" would be encoded as the following:

$$V_m = [len(''APU''), len(''error,''), len(''slot''), len(''6''), len(''(npu''), \\ len(''cold''), len(''restart)'')] = [3,6,4,1,4,4,8]$$

The tokens of the message are also stored in a new word vector $W_m$. This vector contains the template of the message. After this, an $S_m$ similarity value is calculated between the actual message and all clusters with the same event size by the use of cosine similarity. More formally,

$$S_c = \frac{\mathbf{V_c} \cdot \mathbf{V_m}}{\parallel \mathbf{V_c} \parallel \parallel \mathbf{V_m} \parallel} = \frac{\sum_{i=0}^{n} v_{c,i} v_{m,i}}{\sqrt{vv_{c,i}^2}\sqrt{\sum_{i=0}^{n} v_{m,i}^2}}. \quad (1)$$

where $V_C$ is the word length vector of cluster $c$ and $V_m$ is the word length cluster of the actual message, while $v_{c,i}$ and $v_{m,i}$ are the length of the $i$th word in both the cluster and the actual message. This, however, is not enough to correctly cluster all the messages. The comparison between the constant tokens in the same position is also important. This is done by calculating the positional similarity index $S_p$, which was also introduced in [28]. The $S_p$ is defined as:

$$S_p = |\{w_{c,i} = w_{m,i}(w_{c,i} \in \mathbf{W_c}, w_{m,i} \in \mathbf{W_m})\}|, \quad (2)$$

where $\mathbf{W_c}$ and $\mathbf{W_m}$ are the word vectors and $i$ is the position in them. If the message's $S_p$ value is higher than the threshold the message is considered to be in the cluster. Afterward, two things can happen. If there is no cluster with a similarity score greater than the threshold, the message is used to create a new cluster. It is important to note that higher threshold values could result in an increased cluster production. Otherwise, the word length vector of the cluster with the highest score is updated. If there is a word in the cluster whose length does not equal to the length of the word in the new message at the same position, the value is updated to the length of the word in the new message. For example, if we take the message "APU error, slot 6 (device cold restart)", and the previous example as our $c$ cluster, the new $V_c$ vector would be:

$$V_c = [3,6,4,1,7,4,8]$$

The word vector is also updated in a similar fashion. If there is a word in the cluster that is different from the word in the actual message at the same position, the word is changed to a wildcard mask, indicating that it is a variable token. In the end, each cluster represents a message type.

### *3.4. Spell*

Spell views log messages as sequences and uses *LCS (Longest Common Subsequence)* to extract message types [33]. Every word is considered to be a token. Log lines are then converted into token sequences, and a unique ID is assigned to them.

A special data structure called *LCSObject* is created to store *LCSseq*, which is the LCS of numerous log messages. It is also seen as a possible message type for these log messages. That said, LCSObject also contains a list of the line IDs of the corresponding lines. The already parsed LCSObjects are stored in a list called *LCSMap*.

The algorithm works as follows. When a new log line $l_i$ is parsed, $s_i$ token sequence is created from it, and a search through the LCSMap is initiated. Consider the LCSseq in the $i$th LCSObject as $q_i$ and calculate $l_i$, indicating the length of LCS($q_i, s_i$). During the search, $l_{max}$ (the largest $l_i$) and $q_j$, the index of the LCSObject that resulted in $l_{max}$, are stored. In the end, the LCSseq of $q_j$ and $s_i$ are believed to have the same message type, if $l_{max}$ is greater than the given threshold. In the case of multiple LCSObjects with $L_{max}$, the one with the smallest $|q_j|$ value is chosen. After that, the new LCS sequence describing the message type for $l_i$ and all entries in the $j$th LCSObject is created via backtracking. While backtracking, the positions where two sequences are different are denoted with the "*" wildcard mask. After this, $l_i$'s line ID is added to the line IDs of the $j$th LCSObject and its $q_j$ is changed to LCS($q_j, s_i$). If there is no such LCSObject that has an LCS with $s_i$ larger than $|s_i|/2$, then a new LCSObject is created with $s_i$ as its LCSseq and $l_i$'s line ID as its line ID list.

### *3.5. Drain*

Drain is a fixed-depth tree-based online log parsing method that was introduced in "An online log parsing approach with fixed depth tree" [26]. The parse tree consists of three types of nodes. At the top of the parse tree is the *root node*, which is connected to the *internal nodes*. They do not involve any log groups since they are designed to contain specially constructed rules that control the search process. The bottom layer of the parse tree is made from the *leaf nodes*. These nodes hold the log groups, and they can be reached by a path from the root node. The log groups are made log line IDs and log events. The log event is used to indicate the message type that is best suited for the log messages in the group. The name suggests that all leaf nodes have a fixed predefined depth, and Drain only traverses through $(depth - 2)$ internal nodes before reaching a leaf node.

The first step is to search by the length of the log message that is equal to the number of words in the message. Log groups with a different number of tokens are expressed by the first-layer nodes. A path to the first node representing the same length as the actual log line length is selected. For example, in the case of "NPU cold restart" the internal node representing "Length-3" is chosen.

The second step uses the presumption that log messages that have the same message type usually have the same constant token in the first position of the message. The next node is selected by this assumption. For example, in the case of the previous "NPU cold restart" message, the 2nd layer node encoding that the message starts with "NPU" is picked. Messages beginning with a parameter can lead to branch explosion. Tokens that only contain digits are considered to be special. In the case of such messages, a special "*" node is selected in this step.

The third step is to search by token similarity. By this step, the search has already reached a leaf node containing multiple log groups. The most appropriate log group is selected based on the similarity of the actual message and the log event of each group. The similarity, $simSeq$, is defined as:

$$simSeq = \frac{\sum_{i=1}^{n} equ(seq_1(i), seq_2(i))}{n}, \tag{3}$$

where $n$ is the number of tokens in the message and $seq_1$ is the actual log message, $seq_2$ is the log event of the group and $seq(i)$ denotes the $i$th token in the message. The function $equ$ is defined as the following:

$$equ(t_1, t_2) = \begin{cases} 1, & \text{if } t_1 = t_2 \\ 0, & \text{otherwise,} \end{cases} \tag{4}$$

where $t_1$ and $t_2$ are the two tokens. If the greatest similarity reaches the predefined threshold, the log group that reached the largest similarity is returned; otherwise, a flag is returned to indicate that the message does not fit into any log group.

The last step is to update the parse tree. If the output of the previous step is a log group, the ID of the actual log line is added to its ID list and the log event of the group is updated. This is done by checking if the tokens are the same in the different positions of the actual message and the log event. If they differ, a wildcard mask "*" is set in the log event at that position; otherwise, nothing happens. If a previous step resulted in the flag, a new log group is created from the actual message, only with the ID of the actual log line.

### *3.6. MoLFI*

MoLFI was proposed in "A search-based approach for accurate identification of log message formats" [34] and employs the standard NSGA-II for the log parsing problem. While pre-processing the data, trivial constants are replaced with a unique #spec# token that cannot be changed in the later steps. The messages are also sorted into buckets based on their token count.

A new two-level encoding schema is applied: each chromosome $C$ is a set of groups $C = \{G_1, \ldots, G_{max}\}$ where each group $G_N = \{t_1, \ldots, t_j\}$ is a collection of templates (message types) with the token count $N$. This schema ensures that only messages and templates of the same length are matched in the later steps.

In the first step, the initial population is created. Let $M$ denote the pre-processed log messages. After a chromosome is created, it is filled with groups of templates. Each group contains pre-processed log messages $M_N \in M$ with the same $N$ length. Initially, all messages are in a special set called unmatched. In every turn, a message is randomly selected from here, and a template $t$ is created based on it. The template is identical to the message except for a randomly selected token, which is changed to "*". This is then added to the group $G_N$, and the unmatched set is updated (the message is removed). This loop stops when the unmatched set becomes empty.

The next step is to Crossover, which is achieved by the use of the uniform crossover [35] operator. Two parents are taken and two offsprings are created by this operator by mixing the attributes of the parents. The templates between the parents are exchanged without changing the set of templates constructing each group. The offsprings contain all the already-processed messages and do not overlap on any template.

The next step is to mutate the offsprings. This is done by randomly altering a template in each of its groups. Let $t_i = \{token_1, \ldots, token_n\}$ be the selected template. Each token has $\frac{1}{n}$ chance to be changed. If a token is modified, the following can happen: If it is a variable token, it is changed to a constant token randomly selected from the fixed tokens at that position in the messages that match $t_i$. If it is a constant token, it is replaced by a wildcard "*" token. If it is the special #spec# token, nothing happens. At this point, a correction algorithm is used to remove overlapping templates and to add random templates to groups that do not match all of their messages. In the end, the variable tokens that do not influence the frequency scores are removed by inspecting if their deletion changes the messages that match their template or not. If there is a change, they are added back to the template; otherwise, the template remains unchanged. This algorithm results in numerous optimal solutions, from which the knee point is chosen as the final output.

## 4. Encryption Techniques

Encryption is a basic term in cryptography that stands for the process of encoding information. The original human-readable data, plaintext, are transformed into an incomprehensible text, a ciphertext that appears to be random. The process requires the use of a secret key that both the sender and the recipient know. These keys are usually pseudo-random generated keys. Ideally, with the use of the key, only authorized parties can decode the ciphertext and access the initial message. While it is possible to decrypt the message without the key, it requires a computing capacity that modern computers cannot deal with.

### *4.1. AES*

The Advanced Encryption Standard (AES) is a symmetric encryption that is a subset of the Rijndael block cipher introduced in "AES proposal: Rijndael" [29]. It is considered to be one of the best encrypting algorithms. It has a key size of either 128, 192, or 256 bits and a fixed block size of 128 bit and encrypts only one block at a time. It operates on the *state*, which is a $4 \times 4$ column-major order array of bytes. The algorithm's input is the plaintext, which is converted into the output (ciphertext) via a number of transformation rounds. The number of rounds $N_r$ depends on the size of the key. In the case of 128-bit keys, 10 round is used, 12 rounds in case of 192-bit keys and 14 rounds for 256-bit keys. A high-level description of the operation of the method is the following.

The first step is the *KeyExpansion* that derives round keys from the cipher key based on the AES key schedule. This is followed by an initial *AddRoundKey* that uses bitwise xor to combine each byte of the state with a byte of the round key.

This is followed by $N_r - 1$ rounds consisting of four phases. The first is *SubBytes*, which replaces each byte with another based on a lookup table in a non-linear fashion. This is followed by *ShiftRows*, a cyclical shift in the last three rows of the state. After this, the four bytes in each column of the state are combined by the *MixColumns* operation. Lastly, another AddRoundKey is used. The $N_r - 1$ rounds are followed with a final round, which is composed of SubBytes, ShiftRows, and AddRoundKey. To decode the ciphertext with the use of the same encryption key, a set of reverse rounds is used. The algorithm can be seen in Figure 2.

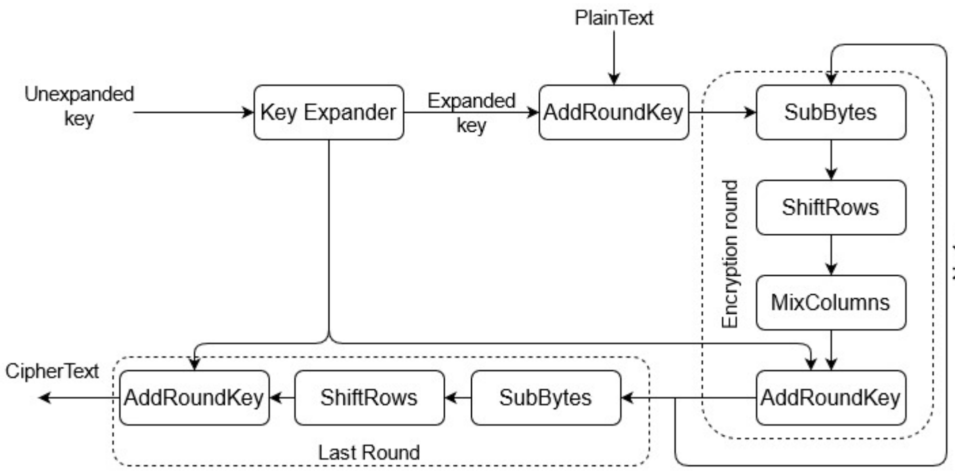


**Figure 2.** The encryption rounds of the AES.

### 4.2. Blowfish

Blowfish was designed by Bruce Schneier in "Description of a new variable-length key, 64-bit block cipher (Blowfish)" [30], 1993, to replace DES. It is a fast and free public encryption software; therefore, it is neither licensed nor patented. It is also a Feistel cipher [36]. Blowfish is a symmetric block cipher with a fixed block size of 64 bits, which means that it divides the input into fixed 64-bit blocks while encrypting and decrypting. It has a variable key length that can vary from 32 bits up to 448 bits. The encryption schedule of Blowfish can be seen in Figure 3. There are 18 subkeys stored in a *P-array*, with each being a 32-bit entry. Four *Substitution boxes (S-box)* $S_1$, $S_2$, $S_3$, $S_4$ are used by the algorithm, each consisting of 256 entries with a size of 32 bits.

First, the P-array and the S-boxes are initialized with the use of Blowfish's key schedule: values are generated from the hexadecimal digits of pi. This is followed by 16 rounds, each consisting of four operations. Each $R_i$ round takes two inputs, the corresponding subkey and the plaintext (data), from the output of the previous round. Let $DL_I$ and $DR_I$ denote the left and right sides of the data. The first step is to XOR the *i*th subkey in the P-array, $P_I$, with $DL_I$.

The second step is to use this XORed data as the input of the *F-function*. The function works as follows: four 8-bit quarters $X_1$, $X_2$, $X_3$, $X_4$ are created from the 32-bit input and are used as the input of the S-boxes, which then creates 32-bit values $X_A$, $X_B$, $X_C$, $X_D$ from them. $X_A$ and $X_B$ are added modulo $2^{32}$; next, the result is XORed with $X_C$. This value is added with $X_D$, generating the output of the F-function.

In the third step, the $DR_I$ is XORed with the output of the function. The final step swaps the left side and the right side. The output of the 16th round is then post-processed (output whitening). The last swap is undone and $P_{18}$ is XORed with $DL_{17}$, while $P_{17}$ is XORed with $DR_{17}$. Decryption works in the same way as encryption, except that the subkeys in P-array are used in reverse order.

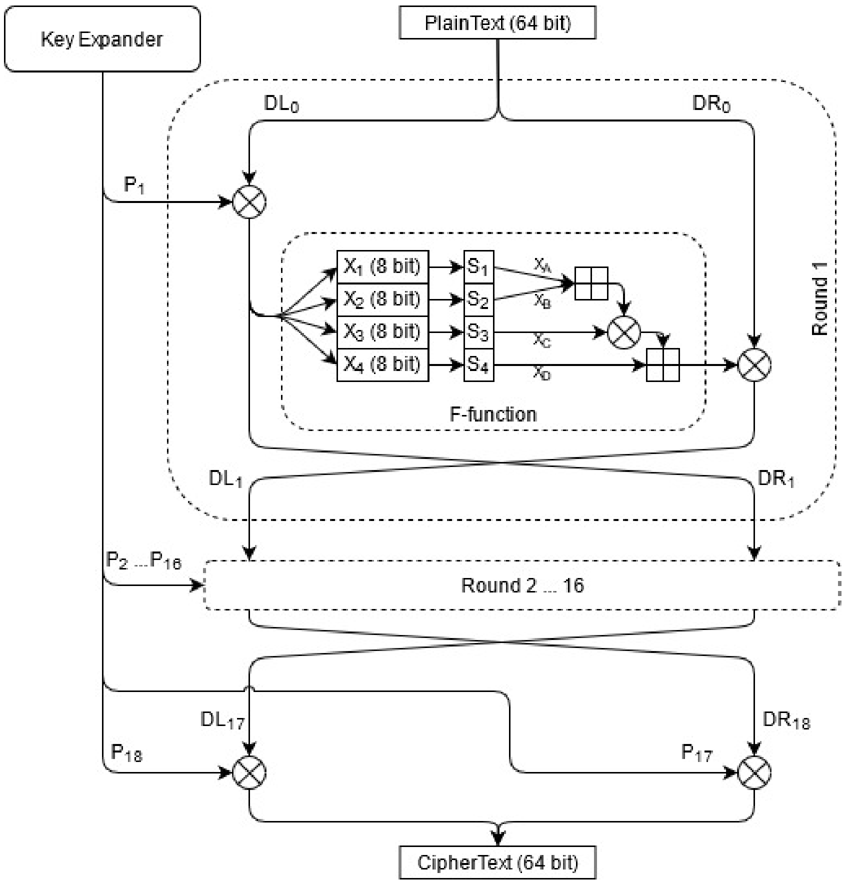


**Figure 3.** The encryption schedule of Blowfish.

### *4.3. FFX*

FFX is a *Format-preserving, Feistel-based encryption* method that was introduced in "Format-Preserving Feistel-Based Encryption Mode" [37]. Format-preserving encryption means that the ciphertext has the same format as the plaintext input. For example, the encryption of a 16-digit credit card number would result in a ciphertext consisting of 16 digits. FFX takes three parameters as its input, the plaintext that is to be encoded, the key that will be used as the round key, and a tweak [38]. A tweak is a nonempty set of strings that are used to modify the round key. As its name suggests, the algorithm is based on the use of a Feistel network [36].

The core of each Feistel network is a *round function* that takes a subkey and a data block as its inputs and returns an output with the same size of the data block. Each round in the network consists of two main operations. The first one is to run the round function on half of the data, and the second is to XOR the output of the function with the other half. FFX uses AES as the round function for its Feistel network. Only one secret key is used as the round keys of AES; however, it is marginally tweaked every round.

## **5. Results**

### *5.1. Data*

Our data consisted of log lines produced by different network devices used at the Ericsson-ELTE Software Technology Lab. We tested the compression efficiency on four distinct log message collections varying in size. Table 1 contains the details about the collections.

**Table 1.** Size of the datasets.

| Name | Number of Messages | Size in Bytes | Size in Kilobytes |
|---|---|---|---|
| Biggest | 637,369 | 23,350,348 | 22,840 KB |
| Large | 280,002 | 10,441,998 | 10,198 KB |
| Mid | 124,433 | 4,717,484 | 4607 KB |
| Small | 39,139 | 1,588,912 | 1152 KB |

### *5.2. Experimental Analysis*

Several experiments were conducted to verify the compression efficiency of our method with the use of different template miners. Evaluating the accuracy of these methods is not part of this paper. The change in size before and after different encryptions was also examined. For our experiments, we used the template miner implementations proposed in [39]. The experimental analyses are divided into three parts and are explained below.

#### 5.2.1. Experiment 1: Comparing the Size of the Compressed Messages with Our Method and Different Template Miners

In order to compress the log messages, we employ five template miners namely IPLoM, LenMa, Spell, Drain, and MoLFI to attain the different message types from the processed log messages. Log messages are then sorted based on their matching message template. After this, an ID is assigned to each message type, and thus a dictionary is created. This dictionary is then used to encode each message. If a message does not contain any parameter, it is changed to the ID corresponding to its message type. If it contains parameters, then the ID is followed by the parameter list. For example, if the ID "1" indicates the template "XF_Restart", then a log message "XF_Restart" would be changed to "1". The "NPU cold restart" message, where "cold" is a parameter and its ID is "2" would be changed to "2 cold". We encoded all of our datasets based on this principle, and the numerical differences between the sizes in bytes and kilobytes can be seen in Table 2.

**Table 2.** Compressed data sizes on the four datasets.

| | **Biggest Dataset** | | |
|---|---|---|---|
| **Template Miner** | **Compressed File Size** | **Dictionary Size** | **Overall Size** |
| IPLoM | 9,904,236 (9673 KB) | 4396 (5 KB) | 9,908,632 (9677 KB) |
| LenMa | 7,728,034 (7547 KB) | 6171 (7 KB) | 7,734,205 (7553 KB) |
| Spell | 13,482,906 (13,167 KB) | 3250 (4 KB) | 13,486,156 (13,171 KB) |
| Drain | 10,926,091 (10,671 KB) | 4161 (5 KB) | 10,930,252 (10,675 KB) |
| MoLFI | 9,846,833 (9617 KB) | 4568 (5 KB) | 9,851,401 (9621 KB) |
| | **Large Dataset** | | |
| **Template Miner** | **Compressed File Size** | **Dictionary Size** | **Overall Size** |
| IPLoM | 4,597,412 (4490 KB) | 4342 (5 KB) | 4,601,754 (4494 KB) |
| LenMa | 3,822,516 (3733 KB) | 6033 (6 KB) | 3,828,549 (3739 KB) |
| Spell | 5,164,654 (5044 KB) | 3233 (4 KB) | 5,167,887 (5047 KB) |
| Drain | 5,005,779 (4889 KB) | 4301 (5 KB) | 5,010,080 (4893 KB) |
| MoLFI | 4,375,515 (4272 KB) | 4717 (5 KB) | 4,380,232 (4278 KB) |
| | **Mid Dataset** | | |
| **Template Miner** | **Compressed File Size** | **Dictionary Size** | **Overall Size** |
| IPLoM | 1,839,053 (1796 KB) | 4368 (5 KB) | 1,843,421 (1801 KB) |
| LenMa | 1,466,446 (1433 KB) | 7376 (8 KB) | 1,473,822 (1440 KB) |
| Spell | 2,319,417 (2666 KB) | 3217 (4 KB) | 2,322,634 (2669 KB) |
| Drain | 2,040,298 (1993 KB) | 4138 (5 KB) | 2,044,436 (1997 KB) |
| MoLFI | 1,773,076 (1732 KB) | 4557 (5 KB) | 1,777,633 (1736 KB) |
| | **Small Dataset** | | |
| **Template Miner** | **Compressed File Size** | **Dictionary Size** | **Overall Size** |
| IPLoM | 691,051 (675 KB) | 3053 (3 KB) | 694,104 (678 KB) |
| LenMa | 470,057 (460 KB) | 6436 (7 KB) | 476,493 (466 KB) |
| Spell | 584,083 (571 KB) | 2271 (3 KB) | 586,354 (573 KB) |
| Drain | 620,965 (607 KB) | 3253 (4 KB) | 624,218 (610 KB) |
| MoLFI | 560,232 (548 KB) | 3924 (4 KB) | 564,156 (551 KB) |

It can be seen that there is no significant difference between the size of the dictionaries, only LenMa produces light overhead. Despite that, the use of a dictionary increases the overall size, it is negligible in comparison with the log file sizes. Figures 4–7 present the compression rates on the datasets.

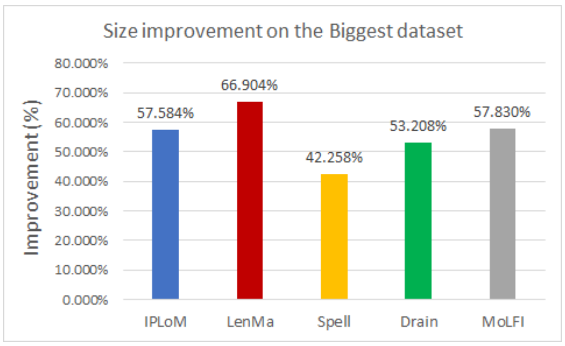


**Figure 4.** Compression achieved by the different template miners on the Biggest dataset.

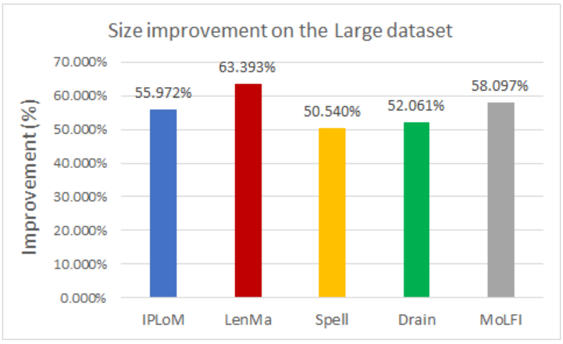


**Figure 5.** Compression achieved by the different template miners on the Large dataset.

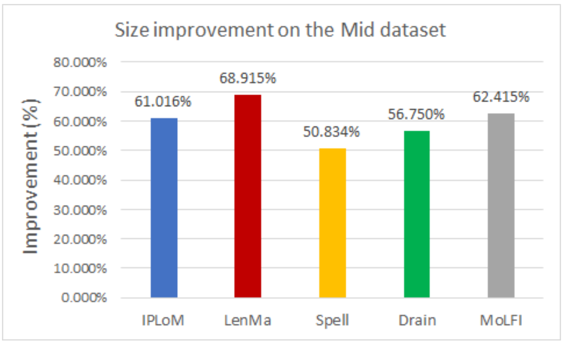


**Figure 6.** Compression achieved by the different template miners on the Mid dataset.

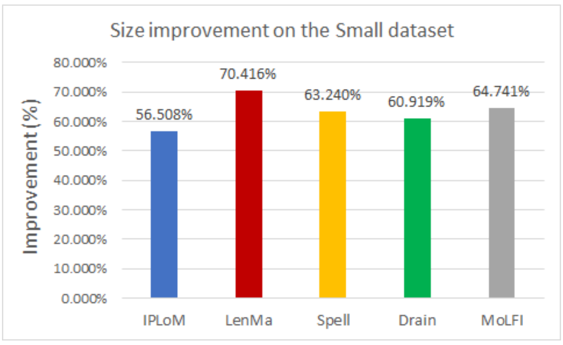


**Figure 7.** Compression achieved by the different template miners on the Small dataset.

Out of all of the investigated methods, LenMa has the best compression rate with our method with an average of 67.407%, while Spell's performance is the worst with an average of 51.718%. This can be explained by the differences in dictionary sizes. LenMa creates a slightly bigger dictionary containing more templates than the other methods, which can be explained with the sensitivity of its threshold parameter; however, this pays off during the encoding of the file. Having more message types results in a larger number of IDs and fewer parameters. Its also notable that using any of the discussed template miners and our dictionary method, the size of a log file can be reduced by at least 50%.

#### 5.2.2. Experiment 2: Comparing Size of the Compressed and Encrypted Messages

Sometimes, log files often contain sensitive user data. These pieces of information are viewed as parameters by the template miners, so they are stored as plain text when our previously discussed method is used. For this reason, we employed three different encryption methods to make the compressed files more secure. The size of the encrypted datasets can be seen in Table 3.

**Table 3.** Compressed data sizes on the four datasets using encryption.

| **Biggest Dataset** | | | | |
|---|---|---|---|---|
| **Template Miner** | **Without Encryption** | **AES** | **Blowfish** | **FFX** |
| IPLoM | 9,908,632 (9678 KB) | 9,904,535 (9673 KB) | 26,679,293 (26,054 KB) | 9,359,864 (9141 KB) |
| LenMa | 7,734,205 (7554 KB) | 7,728,343 (7548 KB) | 20,336,656 (19,861 KB) | 7,161,668 (6994 KB) |
| Spell | 13,486,156 (13,171 KB) | 13,483,207 (13,168 KB) | 36,602,389 (35,745 KB) | 12,973,985 (12,670 KB) |
| Drain | 10,930,252 (10,676 KB) | 10,926,391 (10,671 KB) | 29,439,879 (28,750 KB) | 10,391,499 (10,148 KB) |
| MoLFI | 9,851,401 (9622 KB) | 9,847,143 (9617 KB) | 25,962,525 (25,355 KB) | 9,302,020 (9085 KB) |
| **Large Dataset** | | | | |
| **Template Miner** | **Without Encryption** | **AES** | **Blowfish** | **FFX** |
| IPLoM | 4,601,754 (4495 KB) | 4,597,719 (4490 KB) | 12,308,542 (12,021 KB) | 4,360,391 (4259 KB) |
| LenMa | 3,828,549 (3739 KB) | 3,822,823 (3734 KB) | 10,059,770 (9824 KB) | 3,577,564 (3494 KB) |
| Spell | 5,167,887 (5048 KB) | 5,164,951 (5044 KB) | 14,062,556 (13,733 KB) | 4,933,469 (4818 KB) |
| Drain | 5,010,080 (4894 KB) | 5,006,087 (4889 KB) | 13,477,403 (13,162 KB) | 4,773,055 (4662 KB) |
| MoLFI | 4,380,232 (4242 KB) | 4,375,815 (4274 KB) | 11,722,968 (11,449 KB) | 4,136,272 (4040 KB) |
| **Mid Dataset** | | | | |
| **Template Miner** | **Without Encryption** | **AES** | **Blowfish** | **FFX** |
| IPLoM | 1,843,421 (1801 KB) | 1,839,351 (1797 KB) | 4,962,083 (4846 KB) | 1,731,735 (1692 KB) |
| LenMa | 1,473,822 (1441 KB) | 1,466,743 (1433 KB) | 3,821,890 (3733 KB) | 1,355,323 (1324 KB) |
| Spell | 2,322,634 (2670 KB) | 2,319,719 (2266 KB) | 6,350,991 (6203 KB) | 2,216,914 (2165 KB) |
| Drain | 2,044,436 (1998 KB) | 2,040,599 (1993 KB) | 5,503,321 (5375 KB) | 1,935,082 (1890 KB) |
| MoLFI | 1,777,633 (1732 KB) | 1,773,383 (1732 KB) | 4,699,949 (4590 KB) | 1,665,011 (1626 KB) |
| **Small Dataset** | | | | |
| **Template Miner** | **Without Encryption** | **AES** | **Blowfish** | **FFX** |
| IPLoM | 694,104 (678 KB) | 691,351 (676 KB) | 1,862,264 (1819 KB) | 658,531 (644 KB) |
| LenMa | 476,493 (467 KB) | 470,359 (460 KB) | 1,232,970 (1205 KB) | 435,282 (426 KB) |
| Spell | 586,354 (574 KB) | 584,391 (571 KB) | 1,567851 (1532 KB) | 550,485 (538 KB) |
| Drain | 624,218 (611 KB) | 621,271 (607 KB) | 1,646,740 (1609 KB) | 587,605 (574 KB) |
| MoLFI | 564,156 (552 KB) | 560,535 (548 KB) | 1,486,109 (1452 KB) | 526,304 (514 KB) |

Out of the three algorithms, Blowfish has the worst performance, it makes the files more than two and a half times larger, which is almost as big as the original datasets. AES does not alter the size of the files, while FFX slightly reduces their size.

### 5.2.3. Experiment 3: Comparing the Speed of the Different Template Miners

The speed of a method is also an important feature. Using a template miner with high speed is essential since there could be many log files that need to be compressed. We evaluated the speed of the different methods on our datasets. The results can be seen in Figures 8–11.

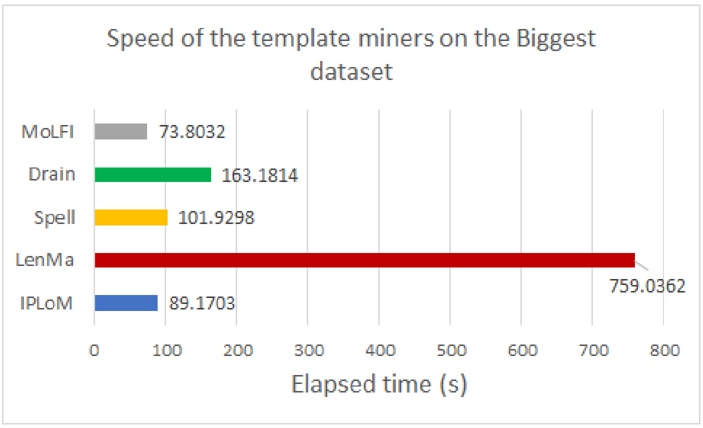


**Figure 8.** Speed of the different template miners on the Biggest dataset.

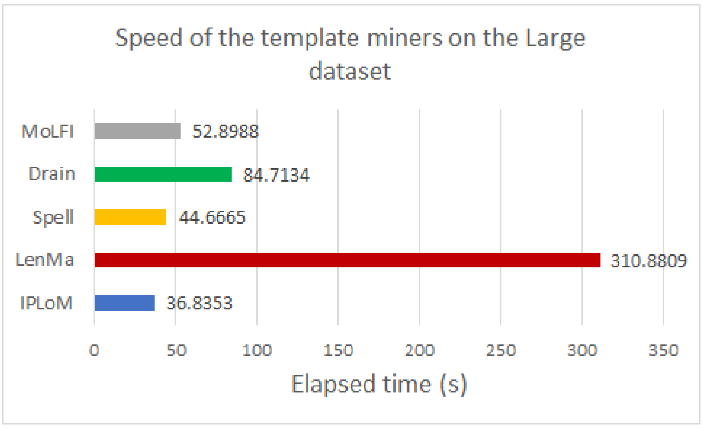


**Figure 9.** Speed of the different template miners on the Large dataset.

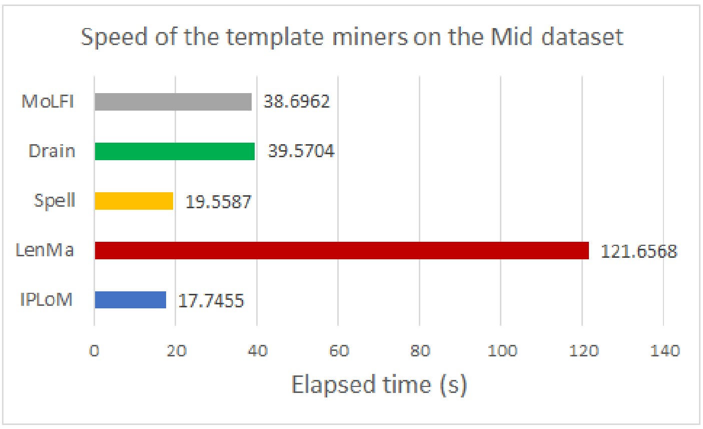


**Figure 10.** Speed of the different template miners on the Mid dataset.

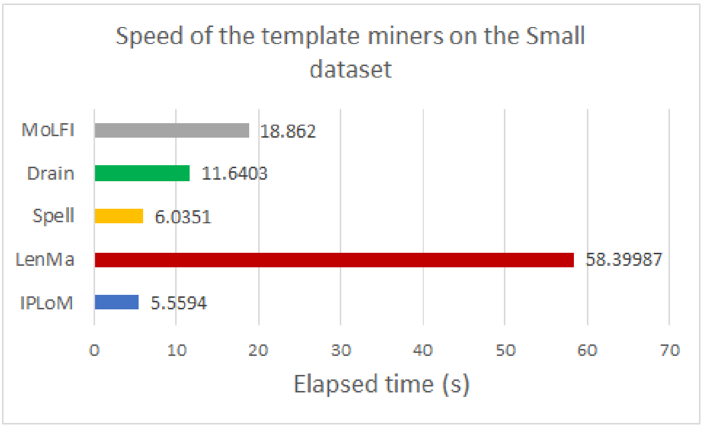


**Figure 11.** Speed of the different template miners on the Small dataset.

It can be seen that although LenMa has the best compression rate, it takes four times more time than the other template miners. This could be explained by the slow speed of cosine similarity. MoLFI is faster on larger datasets but becomes slow on smaller datasets. This is because of the initialization of the multiple chromosomes. In almost every case, IPLoM is the fastest algorithm, since it uses specifically designed heuristics.

## 6. Discussion and Conclusions

In this paper, we evaluated the compression capacity of our dictionary method with the use of various template miners. These measures acquire message events from log files. Events consist of variable tokens (parameters) and constant tokens. We use these templates to create a dictionary where each ID represents a message event. The ID of the corresponding template is assigned to each log line. We then use the dictionary to encode the messages based on the principle that we only store the ID and the parameter list. Since parameters could contain confidential information, the compressed files are encrypted as well.

To analyze the performance of the template miners in pair with this encoding method, several experiments were conducted. The experimental results showed that using any type of template miner with the generated directory results in around 50% compression. Out of all the investigated measures, LenMa proved to be the best with an average of 67.407%. It produced a bigger dictionary, and because of that, fewer parameters had to be encoded, which resulted in smaller file sizes. In the case of the encryption methods, the results showed that the use of FFX slightly decreases the size of the compressed file. The speed of the template miners was also compared. Based on our experiments, LenMa was outstandingly slower than the other methods, despite its good performance at compression. Our results yielded that IPLoM is the fastest among the examined methods.

Based on our experiments, we would suggest the combination of IPLoM and FFX to achieve the best results; however, using any of the log miners with the dictionary method greatly reduces the size of the log file.

While we only investigated these five template miners, it is possible that other methods could yield better results. We only evaluated the performance on log files, and it would be interesting to measure the performance in the case of stream-like data. It would be also beneficial to compare the performance of our method and the performance of the existing general compressors. The compression rate achieved by the combination of our algorithm and the general compressors could be also investigated.

**Author Contributions:** Conceptualization, P.M., P.L.-K., and A.K.; methodology, P.M., P.L.-K., and A.K.; software, P.M. and P.L.-K.; validation, P.M., P.L.-K., and A.K.; investigation, P.M., P.L.-K., and A.K.; writing—original draft preparation, P.M., P.L.-K., and A.K.; writing—review and editing, P.M., P.L.-K., and A.K.; supervision, A.K.; project administration, A.K.; All authors have read and agreed to the published version of the manuscript.

**Funding:** The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002). This research was also supported by grants of "Application Domain Specific Highly Reliable IT Solutions" project that has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Subprogramme) funding scheme.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** The data was provided by the Ericsson-ELTE Software Technology Lab. Authors can confirm that all relevant data are included in the article.

**Acknowledgments:** This publication is the partial result of the Research & Development Operational Programme for the project "Modernisation and Improvement of Technical Infrastructure for Research and Development of J. Selye University in the Fields of Nanotechnology and Intelligent Space", ITMS 26210120042, co-funded by the European Regional Development Fund. The project was also supported by the Ericsson-ELTE Software Technology Lab.

**Conflicts of Interest:** The authors declare no conflict of interest.

## Abbreviations

The following abbreviations are used in this manuscript:

| | |
|---|---|
| IPLoM | Iterative Partitioning Log Mining |
| LenMa | Length Matters |
| SPell | Streaming Parser for Event Logs using LCS |
| Drain | Depth Tree Based Online Log Parsing |
| MoLFI | Multi-objective Log message Format Identification |
| AES | Advanced Encryption Standard |
| FFX | Format-preserving, Feistel-based encryption |

## References

1. Breier, J.; Branišová, J. Anomaly Detection from Log Files Using Data Mining Techniques. In *Lecture Notes in Electrical Engineering*; Springer: Berlin/Heidelberg, Germany, 2015; Volume 339, pp. 449–457.
2. Zimek, A.; Schubert, E. Outlier Detection. In *Encyclopedia of Database Systems*; Springer: New York, NY, USA, 2017; pp. 1–5.
3. Nagaraj, K.; Killian, C.; Neville, J. tructured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*; USENIX Association: San Jose, CA, USA, 2012.
4. Sun, Y.; Li, H.; Councill, I.G.; Huang, J.; Lee, W.C.; Giles, C.L. Personalized Ranking for Digital Libraries Based on Log Analysis. In *Proceedings of the 10th ACM Workshop on Web Information and Data Management*; Association for Computing Machinery: New York, NY, USA, 2008.
5. Reidemeister, T.; Munawar, M.A.; Jiang, M.; Ward, P.A.S. Diagnosis of Recurrent Faults Using Log Files. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*; IBM Corp: Armonk, NY, USA, 2009.
6. Basin, D.; Caronni, G.; Ereth, S.; Harvan, M.; Klaedtke, F.; Mantel, H. Scalable Offline Monitoring. In *Lecture Notes in Computer Science*; Springer International Publishing: Cham, Germany, 2014; pp. 31–47.
7. Kwan, I.S.; Fong, J.; Wong, H. An e-customer behavior model with online analytical mining for internet marketing planning. *Decis. Support Syst.* **2005**, *41*, 189–204. [CrossRef]
8. He, S.; He, P.; Chen, Z.; Yang, T.; Su, Y.; Lyu, M.R. A Survey on Automated Log Analysis for Reliability Engineering. 2020. Available online: https://arxiv.org/abs/2009.07237 (accessed on 6 June 2021).
9. Gaddam, S.R.; Phoha, V.V.; Balagani, K.S. K-Means+ ID3: A novel method for supervised anomaly detection by cascading K-Means clustering and ID3 decision tree learning methods. *IEEE Trans. Knowl. Data Eng.* **2007**, *19*, 345–354. [CrossRef]
10. Munir, M.; Siddiqui, S.A.; Dengel, A.; Ahmed, S. DeepAnT: A deep learning approach for unsupervised anomaly detection in time series. *IEEE Access* **2018**, *7*, 1991–2005. [CrossRef]
11. Akcay, S.; Atapour-Abarghouei, A.; Breckon, T.P. GANomaly: Semi-supervised Anomaly Detection via Adversarial Training. In *Asian Conference on Computer Vision*; Springer: Cham, Switzerland, 2018; pp. 622–637.
12. He, S.; Zhu, J.; He, P.; Lyu, M.R. Loghub: A large collection of system log datasets towards automated log analytics. *arXiv* **2020**, arXiv:2008.06448.
13. Yao, K.; Li, H.; Shang, W.; Hassan, A.E. A study of the performance of general compressors on log files. *Empir. Softw. Eng.* **2020**, *25*, 3043–3085. [CrossRef]

14. Gazdag, A.; Buttyan, L.; Szalay, Z. Efficient lossless compression of CAN traffic logs. In Proceedings of the 2017 25th International Conference on Software, Telecommunications and Computer Networks (SoftCOM), Split, Croatia, 21–23 September 2017.
15. Hassan, A.E.; Martin, D.J.; Flora, P.; Mansfield, P.; Dietz, D. An industrial case study of customizing operational profiles using log compression. In Proceedings of the 30th International Conference on Software Engineering, New York, NY, USA, 10–18 May 2008.
16. Fei, P.; Li, Z.; Wang, Z.; Yu, X.; Li, D.; Jee, K. SEAL: Storage-efficient Causality Analysis on Enterprise Logs with Query-friendly Compression. In *30th USENIX Security Symposium (USENIX Security 21)*; USENIX Association: Berkeley, CA, USA, 2021.
17. Liu, J.; Zhu, J.; He, S.; He, P.; Zheng, Z.; Lyu, M.R. Logzip: Extracting Hidden Structures via Iterative Clustering for LogCompression. In Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 11 November 2019.
18. Yao, K.; Sayagh, M.; Shang, W.; Hassan, A.E. Improving State-of-the-art Compression Techniques for Log Management Tools. *IEEE Trans. Softw. Eng.* **2021**, 1, doi:10.1109/TSE.2021.3069958. [CrossRef]
19. Ryabko, B. Time-universal data compression and prediction. In Proceedings of the 2019 IEEE International Symposium on Information Theory, Paris, France, 7–12 July 2019; pp. 562–566.
20. Vestergaard, R, Zhang, Q, Lucani, D.E. Lossless compression of time series data with generalized deduplication. In Proceedings of the IEEE Global Communications Conference (GLOBECOM), Waikoloa, HI, USA, 9–13 December 2019; pp. 1–6.
21. Doblander, C.; Khatayee, A.; Jacobsen, H.A. Predict: Predictive dictionary maintenance for message compression in publish/subscribe. In Proceedings of the 19th International Middleware Conference, Rennes, France, 10–14 December 2018; pp. 174–186.
22. Bateni, M.; Chen, L.; Esfandiari, H.; Fu, T.; Mirrokni, V.; Rostamizadeh, A. Categorical feature compression via submodular optimization. In Proceedings of the International Conference on Machine Learning, Long Beach, CA, USA, 9–15 June 2019; pp. 515–523.
23. Hoogeboom, E.; Peters,W.; Berg, R.V.; Welling, M. Integer discrete flows and lossless compression. *arXiv* **2019**, arXiv:1905.07376.
24. Wei, J.; Zhang, G.; Wang, Y.; Liu, Z.; Zhu, Z.; Chen, J.; Sun, T.; Zhou, Q. On the Feasibility of Parser-based Log Compression in Large-Scale Cloud Systems. In Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST 21), Berkeley, CA, USA, 23–25 February 2021; pp. 249–262.
25. Ji, C.; Chang, L.P.; Pan, R.; Wu, C.; Gao, C.; Shi, L.; Kuo, T.W.; Xue, C.J. Pattern-Guided file compression with user-Experience enhancement for log-Structured file system on mobile devices. In Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST 21), Berkeley, CA, USA, 23–25 February 2021; pp.127–140.
26. He, P.; Zhu, J.; Zheng, Z.; Lyu, M.R. Drain: An online log parsing approach with fixed depth tree. In Proceedings of the IEEE International Conference on Web Services (ICWS), Honolulu, HI, USA, 25–30 June 2017.
27. Li, L.; Man, Y.; Chen, M. A Method of Large—Scale Log Pattern Mining. In *Lecture Notes in Computer Science*; Springer International Publishing: Cham, Switzerland, 2018; pp. 76–84.
28. Shima, K. Length matters: Clustering system log messages using length of words. *arXiv* **2016**, arXiv:1611.03213.
29. Daemen, J.; Rijmen, V. *AES Proposal: Rijndael*; Gaithersburg, MD, USA, 1999. Available online: https://www.cs.miami.edu/home/burt/learning/Csc688.012/rijndael/rijndael_doc_V2.pdf (accessed on 6 June 2021).
30. Schneier, B. Description of a new variable-length key, 64-bit block cipher (Blowfish). In *Fast Software Encryption*; Springer: Berlin/Heidelberg, Germany, 1994; pp. 191–204.
31. He, P.; Zhu, J.; He, S.; Li, J.; Lyu, M.R. An Evaluation Study on Log Parsing and Its Use in Log Mining. In *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*; IEEE: Toulouse, France, 2016.
32. Makanju, A.; Zincir-Heywood, A.N.; Milios, E.E. A Lightweight Algorithm for Message Type Extraction in System Application Logs. *IEEE Trans. Softw. Eng.* **2012**, *24*, 1921–1936. [CrossRef]
33. Du, M.; Li, F. Spell: Streaming parsing of system event logs. In Proceedings of the IEEE 16th International Conference on Data Mining (ICDM), Barcelona, Spain, 12–15 December 2016.
34. Messaoudi, S.; Panichella, A.; Bianculli, D.; Briand, L.; Sasnauskas, R. A search-based approach for accurate identification of log message formats. In Proceedings of the IEEE/ACM 26th International Conference on Program Comprehension (ICPC), Gothenburg, Sweden, 27 May–3 June 2017.
35. Sivanandam, S.N.; Deepa, S.N. *Introduction to Genetic Algorithms*, 1st ed.; Springer: Berlin/Heidelberg, Germany, 2008; pp. 15–37.
36. Nyberg. K. Generalized Feistel networks. In *ASIACRYPT 1996 -Lecture Notes in Computer Science*; Springer: Berlin/Heidelberg, Germany, 1996; pp. 91–104.
37. Bellare, M.; Ristenpart, T.; Rogaway, P.; Stegers, T. Format-preserving encryption. In *International Workshop on Selected Areas in Cryptography*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 295–312
38. Liskov, M.; Rivest, R.L.; Wagner, D. Tweakable block ciphers. In *Annual International Cryptology Conference*; Springer: Berlin/Heidelberg, Germany, 2002; pp. 31–46.
39. Zhu, J.; He, S.; Liu, J.; He, P.; Xie, Q.; Zheng, Z.; Lyu, M.R. Tools and benchmarks for automated log parsing. In Proceedings of the IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), Montreal, QC, Canada, 25–31 May 2019.