ELISE: A Storage Efficient Logging System Powered by Redundancy Reduction and Representation Learning

Hailun Ding, Shenao Yan, Juan Zhai, and Shiqing Ma, *Rutgers University*

https://www.usenix.org/conference/usenixsecurity21/presentation/ding

This paper is included in the Proceedings of the
30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the
30th USENIX Security Symposium
is sponsored by USENIX.

ELISE: A Storage Efficient Logging System Powered by Redundancy Reduction and Representation Learning

Hailun Ding
Rutgers University

Shenao Yan
Rutgers University

Juan Zhai
Rutgers University

Shiqing Ma
Rutgers University

Abstract

Log is a key enabler of many security applications including but not limited to security auditing and forensic analysis. Due to the rapid growth of modern computing infrastructure size, software systems are generating more and more logs every day. Moreover, the duration of recent cyber attacks like Advanced Persistent Threats (APTs) is becoming longer, and their targets consist of many connected organizations instead of a single one. This requires the analysis on logs from different sources and long time periods. Storing such large sized log files is becoming more important and also challenging than ever. Existing logging systems are either inefficient (i.e., high storage overhead) or designed for limited security applications (i.e., no support for general security analysis). In this paper, we propose ELISE, a storage efficient logging system built on top of a novel lossless data compression technique, which naturally supports all types of security analysis. It features lossless log compression using a novel log file preprocessing and Deep Neural Network (DNN) based method to learn optimal character encoding. On average, ELISE can achieve 3 and 2 times better compression results compared with existing state-of-the-art methods Gzip and DeepZip, respectively, showing a promising future research direction.

1 Introduction

Log is a valuable source for many security applications such as forensic analysis [33, 41, 42, 50], system auditing [48, 66], Denial of Service (DoS) detection [13, 54] and intrusion detection [21, 23, 26, 27, 45]. In many scenarios like forensic analysis, the attacker has already left the system before analysis, and log is the only information that we can leverage to backtrack the attack fingerprints and understand the attack consequences [33]. During the last years, modern attacks like Advanced Persistent Threats (APTs) are becoming more and more frequent. In these attacks, the adversary can maintain secret access to highly confidential systems for a long time [56]. Moreover, APT groups tend to attack a few connected or associated targets together to gain more profitable information. For example, in the recent 2020 United States federal government data breach [14], attackers started to compromise the supply chain before October 2019, and the attack was not acknowledged until December 2020. It is suspected that attacks maintained secret access and performed data breach for over 8 months. This attack targeted over 10 U.S. federal, state and local governments, and 15 private sectors, including these that have well-trained employees and state-of-the-art (SOTA) defense techniques and products, such as Department of Defense and security firms like Palo Alto Networks. Performing log based analysis on such sized attacks requires examining a huge amount of data because of the large number of involved parties and long duration of the attack. Moreover, daily used programs generate a huge amount of data every day. According to previous studies [25, 29, 37], a single end user computer generates GBs log every day. Servers generate even larger sized log because of heavier workloads [63]. As such, storing logs is important and essential for security analysis, but also challenging because of huge storage overheads for large enterprises and organizations [26, 37, 63].

There are two existing mainstream methods to reduce the storage overhead. One is to directly remove redundant information from the log, and the other one is to compress log to reduce log file size. Many existing approaches [29, 37, 42, 63] proposed a set of rules to identify *redundant events* in logs and remove them without affecting the analysis result. However, these methods limit the analysis that can be applied on the log [29, 63]. For many security analyses, it is hard or even impossible to define what is redundant. In dependence based security analysis, examples of redundant events are repeated `read` or `write` system calls on the same system objects, e.g., a socket. While Machine Learning (ML) based methods identify possible DoS attacks by analyzing the frequency of `read` and `write` system calls to certain sockets. In summary, redundant events in one security analysis are no longer redundant in another scenario. As such, data reduction is not general to all downstream applications. Similarly, lossy data compression is not acceptable either, because it can lose critical information required by some security analysis. In conclusion, to provide

general service to various types of security analysis and reduce the storage overhead at the same time, it is essential to perform lossless data compression.

Traditional lossless data compression solutions [20, 30, 68] usually perform rule based processing. For example, the most commonly used compression method, *Gzip* [20], uses the LZ77 algorithm [68] and Huffman encoding [30] to compress files. Such methods can capture certain redundancy in the data, but they are usually not optimal. Recently, ML based data compression has been proposed [8, 43, 55, 61], and benefiting from the recent advances in DNN research, they [6, 19] have achieved lower compression ratios[1]. DNNs can better estimate the character distribution and catch the redundancy in given contexts compared to methods like Gzip [6, 19]. As a result, it can generate shorter encodings to represent the same data using less space. However, existing DNN based compression methods such as DeepZip [19], have a few drawbacks in compressing log files. Firstly, training DNN models is quite challenging. Logs contain natural language (NL) tokens, and training models for such tasks is well known to be hard [53]. SOTA models are huge, difficult to train, and cost a lot of resources. Secondly, existing methods cannot fully disclose the redundancy of log files [6, 19]. Different from a general NL artifact, log entries are well formatted, and hence, much contextual information is hidden. This causes extra difficulty in extracting the redundancy and compressing them for methods like DeepZip.

In this paper, we propose ELISE (Efficient LoggIng SystEm), a storage efficient logging system. It combines redundancy reduction and representation learning to fully uncover the redundancy in logs, and produces optimally sized log files. It creates a dictionary (referred as a reference table) to memorize structural redundancy in logs, and converts NL artifacts to numerical representations to simplify and speed up the process of training an encoder. After that, it leverages the trained encoder and arithmetic encoding to create the optimal representation in binary string format, which takes the minimal space to store. By doing so, ELISE can achieve lower compression ratios compared with existing methods. Our prototype is evaluated on various sized log files from five different systems including Linux, Windows, Apache, MySQL, and FreeBSD. One highlighted result is that ELISE achieves 9 times better compression ratio on HTTP logs compared with the traditional method Gzip. Moreover, it improves the runtime of DeepZip by a factor of 6.

In summary, we make the following contributions:

- We perform a thorough analysis of existing logging systems, and identify their limitations. They are either designed for a limited number of security analysis applications or storage inefficient (i.e., high storage overhead).
- We identify structural and contextual redundancies in log files, and propose a novel lossless log compression technique by using redundancy reduction and optimized encoder. It is more effective at capturing redundancies in logs by leveraging a novel preprocessing process and learning a high quality DNN encoder. It also optimizes the efficiency by converting NL artifacts to numerical representations.
- We build a prototype ELISE based on our proposed idea, and our results show that on average, ELISE outperforms existing methods, Gzip and DeepZip by 1.84 times in terms of compression ratios, and reduces the time cost by 5.63 times compared with methods in its kind.

Roadmap: In Section 2, we provide the background knowledge of log reduction and compression, a motivating example to show the limitations of existing work and a comparison of different methods including ours. Section 3 presents the design of ELISE, our storage efficient system. Section 4 shows the experiments we performed to evaluate the effectiveness, efficiency and security analysis support of ELISE, and an ablation study of ELISE. In Section 5, we discuss the advantages and limitations of ELISE and future research directions. We summarize related work in Section 6 and conclude this paper in Section 7.

2 Background and Motivation

Log analysis is an essential part of system development, which can be used for many tasks such as debugging [34, 47, 57, 60], performance measurement and trouble-shooting [16, 58, 62, 67], as well as many security applications including but not limited to intrusion detection [21, 23, 26, 27, 45], system monitoring [13, 24], attack investigation and provenance analysis [33, 41, 42, 50]. For example, Apache HTTP access log provides rich information for security auditing. In investigating APTs where the adversary customizes malware and residents in the system for months to years, log is the only source that cyber analysts can leverage to understand the ramifications (i.e., damages made by the attack) and root causes. Many security analytic systems for APTs and other cyberattacks are based on system level audit logs or program logs [26, 41, 50, 64].

One fundamental challenge of existing log based systems is the large volume of log data to store. In previous work [17, 26, 31, 37, 40–42, 63], researchers observed that a small sized enterprise needs to store hundreds of gigabytes log files even only for system level events. We also have observed the same phenomenon in our testbed. Notice that APTs can last for years. To support cyber attack analysis, logs have to be stored for years, causing a huge burden. Consistently collecting and storing such large amounts of log for months or even years waste too much storage space and also hinder the development of large-scale log security applications.

[1] Compression ratio is defined as the compressed file size over the original file size. The smaller, the better.

There are two typical approaches to solve this problem. One [37, 42, 63] is to remove redundant events to reduce storage overhead for specific security related investigations. For example, LogGC [37] observes that many system call events represent the same semantics, e.g., a sequence of *read* system calls reflect only one file read operation, and proposes to shrink the log by keeping only one of them. Despite that they have great effects on reducing the log size, these methods assume using analysis methods whose result will remain accurate without removed events, such as dependence analysis [37, 42] where all *read* system calls represent the same dependency. Therefore, the definition of "redundant events" is specific to analysis methods. As such, this approach can not be generally applied to different security applications: for example, an event frequency based anomaly detection method requires all events including the ones that are defined as "redundant" in dependence based analysis [37, 42]. The other approach, data compression, which is more general, stores the same information with less space. Data compression methods can be roughly divided into lossless compression and lossy compression. Because of the data integrity requirement of most security analyses, lossy compression is not suitable. Thus, lossless data compression is the most general and commonly accepted method for log storage optimization.

2.1 Lossless Data Compression

The goal of lossless data compression is to generate another encoding for the same contents so that the space usage is reduced. The basic idea is to use shorter encodings for more frequent elements. Such an embedding schema can be produced by using either traditional rule based approaches or machine learning based approaches. Traditional rule based approaches compress data by using observable and definable redundancy rules first, and symbol frequencies based encoding algorithms such as Huffman [30] encoding later. Represented by *Gzip*, most modern commercial and open-source compression systems use such a schema. ML approaches train probabilistic models to learn the statistical structure of data that can be coupled to arithmetic encoding, a stronger encoding algorithm than Huffman encoding, to better exploit the statistical redundancy in the input and improve compression results. Along this line of work, DNNs have achieved state-of-the-art results [10, 43, 55].

Rule based lossless compression. *Gzip*, the most representative rule based data compression method, works by first replacing repeated content blocks in the text with shorter mark strings, and then using Huffman encoding to encode the characters. It first uses a variant of the LZ77 algorithm [68], which detects all repeated contents in the file and replaces them with shorter marks: if we know the position and size of the first matched content, we can replace the following identical ones with a mark including the distance between these two and the length of the repeated contents. After obtaining the preprocessed file, Gzip then uses Huffman algorithm [30] to encode characters. The more frequently occurring characters in the file are encoded with fewer bits, thus compressing the file further. The mapping between a single character and its encoding will be recorded into a table, which is usually referred as the *reference table*.

ML based lossless compression. *DeepZip* [19] is state-of-the-art ML based lossless compression method. It uses a DNN and arithmetic encoding to better locate the statistical redundancy in inputs and improve compression effectiveness. DeepZip first determines a fixed-window size n, and uses a sequence of n characters in the input as an input to the DNN. Based on the given input, the DNN is trained to predict the distribution of the next character with a standard backward propagation method. Then, the arithmetic encoder encodes the character using the obtained predicted possibility distribution. If the character is predicted accurately, i.e., the character has the highest predicted value, it will be encoded with the fewest number of bits using arithmetic encoding.

Arithmetic encoding works differently from Huffman encoding, and is also used by many existing compression methods [9, 61]. It uses a probabilistic model that constantly updates the occurrence probability of each character at the current location based on the prediction of a certain predictor, and encodes them so that a character with a higher predicted probability will get fewer bits. As a result, better prediction results will lead to lower compression ratios, and it can guarantee that the compression is lossless.

2.2 Log Compression

Traditionally, Gzip is the most widely used method for log compression [12, 52]. However, its compression ratio is higher than ML based methods. Existing work observed that DNN based lossless data compression methods can achieve far better results than Gzip because of the capability of identifying statistical redundancy in data. Our evaluation results (see Section 4) also confirm such findings. For example, on Linux system log, DeepZip and Gzip achieve 1.60% and 3.57% compression ratio, respectively. Namely, files compressed by DeepZip take less than half space compared with Gzip. Considering the log size in large enterprises can be in PB or even larger, such a lower compression ratio can lead to significant savings in storage maintenance.

Despite the amazing effect in compressing textual data, DNN based compression methods have not been widely used in log compression. This is mainly because of its low efficiency. As shown in Section 4.3, DeepZip takes several hours to compress a small log file, e.g., 12.7 hours for a 0.8 GB file, which is unacceptable. The decompression process also takes longer time compared with methods like Gzip. Besides, existing compression methods are designed for general textual data, and do not leverage the domain knowledge of log files, which leads to non-optimal compression ratios.

2.3 Motivating Example

Motivated by the fact that all existing log collection systems are not storage efficient, we propose ELISE to solve this problem. Figure 1 illustrates how different methods work on a simplified log entry from the Linux Auditd system. It logs a system call event with syscall number 20 (`syscall` and `type`) and related context including `pid`, timestamp (`ts`), sequence number of this log entry in this logging session (`counter`), file paths (`path`) and so on.

The compression process can be roughly divided into three steps. The first step is to preprocess the original log from its original format to a compression friendly format. The second step is to produce an encoder that knows the representation of the log. The last step is to encode the log with the encoder and compress it. Gzip uses a deterministic encoding algorithm (i.e., Huffman encoding), so it does nothing in step 2. DeepZip does not perform any analysis on the original log file, and it has no preprocessing logic. ELISE has its own preprocessing step and an improved training procedure in step 2.

***Gzip*.** The workflow of Gzip is shown in Box **A** of Figure 1. In the preprocessing step, Gzip leverages the LZ77 algorithm to replace repeated strings. LZ77 algorithm uses a buffer to store recently scanned data and looks for new common substrings that are longer than a threshold (typically 3). When such a substring is found, LZ77 will replace it with a mark which is shorter than the substring (i.e., the substring length threshold has to be larger than the length of the mark). For the given example in Figure 1, LZ77 can find that `syscall` is a repeated string in ①, and then replaces it with the new mark $(16,7)$. The first number in this mark denotes the distance to the last appearance of this string (i.e., the second `syscall` is 16 characters away from its last appearance), and the second number represents the length of this repeated string (i.e., `syscall` is a 7-char long string).

In the log compression step (step 3), Gzip applies Huffman encoding to compress the log file. Huffman encoding counts the frequency of all characters in the file and encodes more frequent characters with shorter binary codes to ensure that the number of bits needed to encode the entire file is the minimal. For example, digit `6` has higher frequency than `1`, and as a result, encoding of `6` (i.e.,"`10`") is shorter than that of `1` (i.e., "`110`"). Moreover, such binary formats are shorter than the original encoding where all characters have the same length of binary bits.

***DeepZip*.** In Figure 1, Box **B** presents the overview of DeepZip. It does not perform any preprocessing for the target file ③. It trains a DNN based classifier as encoder on the given log file. The DNN takes a string in the log file as input and tries to predict the next character. For example, it uses the sequence `counter:13,syscall:20,type:syscal` to predict the next character. The output of the model will give a probability pair $(low, high)$ for all possible characters. In this case, it assigns $(0.1, 0.7)$ to letter "`l`". The first value denotes the sum of probabilities for all characters before "`l`" (e.g., letters `a` to `k`), and the second value adds the probability of letter "`l`". In this case, the probability of the next letter being letters before "`l`" is 0.1 and letter "`l`" itself is 0.6, thus the second value is 0.7.

DeepZip uses arithmetic encoding to compress logs in step 3. A detailed example of arithmetic encoding is described in Section 3. Compared with Gzip, DNNs have better distribution estimations for characters than Huffman encoding, enabling lower compression ratios.

ELISE. Because of DNN based encoder, DeepZip is more effective in compressing files compared with Gzip. However, we observe that DeepZip is still not optimal. Firstly, DeepZip works on non-optimal log formats. Logs are highly redundant, and DeepZip ignores such redundancy. Secondly, training a good DNN model in DeepZip is very difficult and resource consuming. This is because it directly trains on all possible characters, and the input space is huge.

Log file is a special type of inputs for compression tasks. It has a few unique characters. Firstly, all log files can be or have already been well formatted. Entries in a log are usually generated by `printf`-family functions or similar libraries in other programming languages. These functions require a format string, which essentially provides a template to parse the log entry [65]. Existing projects like LogStash [1] can help format log files from various sources. Secondly, log files have a limited vocabulary. Besides words in log statement templates, variables in each log entry are mostly either well formatted (e.g., IP addresses) or consistently appear in the log (i.e., process names).

Based on the analysis of existing methods and log files, we propose ELISE (BOX **C** in Figure 1). It features a preprocessing step that reduces all structural redundancy from log entries, which effectively reduces the file to compress; and converts all strings/characters to a numerical representation, which reduces the input/output space for DNN based encoder and numerical values make it easier to train. For the example in Figure 1, ELISE applies four different preprocessing rules to convert the entry to a much shorter one, and also converts them into numerical formats making it easier to train the encoder. The DNN training and encoding steps are very similar to that of DeepZip. But with modified log Entries, it can compress the log with shorter bit strings and a faster speed. Based on our evaluation in Section 4, ELISE is around 5.63 times faster than DeepZip.

Compared with Gzip, DNN based encoding has a stronger capability of capturing information redundancy and provides better encodings, and hence has lower compression ratios [19]. That is why both DeepZip and ELISE have lower compression ratios than Gzip. ELISE further improves DeepZip by applying preprocessing rules to capture different levels of redundancy and converting all data into numerical formats to improve the training speed.

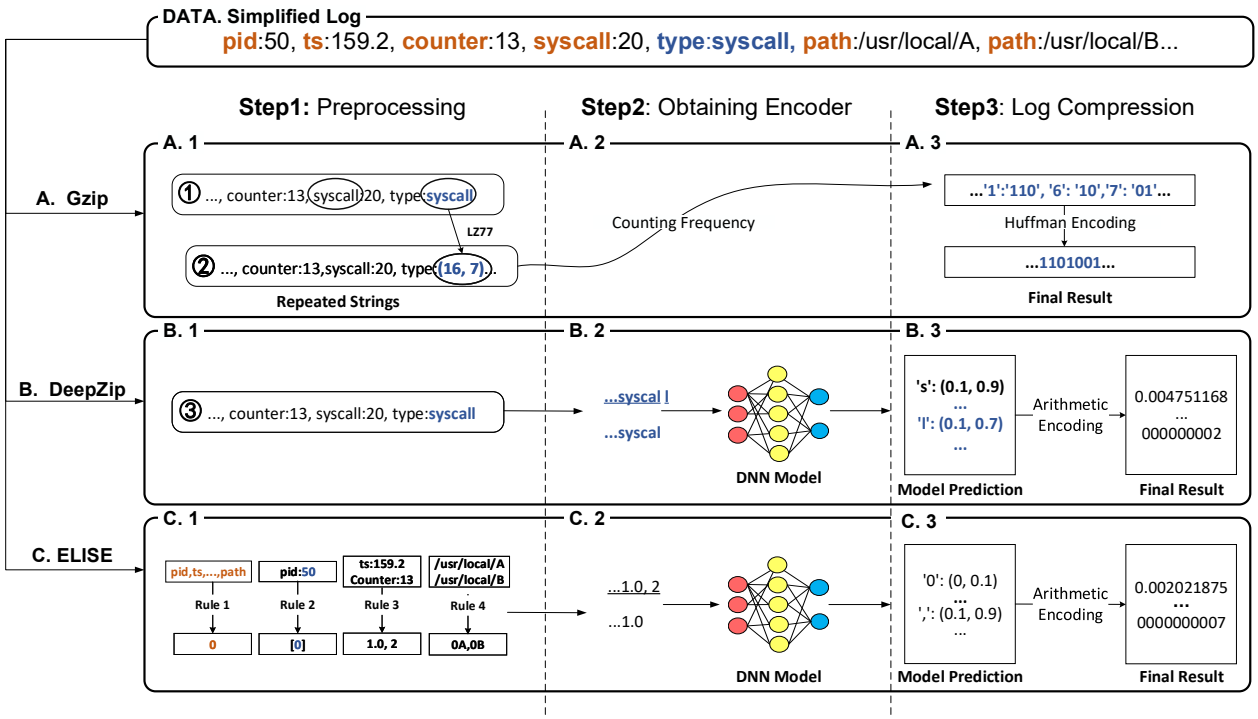


Figure 1: Example of Gzip, DeepZip and ELISE.

3 Design of ELISE

In this section, we first give an overview of ELISE design and a definition of our threat model, and then introduce each component including preprocessing, DNN based encoder and data compression and decompression.

3.1 Overview of ELISE

The overall workflow of ELISE is shown in Figure 2. After receiving logs from various sources, ELISE first converts them into a united format, and then splits large files into small ones for concurrent processing (component A in Figure 2). Then, the log files are preprocessed to remove the redundancy (component B). For each of them, we train a small DNN as its encoder (component C). When compressing the log, we leverage the trained encoder and arithmetic encoding to produce the final output. The dashed box includes all the artifacts (including a DNN model, a reference table and the compressed data file) that are required to reproduce the raw log. In the following sections, we will introduce each component and how ELISE can be deployed in real world scenarios (e.g., avoiding retraining for all files and improving prediction accuracy to improve compression performance).

Scope of the paper. ELISE is designed to be part of an enterprise security infrastructure. It provides the capability of storing large size logs with the minimal space without information loss. ELISE is suitable for enterprise level systems which generate a large amount of log data and store them for various analysis, and hence the storage overhead is high. Also, ELISE is designed for centralized log storage. Namely, instead of storing logs on individual end user computers or servers, logs from different sources are stored in a well protected server. This is a common practice for modern enterprises. A centralized server can provide better data integrity protection and storage optimization. ELISE guarantees the integrity of logs during its processing, and assumes the integrity of logs from sources (both in compression and decompression).

3.2 Log Formatting

ELISE accepts logs from different sources, and the first job it does is to normalize them into the same format. To do this, ELISE leverages LogStash [1] to convert all files to JSON format. Yuan et al. [65] demonstrated that most logs are generated by the `printf`-family functions or their variants in other languages. Such logs use the format string as their first parameter. As a result, log entries can be organized in a `(key:value)` pattern with constant strings in the format string as keys and runtime variable values as values. Thus, JSON is commonly used to store logs. LogStash is a log normalization tool which can parse log and convert their formats based on given rules. It has built-in support for many popular applications and systems already. After formatting all logs, ELISE also splits the large files into smaller ones to enable parallel data compression and achieve high efficiency.

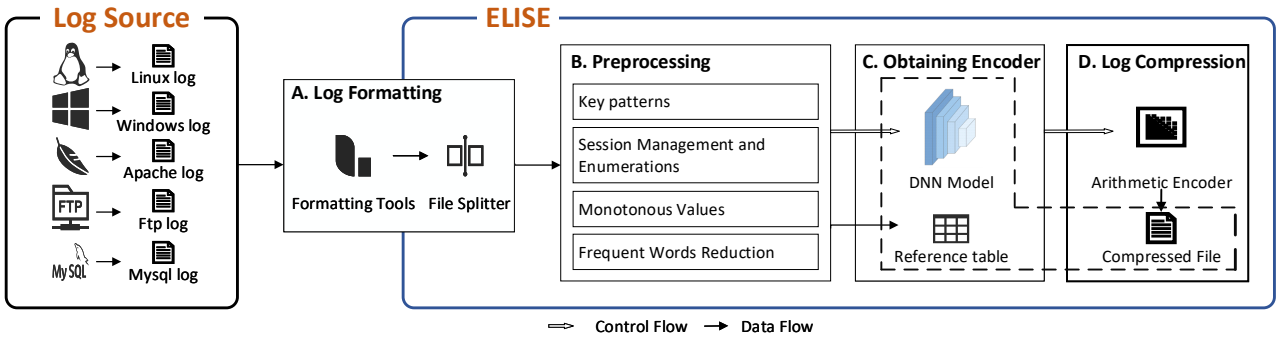


Figure 2: Workflow of ELISE.

3.3 Log Preprocessing

Before obtaining the encoder and compressing the log file, ELISE first applies a set of rules to remove the redundancy and prepare them for DNN based encoder training.

Preprocessing 1: Key patterns. Different from other rules, preprocessing rule 1 focuses on reducing keys belonging to the same type of log entries (e.g., the keyword "pid", "ts" and "path" in system logs) with short numerical codes. Other preprocessing rules reduce redundant values (e.g., the pid number "50"). Recall that each log entry consists of a constant template part and some runtime variable values (Section 3.2). There are a limited number of templates in a program, but they can generate countless log entries. Even though the log formatting step in ELISE can remove some redundant items (e.g., by shrinking a natural language sentence to a single word), the JSON log file still has a lot of redundancy, especially the repeated keywords and their orders. For example, in Linux Auditd logs, there are only four different types of logs despite millions of entries.

ELISE automatically converts all fixed keywords into a numerical sequence with the minimal number of digits. For example, the log entry `(pid:50, ts:159.2, counter:13, syscall:20, type:syscall, rs:success)` will be converted to a new string `(0: {50, 159.2, 13, 20, syscall, success})` with a reference code r. Then, we create a reference table containing the rule to convert the numerical value back to its original format. In this case, we mark the rule corresponds to the reference code r as *(0)* $\overset{r}{\mapsto}$ *(pid, ts, counter, syscall, type, rs)*. During decompression, ELISE will query the reference table to find the translation rules and then apply them. To automatically discover such keys, we leverage LogStash rules. When LogStash parses the log, it also detects the constants and variables in these entries, and ELISE directly uses the constants keys in the converted log as our key patterns.

Preprocessing 2: Session management and enumerations. Many system and software activities from different users (or clients) use sessions. Log entries belonging to the same session will share a lot of variables and hence, they have many repeated values. For example, in Linux system log, log entries belonging to the same process have the same values for `pid`, `ppid` (parent process id), `hostname` and `arch` (system architecture), and most of them have the same values for `uid`, `gid` and so on.

Storing such repeated values will lead to higher overhead. To alleviate this problem, we propose to summarize session related fields into a tuple stored in the reference table. Similar to key patterns, we define a translation rule between the common values and compressed values, and then assign the compressed log entry with a reference code. For example, for `(pid:50, arch:03, uid:3345)`, we map it to `(pid:0, arch:0, uid:0)` with a reference code t. In the reference table, we have *(0,0,0)* $\overset{t}{\mapsto}$ *(50, 03, 3345)*. Notice that even less common, values of some fields belonging to the same process do change from one to another such as `uid`. Functions like `setuid()` can change this value in a session. In this case, we just add a new translation rule in the reference table and increase the compressed code for these fields (e.g., from `0` to `1`). Moreover, we reorganize logs into sessions by aggregating log entries in the same session into the same region to reduce storage overhead. Different from preprocessing 1 which analyzes the redundancy in keys, preprocessing 2 focuses on the repeated patterns in values. Automatically discovering such patterns is a classic data mining problem. To solve this problem, we first extract all values belonging to the same keys (obtained from preprocessing 1). Then, we perform an unsupervised clustering analysis on corresponding to find such patterns. Specifically, we use a TF-IDF to get the frequently values, and then uses the K-means algorithm to cluster all logs. To determine the optimal value of k, we use the Silhouette method. After that, we also manually check whether these keys are correct. Lastly, we summarize these patterns into rules, apply them to the original log, and update the reference table accordingly.

Notice that even though some fields are not related to sessions, they are also clustered because of a limited number of possible values in a given set of log files. For example, forensics systems usually only log I/O and process related system calls, and their logs have a finite set of system call numbers.

Even though they are not session related fields, we also do similar preprocessing for them to reduce the redundancy.

Preprocessing 3: Monotonous values. In various types of logs, it is common to see some fields that have monotonous values even though they are not identical. Because of this, continuous log entries may share a lot of common characters or numbers. Such fields include timestamps, counters used for logging statistical information, transaction identifiers for databases, etc. For example, most logs use the UNIX time to record when the event happens. A single UNIX timestamp in Linux Auditd log is a 14-character long string including 13 digits and a dot symbol. Recall that ELISE separates huge log files into smaller ones (Section 3.2) for parallel processing. Most timestamps in the same log file have identical digits at the beginning representing the same year, month and day, which is redundant.

For these monotonous value fields, we first record the smallest value and then replace the original value with the offset values in the rest of the log. Such incremental logging can help remove a lot of unnecessary digits. When decompressing, we recalculate the actual values by using the smallest value and offsets. Discovering such monotonous value fields is also simple. We first choose the numerical fields, and then simply test whether they are monotonous in log sequence order.

Preprocessing 4: Frequent words reduction. Besides the previous redundancies, there are still word and string level redundancies. For example, folder and file paths commonly share a long prefix. Moreover, folder names in paths may share a lot of common substrings with other fields like process names and binary names. To remove such redundancies, we introduce a two layer frequent words compression technique leveraging existing algorithms (i.e., finding the longest substrings).

Firstly, we compress strings that belong to the same type of log entries. For example, in Linux Auditd log, we gather all `PATH` type entries, and find common substrings among them. The log entry type can be identified by its keyword patterns. Secondly, we apply the algorithm again globally to reduce word level redundancy. One key difference of ELISE in this step from existing algorithms (e.g., LZ77) is that instead of replacing the strings with a mark with offsets and length, ELISE directly substitute them with numerical values (similar to preprocessing 1). In log files, all formats and fields are well-defined, and using a translation rule plus numerical value saves more space.

3.4 Encoder and Data Compression

Similar to DeepZip, ELISE uses a trained encoder and arithmetic encoding to compress a data file. In this section, we will show how to obtain such an encoder and leverage it to perform data compression and decompression.

Encoder. Theoretically, all model architectures that support processing sequential data potentially can be used as our encoder. In our implementation, the design of DNN is shown in Figure 3. It is a Long Short Term Memory (LSTM) model, $\mathcal{M} : S^L \mapsto [0,1]^N$. This model takes an L-length string $s = \{s_1 s_2 \dots s_L\}, s \in S$ as the input to predict the next character s_{L+1} in this sequence. It consists of two LSTM layers, a batch normalization layer, two fully connected layers, and the last layer, a SoftMax layer outputs an N-length long vector $y = \{y_1, y_2, \dots, y_N\}$, where N is the total number of possible characters and each $y_i \in [0,1], arg(y_i) \in [1,N]$. Each label of the output vector represents a possible character (e.g., 'a' to 'z'). For the i-th character s_i, we use string $\{s_{i-L} \dots s_{i-1}\}$ to predict s_i. Training such a model is a typical classification task where the input is a fixed length vector and the output is a one-hot encoding of characters. We used categorical cross-entropy loss function and Adam [35] optimizer with default settings to train the models.

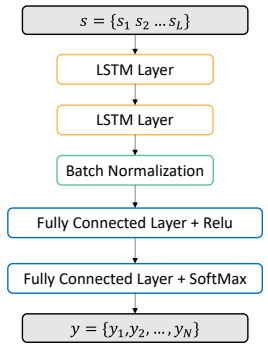


Figure 3: Model Architecture of DNN Model in ELISE.

Pre-trained encoder and partial data training. Training an encoder from scratch can take a long time. We notice that system logs are highly redundant which enables us two optimizations: partial data training and using pre-trained models. Partial data training means we do not train the encoder on the entire dataset but just a small part of it. The other solution is to use pre-trained models and only finetune them on new data files. Also, we can combine these two approaches together: finetuning a pre-trained encoder on partial data. Notice that even if the trained encoder cannot achieve high prediction accuracy on test data, it still can be used for lossless data compression and decompression. Models with higher prediction results will lead to lower compression ratios (i.e., better results) and vice versa. Thus, these optimizations are trade-offs between compression ratios and compression time. In Section 4.4.1, we perform a study on how such optimizations affect ELISE.

Encoding and data compression. After obtaining the trained encoder, we combine it with arithmetic encoding to encode characters. First, we use the trained model to predict each character in the file F^c whose length is c and get their outputs. For the i-th character s_i, we get $y^i = \{y_1^i, y_2^i, \dots, y_N^i\}$. Also, we initialize two variables (A,B) as $(0,1)$ which be

used to store intermediate results, and perform encoding by applying the following equations:

$$\begin{aligned} &arg(y^i_{t_i}) = s_i,\ \ i \in \{1,\ldots,c\},\ \ o^i_j = \frac{\sum_{k=1}^{j} y^i_k}{\sum_{k=1}^{N} y^i_k} \\ &A_i = A_{i-1} + (B_{i-1} - A_{i-1}) * o^i_{t_{i-1}} \\ &B_i = A_{i-1} + (B_{i-1} - A_{i-1}) * o^i_{t_i}, \end{aligned} \tag{1}$$

After doing this for all c characters, we can get a (A_c, B_c), and to encode the whole log file, we only need to pick a number f which satisfies $A_c \leq f < B_c$ and f has the shortest binary representation as our final compressed data file.

Notice that after updating A and B for several iterations, it can get small, and we have to use customized data types to represent such small numbers. As we will show in Section 4, storing a single number like this can take a few MB. For the first L characters at the beginning of a file, we cannot find a corresponding input to the model. A common practice to solve this problem is just assuming a fixed distribution.

Data decompression. Decompressing the data from f is a reverse process of data compression. For the i-th characters s_i, we first obtain the prediction result $y^i = \{y^i_1, y^i_2, \ldots, y^i_N\}$ via the DNN model. Similar to the compression process, we need (A, B) whose initial values are $(0, 1)$ to store our intermediate results and apply the following rules for decompression:

$$\begin{aligned} &o^i = \{o^i_1, o^i_2, \ldots, o^i_N\},\ \ o^i_j = \frac{\sum_{k=1}^{j} y^i_k}{\sum_{k=1}^{N} y^i_k},\ \ i \in \{1,\ldots,c\} \\ &s_i = arg(y^i_z),\ \ f \in [A_i, B_i),\ \ z \in \{1,\ldots,N\} \\ &A_i = A_{i-1} + (B_{i-1} - A_{i-1}) * o^i_{z-1} \\ &B_i = A_{i-1} + (B_{i-1} - A_{i-1}) * o^i_z, \end{aligned} \tag{2}$$

Recall that the trained encoder cannot predict the first L characters in a given file, and they are encoded by using a fixed distribution. For these characters, we reuse this fixed distribution during decompression. After this, we leverage the stored reference table to undo all the preprocessing operations to recover the raw log.

4 Evaluation

We built a prototype based on our proposed idea, and evaluate it using real world data to validate its effectiveness, efficiency and support of real word security analysis applications. We first introduce our setup for our experiments including configurations of the server and datasets (Section 4.1). In Section 4.2, we evaluate its effectiveness by comparing with existing methods DeepZip and Gzip on different sized log files. To measure the efficiency of ELISE, we measure the time cost of individual steps including preprocessing, encoder training, data compression and decompression; and the usage of memory. Moreover, we perform an ablation study on the configurable parameters in ELISE and also alternative designs that can help speed up model training (partial data training and using pre-trained models. Lastly, we use one real world security application, forensics analysis to validate if ELISE can guarantee the log integrity.

4.1 Experiment Setup

Our prototype of ELISE is implemented in Python using Keras [7] with TensorFlow as the backend [4]. If not specified, all experiments are conducted on a Ubuntu 18.04 machine equipped with a GeForce RTX 6000 GPU, 64 CPUs and 376 GB main memory.

Datasets. Our evaluation datasets are collected from 3 different operating systems and 3 popular server applications. We follow the standard guidance to collect datasets on our experimental machine properly. Specifically, we randomly start the data collection procedure, guarantee long enough collection durations, and perform a manual post-modern check to reduce biases and make sure that used workloads are representative. Details of these files are listed in Table 1. We collect system logs from Linux, Windows and FreeBSD and application logs from Apache2, VSFTP and MySQL which run on top of Linux. For system logs, we collect system events which include but are not limited to system calls, monitored process (by default, all processes), specific files (e.g., `/etc/passwd`) and user account information (i.e., `euid`). On Linux, we use the built-in system event collector `auditd`. On Windows, we utilize the `Sysinternal` tools such as *Process Monitor*. On FreeBSD, we leverage `DTrace` to gather such information. All applications we use have their own application logs, and we directly use their built-in tools and default configurations. Apache2 and VSFTP logs mainly contain the connection information (e.g., source IP address, port number, client information) and access information (e.g., file access and downloading behaviors). MySQL logs not only the connection information and queries, but also its internal transaction information. In Table 1, we also list the size of the studied log in the last column, and for short, we give each log file a name which is listed in the first column. The logs are collected with typical workloads. For Linux, Windows and FreeBSD operating systems, they are used as end user machines running office software suites, browsers, editors, note-taking software, email clients, calendars and so on. The Apache2 server is hosting both static and dynamic websites such as blogs and wiki sites. The FTP server provides file sharing service for our organization, and the MySQL databases contain the records for several relational databases. To ensure the consistency of workloads, we collect log data under the same condition and we manually compare them with logs collected in the production environment.

4.2 Effectiveness of ELISE

Experiments. We use log files generated by different systems and applications to evaluate the effectiveness of ELISE, and

Table 1: The Overview of Datasets*.

Name	OS	Collector	Event Type	Size (GB)
Lin	Linux	Auditd	System calls, I/O information	0.8, 7.7, 16.1
Win	Windows	Sysinternal	Process and user information	0.7, 7.4
Htp	Linux	Apache2	HTTP connection and access	2.4, 24.3
Ftp	Linux	VSFTP	FTP connection and access	0.9, 9.5
Sql	Linux	MySQL	MySQL connection and actions	1.0, 10.0
BSD	FreeBSD	DTrace	System calls, I/O information	0.8, 8.3

* In this paper, we use `Name-Size` to refer one dataset.

compare it with existing methods Gzip and DeepZip by measuring the compression ratio (*CR*) which is defined as the total size after compression (including models and compressed data) over the original file size. Gzip embeds its reference table as part of its final output file (with some other engineering optimizations to reduce its size), and thus we just need to measure the size of the output file to determine its total size after compression. For ELISE, the final results contain two parts, i.e., the trained model and the data file containing the reference table and compressed contents. The total size after compression for DeepZip will be the sum of the size of the model and the size of the data file.

ELISE has a set of configurable parameters, and we use the default setting in our experiments. To be more specific, we use 10 parallel processes to perform the compression. For a fair comparison, ELISE uses the same model architecture as DeepZip. More results on the effects of different configurations are presented in Section 4.4.1. Because DeepZip is slow and takes a significantly long time for large files, we choose to use 13 hours as a time limit for the compression process. If one method is taking too much time (i.e., longer than the threshold), we will mark it as `T/O` (i.e., time out).

Results and analysis. Results of our effectiveness evaluation are summarized in Figure 4. In Figure 4(a), the X-axis denotes used datasets and Y-axis shows the compression ratio for each method. For the rest figures (Figure 4(b) to Figure 4(h)), we use 0 to 9 in the X-axis to represent individual files after splitting. The dash lines in each figure show the average compression ratios on all these files for the two methods, and their concrete values are marked in the Y-axis. Notice that we do not show results for DeeZip in these figures. The reason is that we set our timeout threshold value to be 13 hours. That is, when the compression takes longer than 13 hours, we stop it and cannot report results for corresponding experiments. DeepZip timeouts on all large files. As will be discussed in Section 4.2, DeepZip takes 12.7 hours to process a 0.8 GB sized Linux system log, and cannot scale to large files.

From results on small sized files in Figure 4, we can see that although compression ratios vary for different log files, ELISE achieves the best compression ratios among the 3 compared methods. Overall, ELISE is 1.13 $\sim$ 12.97 times better than Gzip and DeepZip. Different compression ratios of three methods show that all three methods can successfully detect the redundancies in the log files, but at different levels. Our results indicate the advantage of ELISE in log compression compared to existing compressors and good generalization to different types of log files. All three methods achieve the best compression result on HTTP logs and the worst on BSD logs. This is because HTTP logs have more redundancies than others and BSD logs contain less redundant information. We can see that ELISE has lower compression ratios for all datasets. On average, its compression ratio is only 36.96% and 54.41% that of Gzip and DeepZip. In other words, it can shrink the data size by half or even more compared to the other two methods. The best results are both achieved on HTTP with over 11.08 times improvement on average, and the worst results are both obtained on the BSD log.

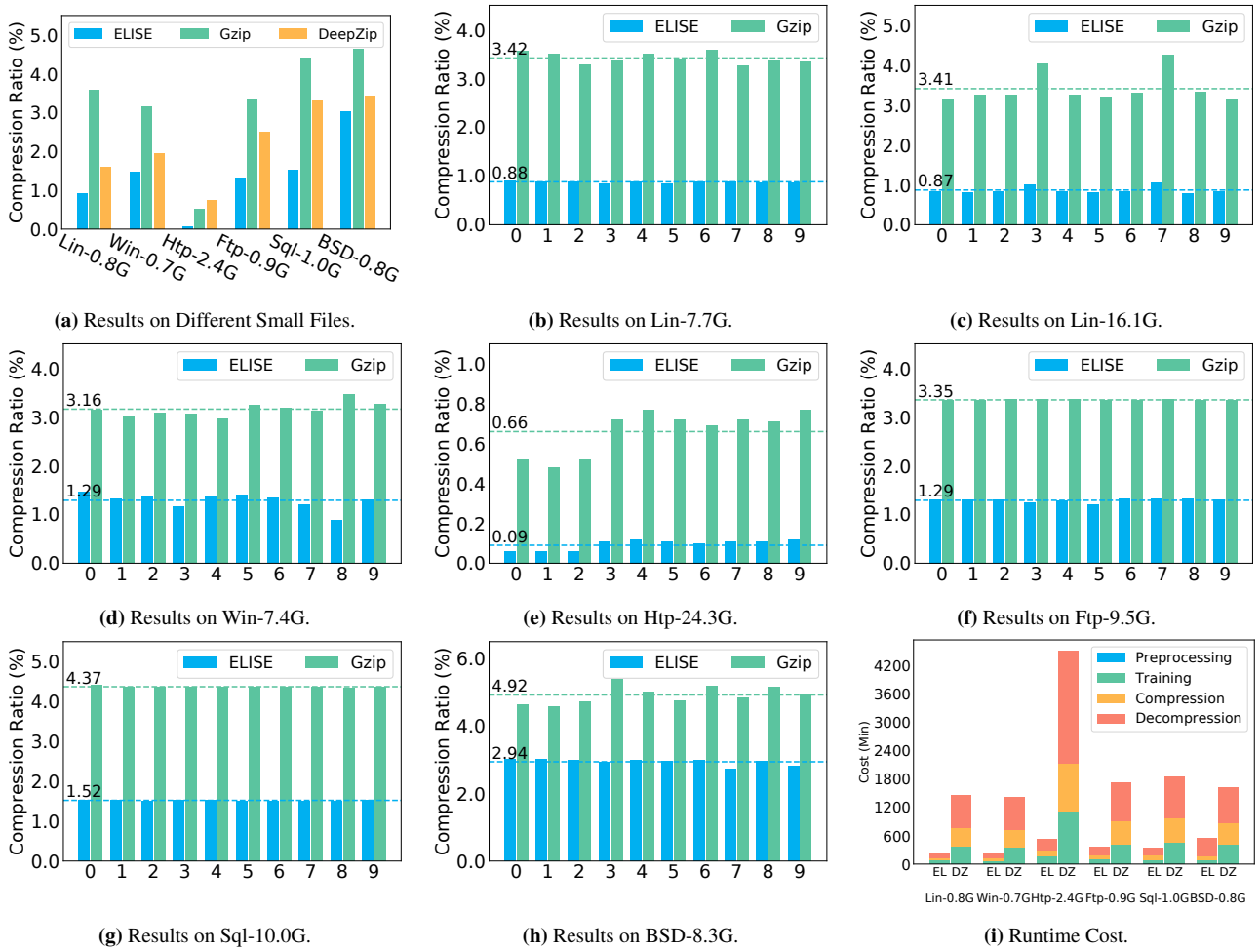
Observations on large sized logs are consistent with small sized logs. For the same type of logs, they show similar compression ratios for all methods and the relative compression results of different methods are also the same. Individual small files split from the same large file have similar compression ratios with Gzip and DeepZip, leading to consistent final results. Notice that the compression ratio for a certain method can be affected by the size of the file. The statistic numbers for a character in small sized files tend to be biased. However, log files are highly redundant, and we do not observe such phenomena in our experiments.

4.3 Efficiency of ELISE

We further evaluate both the runtime cost and memory usage of ELISE to understand its efficiency. In these experiments, we use the same settings for DeepZip and ELISE including the same model architecture and training parameters. Specifically, we use the model described in Figure 3. The batch size is 4096 and the learning rate is 0.001. The selection of batch size and learning rate are discussed in Section 4.4

4.3.1 Runtime cost of ELISE

The runtime cost can be divided into four parts in ELISE: log file splitting and preprocessing, DNN model training, data compression and decompression. To measure the cost of each step, we profile the execution of ELISE on 6 datasets and log the runtime of each step. Gzip does not require training a model, and its runtime cost is divided into only two parts: compression and decompression. For DeepZip, we measure its training time, compression time and decompression time. Results are summarized in Figure 4(i). Each stacked bar represents the time used for processing the log file, and its four components denote the time for each step. The X-axis shows the dataset and compression method, and the Y-axis measures the time cost in minutes. The time cost of Gzip is not shown in this figure because of different compression steps. We include a discussion in the following result analysis. Also, we only show the results for small log files to avoid `T/O` for DeepZip. In practice, the time cost is almost linear with the log size if



(a) Results on Different Small Files.

(b) Results on Lin-7.7G.

(c) Results on Lin-16.1G.

(d) Results on Win-7.4G.

(e) Results on Htp-24.3G.

(f) Results on Ftp-9.5G.

(g) Results on Sql-10.0G.

(h) Results on BSD-8.3G.

(i) Runtime Cost.

Figure 4: Effectiveness Evaluation on Different Files (EL is short for ELISE and DZ is short for DeepZip).

the platform and used applications are the same. Thus, small log files can represent the relative cost of different methods. Many parameters in DeepZip and ELISE can affect the runtime costs such as batch size, and we perform an ablation study in Section 4.4.

Results and analysis. First, we do not include Gzip in Figure 4(i). It is the fastest method among all methods. On average, it takes Gzip 0.17 minutes to compress and 0.06 minutes to decompress the tested 6 small logs. Compared to Gzip, DeepZip and ELISE are slower mainly because they require training a DNN model and querying the model many times during compression and decompression. Such techniques are new and have not been optimized on both the hardware and software stacks. Considering that DeepZip and ELISE have better compression ratios than Gzip (over 11.08 times better, Section 4.2), we do envision DNN based compression as a promising research direction.

On average, DeepZip is 5.63 times slower than ELISE, even though ELISE has an additional preprocessing step. This shows that ELISE is more efficient compared to DeepZip, which is the benefit of our design. By leveraging domain specific knowledge of log files, ELISE converts most of the texts in the log to numerical formats which enables less training time and more efficient data compression/decompression. The preprocessing step takes negligible time compared to the other three steps in ELISE. Overall, it only scans the log file, applies lightweight rules to generate the reference table and converts log format.

Breaking down to individual steps, both ELISE and DeepZip spend most of the time on data decompression which takes almost as long as the other steps combined. In arithmetic encoding based methods, the decoding phase commonly takes a longer time, as also observed and analyzed by existing work [46]. This is because it requires searching the correct value for variable z in Equation 2. Another interesting observation is that the training phase does not take too much time compared with data compression. This is because arithmetic computations in training (mostly FP32 computations) are faster on modern systems. For data compression (and decompression), we have to use customized data types to store

intermediate results, which leads to higher runtime overhead. Even though training requires a lot of time, these operations have been optimized on modern software and hardware stacks. DeepZip takes less time on log files compared with other general NL based tasks. This is mainly because logs have less vocabulary, and their distribution is simpler than other NL artifacts. For the same log file, ELISE spends 578% less time on training compared with DeepZip. This shows the benefits of preprocessing them by removing the redundancies and converting them to numerical formats.

For files in the same type (e.g., Linux system logs or BSD logs) , the processing time and file size should have a linear relationship for DNN based compression methods because the compression methods need to compress more data. Surprisingly, this is also true for most files from different sources. For example, it takes similar time for both DeepZip and ELISE to compress and decompress `Win-0.7G` and `Lin-0.8G`, while `Htp-2.4G` costs 3 times longer compared with these two datasets. `BSD-0.8G` is an exception. The main reason is that, unlike others, BSD logs file has a lot of nested structures. Therefore, handling such a complex log structure requires more time.

4.3.2 Memory Cost

We measure both GPU and main memory cost of ELISE during its execution. Main memory usage is mainly affected by the buffer that is used to store raw data, and concrete numbers are omitted. The GPU and main memory usage of ELISE are shown in Figure 5. In this experiment, we use the default parameters (i.e., learning rate, split size, fine-tuning ratio and batch size) for all systems, which ensures that the comparison is fair. Also, as reported in Section 4.4, default parameters lead to the overall best result for DeepZip. The X-axis represents datasets, and the Y-axis represents memory costs in megabyte scale. From the figure, we observe that processing different files consumes the same amount of GPU memory for individual stages. For different stages, training uses more GPU memory than compression and decompression. Specifically, model training costs 1,632 MB and compression/decompression takes 700 MB, which can be supported by all mainstream GPUs. The consumption of GPU is dominated by the size of the model and the number of samples we use in each batch. Since our LSTM model is small and the default batch sizes are the same, the GPU memory consumption is also tiny and similar. Training takes larger GPU memory because it stores gradients to support backward propagation.

4.4 Ablation Study

In this section, we perform an ablation study for ELISE. Based on this, we try to answer how to select optimal values in practice.

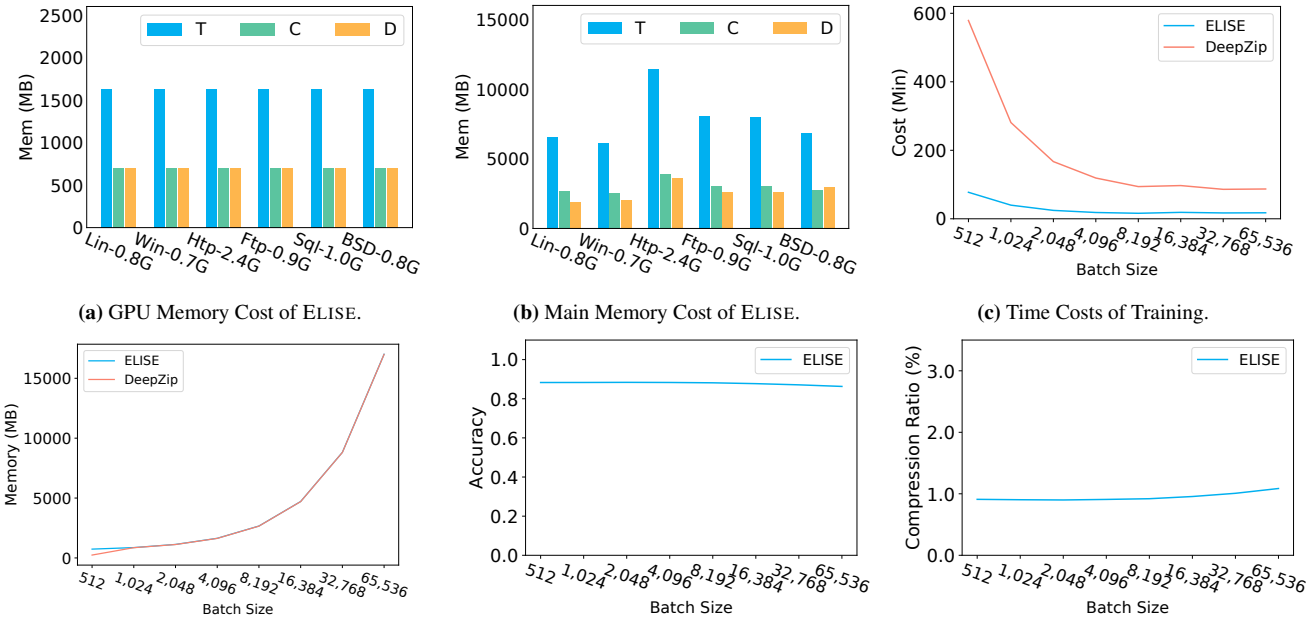
4.4.1 Ablation Study for ELISE

ELISE has a few configurable parameters that may affect its performance: the size of split files, batch size and learning rate in model training. We choose the hyperparameters by following the standard procedure in machine learning. Specifically, we leverage a small dataset randomly drawn from the training dataset and train with different hyperparameters. By comparing different configurations, we can pick optimal hyperparameter values to train on the whole dataset. We then evaluate the impact of each parameter by varying them independently. In Section 3, we also mentioned that ELISE can use partial data for training and pre-trained models to optimize its runtime. We also evaluate the effects of these two techniques in this section.

Split file size. When compressing a large file, ELISE automatically divides the file into smaller ones and compresses them independently. By default, we split large files into 10 smaller ones and make sure their sizes are between 0.7 GB to 4.0 GB (constrained by our memory). The size is configurable. To test the effect of this parameter, we use Lin-7.7G as our dataset, and configure the size from 0.1 GB to 1.2 GB. The results are shown in Figure 6(a). The X-axis shows the split size. The blue line shows the final compression ratio (Y-axis on the left), and the red line reflects the accuracy of the trained model (second Y-axis on the right). Lastly, the green line shows the ratio of compressed model over the total final artifacts. Because we use a fixed model architecture, the model size is a fixed value. When the split size is small and model accuracy is similar, the size of the model itself becomes the dominant factor of the total final size (and hence the compression ratio). This green line is used to study this phenomenon.

From this figure, we can see that when the size is larger than 0.6 GB, the compression ratio reaches a saturation state. This is because the accuracy of DNN cannot be further optimized as indicated by the red line. With the same accuracy of the model, the probability distribution of individual letters will not change much. As a result, we end up with similar encodings for individual letters which leads to saturation in compression ratios. On the other hand, when the size is smaller than 0.6 GB, the DNN encoder has high accuracy but the compression ratio is high. Notice that for different split sized files, the DNN model has the same architecture and size, which is a dominant factor when files are small. This leads to such high compression ratios. On different log files, the saturation points are slightly different, but such a phenomenon exists on all of them. Notice that the results do not mean that ELISE will be sensitive to the split size in practice. This is because in the real world, common log file sizes are far larger than 0.6 GB, which means the compression ratio will be stable and small.

Batch size. Batch size is a typical parameter that can affect the DNN training and prediction including its training time, accuracy, resources consumption, and transitively, compres-



(a) GPU Memory Cost of ELISE.

(b) Main Memory Cost of ELISE.

(c) Time Costs of Training.

(d) Memory Costs of Training.

(e) Accuracy of Training.

(f) Compression Ratios.

Figure 5: Memory Cost of ELISE and Evaluation Results with Different Batch Size Settings (T, C and D are shorts for training, compression and decompression) .

sion ratio. To study its effects, we use different batch sizes on Lin-0.8G dataset and collected the time cost, memory usage, model accuracy and the size of compressed files. The batch size varies from 512 to 65,536. As a comparison, we also do the same experiments on time costs and memory costs evaluation using DeepZip. The experiment results are shown in Figure 5.

Overall, the model accuracy (0.86~0.88) and the compression ratios (0.90%~1.09%) are impervious to the change of batch size. This is mainly because of our preprocessing, which converts training on natural language artifacts to numerical values. This makes training easier and scalable to larger training batch sizes. Here we do not compare ELISE with DeepZip as similar results have been presented in Section 4.2.

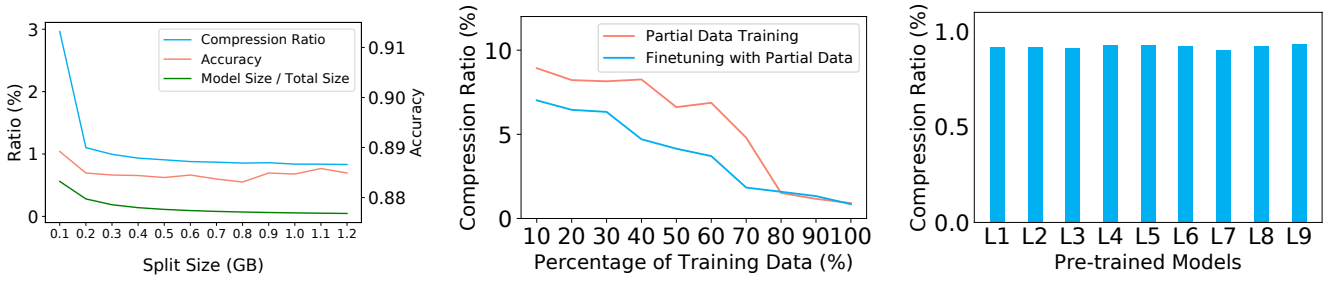
On the other hand, Figure 5(c) and Figure 5(d) show that batch size impacts training time and GPU memory occupation. With the increase of batch size from 512 to 65,536, the GPU memory occupation rises from 736 MB to 17,008 MB and the training time decreases from 77.40 to 16.07 minutes per epoch. Similarly, as the batch size increases, DeepZip consumes more GPU memory but spends less time in training. Comparing the two methods, the time cost of DeepZip is significantly higher than that of ELISE, but its GPU memory usages are comparable. The results reveal that increasing batch size could significantly speed up the training process and the improvement is not linear. The training speed does not increase any more after the batch size is large enough. This conclusion is consistent with previous work [18].

Learning Rates. Learning rates can affect the model accuracy and transitively, the final compression ratio. To measure its effects, we adopt 5 different learning rates (from 0.1 to 0.00001) to train DNN models and collect final compression ratios on the Lin-0.8G log. The results are shown in Table 2. As shown in the table, larger learning rates lead to lower prediction accuracy of DNN and higher final compression ratios, which is consistent with existing work. This is because higher learning makes it easier to skip optimal values during optimization, leading to non-optimal results. On the other hand, small learning rates result in slow convergence and make it hard to skip local optimal values, which is also undesired. To solve this problem, we follow the standard recommendations in ML community, and use adaptive optimization methods (i.e., optimizer will dynamically adjust learning rates) with a relatively large learning rate 0.001 as our default setting.

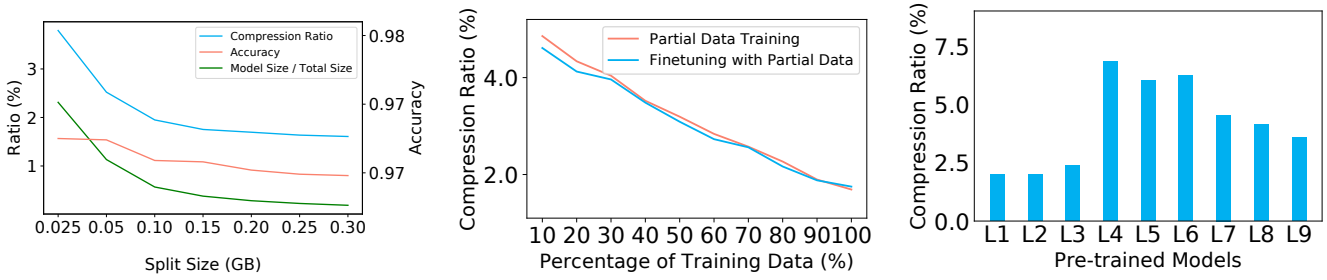
Table 2: Evaluation Results with Different Learning Rate Settings.

Learning Rate	Compression Ratio (%)		Accuracy	
	ELISE	DeepZip	ELISE	DeepZip
0.1	6.11	60.91	0.30	0.20
0.01	0.97	2.85	0.87	0.95
0.001	0.91	1.69	0.88	0.97
0.0001	1.12	1.85	0.86	0.97
0.00001	1.51	2.42	0.82	0.96

Partial data training. Due to the high redundancy in log files, it is possible to use only part of the data for training and perform the compression on the whole data file (Section 3). To evaluate its practical effects, we split a 200 MB file from Lin-7.7G and then divide it into 10 equal parts, and



(a) Results with Different Split Sizes (ELISE). **(b)** Finetuning and Training with Partial Data (ELISE). **(c)** Finetuning with Entire Data (ELISE).



(d) Results with Different Split Sizes(DZ). **(e)** Finetuning and Training with Partial Data (DZ). **(f)** Finetuning with Entire Data (DZ).

Figure 6: Ablation Study Results (DZ is short for DeepZip).

use different percentage of the whole data, i.e., 10%, 20%, $\cdots$,100%, to train the encoder. For each compression, we measure the final compression ratio on the whole file. The red line in Figure 6(b) shows the results for different data split size settings. We observe that when we use more training data, the compression ratio decreases. Meanwhile, when the data size increases, the training takes more time because we need to train the DNN model on more data. How to determine the size of partial data to use is a trade-off in real world scenarios with other constraints. We would like to mention that this experiment is done with file size smaller than 0.6 GB. Namely, the compression has not reached the saturation point. As a result, training with different percentages of the file will lead to the model accuracy change. We believe this experiment is still valuable even though ELISE is not sensitive to training size change when it is larger than a threshold value (i.e., 0.6 GB). This is because training with partial data is an optimization aiming for using less time and resources to achieve acceptable results. In time sensitive scenarios, system administrators can choose this optimization to speed up the system while having non-optimal compression results.

Finetuning with entire data. In a real world scenario, it is highly likely that we only need to store a limited number of logs for a long time. As a result, for a given data file, it is highly likely that we have encountered similar files many times. Therefore, we can use pre-trained model plus finetuning to compress the new file, which is much faster. To validate this idea, we pre-train a model on one of the small files and use it to compress another 9 different files of the same type. For each new file, the model is finetuned only for one epoch, and the compression ratios are measured. The finetuning results are shown in Figure 6(c). The X-axis (i.e., L1~ L9) means different pre-trained models, and the Y-axis is the compression ratio. The results show that the models finetuned on different files have similar compression ratios (i.e., on average 0.92%) and are comparable with results obtained by training on the whole data (see Figure 4). This indicates that using pre-trained models can effectively reduce training costs while maintaining similar compression ratios.

Finetuning with partial data. A natural way to further optimize the runtime of ELISE is to use a pre-trained model and only finetune it with partial data. To test this, we combine these two methods and evaluate its performance. The finetuned model is the same as previous experiments, and file splitting setting is from 10% to 100%. The results are shown as blue line in Figure 6(b). From this figure, we can get similar conclusions with previous experiments: compression ratios drop along with the increase of training data. However, the compression ratios are lower than those in the previous experiments under the same condition (i.e., use same sized partial data to train the model). When we use partial data to train the model, Figure 6(b) shows that training with a pre-trained model always yields better results when the percentage of training data is lower than 80%. This demonstrates the advantages of using a pre-trained model.

4.4.2 Ablation Study for DeepZip

We also perform an ablation study for DeepZip. Due to its inefficiency, all experiments are performed on two portions

of the Lin-7.7G and Lin-16.1G log files.

Split file size. To understand the effects of split file size for DeepZip, we first split the Lin-7.7G file into different sizes (from 0.025 GB to 0.30 GB). Then, we train DNN models on these split files, compress them and measure compression ratios. The result is shown in Figure 6(d), which has a consistent format with Figure 6(a): the blue, red and green lines denote the compression ratio, accuracy of the model and the ratio of model size over the total size (after compression), respectively. Similar to ELISE, with the increase of split size, all three values decrease and then reach a saturation point. What is different is that the absolute values of the split sizes are different. For ELISE, it reaches the saturation point when the size is around 0.6 GB, and for DeepZip, it is smaller than that. This is because ELISE trains the model on numerical values while DeepZip trains the model on discrete values. On the one hand, using raw English letters (i.e., discrete values) makes training harder to converge and takes a longer time. It also does not capture all redundancies like ELISE, leading to high compression ratios. On the other hand, it can identify all repeated substrings with a small size of data, because preprocessings in ELISE have made the distribution of individual letters more complex. For example, the letter "0" in different positions of ELISE preprocessed log has different meanings, which is interpreted by defined rules and reference table. As a result, DeepZip reaches the saturation point with less data than ELISE.

Learning Rates. To measure the effect of learning rates, we train models with different learning rate settings, from 0.1 to 0.00001 on the same file, and measure the compression ratio. Results are shown in Table 2. The results show that, using a large learning rate can significantly increase the compression ratio. When we use a smaller learning rate, the compression ratio becomes smaller and then larger. This observation is consistent with the findings of ELISE, and further proves that a large learning rate and a very small learning rate are not practical.

Partial data training. We also evaluate the effects of DeepZip training with partial data. Specifically, we first split the log file into 10 small ones ranging from 10% to 100% of the original one. Then, we train DNN models on these small files and measure the compression ratio using the original large file. The redline in Figure 6(e) represents this result. Consistent with the results of ELISE, when using more and more data to train the model, the compression ratio gets lower and lower.

Finetuning with entire data. Similar to the experiments for ELISE, we also evaluate the effect of finetuning DeepZip by splitting 10 files (L0 to L9) from the Lin-16.1G log file. The size of each file is 200 MB. A model is pretrained on data file L0, and finetuned on the rest 9 files, L1 to L9. We present the result in Figure 6(f). As we can see, some compression ratios are low while others are high. This is because DeepZip can only identify simple contextual redundancy (but not structural and complex contextual redundancy, such as monotonous values and sessions). As a result, when the contexts in test files (i.e., L1 to L9) and the training file (i.e., L0) are similar, using this pre-trained model can get low compression ratios, and vice versa. This result demonstrates the advantage of using ELISE. By applying preprocessing rules, ELISE can capture all types of contextual and structural redundancies, and achieve good results even when the workloads are different.

Table 3: The Results of Forensic Analysis of 3 Starting nodes.

Experiment	Number of Related Activities		Graph Match
	Original Data	ELISE	
1	320	320	✓
2	199	199	✓
3	69743	69743	✓

Finetuning with partial data. Similar to ELISE, we also try finetuning with partial data in DeepZip. Specifically, we pretrain a model on a 200 MB file split from Lin-7.7G, and finetune it on different percentage of another 200 MB file. The blue line in Figure 6(e) shows the compression ratios in different settings. It shows a similar trend with using partial data training, and also a consistent trend with that of ELISE.

4.5 Supports of Security Investigation

As a lossless compression, ELISE naturally supports all log based security applications. To verify this, we perform log based security incident investigations using the log from DARPA transparent computing (TC) project Engagement 5 and compare the results with existing work [41,42] to see if ELISE can produce the same results. These tasks are forensic analysis aiming to generate a provenance graph from log data and analyze the attack activity by searching ① backward in the graph to find all malicious events that may have led to this activity, and ② forward to find affected files, processes, etc. For each task, we start from a system subject or object as the starting point, and compare the results of using ELISE and raw log. Table 3 summarizes the results for these three experiments including the number of nodes shown in the generated graphs using unmodified log and ELISE (columns 2 and 3, respectively). We also manually check if the graphs match or not (results in column 4 of Table 3). As indicated by the table, ELISE can fully support the log based security analysis.

5 Discussion

As demonstrated by DeepZip and ELISE, DNN based data compression has shown great potential. In some cases, it can reduce 10 times or even more space overhead compared with traditional compression methods like Gzip. On the other hand, DeepZip has a significantly high runtime overhead. ELISE

has proposed novel techniques to alleviate this problem, making it practically usable. Gzip still outperforms DNN based compressions. Since real world logs are quite large, even a 1% lower compression ratio can save GBs of storage space per day in a large organization. Meanwhile, individual log files are only decompressed when needed. We think that ELISE is still valuable in practice. We also envision that with better DNN inference optimization techniques such as inference accelerate hardware (e.g., AI chips) and frameworks, model compression and other potential techniques, the runtime of ELISE can be further optimized.

In the future, there are a few promising research directions based on proposed work. i) *Optimizing* ELISE *runtime.* As mentioned earlier, ELISE still suffers from high runtime overhead compared with Gzip. Optimizing the runtime of ELISE is important. Based on results in Section 4, we know that the inference of DNN is the most time-consuming step in ELISE, which can be optimized and is currently an important topic in the ML community as well. There are already existing methods, such as using inference frameworks or hardware, leveraging pre-trained model, and compressing large model. ii) *Optimizing preprocessing and model training.* Finding other redundancies in log file and designing new preprocessing for it can potentially improve the compression efficiency. Similarly, the training procedure can potentially be optimized by using better model architectures or loss functions. We suspect that a lot of existing AutoML techniques can be altered to fit in this application scenario and provide better results. iii) *Integrating* ELISE *in Existing Systems.* ELISE is orthogonal to many existing techniques, such as redundancy reduction techniques like LogGC [37] when the security application is fixed. Thus, we believe that ELISE can be integrated to existing provenance system and logging system pipelines. Exploring how this can be done is also an interesting direction.

6 Related Work

Besides the directly related works regarding data compression and log reduction discussed in Section 2, ELISE is highly related to log analysis.

Existing threat detection approaches usually learn the pattern of normal behaviors from logs and detect threats if they behave differently. Some work defines normal patterns as single event matching rules [2, 3, 39], and detect potential threats by comparing an activity with such predefined rules learned from historical logs. Forrest et al. [15] uses a fixed size sequence of syscalls to help identify normal behaviors of UNIX processes. Other work [11, 59] improves the intrusion using variable size sequence.

For APTs, many approaches leverage contextual information of events to analyze the provenance graph so that it reduces the false positive alarms in detection. Some of them [22, 44] utilize the information provided by log to create either static or dynamic normal behavior models for threat detection. If an activity does not belong to any normal behavior models, it will be treated as an intrusion. SOTA system in this kind, Unicorn [21] achieves better detection results by learning several normal behavior models from log data.

Moreover, NoDoze [26] exploits historical system execution information integrated in system logs to learn normal behavior patterns. It assigns higher anomaly values to rarely occurring events counted from the historical log. Then, it propagates and updates the anomaly value of each event with its casually related events. Finally, it reduces a large number of false intrusion alerts by filtering out warnings with lower anomaly values. These methods, which utilize historical logs to learn the contextual information, require logs collected for a long time.

Many security analyses leverage provenance graphs, such as forensic analysis and attack attribution [5, 51]. Provenance graphs provide a big picture of the whole attack by backward tracing [32] the events that led to the alert, and forward searching [38] the consequences of the attack. HERCULE [50] reconstructs the attack history using community discovery on correlated log graphs. NoDoze [26] selects the malicious path in the provenance graph. Because provenance analysis is usually performed on a large provenance graph that contains much information, Lee et al. [36] propose to use execution partition to simplify the provenance graph. ProTracer [42] further designs a lightweight tracing system to mitigate this problem and reduce runtime overhead. A lot of existing work [28, 49] focuses on similar topics and removes redundant information of the provenance graph to improve usability.

7 Conclusion

In this paper, we propose and build novel lossless data compression techniques to build a storage efficient logging systems, ELISE. By leveraging a few preprocessing steps, ELISE is able to reduce structural and contextual redundancies that existing techniques cannot reduce, and convert hard to train natural languages in logs to numerical formats. Moreover, it uses a deep neural network based representation learning technique to train an optimal encoder that can represent the same contents with shorter binary strings. Our evaluation shows that ELISE beats existing lossless compression techniques by $1.13 - 12.97$ times with less than 20% overhead compared with methods in its kind.

Availability

ELISE is hosted on GitHub. To facilitate the reproducibility of this paper and deployment of such systems, we also provide ready-to-use containers. The code can be found at https://github.com/dhl123/ELISE-2021

References

[1] Logstash. Website, 2020. https://www.elastic.co/cn/logstash.

[2] How many alerts is too many to handle. https://www.fireeye.com/offers/rpt-idc-the-numbers-game.html., 2021.

[3] Insider threat detection. https://www.netwrix.com/insiderthreatdetection.html., 2021.

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 265–283. USENIX Association, 2016.

[5] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. Trustworthy whole-system provenance for the linux kernel. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 319–334, 2015.

[6] Fabrice Bellard. Lossless data compression with neural networks. https://bellard.org/nncp/nncp.pdf, 2019.

[7] François Chollet et al. Keras. https://keras.io, 2015.

[8] John G. Cleary and Ian H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. Commun.*, 32(4):396–402, 1984.

[9] John G. Cleary and Ian H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. Commun.*, 32(4):396–402, 1984.

[10] David Cox. Syntactically informed text compression with recurrent neural networks. *arXiv preprint*, arXiv:1608.02893, 2016.

[11] Hervé Debar, Marc Dacier, Mehdi Nassehi, and Andreas Wespi. Fixed vs. variable-length patterns for detecting suspicious process behavior. In Jean-Jacques Quisquater, Yves Deswarte, Catherine A. Meadows, and Dieter Gollmann, editors, *Computer Security - ESORICS 98, 5th European Symposium on Research in Computer Security, Louvain-la-Neuve, Belgium, September 16-18, 1998, Proceedings*, volume 1485 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1998.

[12] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. Eidetic systems. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 525–540. USENIX Association, 2014.

[13] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1285–1298. ACM, 2017.

[14] En.Wikipedia.Org. 2020 united states federal government data breach. https://en.wikipedia.org/wiki/2020_United_States_federal_government_data_breach, 2021.

[15] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for unix processes. In *1996 IEEE Symposium on Security and Privacy, May 6-8, 1996, Oakland, CA, USA*, pages 120–128. IEEE Computer Society, 1996.

[16] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In Wei Wang, Hillol Kargupta, Sanjay Ranka, Philip S. Yu, and Xindong Wu, editors, *ICDM 2009, The Ninth IEEE International Conference on Data Mining, Miami, Florida, USA, 6-9 December 2009*, pages 149–158. IEEE Computer Society, 2009.

[17] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The taser intrusion recovery system. In Andrew Herbert and Kenneth P. Birman, editors, *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, pages 163–176. ACM, 2005.

[18] Noah Golmant, Nikita Vemuri, Zhewei Yao, Vladimir Feinberg, Amir Gholami, Kai Rothauge, Michael W. Mahoney, and Joseph Gonzalez. On the computational inefficiency of large batch sizes for stochastic gradient descent. *arXiv preprint*, arXiv:1811.12941, 2018.

[19] Mohit Goyal, Kedar Tatwawadi, Shubham Chandak, and Idoia Ochoa. Deepzip: Lossless data compression using recurrent neural networks. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Data Compression Conference, DCC 2019, Snowbird, UT, USA, March 26-29, 2019*, page 575. IEEE, 2019.

[20] Gzip.Org. The gzip home page. https://www.gzip.org/., 2021.

[21] Xueyuan Han, Thomas F. J.-M. Pasquier, Adam Bates, James Mickens, and Margo I. Seltzer. Unicorn: Runtime provenance-based detector for advanced persistent threats. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[22] Xueyuan Han, Thomas F. J.-M. Pasquier, Tanvi Ranjan, Mark Goldstein, and Margo I. Seltzer. Frappuccino: Fault-detection through runtime analysis of provenance. In Eyal de Lara and Swaminathan Sundararaman, editors, *9th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2017, Santa Clara, CA, USA, July 10-11, 2017*. USENIX Association, 2017.

[23] Xueyuan Han, Thomas F. J.-M. Pasquier, and Margo I. Seltzer. Provenance-based intrusion detection: Opportunities and challenges. In Melanie Herschel, editor, *10th USENIX Workshop on the Theory and Practice of Provenance, TaPP 2018, London, UK, July 11-12, 2018*. USENIX Association, 2018.

[24] Xueyuan Han, Xiao Yu, Thomas F. J.-M. Pasquier, Ding Li, Junghwan Rhee, James Mickens, Margo I. Seltzer, and Haifeng Chen. SIGL: securing software installations through deep graph learning. *CoRR*, abs/2008.11533, 2020.

[25] Wajih Ul Hassan, Adam Bates, and Daniel Marino. Tactical provenance analysis for endpoint detection and response systems. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1172–1189. IEEE, 2020.

[26] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[27] Md Nahid Hossain, Sadegh M. Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R. Sekar, Scott D. Stoller, and V. N. Venkatakrishnan. SLEUTH: real-time attack scenario reconstruction from COTS audit data. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 487–504. USENIX Association, 2017.

[28] Md Nahid Hossain, Sanaz Sheikhi, and R. Sekar. Combating dependence explosion in forensic analysis using alternative tag propagation semantics. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1139–1155. IEEE, 2020.

[29] Md Nahid Hossain, Junao Wang, R. Sekar, and Scott D. Stoller. Dependence-preserving data compaction for scalable forensic analysis. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 1723–1740. USENIX Association, 2018.

[30] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[31] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Intrusion recovery using selective re-execution. In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 89–104. USENIX Association, 2010.

[32] Samuel T. King and Peter M. Chen. Backtracking intrusions. In Michael L. Scott and Larry L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 223–236. ACM, 2003.

[33] Samuel T. King and Peter M. Chen. Backtracking intrusions. *ACM Trans. Comput. Syst.*, 23(1):51–76, 2005.

[34] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines (awarded general track best paper award!). In *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pages 1–15. USENIX, 2005.

[35] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[36] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.

[37] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. Loggc: garbage collecting audit log. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 1005–1016. ACM, 2013.

[38] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[39] Logrhythm. Endpoint threat detection and response monitoring. https://logrhythm.com/solutions/security/endpoint-threat-detection/, 2021.

[40] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela F. Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. Kernel-supported cost-effective audit logging for causality tracking. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 241–254. USENIX Association, 2018.

[41] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. MPI: multiple perspective attack investigation with semantic aware execution partitioning. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 1111–1128. USENIX Association, 2017.

[42] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.

[43] Matthew V. Mahoney. Fast text compression with neural networks. In James N. Etheredge and Bill Z. Manaris, editors, *Proceedings of the Thirteenth International Florida Artificial Intelligence Research Society Conference, May 22-24, 2000, Orlando, Florida, USA*, pages 230–234. AAAI Press, 2000.

[44] Emaad A. Manzoor, Sadegh M. Milajerdi, and Leman Akoglu. Fast memory-efficient anomaly detection in streaming heterogeneous graphs. In Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi, editors, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 1035–1044. ACM, 2016.

[45] Sadegh Momeni Milajerdi, Rigel Gjomemo, Birhanu Eshete, R. Sekar, and V. N. Venkatakrishnan. HOLMES: real-time APT detection through correlation of suspicious information flows. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1137–1152. IEEE, 2019.

[46] Alistair Moffat, Radford M. Neal, and Ian H. Witten. Arithmetic coding revisited. *ACM Trans. Inf. Syst.*, 16(3):256–294, 1998.

[47] Adam J. Oliner and Alex Aiken. Online detection of multi-component interactions in production systems. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011*, pages 49–60. IEEE Compute Society, 2011.

[48] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher W. Fletcher, Andrew Miller, and Dave Tian. Custos: Practical tamper-evident auditing of operating systems using trusted execution. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[49] Thomas F. J.-M. Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David M. Eyers, Jean Bacon, and Margo I. Seltzer. Runtime analysis of whole-system provenance. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1601–1616. ACM, 2018.

[50] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, Shiqing Ma, Fei Wang, Zhiwei Zhang, Luo Si, Xiangyu Zhang, and Dongyan Xu. HERCULE: attack story reconstruction via community discovery on correlated log graph. In Stephen Schwab, William K. Robertson, and Davide Balzarotti, editors, *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, pages 583–595. ACM, 2016.

[51] Devin J. Pohly, Stephen E. McLaughlin, Patrick D. McDaniel, and Kevin R. B. Butler. Hi-fi: collecting high-fidelity whole-system provenance. In Robert H'obbes' Zakon, editor, *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012*, pages 259–268. ACM, 2012.

[52] Andrew Quinn, David Devecsery, Peter M. Chen, and Jason Flinn. Jetstream: Cluster-scale parallelization of information flow queries. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 451–466. USENIX Association, 2016.

[53] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8), 2019.

[54] Mohammed Salem and Helen Armstrong. Identifying dos attacks using data pattern analysis. *Australian Information Security Management Conference*, 2008.

[55] Jürgen Schmidhuber and Stefan Heil. Sequential neural text compression. *IEEE Trans. Neural Networks*, 7(1):142–146, 1996.

[56] W Symantec. Advanced persistent threats: A symantec perspective. *Symantec World Headquarters*, 2011.

[57] Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Mochi: Visual log-analysis based tools for debugging hadoop. In Sambit Sahu and Prashant J. Shenoy, editors, *Workshop on Hot Topics in Cloud Computing, HotCloud'09, San Diego, CA, USA, June 15, 2009*. USENIX Association, 2009.

[58] Brian Tierney, William E. Johnston, Brian Crowley, Gary Hoo, Christopher X. Brooks, and Dan Gunter. The netlogger methodology for high performance distributed systems performance analysis. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing, HPDC '98, Chicago, Illinois, USA, July 28-31, 1998*, pages 260–267. IEEE Computer Society, 1998.

[59] Andreas Wespi, Marc Dacier, and Hervé Debar. Intrusion detection using variable-length audit trail patterns. In Hervé Debar, Ludovic Mé, and Shyhtsun Felix Wu, editors, *Recent Advances in Intrusion Detection, Third International Workshop, RAID 2000, Toulouse, France, October 2-4, 2000, Proceedings*, volume 1907 of *Lecture Notes in Computer Science*, pages 110–129. Springer, 2000.

[60] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 77–90. USENIX Association, 2004.

[61] Frans M. J. Willems, Yuri M. Shtarkov, and Tjalling J. Tjalkens. The context-tree weighting method: basic properties. *IEEE Trans. Inf. Theory*, 41(3):653–664, 1995.

[62] Wei Xu, Ling Huang, Armando Fox, David A. Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In Johannes Fürnkranz and Thorsten Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 37–46. Omnipress, 2010.

[63] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. High fidelity data reduction for big data security dependency analyses. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 504–516. ACM, 2016.

[64] Runqing Yang, Shiqing Ma, Haitao Xu, Xiangyu Zhang, and Yan Chen. Uiscope: Accurate, instrumentation-free, and visible attack investigation for GUI applications. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[65] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, pages 143–154, 2010.

[66] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael Mihn-Jong Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 293–306. USENIX Association, 2012.

[67] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. *ACM Trans. Comput. Syst.*, 30(1):4:1–4:28, 2012.

[68] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977.