Computers & Security 151 (2025) 104312

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Full length article

Using approximate matching and machine learning to uncover malicious activity in logs

Rory Flynn, Oluwafemi Olukoya *

School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, United Kingdom

ARTICLE INFO

Keywords:
Approximate matching
Semi-structured data
Machine learning
Log analysis
Digital forensics
Network security
Fuzzy hashes

ABSTRACT

The rapid expansion of digital services has led to an unprecedented surge in digital data production. Logs play a critical role in this vast volume of data as digital records capture notable events within systems or processes. Large-scale systems generate an overwhelming number of logs, making manual examination by analysts infeasible during critical events or attacks. While hashes, whether cryptographic or fuzzy, are widely used in digital forensics because they serve as the foundation for software integrity and validation, authentication and identification, similarity analysis, and fragment detection, this study investigates and extends the use of approximate matching (AM) algorithms in semi-structured data, such as logs. Existing AM algorithms such as *ssdeep*, *sdhash*, *TLSH*, and *LZJD* struggle particularly with semi-structured data due to the size of the input data being comparatively small, with syntactical and structural information comprising a significant amount of the data. We present a novel approximate matching algorithm for application across a range of semi-structured data types, which requires no knowledge of the underlying data structure. The algorithm produces digests that serve as input to a machine learning classifier, classifying the behaviour of the underlying logs the hashes represent. Experimental results on a benchmark dataset of IoT network traffic show that the proposed framework can correctly discern malicious logs from benign records with a 95% accuracy, with an F1 score of 0.98. The behaviour of the records deemed malicious was then correctly identified with a 99% accuracy when evaluated using a test data set, producing an average F1 score of 0.99. Additionally, we demonstrate that this approach provides a faster and lightweight framework to perform classification with high accuracy on a list of logs, producing those indicative of an attack for review.

1. Introduction

The rapid expansion of digital services has led to a significant increase in digital data production (Rydning et al., 2018). Logs are a byproduct of this growth and serve as a digital record of notable events within a system, network, service or process. These are often the only records of runtime data within a system and have become vital reliability assurance mechanism (Cinque et al., 2013; Wang et al., 2019; He et al., 2021) and source of valuable information to forensic investigations (Studiawan et al., 2019; Khan et al., 2023; Arasteh et al., 2007; Ibrahim et al., 2011). This reliance has led to an exponential increase in the volume of software and hardware logs, with additional factors including increasing sophistication of cyber-attacks and government regulations and compliances fuelling the growth. For a well-designed software system, the developer will use the minimal amount required to keep a complete picture of an event in a system, as these logs have an accompanying storage cost. However, even with these optimisations a large-scale system produces an infeasible number of logs for an analyst to manually examine in the event of an attack (Lambert, 2024; Lillis et al., 2016).

To address these challenges, a significant amount of work has been done by researchers focusing on best practices and analysis methodologies (Günther and van der Aalst, 2006; Xu et al., 2009; Shang, 2012; He et al., 2022b; Zhang et al., 2019; El-Masri et al., 2020; Zhu et al., 2019). Techniques such as anomaly detection (Vaarandi, 2003; Xu et al., 2009; Fu et al., 2009; Oliner et al., 2012; He et al., 2016b) have been widely used for decades, focusing on using pattern-matching techniques to cluster data and detect anomalies in event logs. However, these earlier techniques relied heavily on an expert in each domain, platform and system to create a suitable database of patterns. In recent years, state-of-the-art (SOTA) implementations have aimed to tackle this problem by using advancements in artificial intelligence (AI) to predict and classify logs based on a categorical sentiment (Meng et al., 2021; Sipos et al., 2014). The exponential growth of big data has further accelerated the utility and accuracy of these models, allowing them to identify

* Corresponding author.
E-mail addresses: rflynn12@qub.ac.uk (R. Flynn), o.olukoya@qub.ac.uk (O. Olukoya).

https://doi.org/10.1016/j.cose.2025.104312
Received 25 June 2024; Received in revised form 3 December 2024; Accepted 2 January 2025
Available online 9 January 2025

regular use for a system to identify anomalies through supervised and unsupervised learning (Berlin et al., 2015; Du et al., 2017). Even with these advancements, detection systems are not perfect due to a range of factors, such as data quality, which is highly dependent on an organisation's logging consistency and methodology, which is often subpar (Shang, 2012; Pecchia et al., 2015; Oliner et al., 2012; He et al., 2018; Fu et al., 2014). These inconsistencies challenge the representation of the data uniformly, with minimal loss of context.

Logs are commonly used in the industry for network and system observability (Sridharan, 2018; Gatev and Gatev, 2021). Various stakeholders, such as software developers, site reliability engineers, security researchers, threat hunters, cyber analysts, and incident responders require a method to monitor system behaviours, conduct post-mortem analysis, cluster, hunt, and pivot across a range of log files in an efficient and scalable manner. We propose using hashing to represent the logs. The underlying hypothesis of this research is that the hash calculated from the log samples will serve as a data point for analysing, parsing, clustering, and pivoting between different log samples. Hashing enables the stakeholders to condense an unwieldy data point into a concise figure that is easily searchable and pivotable (Van Dijk, 2023; Wilson, 2023; Versteeg and Moodley, 2023). Some log samples can be huge, spanning multiple thousands of characters. Reducing this complex log sample into a single unique 64-character value or any other form of compact encoding technique makes handling this data more feasible and consistent without losing context (Ianni and Masciari, 2023).

Hashing algorithms have existed for decades, focusing on cryptographic hash functions (e.g. SHA, MD5), as these functions are extensively used in digital forensics for tasks such as data fingerprinting (Roussev, 2010). Unlike cryptographic hashing, approximate matching (AM), or fuzzy hashing algorithms can produce similar or identical hashes for data with minor differences. The utility of this type of algorithm is to gauge similarity among two sets of data, by calculating a similarity score based on the sample's hash. At higher levels, approximate matching can incorporate more abstract analysis allowing a system to interpret the content represented by a byte sequence (Breitinger et al., 2014). A categorisation based on the behavioural characteristics of this stage was proposed in Martín-Pérez et al. (2021), and consists of three categories:

- **Feature Sequence Hashing** (FSH): This category groups algorithms that split an input and extract a set of features, measuring their similarity by feature sequences.
- **Locality-Sensitive Hashing** (LSH): This category consists of algorithms that map objects into buckets, grouping similar objects with high probability.
- **Byte Sequence Hashing** (BSH): This class includes algorithms that determine if byte sequences, sometimes known as blocks, are present in the input. The number of common blocks across similarity digests is compared to determine the similarity score.

This categorisation makes it possible to group algorithms using similar methodologies and analyse if there are trends between these behavioural characteristics and their suitability for use on logs. Various fuzzy hashing algorithms exist, with most focusing on identifying similarities across different executables (He et al., 2022a; Mercês and Costoya, 2020). As a result, new fuzzy hashing algorithms have been developed for other use cases and domains, including ELF files (Mercês and Costoya, 2020), declared permissions in Chromium-based browser extensions and APKs (Wilson, 2023), and more. This research aims to create a new fuzzy hashing algorithm for semi-structured data, such as log entries. This will enable large-scale comparison of log entries during forensic investigations, aiding incident responders and cyber security analysts in identifying malicious activity (Versteeg and Moodley, 2023), as well as facilitating log analysis and anomaly detection (Zhu et al., 2023; Landauer et al., 2023).

This approach uses a newly developed locality-sensitive hashing algorithm, `SSDHash`, to accurately represent log data. This algorithm operates at a high level of abstraction, meaning it can interpret the underlying log data and will dynamically remove certain field types such as UUIDs and timestamps to represent the logs' content better. This hash representation is processed and is passed through a multistage machine learning model. The tool will initially perform binary classification to determine if it is benign or malicious. If the given entry is deemed malicious the tool will perform multiclass classification to classify the behaviour of the entry e.g. denial of service (DOS), command and control (C&C). The primary research question of this work is; "*Given a log record, to what extent can we accurately automate log analysis, by categorising entries on their behaviour using a novel approximate matching algorithm designed primarily for semi-structured data in combination with machine learning?*".

This paper makes the following key contributions:

- A novel fuzzy hashing algorithm, `SSDHash`, is designed to accept semi-structured data inputs.
- A novel framework for using the fuzzy hashes of log records to perform behaviour classification.
- A technique for embedding fuzzy hashes using positional word encoding and image transformation such that certain sequences and patterns identify similarities among malicious log records.
- An investigation of the impact of various data transformations on improving the accuracy of a fuzzy hash-based classifier.
- A comparative analysis of the performance of hashed logs versus the state-of-the-art vector representation of raw log events using *Word2Vec* algorithms.
- The code for `SSDHash`, the log analysis tool, datasets and the relevant experiments are publicly available[1] to enhance the reproducibility of research.

The rest of the paper is structured as follows: Section 2 provides the background into AM algorithms and their stages. Section 3 reviews relevant literature on current AM algorithms and existing log analysis solutions, which is the domain of evaluation of the proposed AM scheme. An exploratory study to assess the capabilities of the current AM algorithms in handling semi-structured data is also presented. Section 4 provides an overview of the system architecture of the multistage log analysis overview, while Section 5 provides the design details of the novel `SSDHash` scheme. Section 6 describes the preprocessing steps and classification methods for combining the newly proposed AM scheme with ML. The multistage log analysis solution including fuzzy hashes embedding, datasets, data-type processing, machine learning classification and performance metrics is also presented in Section 6. Evaluation results are presented in Section 7. Section 8 discusses the limitations of the proposed approach and future work, while Section 9 concludes the paper.

2. Background

Digital forensics and malware detection have long relied on cryptographic hashing algorithms such as MD5 and SHA-256 to provide a unique identifier for a file (National Institute of Standards and Technology (NIST) and Dang, 2015; Hahn, 2021; Mercês and Costoya, 2020). These algorithms take an input file and output a fixed-length corresponding value. One of the key attributes of these algorithms is that for a given input, any minute difference will produce a drastically different output. Secondly, while it is theoretically possible for collisions to occur for a given hash, it is computationally improbable that a system can find another input that produces the same hash (Uhlig et al., 2023). This ability to generate a digital fingerprint for a file allowed analysts to build a set of hashes for known files that could

[1] https://github.com/roryflynn9401/SSDHash

Table 1
Summary of related works on combining approximate matching and machine learning.

Type	Attribute	Our work	Uhlig et al. (2023)	Kida and Olukoya (2023)	Rodriguez-Bazan et al. (2023b)	Peiser et al. (2020)
File	PDF		✓			
	XLSX		✓			
	JSON	✓				
	XML	✓				
	CSV	✓				
	Android APK				✓	
	Binary			✓		
	JavaScript		✓			✓
ML Input	Fuzzy Hashes	✓	✓	✓	✓	✓
Encoding	Word	✓		✓		
	Integer		✓			✓
	Position Inclusive	✓	✓			
	Image Representation	✓			✓	
Classifiers	Binary	✓	✓			✓
	Multiclass	✓		✓	✓	

be used to compare new files. This allowed analysts to check whether any new hashes produced matched the list of known files and could be disregarded (Uhlig et al., 2023). This concept of fingerprinting a file birthed popular malware sample exchange platforms, allowing users to query a list of known samples or scan suspicious files, URLs, IP addresses, domain or file hashes to detect malware and other breaches. These platforms, including services such as VirusTotal,[2] VirusShare[3] and MalwareBazaar,[4] utilise many tools, including cryptographic and fuzzy hashing algorithms, providing users with a signature for known malware variants.

This methodology of signature-based malware detection is effective in combating known variants of malware but relies heavily on a community providing such samples. This reliance makes it vulnerable to new malware families and has pushed researchers to derive new methodologies for detection. Traditional approaches determine similarity by measuring the "distance" between the fuzzy hashes to calculate a similarity score. The similarity score is a representation of common fragments of the underlying data. However, the effectiveness of similarity scores is constrained, as demonstrated by Göbel et al. (2022), often falling short in identifying common elements or generating inaccurate similarities. Related papers such as Uhlig et al. (2023) propose using machine learning algorithms on the fuzzy hashes to perform fragment detection on files to identify known malicious fragments to perform classification. Additionally, a case study released by Microsoft Threat Intelligence (Lazo, 2021), highlighted the effective use of combining fuzzy hashing and deep learning methods to identify a variant of GoldMax malware. The observably similar `ssdeep` (Kornblum, 2006) and TLSH (Oliver et al., 2013) hashes to the first GoldMax variant were crucial in early detection. This proactive approach enabled Microsoft to swiftly respond and protect its customers by blocking the identified malicious PE (portable executable) file.

Fuzzy hashing algorithms follow three main stages: artefact processing, digest generation and digest comparison (Martín-Pérez et al., 2021). Current AM algorithms will implement all three steps but differ primarily in the artefact processing stage due to varying goals and approaches. These different approaches operate on varying levels of abstraction, with the lowest level approximating the similarity between byte sequences with no regard to the structure or meaning of the data.

2.1. Artefact processing

The artefact processing stage is where most commonly used algorithms differ. While algorithms like `sdhash` operate at a lower level, directly hashing files, higher-level algorithms such as `DexoFuzzy` (Lee et al., 2019a,b) and `PermHash` (Wilson, 2023) differ by instead analysing opcodes and permissions, respectively. This diversity in methodologies yields distinct advantages in various applications, particularly in fragment detection and malware analysis. Fragment detection is a fundamental part of digital forensics and relies on algorithms capable of identifying and reconstructing fragmented files from disparate pieces of data. Here, algorithms like `sdhash` excel because they can efficiently hash files and detect similarities even in fragmented datasets. By comparing hash values, the algorithm facilitates the reconstruction of fragmented data, aiding forensic investigators in piecing together digital evidence.

Historically, malware analysts used low-level algorithms such as `ssdeep` and `sdhash` to identify common code fragments and detect known malware variants. However, several of these algorithms, including `ssdeep`, `sdhash` and `TLSH` have known methods to make small, but smart modifications to a file's contents or semantics to produce drastically different hashes, potentially obscuring them from detection (Chang et al., 2019; Breitinger et al., 2012; Fuchs et al., 2023). Recent approaches such as `DexoFuzzy` and `PermHash` attempt to identify commonalities in an application's behaviour. Both algorithms' primary utility is malware identification for Android apps (APKs) in which they hash an app's respective opcodes and permissions. The logic dictates that while it is possible for a developer to obfuscate code and logic, when the context is restricted to instructions or permissions the underlying task the malicious actor aims to achieve requires similar steps, such as accessing the device's files etc. This approach of targeted data extraction has proved to be a major inspiration in the development of `SSDHash`, with resilience demonstrated by these high-level algorithms in practice being a major factor.

2.2. Digest generation

The next step in approximate matching algorithms is digest generation following artefact processing. The process of digest generation varies across different algorithms, with some such as `sdhash` using the cryptographic hashing function SHA-1 paired with a bloom filter (Roussev, 2010), while others such as `ssdeep`, use the non-cryptographic Fowler–Noll–Vo (FNV) hash function (Kornblum, 2006).

In this phase, the algorithms transform the processed artefacts into digests, compact representations suitable for comparison and analysis. For instance, algorithms like `ssdeep` and `sdhash` generate digests by hashing certain features extracted from the artefacts. These features may include byte sequences, n-grams, or other data structural elements. By hashing these features, the algorithms produce fixed or variable-length digests, that capture essential information about the artefacts while minimising storage and computational requirements.

[2] https://www.virustotal.com
[3] https://virusshare.com/
[4] https://bazaar.abuse.ch/

Table 2
Relevant State-of-the-art Approximate Matching tools for unique file types beyond executables.

Tool	Inputs	Artefacts	Digest generation algorithm	Digest comparison
JsonHash	JSON	Key–Value Pairs	Pearson Hash	Similarity Score
Permhash	APK,CRX	Permissions	SHA256	–
DexoFuzzy	Dex	OpCodes	ssdeep	Jaccard Index, Clustering
SSDHash	JSON, XML, CSV	Key–Value Pairs	Spooky64	Similarity Score

2.3. Digest comparison

Once the digests have been generated for the artefacts, the next step for approximate matching algorithms is digest comparison. In this phase, the algorithms compare the generated digests to identify similarities and potential matches among the artefacts. The comparison process involves assessing the similarity between pairs of digests, with some common methods being the Jaccard Index and Hamming distance (Bookstein et al., 2002). These metrics quantitatively measure the difference between two digests, allowing algorithms to establish similarity thresholds for determining matches. However, these metrics may not always capture semantic similarities effectively, so in more recent work, machine learning has been extensively used to address similar limitations in these basic comparison approaches (Peiser et al., 2020; Rodriguez-Bazan et al., 2023b; Kida and Olukoya, 2023; Uhlig et al., 2023; Lazo, 2021; Rodriguez-Bazan et al., 2023a). By training models on labelled datasets, these algorithms can learn to recognise patterns indicative of similarity. These approaches cover a range of domains which utilise both conventional machine learning and deep learning techniques. For example, Peiser et al. (2020) used fuzzy hash inputs to train a neural network to identify JavaScript malware. Similarly, Kida and Olukoya (2023) used fuzzy hash inputs to train a range of machine learning classifiers to identify similarities in malware that can be used to attribute attacks. Regardless of domain, this data-driven approach improves the algorithm's ability to handle diverse datasets and evolving threat landscapes, making it more effective for real-world applications. Table 1 shows a comparative analysis of related works on machine learning-assisted approximate matching and the proposed study.

3. Related work

This section covers two main phases of relevant literature. The first phase examines current state-of-the-art approximate machine algorithms and their taxonomy, while the second phase focuses on existing log analysis solutions. We also conduct an initial study to assess the current capabilities of existing AM algorithms on semi-structured data.

3.1. Approximate matching algorithms and their taxonomy

Early applications of fuzzy hashing algorithms, which include the spam email detector *spamsum*, laid the foundation for adoption in a broader range of domains, including natural language processing (Kalyanathaya et al., 2019) and notably cyber security. In the following, several of the current industry standard algorithms are reviewed, with the key steps that were introduced earlier in the paper explained and reviewed, with a summary of some of the relevant state-of-the-art algorithms shown in Table 2.

- **ssdeep** (Kornblum, 2006): ssdeep is the most widely documented and tested algorithm out of this list. It operates similarly to TLSH, by converting an input binary into a digest. The primary differences lie in the formatting of the output digest and the hash computation. ssdeep employs a rolling hash algorithm, meaning that changes to the file result in only localised changes in the signature. ssdeep splits the input files into block segments and computes a cryptographic hash for each block to produce the final ssdeep hash digest. Block size and the input file are used to calculate trigger points, which are then used to build the blocks. The similarity measure to compare two files is the edit distance between the two digests. ssdeep belongs to the feature sequence fuzzy hashing algorithms class, which determines file similarity by comparing feature sequences.
- **sdhash** (Roussev, 2010): sdhash is a tool that facilitates the comparison of arbitrary blobs based on common strings of binary data. One of its common use cases is in fragment detection where there is a need to find smaller pieces of data from larger chunks. An example of this is Block *vs.* File correlation in which a chunk of data (e.g. a disc block or network packet) can be compared against a reference collection of files to identify whether a chunk originated from any of them. The digests are generally produced by searching for neighbouring 64-byte sequences that are statistically improbable to occur by chance. After being hashed, each feature is run through a bloom filter. The major downside of this approach is that the hash length is not fixed and grows with the input size. This means the produced hashes can be more difficult to compare and analyse with larger files.
- **TLSH** (Trend Micro Locality Sensitive Hash) (Oliver et al., 2013): This algorithm generates a 72-character output digest by applying a sliding window over the input byte string. A key advantage of this approach is that the output size is consistent, eliminating the need for padding when used with machine learning algorithms. To calculate the similarity score between two digests, the distances of the header and body are added together, using quartile ratios and hamming distance, respectively. TLSH belongs to the family of fuzzy hashing algorithms that use n-grams to define file similarity.
- **LZJD** (Raff and Nicholas, 2018): LZJD was created as an effective alternative to existing algorithms such as ssdeep and sdhash for measuring similarities in byte sequences. It converts the input into a set of sub-strings using the Lempel–Ziv algorithm. These substrings are then compared using the Jaccard index to calculate the similarity based on common chunks. Like ssdeep and sdhash, LZJD's primary usage is in digital forensics.
- **JsonHash** (Versteeg and Moodley, 2023): JsonHash was created to compare semi-structured data and find its use in Microsoft's Incident Center, which groups similar logs in threat-hunting scenarios. The given implementation is designed for JSON data. Still, the algorithm will work for any semi-structured data by flattening the data structure and mapping similar objects into buckets. The core operations for digest generation are the same as those for SSDHash, with the main differences being preprocessing, rank calculation, and digest comparison implementations.
- **Permhash** (Wilson, 2023): Permhash is a framework that aims to identify malware in Android APKs and Chromium extensions. It works by extracting the permissions associated with the APK/extension and concatenating them into a feature string. The algorithm then hashes the permission string using the cryptographic hash function SHA256. The framework allows users to draw correlations between samples by clustering based on the output hashes.

3.2. Existing log analysis solutions

Typical methods for detecting malicious activity have relied on signature-based detection tools and rule-based systems. These approaches have demonstrated effectiveness in identifying known

threats, providing a reliable means of recognising patterns associated with established malicious activities. In the context of network logs, signature-based detection, employed by tools like Snort, involves matching log entries against predefined signatures or patterns indicating known IOCs (Kumar and Sangwan, 2012). Similarly, rule-based systems use predefined rulesets to analyse log entries and trigger alerts based on specified conditions. These rules encompass various criteria, allowing for a flexible yet structured approach to network log analysis. When deployed, these methods are often seen as a security information and event management (SIEM) system that can incorporate additional methods such as anomaly detection and behaviour analysis to address each method's shortcomings (Bhatt et al., 2014).

The application of these traditional approaches extends beyond network logs to encompass a broader spectrum of logs, including system event logs. System logs, such as those generated by operating systems, databases, or applications, face unique challenges when traditional methods are applied. The static nature of signature-based detection limits its effectiveness against previously unseen threats in system logs, where attack patterns can be more diverse and subtle (Berlin et al., 2015). Additionally, system event logs often capture a wide range of information related to user activities, application behaviour, and system configurations. While effective in specific scenarios, rule-based systems encounter difficulties adapting to the dynamic nature of system logs and the evolving tactics of adversaries.

Recent advances in these existing approaches leverage machine learning and deep learning advancements for use in log analysis (Landauer et al., 2023; He et al., 2016b). These alternatives offer promising advancements compared to the conventional rule and signature-based systems by leveraging ML techniques such as classification and anomaly detection. These techniques aim to achieve the similar goal of finding patterns in log entries indicative of an attack, with the methods these approaches employ differing depending on the system's goal. Log classification aims to identify two or more distinct categories (Berlin et al., 2015; Meng et al., 2021; Moh et al., 2016), whereas anomaly detection aims to identify outliers from normal behaviour (Chen et al., 2021; Du et al., 2017; Li et al., 2020). Several recent works have leveraged machine learning-assisted anomaly detection in domains like web and system logs, such as Han et al. (2023), Zhang et al. (2019), demonstrating methods for modelling semi-structured log samples as natural language input for a deep learning model to perform anomaly detection on the system logs of a distributed system. Other works such as Le and Zhang (2022), Studiawan et al. (2021) provide other examples of anomaly detection in system logs, with Cao et al. (2017) demonstrating the application of a similar approach to web logs. In these approaches, the textual attributes in the log event are converted into word embeddings learned with Word2Vec (Mikolov et al., 2013a,b) algorithms e.g. FastText (Joulin et al., 2016), GloVe (Pennington et al., 2014) combined with TF-IDF (Sparck Jones, 1972; Luhn, 1957). Then, the word embeddings are combined to create the semantic vector of the log event.

3.3. Exploratory study

We provide a classification of the most popular fuzzy hashing algorithms in Table 3, which covers the input, minimum input size, digest generation algorithm, comparison algorithm, maximum similarity, and the classification category as defined by Martín-Pérez et al. (2021). Fuzzy hashing algorithms such as ssdeep, spamsum, LZJD and TLSH can work on any file and are not limited to portable executables (PE) files. In contrast, algorithms such as sdhash, ImpHash, impfuzzy and MRSH are designed for binary (PE) files (Shiel and O'Shaughnessy, 2019; Namanya et al., 2020; Lee et al., 2019b; Fleming and Olukoya, 2024; He et al., 2022a; Mercês and Costoya, 2020). FbHash is used for uncompressed such as `.txt` files and compressed file format documents such as `docx`, `ppt`, `pdf` etc (Chang et al., 2019). This classification of fuzzy hashing algorithms is consistent with those presented by Mercês and Costoya (2020), Martín-Pérez et al. (2021) and He et al. (2022a).

Log files can be found in structured, semi-structured, or unstructured formats (Sharif, 2022). Although log formats vary significantly across different systems, applications, and tools, some of the most common formats used in security-oriented applications include plain text, CSV (Comma Separated Values), XML, Syslog, and JSON (Anand, 2022; xcitium, 2024). For instance, several application performance monitoring (APM) and observability tools, such as *SigNoz*,[5] (Anand, 2022) *Papertrail*,[6] (xcitium, 2024) *Loggly* (SolarWinds Loggly, 2024), *Coralogix* (Coralogix, 2022), and *Sumo Logic* (Kim(Sumo Logic), 2024), utilise these formats for log management. In addition to observability platforms, various applications generate logs in these formats. Examples include *IBM's Financial Transaction Manager for Multiplatforms* (IBM Corporation, 2024a), the *Panther* SIEM platform (Panther Labs, 2024), *WinSCP* (Prikryl, 2024), *Microsoft Windows Media* (Microsoft, 2021), and *IBM Security Verify Access* (IBM Corporation, 2024b). These formats have become widely recognised as the most convenient standardised options for log files within logging systems. For example, while Docker supports 11 built-in logging drivers, each with distinct log formats, it defaults to the `json-file` logging driver, which caches container logs in JSON format internally (Docker Inc, 2024; Ge et al., 2021).

While most fuzzy hashing algorithms have been designed to compare files such as executables and compressed formats, none are specifically tailored for semi-structured data inputs. This limitation has resulted in a lack of automated analysis for these file types since they do not conform to standard executable formats. As indicated in Table 3, algorithms like TLSH, ssdeep, LZJD, and sdhash can handle any type of file, including plain text (`.txt`) files (Mercês and Costoya, 2020; He et al., 2022a). Plain text files are a significant portion of global logs and logs generated by embedded equipment like routers that utilise the Syslog format. FbHash was designed for both compressed (e.g., `docx` etc.) and uncompressed (e.g., `txt` files) file formats, which also effectively caters to log files formatted in plain text (`.txt`). However, existing fuzzy hashing algorithms perform poorly when applied to semi-structured data, such as JSON blobs, XML files, or lines from log files (Versteeg and Moodley, 2023; Zhai et al., 2017; Breitinger and Baggili, 2014). Consequently, there is a need for an effective approximate matching algorithm capable of handling semi-structured data inputs in formats like JSON, XML, or CSV to enhance the use of fuzzy hashing on log files beyond plain text.

A recent systematic literature review highlighted the challenges faced by practitioners in software logging practices (Batoun et al., 2024). This review covers all aspects of software logging, from the creation of logs to their analysis, and it highlights the gap between researchers' focus and the needs of practitioners. The empirical study revealed significant advancements in automated log analysis techniques, including log parsing, clustering, and mining. However, in practice, developers primarily concentrate on converting large volumes of log data into JSON format and parsing complex log messages already in JSON format. One of the key recommendations by the authors to address the practical challenges of logging and to bridge the gap between research and practices is as follows: *"While a large body of research exists on parsing unstructured log data, we recommend that future studies expand these existing approaches to handle complex log data, such as data presented in JSON format"*. (Batoun et al., 2024).

We conduct an exploratory study to assess the capabilities of existing approximate matching algorithms, with the primary objective of determining how well these algorithms can adapt to semi-structured data. To evaluate an algorithm's suitability effectively, a range of tests were run on each algorithm in Table 3. These tests entailed running

[5] https://github.com/SigNoz/signoz

[6] https://www.papertrail.com/

Table 3
Commonly used approximate matching algorithms.

Algorithm	Input	Minimum input size	Digest generation algorithm	Comparison algorithm	Output	Maximum similarity	Category
ssdeep	File	64 bytes	FNV hash	Edit distance	0–100	0	FSH
sdhash	Binary	512 bytes	SHA-1 and bloom filter	Bloom filter analysis	0–100	0	FSH
TLSH	File	50 bytes	Pearson hash	Distance score	N-0	N	LSH
FbHash	.txt, docx	7 bytes	Rabin Karp rolling hash	Cosine similarity	0–100	0	LSH
LZJD	File	200 bytes	Lempel–Ziv Jaccard Distance	Jaccard similarity	0–1	0	BSH

Table 4
Test results for AM tools on semi-structured data using S5.

Algorithm	Sample 1	Sample 2	Sample 3	Sample 4	Sample 5
ssdeep	43	96	64	82	83
sdhash	–	–	–	–	–
TLSH	–	74	–	66	69
FbHash	100	100	100	100	100
LZJD	0.9	0.92	0.78	0.71	0.83
JsonHash	0.55	0.23	0.50	0.31	0.53

the algorithms in their default configuration using a range of semi-structured log pairs in various formats and sizes (e.g. JSON, XML). The S5 dataset contains five pairs of objects that only differ by a few characters in the values, with all keys remaining the same. When treated as a textual string, these inputs would produce a theoretically low hamming distance (Bookstein et al., 2002) relative to their size. Hamming distance measures the number of positions two strings differ, with the relative characteristics of this dataset shown in Table 8. In practice, the data these structures represent can vary and should provide a range of similarities if only their content is considered. To provide context for the dataset, a record from each of the five samples is shown in Section 7.1, Listing 1 of the evaluation.

The results in Table 4 show the similarity scores of each widely used approximate matching algorithm when the five data samples in JSON and XML format are hashed. Tests using algorithms such as `ssdeep`, `TLSH`, `JsonHash` and `LZJD` produced a range of results for sample data that has slight differences, with the main outliers of this test being `sdhash` and `FbHash` which failed to produce any meaningful results. In the case of `sdhash`, all samples were below the 512-byte minimum input size and thus did not produce any results, whereas `FbHash` did run but failed to find any similarities in the data. In earlier work by Zhai et al. (2017), semi-structured data, like HTML, cannot be accurately detected using `sdhash` for real-time network file similarity detection. Similarly, when `sdhash` was used as a file identification mechanism on network traffic, some HTML files were falsely identified as text files due to their comparable layouts and tables (Breitinger and Baggili, 2014). A similar conclusion was made in Göbel et al. (2021) while evaluating network traffic analysis using approximate matching algorithms, demonstrating that the algorithm's accuracy varied depending on the file type. Other algorithms such as `ssdeep`, `TLSH` and `LZJD` produced reasonable similarity scores across the samples, but when posed with more complex inputs the results tended to diverge towards a low similarity score. However, efforts have been made to enhance the accuracy and runtime performance of `ssdeep`, such as `ssdeeper` (Jakobs et al., 2022), the fundamental drawback of not being intended for semi-structured data and the inconsistent outcomes depending on file sizes and types persists.

When applying existing AM algorithms to semi-structured data, the critical issue is that the input comprises structural and syntactical information. This is especially true of byte-wise algorithms, such that a significant amount of input sanitation is required when working with semi-structured data to produce consistent results. This extends beyond the syntactical information mentioned and encompasses things such as white space, which is extremely common in semi-structured data to improve readability. All white-space characters were removed from the inputs before hashing to provide a consistent representation for these tests. This significant variance in the results highlights the inadequacy of the current approaches and challenges faced in applying existing AM algorithms to semi-structured data. The proposed approach extends this research by combining a fuzzy hashing algorithm for semi-structured data with machine learning to perform multistage classification on log data.

4. Multistage log analysis overview

This section introduces the multistage log analysis tool to achieve log classification based on fuzzy hash inputs. The machine learning model can discern the patterns constituting a regular record and those indicative of malicious intent, subdivided by attack type.

4.1. System architecture

Fig. 1 presents an overview of the architecture that combines fuzzy hashing and machine learning classifiers for log analysis. The system takes log samples as input, which can be in JSON, XML, or CSV format. Logs are primarily plain text in an unstructured format (Zhu et al., 2023; Landauer et al., 2024). However, they are captured, processed, and parsed as textual data in XML, JSON, CSV and other structured formats for log analysis. This conversion maintains high informational value with minimal alterations to the raw data readable for humans and automated systems (He et al., 2016a; Jiang et al., 2024; Zhu et al., 2019; Batoun et al., 2024). The log samples used in this research are sourced from IoT-23 (Garcia et al., 2020), a labelled malicious and benign IoT network traffic dataset parsed in the CSV format. The dataset contains 23 captures of various IoT traffic, including 20 malware captures performed on a Raspberry Pi and three benign captures from real IoT devices, specifically an Amazon Echo, a Philips Hue, and a Somfy door lock. Additionally, the dataset is multi-labelled, with each label containing different types of attacks such as *C&C*, *DDoS*, *Okiru*, *Mirai*, *PartOfAHorizontalPortScan*, etc. These labels describe the relationship between flows linked to malicious or potentially malicious network activities. A *DDoS* label indicates that the infected device is executing a Distributed Denial of Service attack, and a *C&C* label indicates that the infected device was connected to a CC server. A *PartOfAHorizonalPortScan* indicates that the connections perform a horizontal port scan to gather information for future attacks. The *Okiru* label indicates that the connections have the characteristics of an Okiru botnet. In contrast, a *Mirai* label indicates that the connections have the characteristics of a Mirai botnet. A detailed explanation of the labels, distribution, frequency of flows, and feature descriptions are publicly available.[7]

Multiple preprocessing steps are applied to the data before generating the final digest, and preparing it for classification. The data is transformed into 3-grams and encoded using positional word encoding before being fed into the machine learning model. The log sample hashes are also converted into greyscale images as part of the fuzzy hashes encoding. The processed data is then input to the *Support Vector Machine (SVM)*, *Stochastic Dual Coordinate Ascent (SDCA)*, and *K-Means* machine learning models, which perform clustering, binary, and

[7] https://www.stratosphereips.org/datasets-iot23

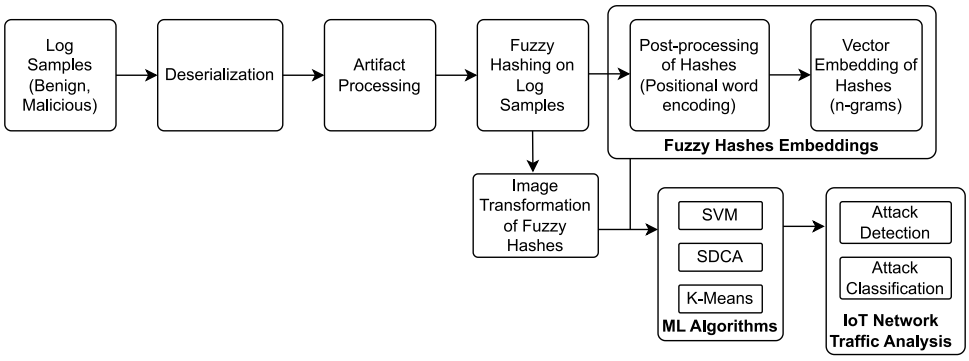

Fig. 1. Architecture overview.

multiclass classification on the sample. The classifier predicts whether the specified sample is malicious or benign for attack detection. For attack classification, the malicious sample is classified into the attack type, such as a DDoS attack. In summary, the objective of the system overview depicted in Fig. 1 is to introduce an innovative approach towards advancing the domain of log analysis. This approach leverages fuzzy hashes as inputs to a machine learning classifier, aiming to speed up the process of attack categorisation. In the context of the logs, the data produced is quite diverse and extremely domain-specific. For this novel implementation, the domain considered is network traffic logs. However, this approach can be adapted for any log record type. The preprocessing steps are designed to remain domain agnostic and with similar work showing that classifiers can be trained for use with other types of logs, such as event logs (Berlin et al., 2015; Landauer et al., 2024), system logs (Zhu et al., 2023), network traffic (Garcia et al., 2020; Naas and Fesl, 2023), activity logs (Ianni and Masciari, 2023; Ombongi et al., 2024; Bash, 2024) or web traffic (Špaček et al., 2022). The generalisability and transferability of preprocessing steps are crucial because the accuracy of approximate matching algorithms, as demonstrated by Göbel et al. (2021), can vary depending on the file type. Consequently, further research by Uhlig et al. (2023), which explored machine learning and approximate algorithms for fragment detection, trained and evaluated one model per file type, which is not generalisable to all file types relevant to the use case.

The proposed multistage log analysis has two main objectives: detecting malicious logs (attack detection) and classifying log behaviour (attack classification). To achieve this, we utilised the IoT-23 network traffic logs, which were converted from raw capture formats such as PCAP files and NetFlow data into labelled log files (`conn.log.labeled` files), as shown in Table 5 with details of the record type which contains column fields of the connection log. These files were subsequently processed into CSV format for our analysis. The IoT-23 has been preprocessed into CSV files for various applications across different domains.[8,9,10] As shown in Table 5, the dataset specifies whether the data is malicious or benign and also identifies the type of malicious traffic. This processing makes the dataset suitable for binary and multiclass classification tasks, as outlined in the proposed multistage log analysis in Fig. 1. All features in the dataset originate from the Zeek[11] processing conducted by the dataset authors. The Zeek interface provides detailed information on the record type, including feature names, descriptions, and data types for all fields in the connection log. Additionally, detailed information about the design of the label assignment is publicly available.[12]

Fuzzy hashing algorithms combined with deep learning algorithms is an emerging area of research. In the literature, two approaches have been explored for converting fuzzy hashes into a suitable input format for machine learning algorithms. Fuzzy hashes are not a natural language, which makes using pre-trained models difficult. The first approach involves creating embeddings of the fuzzy hashes into vectors using techniques such as integer and positional encoding of each character (Lazo, 2021; Uhlig et al., 2023; Lingrong et al., 2023), NATO phonetic encoding (Kida and Olukoya, 2023), and tokenizing the fuzzy hashes using a rolling window approach (Peiser et al., 2020). This method has been used for tasks such as file fragment detection, threat actor attribution, and malware detection and classification. The second approach converts the hashes from approximate matching algorithms into greyscale images for malware detection and classification (Rodriguez-Bazan et al., 2023b,a). This research combines both methods to determine whether creating embeddings for fuzzy hashes or transforming them into RGB images is more effective in identifying malicious indicators in IoT network traffic.

5. A new fuzzy hash: SSDHash

In this section, we first introduce the fuzzy hashing algorithm, *SSDHash*, which is inspired by algorithms such as TLSH (Oliver et al., 2013) and JsonHash (Versteeg and Moodley, 2023). The algorithm is categorised as a Locality-Sensitive hashing algorithm, as its core digest operation is mapping similar objects into buckets. The processes shown in Fig. 2, show the major stages associated with the algorithm.

5.1. Design of SSDHash

Several key terms and notations are defined here and used throughout this section to facilitate a better understanding of the tool.

5.1.1. Notations and terminology

- Anonymous object: An object with no concrete implementation class.
- Flattened object: An object with no dimensionality or nested fields.
- Tokenized: A representation of an input string X, split on whitespace or separator characters producing an array of tokens.
- Bucket: A grouping of similar data points.
- T_i^o : the ith token pair of an object O.
- K_i : the ith key of an object O.
- V_i : the ith value of an object O.

[8] https://www.kaggle.com/datasets/engraqeel/iot23preprocesseddata/data
[9] https://ieee-dataport.org/documents/iot-data-iot-23-nsl-kdd-and-toniot
[10] https://huggingface.co/datasets/19~kmunz/iot-23-preprocessed-minimumcolumns
[11] https://docs.zeek.org/en/master/scripts/base/protocols/conn/main.zeek.html#type-Conn::Info
[12] https://docs.google.com/spreadsheets/d/1HRqgKJp0XoSUIfW3rCQKoD_LnSCJ1k-k61PndJXWq_o/edit#gid=0

Table 5
Sample labelled connection logs from IoT-23 dataset capture.

```
#separator \x09
#set_separator,
#empty_field (empty)
#unset_field -
#path conn
#open 2018-08-08-12-22-16
#fields ts uid id.orig_h id.orig_p id.resp_h id.resp_p proto service duration orig_bytes resp_bytes conn_state local_orig
        local_resp missed_bytes history orig_pkts orig_ip_bytes resp_pkts resp_ip_bytes tunnel_parents label detailed-label
#types time string addr port addr port enum string interval count count string bool bool count string count count count count set[string] string string
1533042911.474174 C5JLGOoxIw2dBZt47 192.168.100.113 123 81.2.254.224 123 udp - 0.005490 48 48 SF - - 0 Dd 1 76 1 76 (empty) Benign -
1547157068.286878 Cv9DbR3aVAeubQOrcj 192.168.1.199 38897 86.136.151.56 80 udp - 300.802367 1744830458 0 S0 - - 0 D 1305034 1781371410 0 0 - Malicious DDoS
1533047044.299948 C3eDKm2wnG96wuJNmc 192.168.100.113 43324 178.128.185.250 50 tcp - 3.151458 0 0 S0 - - 0 S 3 180 0 0 (empty) Malicious C&C
1525879831.015811 CUmrqr4svHuSXJy5z7 192.168.100.103 51524 65.127.233.163 23 tcp - 2.999051 0 0 S0 - - 0 S 3 180 0 0 (empty) Malicious PartOfAHorizontalPortScan
```

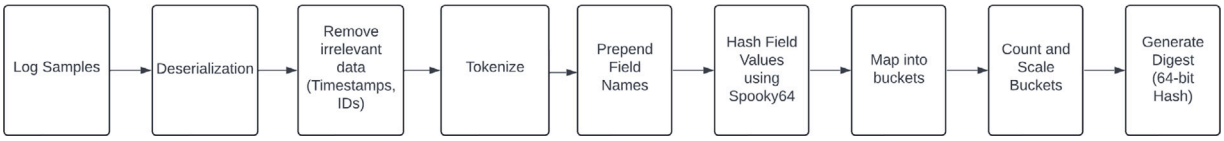

Fig. 2. SSDHash processes.

- B_i : the ith bucket.
- $C(B)$: the count of the bucket(s).
- $H(T_i^o)$: the hash representation of the ith token.
- $Rank_{B_i}$: the scaled and quantised count of the ith bucket, representing a character of the final digest.

5.1.2. Bucket frequency calculation

The calculation of bucket frequency will be a commonality between all locality-sensitive hashing algorithms such as seen in TLSH (Oliver et al., 2013) and JsonHash (Versteeg and Moodley, 2023). A bucket represents a grouping of similar objects, so mapping these objects is the primary differentiator in methodologies. For example, TLSH uses quartile points to ensure certain percentages of the counts lie in each bucket. A differing approach is the one utilised by JsonHash, which hashes each flattened field and calculates the bucket number by deducing the integer remainder when the hash is divided by the length of the output digest. This approach is the basis of SSDHash with the main differences being in choice of hashing algorithm and final rank calculation.

Before the buckets can be considered, the record is deserialized and divided into an array of key–value pairs ready for processing. The aim here is to flatten the object into a standardised representation of the fields containing relevant data, putting less emphasis on the object's structure and more on its content.

1. Let S represent an object with n properties. At this stage, the object can contain nested fields, so the object is flattened into key–value pairs. The transformation applied to S to form S' is shown in Eq. (1).

$$\begin{aligned} S &= \{K_0 : V_0, K_1 : V_1, \ldots, K_{n-1} : V_{n-1}\} \\ S' &= \{K_0 : V_0\}, \{K_1 : V_1\}, \ldots, \{K_{n-1} : V_{n-1}\} \end{aligned} \tag{1}$$

2. The next stage of the flattening process entails prepending any parent object names to the child's field names, so that a key K_i who has a parent named P_j, takes the form of "$P_j.K_i$".
3. This flattened object's values are then tokenized, breaking the values into their constituent words. If object O with n properties, produces i tokens, the transformation would take the form of Eq. (2).

$$\begin{aligned} S' &= \{K_0 : V_0\}, \{K_1 : V_1\}, \ldots, \{K_{n-1} : V_{n-1}\} \\ S^t &= \{K_0 : \{T_0^o, T_1^o\}\}, \{K_1 : \{T_2^o, T_3^o\}\}, \ldots, \{K_{n-1} : \\ &\qquad \{T_{i-2}^o, T_{i-1}^o\}\} \end{aligned} \tag{2}$$

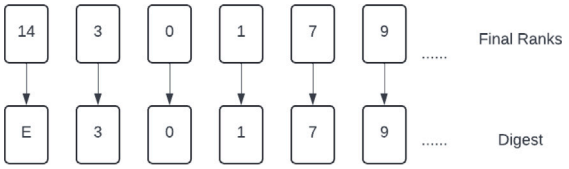

Fig. 3. Digest generation.

4. The key is then prepended to each token, which takes its final form before being hashed, such that the final input would resemble "$K_j : T_i^o$".
5. The function to calculate B_i in Eq. (3) is then applied to each hash to decide its bucket number, with the buckets then counted.

$$B_i = H(T_i^o) \text{ modulus } 64 \tag{3}$$

6. The bucket counts are then scaled with the transformation shown below. As each character in the final digest will represent a hexadecimal value, we divide each bucket count by the total bucket count and multiply by 15 as shown in Eq. (4).

$$C'(B_i) = \frac{C(B_i)}{C(B)} \times 15 \tag{4}$$

7. The scaled counts are then ordered and quantised to produce a rank. If we let j represent the position of bucket B_i in the ordered list and l its number of buckets, the function in Eq. (5) produces a rank, the final value used in the digest.

$$Rank_{B_i} = Round(\frac{j}{l-1} \times 15) \tag{5}$$

5.1.3. Digest generation

To generate the digest, the rank values are converted into hexadecimal as shown in Fig. 3, producing a fixed-length string of length 64.

5.1.4. Digest comparison

Although this research focuses on its use with logs, the algorithm is designed to work for any semi-structured data in the supported formats. Hence, a metric inspired by the Jaccard index exists to compare two digests. This metric is the same that is utilised in JsonHash (Versteeg and Moodley, 2023), where the similarity for two digests D_1 and D_2

would be calculated using Eq. (6), where a value closer to zero indicates a higher similarity.

$$Similarity(D_1, D_2) = 1 - \frac{\sum_{i=0}^{63} min(D_1[i], D_2[i])}{\sum_{i=0}^{63} max(D_1[i], D_2[i])} \quad (6)$$

While this approach is effective for simple clustering or comparing two hashes, we will demonstrate in Section 6 that a domain-specific machine-learning solution yields a better classifier than a general similarity score in the subsequent section.

6. Combining ML and approximate matching

This research aims to propose a new similarity hashing scheme for semi-structured data and evaluate it on a benchmark dataset of IoT network traffic for attack detection and classification. Identifying key cyber-attack behaviours is extremely useful as this approach will try to determine the behaviours regardless of which malware variant or malicious actor instigated the attack. This means that, unlike some tools that detect malware-specific signatures, it is less vulnerable to new variants. The key objectives of this research are as follows:

1. Implementing a fuzzy hashing algorithm tailored for semi-structured data.
2. Evaluating the approach as a fuzzy hashing algorithm tailored for semi-structured data and a comparative analysis with other approaches in the literature.
3. Implementing a malicious log classifier using fuzzy hashes.
4. Evaluating the approach using benchmarks of network logs. Since network and web logs are examples of semi-structured data, they are a good choice for evaluating *SSDHash*.

This section explores combining machine learning techniques with approximate matching to enhance the digest comparison stage. While SSDHash provides the foundation for digest generation, machine learning algorithms offer the capability to discern intricate patterns within the data, thereby improving the analysis process (Kida and Olukoya, 2023; Lazo, 2021; Uhlig et al., 2023; Peiser et al., 2020; Rodriguez-Bazan et al., 2023b,a). We describe in detail the datasets used and data type processing, preprocessing steps, embedding of the fuzzy hashes, and multistage log analysis with machine learning classifiers and performance metrics.

6.1. Datasets

The S1 dataset by Garcia et al. (2020) consists of 104,963 samples from the IoT-23[13] log samples, representing botnet activity in IoT devices at the Stratosphere research laboratory. The IoT-23 has become a standard benchmark dataset for detecting IoT attacks in a typical network (Dutta et al., 2020; Abdalgawad et al., 2021; Ullah and Mahmoud, 2021; Sahu et al., 2021). This dataset was used to derive S2 and S3 for train–test sets for the binary and multiclass models. The datasets contain three common behaviours typical in botnet attacks, as shown in Table 6. The publicly available dataset includes information on the samples, collection methods, and data processing techniques. After generating the hashes for S2 and S3, we parsed them into CSV files containing the SSDHash and the corresponding labels for the binary and multiclass classification settings. A sample of the CSV file generated from Table 5 is shown in Table 7, where an SSDHash is created for each record containing the column fields of the connection log against the label.

Dataset S4 is a 10,000 record dataset of JSON logs typical of a Web API created using the data generator Bogus[14] used for performance testing of SSDHash. S5 is a 5-sample subset derived from S4. These records were then duplicated producing five sets of identical log pairs. One record from each pair was then altered, by changing some of the field values with the data structure remaining the same. Two of these pairs were then converted to XML format to represent an alternate semi-structured data type. The characteristics of S5 are shown in Table 8, with the character length inclusive of all syntactical and structural data, but excludes formatting white-space characters.

Table 6
Dataset characteristics.

Dataset	Use	Data type	Sample count
S1	Performance Testing	Logs	104,963
S2	Classification (Binary)	Benign	24,000
		Malicious	24,001
S3	Classification (Multiclass)	Command and Control	8001
		Denial of Service (DOS)	7999
		Port Scan	7999
S4	Performance Testing	Web API System Logs	10,000
S5	AM tests	Example Web API System Log Pairs	5 × 2
Total Number of Log Samples = 186,973			

6.2. Data-type processing

The primary issue with using conventional fuzzy hashing algorithms on semi-structured objects is the input is composed of a significant amount of syntactical and structural information. This information is necessary to define the data structure in a standard way so inter-system communication is possible. However, it significantly detriments the effectiveness of the commonly used algorithms and was a primary consideration in developing SSDHash. This approach of targeted data extraction was then extended for use specifically in logs, attempting to remove unnecessary information from the structure before a digest is generated. As the idea behind building the machine learning classifier was to identify patterns in the hashes indicative of system behaviour, we wanted to ensure the data being mapped into buckets was relevant for comparison. When the logs are stored in a database, several unique fields are commonly present, representing items such as IDs and timestamps. As we consider individual logs, these fields have no relevance in defining behaviour and will only act as noise in the hashes, potentially reducing accuracy. While the output log contains numerous fields, not all of them are relevant to threat investigation and hunting (Shiva et al., 2024). Similarly, not all functions are relevant and can be discarded when generating a digest of functions for characterising malware using fuzzy hashing (Onieva et al., 2024).

For this novel implementation, two data types were chosen to be excluded to demonstrate this concept. As the tool is implemented in C#, the fields that map to the data types *Guid* and *DateTime* are removed during deserialization. An example is shown in Table 9, of the same input producing two different hashes when the whole input is hashed versus only the core data. This framework could easily be converted to any language using the same steps or by implementing a wrapper class around the library. The system's architecture also permits the creation of a custom converter class to do any custom logic by implementing the *IDataProcessor* interface, requiring little code changes to adapt for a different data set. The primary consideration of this methodology is the availability of a reliable parsing mechanism for each data type that needs to be removed. For example, data-set S1 contains timestamps in the Unix format which would typically not map to Date-Time, requiring a custom mapping function to ensure the data maps to the correct type to facilitate this processing.

[13] https://www.stratosphereips.org/datasets-iot23
[14] https://github.com/bchavez/Bogus

Table 7
Hashes produced by SSDHash on S2 and S3 Samples.

Dataset	SSDHash	Label
S2	10004000D0000000008072A0050003000000090700000000E006000C00000B0	Benign
	90300000000600700011100B4DC0000820000370500B5000000E00A00000900D0	Malicious
S3	803000460004000D305B0EA00A00002800C00000060100000D001070000900B0	C&C
	A0000000D00304060000091B08B60000030D00000800100072E004000000A05C0	DoS
	306D000000070000A00404A80B000109000021000000000500E007D00000000C0	Port Scan

Table 8
Hamming distance of S5 Samples.

Sample no.	Data type	Char length	Hamming distance
1	JSON	96	23
2	XML	218	9
3	JSON	122	6
4	JSON	197	12
5	XML	214	10

6.3. Preprocessing steps

Machine learning models necessitate feature vectors as input (Nogales and Benalcázar, 2023). To accommodate this we employ SSDHash to generate digests representing each log entry. Before the digest can be passed to the machine learning models it undergoes several preprocessing steps to transform it into a suitable set of input features that encapsulate the essential characteristics of the records. The steps are designed to remain consistent for both models, with the details of these steps below.

1. The generated digest is broken into its constituent rank values and encoded using a positional word encoding. This encoding method is similar to the NATO phonetic encoding in Kida and Olukoya (2023). However, the approach does not consider positional information. There are 16 values for each of the 64 character positions in the digest, resulting in 1024 possible encoding values. The encoding id is for a rank $Rank_{B_i}$ is derived using Eq. (7).

$$\text{encodedIndex} = (pos_i \times 16) + Rank_{B_i} \tag{7}$$

2. The encoded hashes are then tokenized, with the output being used to form n-grams. A length of 3 was chosen, with the n-gram extractor using the inverse document frequency (IDF) weighting method to produce the final feature vector inputs for the machine learning models.

6.4. Embedding of Fuzzy hashes

Fuzzy hashes generated by SSDHash are embedded into the feature vectors extracted from the log samples. The fuzzy hashes serve as a compact representation of the log entries, capturing their unique characteristics while minimising the dimensionality of the feature space. This embedding process involves representing each fuzzy hash as a fixed-length vector that can be used to derive the final 3-gram representation of the encoded characters serving as model input. An example of a 3-gram word-encoded hash is shown in Table 10.

In the case of fuzzy hashes, the position of each character is crucial to identifying patterns as is the value of each element. Therefore, for fuzzy hashes to be an effective machine learning input, there is a need for an encoding mechanism that, when executed by the classifier, can learn the relationships between different values and positions within the fuzzy hash vectors. This research tested three main encoding methods which are detailed below.

6.4.1. Positional word encoding

In the positional word encoding method, each character within the fuzzy hash is assigned a unique index that incorporates its position and value. This index is then used to map each character to a unique word. Since the hash input is fixed-length, the proposed encoding is unlikely to cause collisions. However, for workloads with higher input variability, such as in NLP tasks, more sophisticated encoding techniques that consider absolute and relative position have been proposed such as in Ke et al. (2021). An example of an encoded hash in this style is shown in Table 11.

6.4.2. One-hot encoding

In the one-hot encoding method, each character within the fuzzy hash vector is represented as a binary vector, where only one element is hot (set to 1) and all others are cold (set to 0) (Hancock and Khoshgoftaar, 2020). Each unique value within the fuzzy hash vector is assigned a unique position in the binary vector, and the corresponding element is set to 1 to indicate its presence. This encoding scheme creates sparse binary vectors for each character position, where the presence or absence of specific values is encoded explicitly. The major difference between the word encoding method is that an input encoded using one hot encoding requires additional feature inputs to represent the content of the data. In the case of the models developed as a part of this research, the feature input is the combination of the n-grams and the encoded tokens, meaning the dimensionality of the feature column is twice as big compared to the previous encoding. An example of this method used in research is in Kang et al. (2019), where one-hot encoding is used to identify malware using opcodes and API function names. An example of one-hot encoding of an SSDHash digest chunk is shown in Table 12.

6.4.3. Image transformation of hashes

This process starts by extracting the fuzzy hashes from the semi-structured data. The hexadecimal hashes are split into individual characters, converted into integer values, and multiplied by 68. After this, a 3-channel RGB code is created for each integer value to represent the character visually. Each code in the channel represents an RGB value within the range of 0–255 for each colour. Table 13 provides an example of a generated RGB colour tuple and hex colour code for four sample characters (049F) in a hash.

The fuzzy hashes of each log sample are eventually converted to a list of RGB tuples. After that, it is reshaped into a 12 × 16 2D array for balance. Finally, an RGB image is generated from the array in PNG format. The RGB images from two sample SSDHash are shown in Fig. 4.

6.5. Multistage log analysis with machine learning classifiers

In this research, we propose a multistage log analysis tool that utilises fuzzy hashes as inputs for behaviour categorisation. The primary objective is to automate the log analysis process by categorising entries based on their behaviour, distinguishing between benign and malicious activities, and categorising malicious activities by their specific behaviour types. To achieve behaviour categorisation, we employ supervised machine learning algorithms, which leverage labelled data to train classifiers capable of distinguishing between log behaviours. Support Vector Machines (SVM) and Stochastic Dual Coordinate Ascent (SDCA) classifiers are trained on labelled log samples to learn patterns

Table 9
Hashes produced by the same input.

Log mode	00000**6**0**C**0000000**0**0000000**4**000008000**0E**000000000**2**00**A**0000000000000000
Normal	00000**8**0**C**0000000**3**0000000**6**002009000**0E**000000000**5**00**B**0000000000000000

Table 10
Example 3-gram word-encoded hash.

Encoded hash	3-Grams
This Thousand Toward Two Violence Weight Woman World	"This Thousand Toward", "Thousand Toward Two", "Toward Two Violence", "Two Violence Weight", "Violence Weight Woman", "Weight Woman World"

Table 11
Word-encoded hashes.

Digest chunk	Encoded hash
C00000B0	This Thousand Toward Two Violence Weight Woman World

Table 12
One hot encoded Hashes.

Digest chunk	Encoded vectors
C00000B0	[1,0,0, 0,1,0] [0,1,0, 0,1,0] [0,1,0, 0,1,0] [0,0,1, 0,1,0]

Table 13
Assigning RGB Tuple to Hashes.

Character	Scaling (×68)	RGB colour tuple	Hex colour code
0	0	(0,0,0)	#000000
4	272	(0,1,16)	#000110
9	612	(0,2,100)	#000264
F	1020	(0,3,252	#0003FC

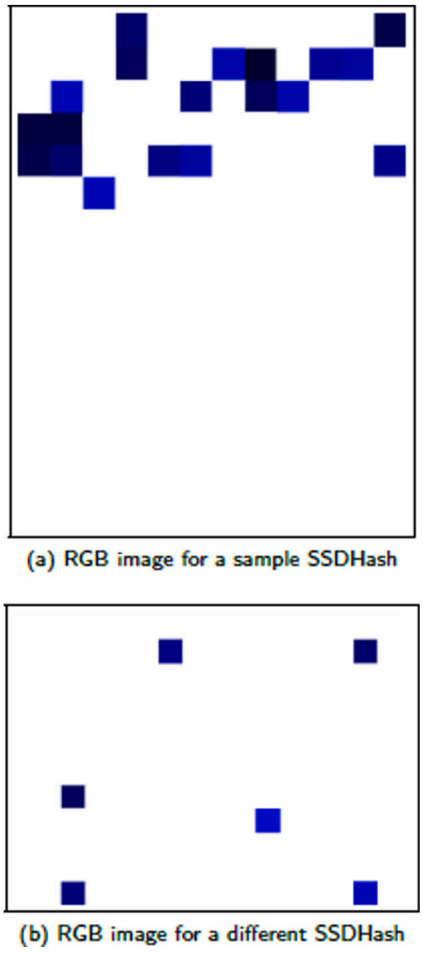

Fig. 4. SSDHash image representation.

indicative of benign and malicious behaviour and different behaviour types. In addition to supervised learning, we integrate unsupervised learning techniques to explore the underlying structure of the log data. One such technique is K-Means clustering, which is used to group log entries based on similarity in behaviour. While supervised classifiers aim to provide concrete behaviour categorisation results, K-Means clustering is utilised to investigate the effectiveness of grouping log entries based on fuzzy hash inputs.

For the binary classification task distinguishing between benign and malicious log entries, the Support Vector Machine (SVM) algorithm is used. SVM was chosen because it can effectively classify data points in high-dimensional spaces, making it ideal for handling the feature-rich inputs generated through the preprocessing steps (Cortes and Vapnik, 1995). Secondly, the Stochastic Dual Coordinate Ascent (SDCA) algorithm categorises malicious log entries into specific attack types. SDCA was chosen due to its efficient convergence properties. These properties make it suitable for handling the complex decision boundaries of distinguishing between attack types. An example of using this algorithm in practice is studied in Oros and Bâcu (2022), where a framework including an SDCA classifier is used to identify malicious bots in online multiplayer games based on their behaviour. There have been several multilabel, top-k and multiclass performance optimisations designed based on SDCA (Lapin et al., 2018; Fan et al., 2008; Chu et al., 2018; Lapin et al., 2015; Shalev-Shwartz and Zhang, 2014) to solve large-scale supervised learning problems. These optimisations are suitable for the IoT-23 dataset (Garcia et al., 2020), a heterogeneous, multi-labelled, and multiclass dataset. In this dataset, the labels represent different attacks, and the classes can combine different attacks. Although three machine learning models were developed as a part of the research, only two are used for classification. The clustering models were used to produce visualisations to determine the suitability of a dataset for classification.

For the model implementation, the ML.NET[15,16] API was used for developing the given models. Like the use of C# for developing SSDHash, this choice of ML framework is interchangeable due to the cross-platform format ONYX, used for representing the ML models. The classifiers are assessed to gauge the applicability of fuzzy hashes for behaviour categorisation. Performance evaluation metrics, such as recall and precision, are derived from the confusion matrix with their formulas outlined in Table 14. The details of the performance metrics are given below:

- **True Positive (TP)**: The model correctly predicts that a log entry is malicious or exhibits a specific behaviour category.

[15] https://dotnet.microsoft.com/en-us/apps/machinelearning-ai/ml-dotnet
[16] https://learn.microsoft.com/en-us/dotnet/machine-learning/how-to-choose-an-ml-net-algorithm

Table 14
Performance metrics.

Metric	Formula
Accuracy	$\frac{TP+TN}{TP+TN+FP+FN}$
Recall	$\frac{TP}{TP+FN}$
Precision	$\frac{TP}{TP+FP}$
F1	$\frac{TP}{TP+\frac{1}{2}(FP+FN)}$

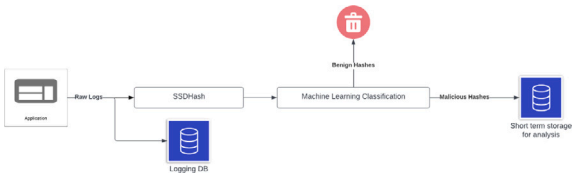

Fig. 5. Tool usage in application.

- **True Negative (TN)**: The model correctly predicts that a log entry is not malicious.
- **False Positive (FP)**: The model incorrectly predicts that a log entry is malicious or exhibits a behaviour category when it does not, erroneously flagging benign behaviour as malicious.
- **False Negative (FN)**: The model fails to identify a log entry as belonging to a behaviour category when it does, meaning it misses instances of malicious activity.
- **Accuracy**: This metric assesses the model's overall performance by measuring the ratio of correctly classified log entries to the total number of log entries.
- **F1 Score**: This metric combines precision and recall to provide a balanced assessment of the model's performance across all behaviour categories. It considers the model's ability to correctly identify instances of a behaviour category (precision) and capture all instances of that category (recall). A higher F1 score indicates better overall performance in behaviour categorisation.
- **ROC AUC**: The Receiver Operating Characteristics curve and its corresponding area under the curve (AUC) offer insights into the model's ability to distinguish between different behaviour categories across various classification thresholds, with a higher AUC indicating better performance in distinguishing between behaviours.
- **Logarithmic Loss**: A measure of the model performance that quantifies the difference between predicted probabilities and actual values.

The dataset shown in Table 6 demonstrates diversity by including various types of log data, such as network traffic logs (S1–S3) and Web API system logs (S4–S5), for analysis. Since logs are essential for system and network observability, the proposed algorithm's evaluation of these datasets is relevant for addressing emerging security threats. For instance, the OWASP Top 10:2021, a standard reference document for developers and application security, ranks security logging and monitoring as the 9th most critical application and internet security risk (OWASP Top 10 team, 2021). One such security threat is the lack of application and API logs monitoring for suspicious activities, which aligns with our research. We are using fuzzy hashing to analyse malicious activities in logs. This research demonstrates the identification and classification of malicious activities in logs, which can have significant implications for reliability, availability, accountability, visibility, incident alerting, and forensic investigations of cyber incidents (Shiva et al., 2024; OWASP Top 10 team, 2021; He et al., 2022b; Landauer et al., 2024; Zhu et al., 2023).

7. Evaluation

The two main goals of this research are to develop novel approaches and techniques for producing fuzzy hashes for semi-structured data and using these hashes as input to a machine learning classifier to detect the prescience of malware within a network. The evaluation of these statements is split up into the following research questions:

- **RQ1**: How does SSDHash perform as a novel fuzzy hashing algorithm for use with logs?
- **RQ2**: To what extent is fuzzy hashing suitable for log behaviour classification?
- **RQ3**: How do positional word encoding, one-hot encoding, and RGB image encoding compare in representing hashes for classification??
- **RQ4**: How does removing certain data types affect hash computation?
- **RQ5**: Is the tool efficient for real-time analysis?
- **RQ6**: How does the multistage log analysis with the machine learning classifier solution perform when trained on raw log samples instead of fuzzy hashes?

7.1. RQ1: How does SSDHash perform as a novel fuzzy hashing algorithm for use with logs?

To address RQ1, we evaluate the performance of SSDHash using a range of benchmarks. These benchmarks aim to assess whether SSDHash has achieved its key objectives related to performance and accuracy. Although SSDHash is designed to be a fuzzy hashing algorithm tailored for semi-structured data, the primary use in the case of this research is its use within the log analysis tool. The theoretical use of this framework would place it as middleware within an existing logging mechanism, with the fuzzy hashes being an intermediary representation before the machine learning model classifies them. The records deemed malicious can then be stored in short-term storage so that the underlying records can be analysed, with the remainder discarded. An illustration of this process is shown in Fig. 5.

One use case for this tool is in forensic investigation and security operation workflows. Collecting and monitoring logs to identify suspicious activities, discover anomalies, and create baseline usage patterns is crucial for threat investigation and detection (Shiva et al., 2024). Incident responders, cybersecurity analysts, and security operations centre (SOC) analysts often need to analyse large volumes of logs and other types of semi-structured data to identify potential malicious activity. This can involve parsing the logs and grouping them according to the servers they originate from, which helps to identify connections between different logs based on their fuzzy hash similarity. For instance, if an incident responder discovers a malicious log, they can easily identify similar logs supporting threat hunting, clustering, and pivoting. The logs' fuzzy hashes, represented by a 64-dimensional vector of hash bucket counts, can be stored in a vector database to enable rapid similarity comparisons. Additionally, all log entries can be hashed and clustered, and the multistage log analysis architecture allows for detecting outlier clusters and abnormal log entries. Operationalizing SSDHash in the described use case scenario in security operations centres (SOCs) can be significant for detection and threat hunting, grouping and tracking malware campaigns, retroactive hunting and during incident response for identifying new infrastructure (Signalblur, 2024; Versteeg and Moodley, 2023).

Approximate matching algorithms are typically evaluated using the FRASHER framework by Göbel et al. (2022), which uses a range of tests to quantify an algorithm's performance in areas such as efficiency and sensitivity with the capabilities to test functionality like fragment detection and adversarial resilience. The problem with using FRASHER to evaluate SSDHash is that the tests run using document and image files for input (PDF, docx, JPG etc.). For this novel implementation, the algorithm only accepts inputs in JSON, XML, and CSV format, thus making it unsuitable for FRASHER evaluation.

Table 15
SSDHash performance benchmarks.

Method	Sample count	Mean S4 (µs)	Theoretical throughput S4 (samples/s)	Mean S1 (µs)	Theoretical throughput S1 (samples/s)
Individual	1	40.89	24,758	30.86	32,509
Parallel	10	124.43 (12.44 µs/sample)	80,385	108.97 (10.89 µs/sample)	91,827
	100	782.15 (7.82 µs/sample)	127,877	874.15 (8.74 µs/sample)	114,416
	1000	8584.21 (8.58 µs/sample)	116,550	8315.13 (8.32 µs/sample)	120,192
	10,000	87,969.93 (8.79 µs/sample)	113,765	87 590.99 (8.76 µs/sample)	114,115

To provide a comparable benchmark for SSDHash, a series of performance tests related to its proposed use in real-time analysis were created. For this proposed tool to have utility in this area, the processing performance of the hashing algorithm would have to exceed the number of records produced. In this section we focus on the performance of the hashing algorithm, comparing the performance for both an individual record and multiple records in which parallel programming concepts are utilised to increase throughput. To define the levels of performance we introduce the following terms:

- Mean: Arithmetic average of all measurements
- µs: 1 Microsecond (0.000001 s)
- Standard Deviation (SD): The amount of variation of a random sample expected about its mean
- Error: Half of 99.9% confidence interval in measurements

Each ran multiple times to remove variance and ensure the tests remained consistent. Two complex datasets were created to benchmark the system's performance. S4 is a set of typical Web API logs with multiple nested objects; while S1, is a set of network logs. The tests ran benchmarks for both the single-use and batching functionality. In the real world, an individual record would seldom be run, but this typically is a worst-case scenario in terms of performance as the runtime cannot take advantage of things like caching and parallel execution. The algorithm can then be tested with increasing sample sizes to calculate a maximum throughput in real-world terms, as these would be batched in a real implementation. All tests were run using BenchmarkDotNet[17] on a system with an AMD Ryzen 7 5800X CPU with 16 GB of RAM to represent an average server build.

The summarised results for these tests are shown in Table 15, with the full results for each run shown in Table 16. It is worth noting that these results represent the processing time for the batching logic and the hashing of the data. However, they do not account for overheads such as IO that would be typical in a real system, which could involve tasks such as reading records from a database. Additionally, these tests were run using the string input feature with serialisation overheads included in these times. However, the API has functionality to consume objects, which if the data was coming from the database is optimal as it removes the need for additional unnecessary serialisation overheads. The purpose of these tests is to measure if the algorithm can consume data at the same rate as it is produced in a large system, hence the emphasis on the batching capabilities of the tool. The results produced on this modest hardware indicate the algorithm is effective in processing enough data for even the largest workloads, with almost all batch configurations achieving a throughput of over 100,000 samples/s.

Table 17 shows the SSDHash's results of the tests run on the S5 dataset, with the results of the other widely used AM algorithms in Table 4. Samples from S5 are shown in Listing 1. The similarity score is calculated on a scale of 0 (lowest) to 1, with lower numbers indicating poor similarity and higher numbers suggesting great similarity between the two data sets under consideration. When considering the produced results it is worthwhile to note that sample 1 produces the largest hamming distance relative to its size and thus theoretically should be maximally different from other samples. Additionally, sample 5 produces a low hamming distance for its size, but due to the nature of XML requiring both an opening and closing element this value is inflated relative to the data it represents. Sample 5 contains the most unique fields out of all samples tested and thus should produce a low similarity score. These assertions prove true from the results, with samples 1 and 5 producing similarity scores of 0.81 and 0.73, respectively. When compared to JsonHash the similarity levels remain consistent, with the main differences arising from SSDHash's log mode removing fields such as time stamps, which in the case of these samples should decrease similarity as the amount of unique data relative to their size increases.

```
1.
{
    "timestamp": "2023—12—11T08:30:45.123Z",
    "message": "User 'john_doe' successfully logged in."
}

2.
<log>
    <timestamp>2023—12—11T09:15:22.567Z</timestamp>
    <error>
        <code>500</code>
        <description>Internal Server Error</description>
        <details> Failed to process request due to database
        connection failure.</details>
    </error>
</log>

3.
{
    "timestamp": "2023—12—11T10:45:10.890Z",
    "event":{
        "name": "FileUploaded",
        "file": {
            "name": "document.pdf",
            "size": 1024
            }
        }
}

4.
{
    "timestamp": "2023—12—11T14:20:30.500Z",
    "info": {
    "type": "Transaction",
    "amount": 120.75,
    "details": {
        "sender": "user123",
        "recipient": "merchant456",
        "message": "Payment for services rendered."
        }
    }
}
5.
<log>
    <timestamp>2023—12—11T15:45:40.078Z</timestamp>
```

[17] https://github.com/dotnet/BenchmarkDotNet

Table 16
Comprehensive benchmark results for SSDHash.

Test	Batch size	Run	1	2	3	4	5
S1 SSDHash Performance	1	Mean (µs)	40.05	41.14	43.05	40.62	39.61
		Error (µs)	0.349	0.284	0.251	0.205	0.495
		SD (µs)	0.327	0.251	0.210	0.182	0.413
	10	Mean (µs)	125.77	126.24	124.95	122.94	122.23
		Error (µs)	2.450	2.261	1.022	0.889	0.944
		SD (µs)	2.515	2.115	0.956	0.832	0.789
	100	Mean (µs)	774.79	781.38	790.41	774.46	789.71
		Error (µs)	5.344	7.330	15.030	4.881	4.818
		SD (µs)	4.999	6.856	14.059	4.565	4.271
	1000	Mean (µs)	8466.21	8600.86	8575.56	8461.71	8816.69
		Error (µs)	128.266	169.742	107.687	164.032	174.024
		SD (µs)	119.980	188.668	95.462	182.321	226.280
	10,000	Mean (µs)	90,186.16	89,379.38	90,419.62	89,705.01	93,779.54
		Error (µs)	1735.316	1774.216	1646.373	972.363	1875.426
		SD (µs)	2065.770	1972.036	1540.018	909.549	1925.924
S4 SSDHash Performance	1	Mean (µs)	30.66	31.01	30.79	30.76	31.12
		Error (µs)	0.075	0.101	0.141	0.179	0.181
		SD (µs)	0.062	0.089	0.132	0.167	0.160
	10	Mean (µs)	108.29	109.37	109.52	108.23	109.47
		Error (µs)	0.289	0.426	1.119	0.437	1.349
		SD (µs)	0.242	0.356	0.992	0.387	1.262
	100	Mean (µs)	873.76	874.46	874.60	874.11	873.80
		Error (µs)	8.728	9.931	4.911	7.893	9.103
		SD (µs)	8.164	9.289	4.593	7.373	8.069
	1000	Mean (µs)	8287.78	8366.85	8324.14	8275.07	8321.81
		Error (µs)	63.432	95.596	65.394	44.029	80.526
		SD (µs)	59.334	89.420	54.607	34.375	71.384
	10,000	Mean (µs)	87,405.20	87,844.06	88,481.18	86,940.45	87,284.53
		Error (µs)	218.602	367.110	986.832	172.676	163.146
		SD (µs)	170.670	306.553	874.800	134.814	127.374
S1 Classification	1000	Mean (ms)	50.70	50.74	50.08	49.86	51.06
		Error (ms)	1.013	0.641	0.837	0.583	0.912
		SD (ms)	1.08	0.496	0.696	0.460	0.806
	10,000	Mean (ms)	62.81	62.83	62.64	62.53	62.37
		Error (ms)	1.007	1.174	1.191	1.078	1.102
		SD (ms)	0.841	1.257	0.995	0.847	0.466
	100,000	Mean (ms)	193.90	196.40	192.05	195.33	200.27
		Error (ms)	3.30	3.74	1.89	4.02	1.95
		SD (ms)	3.08	3.67	2.31	3.56	2.02

Table 17
SSDHash similarity results using the S5 dataset.

Algorithm	Set 1	Set 2	Set 3	Set 4	Set 5
SSDHash	0.81	0.40	0.41	0.49	0.73

```
    <event>
        <name>RequestReceived</name>
        <details>
            <method>POST</method>
            <path>/api/orders</path>
            <body>{"product": "Laptop",
            "quantity": 2}</body>
        </details>
    </event>
</log>
```

Listing 1: Samples from S5

An example of the difference between the produced digests for SSDHash and JsonHash is demonstrated in Table 18 in which Sample 1, Sample 3 and Sample 4 from dataset S5 are hashed using these algorithms, demonstrating the reduced noise in the digests for SSDHash. As the JsonHash implementation currently only supports JSON inputs, Sample 2 and Sample 5 were omitted because they are in XML format.

Furthermore, we compare the proposed approximate matching algorithm, SSDHash, against existing algorithms like ssdeep, TLSH and JsonHash. Table 19 shows the performance metrics for some of these commonly used fuzzy hashing algorithms tested using datasets S1 and S4. These results were used to calculate a series of metrics to demonstrate the computational efficiency of each algorithm, with the results available in Table 20. The computational efficiency metrics were inspired by those used in Göbel et al. (2022), the details of which are given below.

- E : Execution time
- $C(T)$: Token count
- T^o : Output token
- T^i : Input token
- **Generation Efficiency** reflects the ease with which the respective algorithm can process data to produce the similarity digest. The metric is taken as a collective average across the benchmark, with the metric calculated using the formula:

$$GE = \frac{E}{C(T^i)} \tag{8}$$

- **Comparison Efficiency** expresses the runtime efficiency to compare digest. The metric is taken as a collective average across the benchmark, with the metric calculated using the formula:

$$CE_1 = \frac{E}{C(T^i)} \tag{9}$$

- **Compression Efficiency** evaluates the compression rate of the algorithm by measuring the size of the digest with the input size

Table 18
Hashes produced by SSDHash and JsonHash on S5.

Sample 1	SSDHash	0000000008000000F0B0000000000000000000000040000000000000000
	JsonHash	0001000000011000010000010010000000000000010000201010000020010100
Sample 3	SSDHash	000000000050000000000000000000000000A0000000000000000F0000000
	JsonHash	0002000000200000000000022000000000020000000000000000020000000
Sample 4	SSDHash	0000060D000000000000040000900000F000000000200B0000000000000000
	JsonHash	0500000000000C0000000000020000000A000000F0000000000800000000

Table 19
Performance benchmarks of comparable algorithms using single execution times.

Algorithm	Mean S4 (µs)	Theoretical throughput S4 (samples/s)	Mean S1 (µs)	Theoretical throughput S1 (samples/s)
JsonHash	82.13	12,192	59.03	17,091
ssdeep	32.14	31,113	26.74	37,397
TLSH	75.14	13,308	65.12	15,356

Table 20
Performance & Efficiency metrics of comparable algorithms.

Algorithm	Generation efficiency	Comparison efficiency	Compression efficiency
SSDHash	0.074	50.49	11.59%
JsonHash	0.107	8.07	11.59%
ssdeep	0.048	5.41	13.29%
TLSH	0.118	11.13	12.68%

using the formula:

$$CE_2 = \frac{C(T^o)}{C(T^i)} \times 100 \tag{10}$$

The result of the individual runs for SSDHash is omitted from Table 19, as its performance benchmarks have already been presented in Table 15. The results presented in Table 19 are for the individual runs as conducting parallel runs for the algorithms is not a native feature of the tools. Therefore, all the metrics are based on a single execution time. As shown in Table 20, the compression efficiency of SSDHash and JsonHash produce the same result as they are both fixed 64-character digests acting on the same input, whereas other approximate matching algorithms like ssdeep will vary in length between inputs. The low generation efficiency demonstrates the high performance of the digest generation steps within SSDHash compared to other algorithms. However, the high comparison efficiency metric shows the performance downside of using a machine-learning solution instead of a low complexity hash comparison such as the Jaccard Index utilised by JsonHash.

7.2. RQ2: To what extent is fuzzy hashing suitable for log behaviour classification?

To address RQ2, we discuss the results of using fuzzy hashes as input for a machine-learning classifier. These hashed log samples were encoded using the positional-word encoding and preprocessing steps detailed in Section 6.3. These samples were used as inputs to train two machine learning classifiers using an 80:20 train–test split. The first classifier is designed to filter out benign samples representing the system's normal use. The classifier is trained using a supervised local deep SVM model. Approximately 24,000 benign samples served as input to define this normal use, with an additional 24,001 malicious samples to define a positive result. The multiclass classifier is a supervised SDCA model that can predict the behaviour of a given malicious log, in which there are approximately 8000 records of each class as shown in Table 6. To evaluate these classifiers the metrics defined in Table 14 are used, with the results presented below in Tables 21 and 22.

In answering RQ2, the result of the performance metrics provides compelling evidence in favour of fuzzy hashes as an effective input for a machine learning classifier. When the dataset was investigated using methods such as KMeans clustering to visualise the separation between classes the resulting diagrams displayed good inter-class separation. However, the resulting diagrams alluded to more difficulty when developing the binary model due to the malicious category being a combination of several well-defined clusters. The binary model using a positional word encoding produced 97% accuracy, with 98% F1 score and 99% AUC as shown in Table 21. When developing this model the metrics considered most closely aimed to minimise the number of false negatives. The reasoning behind this choice specifically for the binary model is that it acts as a filter for incoming logs. If this filter is overfit for benign samples the likelihood of a malicious record producing a false negative increases and thus the chance of detection decreases. While the tool is designed to cluster potentially malicious records and reduce the amount of work an analyst is required to do, a slight bias towards malicious is preferable here to increase the overall detection rate. The results for the binary model are marginally lower than the multiclass version, which produced 99% accuracy and F1, with recall and precision metrics at or greater than 99% as shown in Table 22. For this novel implementation, both models are extremely effective for classifying the given behaviours of the underlying logs.

Table 21
Positional Word Encoding — Binary model performance metrics.

Metric	Word encoding
Accuracy	0.97
Recall	0.97
Precision	0.99
F1	0.98
AUC	0.99

Table 22
Positional Word Encoding - Multiclass model performance metrics.

Accuracy	Log-loss	Class	Recall	Precision	F1
0.99	0.008	Port Scan	1.00	0.99	0.99
		C&C	0.99	1.00	0.99
		DoS	1.00	1.00	1.00
		Average:	0.997	0.997	0.993

To better understand the factors contributing to the better performance of the multiclass model over the binary classification model, we conducted additional experiments by training the model using KMeans clustering. The results are shown in Fig. 6, with each colour representing a different class — DoS, C&C, Port Scan and Benign. Fig. 6 illustrates the distinct separation of the different classes, suggesting better performance in the multiclass (Table 22) compared to the binary class (Table 21).

7.3. RQ3: How do positional word encoding, one-hot encoding, and RGB image encoding compare in representing hashes for classification?

To address RQ3, we discuss the effect of using the three tested encoding mechanisms - *positional encoding*, *one-hot encoding* and *image encoding*, as detailed in Section 6.4. When strings are typically encoded for use in ML, methods such as integer encoding at a word level usually suffice. However, from the testing performed in this research and relevant literature such as Peiser et al. (2020), Uhlig et al. (2023),

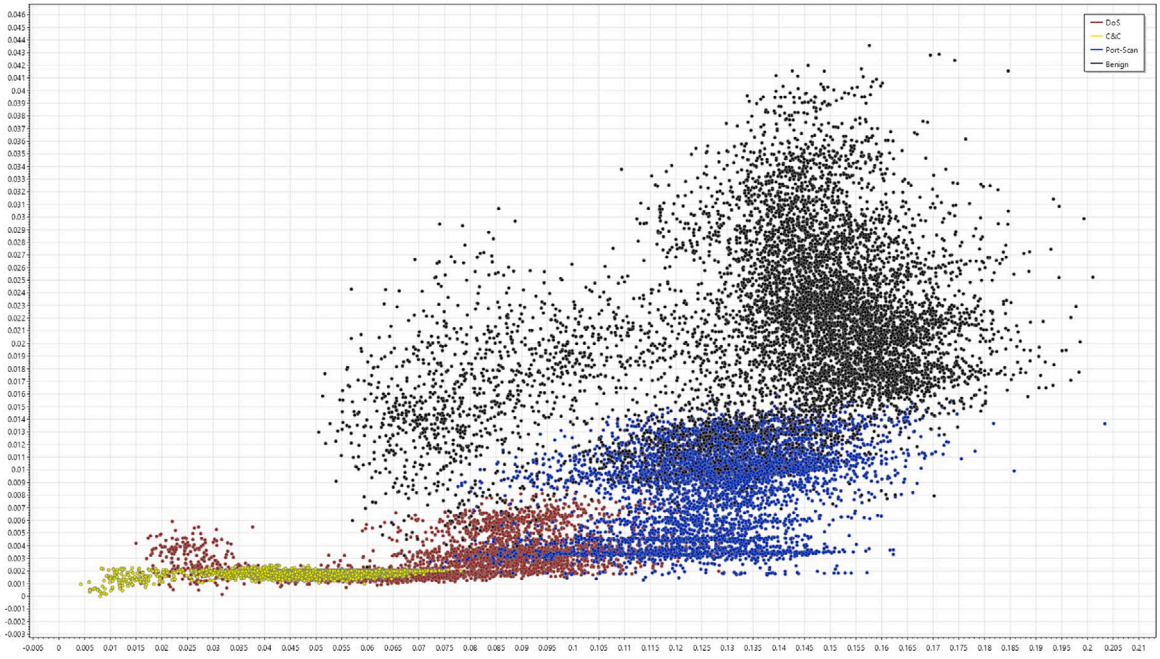

Fig. 6. Clustering diagram derived from KMeans model of the utilised dataset.

Table 23
One-Hot Encoding — Binary model performance metrics.

Metric	One hot encoding
Accuracy	0.95
Recall	0.95
Precision	0.95
F1	0.95
AUC	0.98

Table 24
One-Hot Encoding - Multiclass model performance metrics.

Accuracy	Log-loss	Class	Recall	Precision	F1
0.98	0.041	C&C	0.96	0.99	0.98
		DoS	0.99	0.99	0.99
		Port Scan	0.91	0.88	0.89
		Average:	0.953	0.953	0.953

Table 25
Image Encoding — Binary model performance metrics.

Metric	Image encoding
Accuracy	0.86
Recall	0.80
Precision	0.91
F1	0.85

Table 26
Image Encoding - Multiclass model performance metrics.

Accuracy	Log-loss	Class	Recall	Precision	F1
0.95	0.151	C&C	0.96	0.99	0.97
		DoS	0.99	0.96	0.97
		Port Scan	0.86	0.99	0.92
		Average:	0.937	0.980	0.953

the use of positional encoding at a character level is essential to train an effective classifier when working with fuzzy hash inputs. Without such encoding, the resulting models seldom produced better accuracy than random chance. The model's results using positional encoding for the binary and multiclass classification tasks are presented in Tables 21 and 22.

The primary difference in methodologies between these encoding methods is that when one hot encoding is used the resulting vector captures position information well, but fails to capture the intricacies of the data. The feature input is a concatenated vector of N-grams and these encoded values make it an effective classifier. Compared to the positional word encoding model which encodes the characters before they are mapped to n-grams, it produces a feature vector half the size of the one hot encoded model. This results in more work required to process and thus longer processing times for prediction. The results of the model using one hot encoding are detailed in Tables 23 and 24.

The RGB image encoding approach, which encodes each character of the hash as an RGB pixel, creates an image as an input into a Deep Neural Network (DNN) for classification (Rodriguez-Bazan et al., 2023b,a). The visual nature of the RGB-encoded model allows the DNN to use its capability to identify patterns and anomalies in images. However, this method comes at the cost of increased computational resources and longer training times than the other two methods due to the higher processing complexity of image data processing. Tables 25 and 26 provides a detailed overview of the results obtained from the RGB image encoding approach.

The one hot encoded model shows notable metrics, producing greater than 0.95 in all key measures. Similarly, the image encoding model produced strong metrics, such as an F1-score of 0.85 and 0.95 for the binary and multiclass models respectively. However, a key requirement considered when building this tool was performance, with the theorised use being real-time analysis of logs the resulting model would have to perform well while also being effective enough to prevent a notable backlog. When compared to the positional word encoding, the combination of worse performance and worse metrics resulted in a simple choice for the final tool. The documentation[18] related to one hot encoding suggests it is most effective when working with a small number of positional categories and could potentially explain the difference in performance when compared with the positional word

[18] https://www.analyticsvidhya.com/blog/2020/03/one-hot-encoding-vs-label-encoding-using-scikit-learn/

Table 27
Full Data Input — Binary model performance metrics.

Metric	Full data input
Accuracy	0.88
Recall	0.89
Precision	0.90
F1	0.89
AUC	0.91

Table 28
Full Data Input — Multiclass model performance metrics.

Accuracy	Log-loss	Class	Recall	Precision	F1
		Port Scan	0.91	0.88	0.89
0.93	0.362	C&C	0.93	0.99	0.96
		DoS	0.93	0.91	0.92
		Average:	0.923	0.927	0.923

Table 29
Classifier performance.

Batch size	Mean (ms)	Theoretical throughput (samples/s)
1000	50.48	19,810
10,000	62.64	159,642
100,000	195.59	511,274

Table 30
Raw log samples — Binary model performance metrics.

Metric	One hot encoding
Accuracy	0.99
Recall	0.99
Precision	0.98
F1	0.99
AUC	0.99

encoding. To answer RQ3, the positional word encoding method is the most effective method for transforming fuzzy hashes as input for machine learning classifiers.

7.4. RQ4: How does removing certain data types affect hash computation?

The goal of RQ4 is to assess the robustness of the proposed algorithm by investigating the impact of removing specific data types from the log. This is done to simulate significant changes in the structure or format of logs, such as those coming from different devices or applications. To answer RQ4, we evaluate the effect of removing certain data types to reduce noise in the resulting fuzzy hashes. An example of the difference this can make to the digest is documented in Table 9. When evaluated with the tool hash comparison function, the difference is observable, with the resulting hashes producing a 0.2 similarity score. So while the effect on the hash is clear, this part of the research aims to see its impact on the classifier. In the case of the dataset used, it contains a timestamp which is removed before the hashes are generated.

Tables 27 and 28 show the results of the classifiers when trained on the fuzzy hashes of the same dataset with the timestamps included. The models were trained using the same encoding and methodology as the final model detailed in RQ2, with the only difference being hash generation. The inclusion of the timestamp in the case of our dataset added noise to the hashes with the resulting performance metrics of the models reduced by 9.6% and 7.3% on the binary and multiclass tasks, when compared with the results of the noise-free models in Section 7.2. This result is not surprising, with noise reduction being a key research field in machine learning for numerous years (Nettleton et al., 2010; Zhu and Wu, 2004; Gupta and Gupta, 2019). However, it further emphasises the requirement for quality data sets and the valuable role of data cleaning when training effective classifiers (Xiong et al., 2006).

However, even with the addition of noise, the proposed approach produces an F1-score of 0.89 on the binary classification task, as shown in Table 27, while making an average F1-score of 0.923 in the attack classification task as shown in Table 28. While these results suggest some level of robustness to noise in logs, a systematic noise filtering process that improves the quality of the system logs such as outliers detection (Conforti et al., 2015), imperfection patterns (Suriadi et al., 2017), infrequent behaviour detection (Conforti et al., 2017) might be beneficial before the hash generation to increase robustness.

7.5. RQ5: Is the tool efficient for real-time analysis?

To answer RQ5, we introduce the performance metrics for classifiers, with the results documented in Table 29. Like the performance benchmarks for SSDHash in Section 7.1 Tables 15 and 16, which were aimed to demonstrate its utility in generating the inputs for the classifiers, these tests are aimed to demonstrate the feasibility of using machine learning in real-time analysis. Like the SSDHash tests, the results represent the time taken during classification and do not include things such as IO. The time shown is when a model filters out benign records, with the remaining records classified by the multiclass model. The dataset from which the training data was derived was used for these tests. These represent real log records and thus should be a relevant metric for performance when trained on mid-tier hardware.

Unlike the fuzzy hashing algorithm which runs on a CPU, these models utilise the parallel performance benefits of running on a GPU, in this case, a GTX 1070. The performance benefits of utilising GPUs for machine learning tasks are clear here, as unlike the CPU-bound SSDHash, the parallelisation overhead diminishes with increasing batch size by using the optimised CUDA units for processing. The theorised architecture of the framework shows the hash generation and classification steps separately in Fig. 1. In a deployed implementation, this is the case to allow for efficient scaling of each process as log output levels change, which was the purpose of benchmarking each process individually. The results in Table 29 demonstrate that the framework can meet scalability and performance demands even for a large system.

7.6. RQ6: How does the multistage log analysis with the machine learning classifier solution perform when trained on raw log samples instead of fuzzy hashes?

The goal of RQ6 is to investigate the comparative advantage of hashing logs as opposed to using the logs as it is done in existing log analysis solutions (see Section 3.2). To answer RQ6, we discuss the results of a classifier trained on the raw inputs of log samples. The same dataset and machine learning algorithms utilised for the hash-based classifiers have been reused for these experiments to ensure comparative consistency. The results for these classifiers can be seen in Tables 30 and 31 with the primary difference being in the feature extraction stage where the input for the model would be text. Each text feature must first be converted to a vector to be a suitable input for the classifier. To produce this input, several techniques employed in Ring et al. (2021) were adapted and tested for use in this domain. Timestamps and IDs were omitted as features like the hash-based classifier, with the rest of the data embedded using *GloVe*, obtaining vector representations for the words. *GloVe* was chosen because it is widely used in practice to construct log representations from various domains and platforms (Chen et al., 2020; Fu et al., 2023; Ring et al., 2021).

Other than performance, these two feature extraction approaches have distinct advantages. When approaching the development of a

Table 31
Raw log samples — Multiclass model performance metrics.

Accuracy	Log-loss	Class	Recall	Precision	F1
		C&C	0.99	0.58	0.73
0.75	0.548	DoS	1.00	0.99	0.99
		Port Scan	0.24	1.00	0.39
		Average:	0.743	0.860	0.703

classifier tied to a specific domain, investing time into performing proper feature engineering allows for additional control and can yield a strong classifier, however, this requires significant time and domain knowledge to achieve. In comparison, a hash-based input provides a way of rapidly developing new models using a standardised framework. This results in the ability to deploy and scale a range of systems tailored to a specific domain, while reducing the ongoing development to maintain. For the binary classification, the results show that the evaluation metrics for the binary model (F1-score of 0.99 in Table 30) are comparable with the hash-based equivalent (F1-score of 0.98 in Table 21). However, the multiclass model that utilised the raw logs produced significantly lower metrics (F1-score of 0.703 in Table 31) when compared to the hash-based equivalent (F1-score of 0.993 in Table 22).

8. Limitations

This section describes the limitations of the proposed approach within four key areas that represent the components of the multistage log analysis architecture. We will identify, explain, and evaluate the impact of each limitation and suggest areas for future research to address them.

8.1. Learning architecture

In our research, we have developed a multistage log analysis architecture that utilises Support Vector Machines (SVM) and Stochastic Dual Coordinate Ascent (SDCA) for binary and multiclass classification of network traffic logs. However, we are considering exploring other machine learning algorithms to improve the classification accuracy for detecting malicious activity in the logs. Some of the architectures we plan to investigate for future analysis are multilayer perceptron, random forest, convolutional neural network (CNN), feed-forward neural network and transformer networks. These algorithms have been chosen based on their successful use in transforming fuzzy hashes into suitable inputs for neural networks, traditional sequential models, conventional machine learning networks and attention-based networks for countering malware detection evasion techniques (Lazo, 2021), nation-state Advanced Persistent Threat (APT) malware attribution (Kida and Olukoya, 2023), Android malware detection and classification (Rodriguez-Bazan et al., 2023b,a; Lingrong et al., 2023), Javascript malware detection (Peiser et al., 2020), and file fragment detection (Uhlig et al., 2023).

8.2. Use cases

Our approach focuses on the specific use case of finding malicious activity in logs by analysing, identifying and classifying network traffic and web API system logs. This aligns with the state-of-the-art approaches of using log datasets for AI-driven analytics use cases such as root cause analysis, predictive maintenance, anomaly detection etc. Zhu et al. (2023), Landauer et al. (2024). Our approach is designed to be generalisable across different domains, allowing the training of classifiers to be applied to various log datasets. Most logs are in plain text, with some in JSON, XML, or CSV formats, all of which our proposed architecture can handle. This makes our architecture adaptable to other domains and log data types not specifically addressed in the study. To further improve our multistage log analysis solution, our future work will involve evaluating our hashing algorithm on other log datasets for different use cases. Evaluating the proposed similarity hashing in other use case scenarios and distinct, independent tasks would help in understanding the uses and misuses of similarity hashing functions in log analysis, identification, and classification (Botacin et al., 2021; Pagani et al., 2018).

8.3. Deployment and operation

In Section 7.1, we assessed the proposed fuzzy hashing algorithm for use with logs. We demonstrated the tool's application, presented a use case, and conducted performance benchmarks using relevant metrics. Our future work will involve testing the algorithm in a real-world deployment in collaboration with an industrial partner to validate its performance in an operational environment. Integrating the multistage log analysis as middleware into an existing logging system in an operational environment will create a feedback loop with the artefact processing, digest generation, and digest comparison phases of the hashing algorithm and the system design of the machine learning component. This will help to bridge the gap between deployment and operation, revealing practical limitations and open challenges that will foster further research (Arp et al., 2022; Apruzzese et al., 2023). Integrating closed-world settings with limited diversity with real-world settings for measuring accurate practical impact assessment is worthy of exploration to bridge the gap between similarity hashing for security research and practice (Li et al., 2015; Botacin et al., 2021; Pagani et al., 2018). Engaging with the industry to implement the algorithm in pilot projects would provide valuable insights into their real-world performance and practicality.

8.4. Parameter and refinements

In generating a digest, we took specific preprocessing steps and refinements to handle anomalies and irregularities in the log data, as described in Section 6.2. One of these steps involved removing items such as IDs and timestamps, as we demonstrated that these fields are irrelevant for monitoring suspicious activity in log data. We found that including these items would only add noise to the hash computation and reduce accuracy as demonstrated in Section 7.4. However, identifying the best parameters and refinements for various use cases, domains, and tasks requires additional experiments. This will help understand which parameters or settings within the algorithm are highly sensitive to changes, and how these changes affect the outcomes. Additionally, these experiments will lead to further research into developing a systematic noise filtering process that improves the quality of the logs before the hash computation, thereby improving robustness (El-Masri et al., 2020; Suriadi et al., 2017; Conforti et al., 2015, 2017; Zhang et al., 2019). In the future, research could explore using weights for parameters to indicate their importance, as not all fields, parameters, or attributes may have the same significance in different use-case scenarios (Onieva et al., 2024).

In the design of SSDHash, we use the fast non-cryptographic hashing function Spooky64 (Jenkins, 2012) as part of the token generation step. This function includes a seed parameter, which we have set to 0 for consistency between iterations. The seed parameter will only impact the results if the compared data was generated with a different seed. Our future work will explore other potential digest algorithms such as Piecewise hashing (FNV hash), Pearson Hash, SHA256, MD5, SHA-1, and bloom filter, which existing fuzzy hashing algorithms use. Additionally, we will look into other digest comparison algorithms like edit distance and classical bloom filter analysis to determine optimal design parameters.

9. Conclusion

This study demonstrates that a lightweight framework that uses fuzzy hashes as input for machine learning classifiers could credibly be used to identify common behaviours in logs typical of cyber attacks. The proposed approach demonstrated its effectiveness as an alternative technique for identifying malicious activity regardless of origin. It was evaluated using a public dataset of known malicious activity in IoT devices. For this novel implementation, the scope of behaviours was limited to three of the most common in the networking domain. However, future research would be required to identify its effectiveness when applied to other domains. Extending this approach to cover a wider range of behaviours and assessing its robustness requires further investigation.

Network logs are typically small and have a standard format. Our implementation allowed for additional functionality, such as custom parsing of Unix timestamps, with little performance impact. This effect on larger data structures would need further investigation, with additional work required to deduce the impact of adversarial manipulation of the data structures in domains where logs are not standardised. These domains, such as event logs, can contain developer-written messages that vary widely depending on the organisation.

We recognise that no silver bullet exists to detect malicious activity within a network. Instead, we advocate for a layered security architecture to analysis, wherein various techniques and tools are employed to mitigate inherent uncertainties. This approach provides multiple tools that present useful information for an analyst to make informed decisions. However, the responsibility of verifying leads still lies on the analyst, with this approach aiming to improve their operating efficiency and reduce the time to detection.

CRediT authorship contribution statement

Rory Flynn: Writing – original draft, Visualization, Software, Methodology, Investigation, Data curation. **Oluwafemi Olukoya:** Writing – review & editing, Validation, Supervision, Resources, Project administration, Methodology, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

References

Abdalgawad, N., Sajun, A., Kaddoura, Y., Zualkernan, I.A., Aloul, F., 2021. Generative deep learning to detect cyberattacks for the IoT-23 dataset. IEEE Access 10, 6430–6441. http://dx.doi.org/10.1109/ACCESS.2021.3140015.

Anand, A., 2022. JSON logs | best practices, benefits, and examples. https://signoz.io/blog/json-logs. (date Accessed: 25 Nov 2024).

Apruzzese, G., Laskov, P., Montes de Oca, E., Mallouli, W., Brdalo Rapa, L., Grammatopoulos, A.V., Di Franco, F., 2023. The role of machine learning in cybersecurity. Digit. Threats: Res. Pract. 4 (1), 1–38. http://dx.doi.org/10.1145/3545574.

Arasteh, A.R., Debbabi, M., Sakha, A., Saleh, M., 2007. Analyzing multiple logs for forensic evidence. Digit. Investig. 4, 82–91. http://dx.doi.org/10.1016/j.diin.2007.06.013.

Arp, D., Quiring, E., Pendlebury, F., Warnecke, A., Pierazzi, F., Wressnegger, C., Cavallaro, L., Rieck, K., 2022. Dos and don'ts of machine learning in computer security. In: 31st USENIX Security Symposium. USENIX Security 22, pp. 3971–3988.

Bash, K., 2024. Microsoft graph activity logs is now generally available. https://techcommunity.microsoft.com/t5/microsoft-entra-blog/microsoft-graph-activity-logs-is-now-generally-available/ba-p/4094535. (date Accessed: 04 Sep 2024).

Batoun, M.A., Sayagh, M., Aghili, R., Ouni, A., Li, H., 2024. A literature review and existing challenges on software logging practices: From the creation to the analysis of software logs. Empir. Softw. Eng. 29 (4), 103. http://dx.doi.org/10.1007/s10664-024-10452-w.

Berlin, K., Slater, D., Saxe, J., 2015. Malicious behavior detection using windows audit logs. In: Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security. pp. 35–44. http://dx.doi.org/10.1145/2808769.2808773.

Bhatt, S., Manadhata, P.K., Zomlot, L., 2014. The operational role of security information and event management systems. IEEE Secur. Priv. 12, 35–41. http://dx.doi.org/10.1109/MSP.2014.103.

Bookstein, A., Kulyukin, V.A., Raita, T., 2002. Generalized hamming distance. Inf. Retr. 5, 353–375. http://dx.doi.org/10.1023/A:1020499411651.

Botacin, M., Moia, V.H.G., Ceschin, F., Henriques, M.A.A., Grégio, A., 2021. Understanding uses and misuses of similarity hashing functions for malware detection and family clustering in actual scenarios. Forensic Sci. Int.: Digit. Investig. 38, 301220. http://dx.doi.org/10.1016/j.fsidi.2021.301220.

Breitinger, F., Baggili, I., 2014. File detection on network traffic using approximate matching. J. Digit. Forensics Secur. Law (JDFSL) 9 (2), 23–36. http://dx.doi.org/10.15394/jdfsl.2014.1168.

Breitinger, F., Baier, H., Beckingham, J., 2012. Security and implementation analysis of the similarity digest sdhash. In: 1st International Baltic Conference on Network Security & Forensics. NeSeFo, pp. 1–16.

Breitinger, F., Guttman, B., McCarrin, M., Roussev, V., White, D., 2014. Approximate matching: Definition and terminology. In: NIST Special Publication 800-168. National Institute of Standards and Technology, http://dx.doi.org/10.6028/NIST.SP.800-168.

Cao, Q., Qiao, Y., Lyu, Z., 2017. Machine learning to detect anomalies in web log analysis. In: 2017 3rd IEEE International Conference on Computer and Communications. ICCC, IEEE, pp. 519–523. http://dx.doi.org/10.1109/CompComm.2017.8322600.

Chang, D., Ghosh, M., Sanadhya, S.K., Singh, M., White, D.R., 2019. FbHash: A new similarity hashing scheme for digital forensics. Digit. Investig. 29, S113–S123. http://dx.doi.org/10.1016/j.diin.2019.04.006.

Chen, Z., Liu, J., Gu, W., Su, Y., Lyu, M.R., 2021. Experience report: Deep learning-based system log analysis for anomaly detection. http://dx.doi.org/10.48550/arXiv.2107.05908, arXiv preprint arXiv:2107.05908.

Chen, R., Zhang, S., Li, D., Zhang, Y., Guo, F., Meng, W., Pei, D., Zhang, Y., Chen, X., Liu, Y., 2020. Logtransfer: Cross-system log anomaly detection for software systems with transfer learning. In: 2020 IEEE 31st International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 37–47. http://dx.doi.org/10.1109/ISSRE5003.2020.00013.

Chu, D., Lu, R., Li, J., Yu, X., Zhang, C., Tao, Q., 2018. Optimizing top- k multiclass SVM via semismooth Newton algorithm. IEEE Trans. Neural Netw. Learn. Syst. 29 (12), 6264–6275. http://dx.doi.org/10.1109/TNNLS.2018.2826039.

Cinque, M., Cotroneo, D., Pecchia, A., 2013. Event logs for the analysis of software failures: A rule-based approach. IEEE Trans. Softw. Eng. 39 (6), 806–821. http://dx.doi.org/10.1109/TSE.2012.67.

Conforti, R., La Rosa, M., ter Hofstede, A., 2015. Noise filtering of process execution logs based on outliers detection. https://eprints.qut.edu.au/82901. (date Accessed: 22 Aug 2024).

Conforti, R., La Rosa, M., ter Hofstede, A., 2017. Filtering out infrequent behavior from business process event logs. IEEE Trans. Knowl. Data Eng. 29 (2), 300–314. http://dx.doi.org/10.1109/TKDE.2016.2614680.

Coralogix, 2022. JSON logging: What, why, how, & tips. https://coralogix.com/blog/json-logging-why-how-what-tips. (date Accessed: 25 Nov 2024).

Cortes, C., Vapnik, V., 1995. Support-vector networks. Mach. Learn. 20, 273–297. http://dx.doi.org/10.1007/BF00994018.

Docker Inc, 2024. Configure the delivery mode of log messages from container to log driver. https://docs.docker.com/engine/logging/configure. (date Accessed: 25 Nov 2024).

Du, M., Li, F., Zheng, G., Srikumar, V., 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1285–1298. http://dx.doi.org/10.1145/3133956.3134015.

Dutta, V., Choraś, M., Pawlicki, M., Kozik, R., 2020. A deep learning ensemble for network anomaly and cyber-attack detection. Sensors 20 (16), 4583. http://dx.doi.org/10.3390/s20164583.

El-Masri, D., Petrillo, F., Guéhéneuc, Y.-G., Hamou-Lhadj, A., Bouziane, A., 2020. A systematic literature review on automated log abstraction techniques. Inf. Softw. Technol. 122, 106276. http://dx.doi.org/10.1016/j.infsof.2020.106276.

Fan, R.-E., Chang, K.-W., Hsieh, C.-J., Wang, X.-R., Lin, C.-J., 2008. LIBLINEAR: A library for large linear classification. J. Mach. Learn. Res. 9, 1871–1874. http://dx.doi.org/10.5555/1390681.1442794.

Fleming, M., Olukoya, O., 2024. A temporal analysis and evaluation of fuzzy hashing algorithms for android malware analysis. Forensic Sci. Int. Digit. Investig. 49, 301770. http://dx.doi.org/10.1016/j.fsidi.2024.301770.

Fu, Y., Liang, K., Xu, J., 2023. MLog: Mogrifier LSTM-based log anomaly detection approach using semantic representation. IEEE Trans. Serv. Comput. 16 (5), 3537–3549. http://dx.doi.org/10.1109/TSC.2023.3289488.

Fu, Q., Lou, J.-G., Wang, Y., Li, J., 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In: 2009 9th IEEE International Conference on Data Mining. IEEE, pp. 149–158. http://dx.doi.org/10.1109/ICDM.2009.60.

Fu, Q., Zhu, J., Hu, W., Lou, J.-G., Ding, R., Lin, Q., Zhang, D., Xie, T., 2014. Where do developers log? An empirical study on logging practices in industry. In: Companion Proceedings of the 36th International Conference on Software Engineering. pp. 24–33. http://dx.doi.org/10.1145/2591062.2591175.

Fuchs, G., Nagy, R., Buttyán, L., 2023. A practical attack on the TLSH similarity digest scheme. In: Proceedings of the 18th International Conference on Availability, Reliability and Security. pp. 1–10. http://dx.doi.org/10.1145/3600160.3600173.

Garcia, S., Parmisano, A., Erquiaga, M.J., 2020. IoT-23: A labeled dataset with malicious and benign IoT network traffic (Version 1.0.0) [Data set]. Zenodo, http://dx.doi.org/10.5281/zenodo.4743746, (date Accessed: 09 March 2024).

Gatev, R., Gatev, R., 2021. Observability: Logs, metrics, and traces. Introd. Distrib. Appl. Runtime (Dapr): Simplif. Microservices Appl. Dev. Through Proven Reusable Patterns Pract. 233–252. http://dx.doi.org/10.1007/978-1-4842-6998-5_12.

Ge, S., Xu, M., Qiao, T., Zheng, N., 2021. A novel file carving algorithm for docker container logs recorded by json-file logging driver. Forensic Sci. Int. Digit. Investig. 39, 301272. http://dx.doi.org/10.1016/j.fsidi.2021.301272.

Göbel, T., Uhlig, F., Baier, H., 2021. Evaluation of network traffic analysis using approximate matching algorithms. In: Advances in Digital Forensics XVII: 17th IFIP WG 11.9 International Conference, Virtual Event, February 1–2, 2021, Revised Selected Papers 17. Springer, pp. 89–108. http://dx.doi.org/10.1007/978-3-030-88381-2_5.

Göbel, T., Uhlig, F., Baier, H., Breitinger, F., 2022. FRASHER – a framework for automated evaluation of similarity hashing. Forensic Sci. Int. Digit. Investig. 42, 301407. http://dx.doi.org/10.1016/j.fsidi.2022.301407.

Günther, C.W., van der Aalst, W.M., 2006. A generic import framework for process event logs: Industrial paper. In: International Conference on Business Process Management. Springer, pp. 81–92. http://dx.doi.org/10.1007/11837862_10.

Gupta, S., Gupta, A., 2019. Dealing with noise problem in machine learning data-sets: A systematic review. Procedia Comput. Sci. 161, 466–474. http://dx.doi.org/10.1016/j.procs.2019.11.146.

Hahn, K., 2021. All your hashes are belong to us: An overview of malware hashing algorithms. https://www.gdatasoftware.com/blog/2021/09/an-overview-of-malware-hashing-algorithms. (date Accessed: 27 Aug 2024).

Han, P., Li, H., Xue, G., Zhang, C., 2023. Distributed system anomaly detection using deep learning-based log analysis. Comput. Intell. 39, 433–455. http://dx.doi.org/10.1111/coin.12573.

Hancock, J.T., Khoshgoftaar, T.M., 2020. Survey on categorical data for neural networks. J. Big Data 7 (1), 28. http://dx.doi.org/10.1186/s40537-020-00305-w.

He, P., Chen, Z., He, S., Lyu, M.R., 2018. Characterizing the natural language descriptions in software logging statements. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 178–189. http://dx.doi.org/10.1145/3238147.3238193.

He, S., He, P., Chen, Z., Yang, T., Su, Y., Lyu, M.R., 2021. A survey on automated log analysis for reliability engineering. ACM Comput. Surv. 54 (6), 1–37. http://dx.doi.org/10.1145/3460345.

He, D., Yu, X., Zhu, S., Chan, S., Guizani, M., 2022a. Fuzzy hashing on firmwares images: A comparative analysis. IEEE Internet Comput. 27 (2), 45–50. http://dx.doi.org/10.1109/MIC.2022.3225811.

He, S., Zhang, X., He, P., Xu, Y., Li, L., Kang, Y., Ma, M., Wei, Y., Dang, Y., Rajmohan, S., et al., 2022b. An empirical study of log analysis at microsoft. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1465–1476. http://dx.doi.org/10.1145/3540250.3558963.

He, P., Zhu, J., He, S., Li, J., Lyu, M.R., 2016a. An evaluation study on log parsing and its use in log mining. In: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. DSN, pp. 654–661. http://dx.doi.org/10.1109/DSN.2016.66.

He, S., Zhu, J., He, P., Lyu, M.R., 2016b. Experience report: System log analysis for anomaly detection. In: 2016 IEEE 27th International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 207–218. http://dx.doi.org/10.1109/ISSRE.2016.21.

Ianni, M., Masciari, E., 2023. Scout: Security by computing outliers on activity logs. Comput. Secur. 132, 103355. http://dx.doi.org/10.1016/j.cose.2023.103355.

IBM Corporation, 2024a. CSV log files. https://www.ibm.com/docs/en/ftmfm/4.0.6?topic=logging-csv-log-files. (date Accessed: 25 Nov 2024).

IBM Corporation, 2024b. Format of messages in logs. https://www.ibm.com/docs/en/sva/10.0.8?topic=logging-format-messages-in-logs. (date Accessed: 25 Nov 2024).

Ibrahim, N.M., Al-Nemrat, A., Jahankhani, H., Bashroush, R., 2011. Sufficiency of windows event log as evidence in digital forensics. In: International Conference on E-Democracy. Springer, pp. 253–262. http://dx.doi.org/10.1007/978-3-642-33448-1_34.

Jakobs, C., Lambertz, M., Hilgert, J.-N., 2022. Ssdeeper: Evaluating and improving ssdeep. Forensic Sci. Int. Digit. Investig. 42, 301402. http://dx.doi.org/10.1016/j.fsidi.2022.301402.

Jenkins, B., 2012. SpookyHash: a 128-bit noncryptographic hash. https://burtleburtle.net/bob/hash/spooky.html. (date Accessed: 16 Oct 2024).

Jiang, Z., Liu, J., Huang, J., Li, Y., Huo, Y., Gu, J., Chen, Z., Zhu, J., Lyu, M.R., 2024. A large-scale evaluation for log parsing techniques: How far are we? In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. In: ISSTA 2024, Association for Computing Machinery, pp. 223–234. http://dx.doi.org/10.1145/3650212.3652123.

Joulin, A., Grave, E., Bojanowski, P., Douze, M., Jégou, H., Mikolov, T., 2016. Fasttext.zip: Compressing text classification models. http://dx.doi.org/10.48550/arXiv.1612.03651, arXiv preprint arXiv:1612.03651.

Kalyanathaya, K.P., Akila, D., Suseendren, G., 2019. A fuzzy approach to approximate string matching for text retrieval in NLP. J. Comput. Inf. Syst. 15 (3), 26–32.

Kang, J., Jang, S., Li, S., Jeong, Y.S., Sung, Y., 2019. Long short-term memory-based malware classification method for information security. Comput. Electr. Eng. 77, 366–375. http://dx.doi.org/10.1016/j.compeleceng.2019.06.014.

Ke, G., He, D., Liu, T.-Y., 2021. Rethinking positional encoding in language pre-training. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021. OpenReview.net, pp. 1–14. http://dx.doi.org/10.48550/arXiv.2006.15595.

Khan, S., Parkinson, S., Murphy, C., 2023. Context-based irregular activity detection in event logs for forensic investigations: An itemset mining approach. Expert Syst. Appl. 233, 120991. http://dx.doi.org/10.1016/j.eswa.2023.120991.

Kida, M., Olukoya, O., 2023. Nation-state threat actor attribution using fuzzy hashing. IEEE Access 11, 1148–1165. http://dx.doi.org/10.1109/ACCESS.2022.3233403.

Kim(Sumo Logic), 2024. Parse operators. https://help.sumologic.com/docs/search/search-query-language/parse-operators. (date Accessed: 25 Nov 2024).

Kornblum, J., 2006. Identifying almost identical files using context triggered piecewise hashing. Digit. Investig. 3, 91–97. http://dx.doi.org/10.1016/j.diin.2006.06.015.

Kumar, V., Sangwan, O.P., 2012. Signature based intrusion detection system using SNORT. Int. J. Comput. Appl. & Information Technol. 1 (3), 35–41.

Lambert, J., 2024. Defending with the graph of graphs. In: Presented At MSRC BlueHat India, vol. 1, Microsoft, pp. 1–31, https://github.com/JohnLaTwC/Shared/blob/master/Presentations/2024-05-JohnLa-BluehatIDC.Final.1.pptx. (date Accessed: 29 May 2024).

Landauer, M., Onder, S., Skopik, F., Wurzenberger, M., 2023. Deep learning for anomaly detection in log data: A survey. Mach. Learn. Appl. 12, 100470. http://dx.doi.org/10.1016/j.mlwa.2023.100470.

Landauer, M., Skopik, F., Wurzenberger, M., 2024. A critical review of common log data sets used for evaluation of sequence-based anomaly detection techniques. Proc. the ACM Softw. Eng. 1 (FSE), 1354–1375. http://dx.doi.org/10.1145/3660768.

Lapin, M., Hein, M., Schiele, B., 2015. Top-k multiclass SVM. In: Cortes, C., Lawrence, N., Lee, D., Sugiyama, M., Garnett, R. (Eds.), In: Advances in Neural Information Processing Systems, vol. 28, Curran Associates, Inc, pp. 325–333. http://dx.doi.org/10.5555/2969239.2969276.

Lapin, M., Hein, M., Schiele, B., 2018. Analysis and optimization of loss functions for multiclass, top-k, and multilabel classification. IEEE Trans. Pattern Anal. Mach. Intell. 40 (7), 1533–1554. http://dx.doi.org/10.1109/TPAMI.2017.2751607.

Lazo, E.G., 2021. Combing through the fuzz: Using fuzzy hashing and deep learning to counter malware detection evasion technique. https://www.microsoft.com/en-us/security/blog/2021/07/27/combing-through-the-fuzz-using-fuzzy-hashing-and-deep-learning-to-counter-malware-detection-evasion-techniques. (date Accessed: 09 May 2024).

Le, V.H., Zhang, H., 2022. Log-based anomaly detection with deep learning: How far are we? In: Proceedings of the 44th International Conference on Software Engineering. pp. 1356–1367. http://dx.doi.org/10.1145/3510003.3510155.

Lee, S., Jung, W., Kim, S., Kim, E.T., 2019a. Android malware similarity clustering using method based opcode sequence and Jaccard index. In: 2019 International Conference on Information and Communication Technology Convergence. ICTC, IEEE, pp. 178–183. http://dx.doi.org/10.1109/ICTC46691.2019.8939894.

Lee, S., Jung, W., Kim, S., Lee, J., Kim, J.-S., 2019b. Dexofuzzy: Android malware similarity clustering method using opcode sequence. Virus Bull..

Li, X., Chen, P., Jing, L., He, Z., Yu, G., 2020. Swisslog: Robust and unified deep learning based log anomaly detection for diverse faults. In: 2020 IEEE 31st International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 92–103. http://dx.doi.org/10.1109/ISSRE5003.2020.00018.

Li, Y., Sundaramurthy, S.C., Bardas, A.G., Ou, X., Caragea, D., Hu, X., Jang, J., 2015. Experimental study of fuzzy hashing in malware clustering analysis. In: 8th Workshop on Cyber Security Experimentation and Test. CSET 15, USENIX Association, p. 8. http://dx.doi.org/10.5555/2831120.2831128.

Lillis, D., Becker, B., O'Sullivan, T., Scanlon, M., 2016. Current challenges and future research areas for digital forensic investigation. http://dx.doi.org/10.48550/arXiv.1604.03850, arXiv preprint arXiv:1604.03850.

Lingrong, H., Shibin, Z., Jiazhong, L., Yuanyuan, H., Zhi, Q., 2023. An enhanced fusion model for android malware detection leveraging multi-code fragment features and fuzzy hashing. In: 2023 20th International Computer Conference on Wavelet Active Media Technology and Information Processing. ICCWAMTIP, pp. 1–5. http://dx.doi.org/10.1109/ICCWAMTIP60502.2023.10387082.

Luhn, H.P., 1957. A statistical approach to mechanized encoding and searching of literary information. IBM J. Res. Dev. 1 (4), 309–317. http://dx.doi.org/10.1147/rd.14.0309.

Martín-Pérez, M., Rodríguez, R.J., Breitinger, F., 2021. Bringing order to approximate matching: Classification and attacks on similarity digest algorithms. Forensic Sci. Int. Digit. Investig. 36, 301120. http://dx.doi.org/10.1016/j.fsidi.2021.301120.

Meng, W., Liu, Y., Zhang, S., Zaiter, F., Zhang, Y., Huang, Y., Yu, Z., Zhang, Y., Song, L., Zhang, M., et al., 2021. Logclass: Anomalous log identification and classification with partial labels. IEEE Trans. Netw. Serv. Manag. 18 (2), 1870–1884. http://dx.doi.org/10.1109/TNSM.2021.3055425.

Mercês, F., Costoya, J., 2020. Telfhash: An Algorithm That Finds Similar Malicious ELF Files Used in Linux IoT Malware. Technical Report, Trend Micro.
Microsoft, 2021. [MS-WMLOG]: Windows media log data structure. https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-wmlog/42c620eb-0d77-4350-b070-bcd1e182fe84. (date Accessed: 25 Nov 2024).
Mikolov, T., Chen, K., Corrado, G., Dean, J., 2013a. Efficient estimation of word representations in vector space. http://dx.doi.org/10.48550/arXiv.1301.3781, arXiv preprint arXiv:1301.3781.
Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J., 2013b. Distributed representations of words and phrases and their compositionality. Adv. Neural Inf. Process. Syst. 26, http://dx.doi.org/10.5555/2999792.2999959.
Moh, M., Pininti, S., Doddapaneni, S., Moh, T.-S., 2016. Detecting web attacks using multi-stage log analysis. In: 2016 IEEE 6th International Conference on Advanced Computing. IACC, IEEE, pp. 733–738. http://dx.doi.org/10.1109/IACC.2016.141.
Naas, M., Fesl, J., 2023. A novel dataset for encrypted virtual private network traffic analysis. Data Brief 47, 108945. http://dx.doi.org/10.1016/j.dib.2023.108945.
Namanya, A.P., Awan, I.U., Disso, J.P., Younas, M., 2020. Similarity hash based scoring of portable executable files for efficient malware detection in IoT. Future Gener. Comput. Syst. 110, 824–832. http://dx.doi.org/10.1016/j.future.2019.04.044.
National Institute of Standards and Technology (NIST), Dang, Q., 2015. Secure Hash Standard. Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD, http://dx.doi.org/10.6028/NIST.FIPS.180-4.
Nettleton, D.F., Orriols-Puig, A., Fornells, A., 2010. A study of the effect of different types of noise on the precision of supervised learning techniques. Artif. Intell. Rev. 33, 275–306. http://dx.doi.org/10.1007/s10462-010-9156-z.
Nogales, R.E., Benalcázar, M.E., 2023. Analysis and evaluation of feature selection and feature extraction methods. Int. J. Comput. Intell. Syst. 16 (1), 153. http://dx.doi.org/10.1007/s44196-023-00319-1.
Oliner, A., Ganapathi, A., Xu, W., 2012. Advances and challenges in log analysis. Commun. ACM 55, 55–61. http://dx.doi.org/10.1145/2076450.2076466.
Oliver, J., Cheng, C., Chen, Y., 2013. TLSH–a locality sensitive hash. In: 2013 Fourth Cybercrime and Trustworthy Computing Workshop. IEEE, pp. 7–13. http://dx.doi.org/10.1109/CTC.2013.9.
Ombongi, F.M., Guo, Y., RetYn, l., Norman, M., Buck, A., Graham, L., 2024. Access microsoft graph activity logs. https://learn.microsoft.com/en-us/graph/microsoft-graph-activity-logs-overview. (date Accessed: 04 Sep 2024).
Onieva, J.A., Jiménez, P.P., López, J., 2024. Malware similarity and a new fuzzy hash: Compound code block hash (CCBHash). Comput. Secur. 142, 103856. http://dx.doi.org/10.1016/j.cose.2024.103856.
Oros, B.-I., Bâcu, V.I., 2022. AntiMSA: A framework for detecting malicious software agents in online multiplayer games. In: 2022 IEEE 18th International Conference on Intelligent Computer Communication and Processing. ICCP, IEEE, pp. 283–288. http://dx.doi.org/10.1109/ICCP56966.2022.10053949.
OWASP Top 10 team, 2021. A09:2021 – security logging and monitoring failures. https://owasp.org/Top10/A09_2021-Security_Logging_and_Monitoring_Failures. (date Accessed: 02 Sep 2024).
Pagani, F., Dell'Amico, M., Balzarotti, D., 2018. Beyond precision and recall: understanding uses (and misuses) of similarity hashes in binary analysis. In: Proceedings of the 8th ACM Conference on Data and Application Security and Privacy. pp. 354–365. http://dx.doi.org/10.1145/3176258.3176306.
Panther Labs, 2024. Custom logs. https://docs.panther.com/data-onboarding/custom-log-types. (date Accessed: 25 Nov 2024).
Pecchia, A., Cinque, M., Carrozza, G., Cotroneo, D., 2015. Industry practices and event logging: Assessment of a critical software development process. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 2, IEEE, pp. 169–178. http://dx.doi.org/10.1109/ICSE.2015.145.
Peiser, S.C., Friborg, L., Scandariato, R., 2020. Javascript malware detection using locality sensitive hashing. In: ICT Systems Security and Privacy Protection: 35th IFIP TC 11 International Conference, SEC 2020, Maribor, Slovenia, September 21–23, 2020, Proceedings 35. Springer, pp. 143–154. http://dx.doi.org/10.1007/978-3-030-58201-2_10.
Pennington, J., Socher, R., Manning, C.D., 2014. Glove: Global vectors for word representation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing. EMNLP, pp. 1532–1543. http://dx.doi.org/10.3115/v1/D14-1162.
Prikryl, M., 2024. XML logging. https://winscp.net/eng/docs/logging_xml. (date Accessed: 25 Nov 2024).
Raff, E., Nicholas, C., 2018. Lempel-Ziv Jaccard distance, an effective alternative to ssdeep and sdhash. Digit. Investig. 24, 34–49. http://dx.doi.org/10.1016/j.diin.2017.12.004.
Ring, M., Schlör, D., Wunderlich, S., Landes, D., Hotho, A., 2021. Malware detection on windows audit logs using LSTMs. Comput. Secur. 109, 102389. http://dx.doi.org/10.1016/j.cose.2021.102389.
Rodriguez-Bazan, H., Sidorov, G., Escamilla-Ambrosio, P.J., 2023a. Android malware classification based on fuzzy hashing visualization. Mach. Learn. Knowl. Extr. 5 (4), 1826–1847. http://dx.doi.org/10.3390/make5040088.
Rodriguez-Bazan, H., Sidorov, G., Escamilla-Ambrosio, P.J., 2023b. Android ransomware analysis using convolutional neural network and fuzzy hashing features. IEEE Access 11, 121724–121738. http://dx.doi.org/10.1109/ACCESS.2023.3328314.
Roussev, V., 2010. Data fingerprinting with similarity digests. In: Advances in Digital Forensics VI: 6th IFIP WG 11.9 International Conference on Digital Forensics, Hong Kong, China, January 4-6, 2010, Revised Selected Papers 6, vol. 337 AICT, Springer, pp. 207–226. http://dx.doi.org/10.1007/978-3-642-15506-2_15.
Rydning, D.R.-J.G.-J., Reinsel, J., Gantz, J., 2018. The digitization of the world from edge to core. Framingham: Int. Data Corp. 16, 1–28.
Sahu, A.K., Sharma, S., Tanveer, M., Raja, R., 2021. Internet of Things attack detection using hybrid deep learning model. Comput. Commun. 176, 146–154. http://dx.doi.org/10.1016/j.comcom.2021.05.024.
Shalev-Shwartz, S., Zhang, T., 2014. Accelerated proximal stochastic dual coordinate ascent for regularized loss minimization. In: International Conference on Machine Learning. PMLR, pp. 64–72. http://dx.doi.org/10.1007/s10107-014-0839-0.
Shang, W., 2012. Bridging the divide between software developers and operators using logs. In: 2012 34th International Conference on Software Engineering. ICSE, IEEE, pp. 1583–1586. http://dx.doi.org/10.1109/ICSE.2012.6227031.
Sharif, A., 2022. 6 common log file formats. https://www.crowdstrike.com/en-us/cybersecurity-101/next-gen-siem/log-file-formats. (date Accessed: 25 Nov 2024).
Shiel, I., O'Shaughnessy, S., 2019. Improving file-level fuzzy hashes for malware variant classification. Digit. Investig. 28, S88–S94. http://dx.doi.org/10.1016/j.diin.2019.01.018.
Shiva, P., Dreymann, C., Azed, A., 2024. Hunting with microsoft graph activity logs. https://techcommunity.microsoft.com/t5/microsoft-security-experts-blog/hunting-with-microsoft-graph-activity-logs/ba-p/4234632. (date Accessed: 04 Sep 2024).
Signalblur, 2024. Operationalizing TLSH fuzzy hashing. https://www.magonia.io/blog/operationalizing-tlsh-fuzzy-hashing-for-detection. (date Accessed: 31 Aug 2024).
Sipos, R., Fradkin, D., Moerchen, F., Wang, Z., 2014. Log-based predictive maintenance. In: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 1867–1876. http://dx.doi.org/10.1145/2623330.2623340.
SolarWinds Loggly, 2024. Use cases - JSON logging best practices. https://www.loggly.com/use-cases/json-logging-best-practices. (date Accessed: 25 Nov 2024).
Špaček, S., Velan, P., Čeleda, P., Tovarňák, D., 2022. Encrypted web traffic dataset: Event logs and packet traces. Data Brief 42, 108188. http://dx.doi.org/10.1016/j.dib.2022.108188.
Sparck Jones, K., 1972. A statistical interpretation of term specificity and its application in retrieval. J. Doc. 28 (1), 11–21. http://dx.doi.org/10.1108/eb026526.
Sridharan, C., 2018. Distributed Systems Observability: A Guide to Building Robust Systems. O'Reilly Media.
Studiawan, H., Sohel, F., Payne, C., 2019. A survey on forensic investigation of operating system logs. Digit. Investig. 29, 1–20. http://dx.doi.org/10.1016/j.diin.2019.02.005.
Studiawan, H., Sohel, F., Payne, C., 2021. Anomaly detection in operating system logs with deep learning-based sentiment analysis. IEEE Trans. Dependable Secur. Comput. 18 (5), 2136–2148. http://dx.doi.org/10.1109/TDSC.2020.3037903.
Suriadi, S., Andrews, R., ter Hofstede, A.H., Wynn, M.T., 2017. Event log imperfection patterns for process mining: Towards a systematic approach to cleaning event logs. Inf. Syst. 64, 132–150. http://dx.doi.org/10.1016/j.is.2016.07.011.
Uhlig, F., Struppek, L., Hintersdorf, D., Göbel, T., Baier, H., Kersting, K., 2023. Combining AI and AM – improving approximate matching through transformer networks. Forensic Sci. Int. Digit. Investig. 45, 301570. http://dx.doi.org/10.1016/j.fsidi.2023.301570.
Ullah, I., Mahmoud, Q.H., 2021. Design and development of a deep learning-based model for anomaly detection in IoT networks. IEEE Access 9, 103906–103926. http://dx.doi.org/10.1109/ACCESS.2021.3094024.
Vaarandi, R., 2003. A data clustering algorithm for mining patterns from event logs. In: Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)(IEEE Cat. No. 03EX764). IEEE, pp. 119–126. http://dx.doi.org/10.1109/IPOM.2003.1251233.
Van Dijk, I., 2023. Playbook of the week: Uncovering unknown malware using SSDeep. https://www.paloaltonetworks.com/blog/security-operations/playbook-of-the-week-uncovering-unknown-malware-using-ssdeep. (date Accessed: 29 Aug 2024).
Versteeg, S., Moodley, E., 2023. JSONHash - fuzzy hashing logs to find malicious activity. https://techcommunity.microsoft.com/t5/microsoft-security-experts-blog/fuzzy-hashing-logs-to-find-malicious-activity/ba-p/3786669. (date Accessed: 09 May 2024).
Wang, L., Du, Y., Qi, L., 2019. Efficient deviation detection between a process model and event logs. IEEE/CAA J. Autom. Sin. 6 (6), 1352–1364. http://dx.doi.org/10.1109/JAS.2019.1911750.
Wilson, J., 2023. Permhash - no curls necessary. https://www.mandiant.com/resources/blog/permhash-no-curls-necessary. (date Accessed: 09 May 2024).
xcitium, 2024. Log file formats. https://www.xcitium.com/log-file-formats. (date Accessed: 25 Nov 2024).
Xiong, H., Pandey, G., Steinbach, M., Kumar, V., 2006. Enhancing data analysis with noise removal. IEEE Trans. Knowl. Data Eng. 18 (3), 304–319. http://dx.doi.org/10.1109/TKDE.2006.46.
Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M.I., 2009. Detecting large-scale system problems by mining console logs. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. pp. 117–132. http://dx.doi.org/10.1145/1629575.1629587.

Zhai, A., Xu, F., Cao, Z., Pan, H., Li, Z., Xiong, G., 2017. Real time network file similarity detection based on approximate matching. In: 2017 IEEE Security and Privacy Workshops. SPW, IEEE, pp. 223–228. http://dx.doi.org/10.1109/SPW.2017.16.

Zhang, X., Xu, Y., Lin, Q., Qiao, B., Zhang, H., Dang, Y., Xie, C., Yang, X., Cheng, Q., Li, Z., et al., 2019. Robust log-based anomaly detection on unstable log data. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 807–817. http://dx.doi.org/10.1145/3338906.3338931.

Zhu, J., He, S., He, P., Liu, J., Lyu, M.R., 2023. Loghub: A large collection of system log datasets for AI-driven log analytics. In: 2023 IEEE 34th International Symposium on Software Reliability Engineering. ISSRE, pp. 355–366. http://dx.doi.org/10.1109/ISSRE59848.2023.00071.

Zhu, J., He, S., Liu, J., He, P., Xie, Q., Zheng, Z., Lyu, M.R., 2019. Tools and benchmarks for automated log parsing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice. ICSE-SEIP, IEEE, pp. 121–130. http://dx.doi.org/10.1109/ICSE-SEIP.2019.00021.

Zhu, X., Wu, X., 2004. Class noise vs. attribute noise: A quantitative study. Artif. Intell. Rev. 22, 177–210. http://dx.doi.org/10.1007/s10462-004-0751-8.

Rory Flynn completed the integrated MEng degree in software engineering with placement from the School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast. He is currently working as a freelance software engineer. His research interests include algorithms analysis and application, parallel and distributed computing, secure software development and network security.

Oluwafemi Olukoya received his PhD in Computing Science from the University of Glasgow, United Kingdom. Currently, he is a Lecturer (Assistant Professor) at the School of Electronics, Electrical Engineering and Computer Science at Queen's University Belfast, an EPSRC-NCSC Academic Centre of Excellence in Cyber Security Research (ACE-CSR) where he leads research in privacy, malware analysis, systems security, and cybercrime. His research interest is in the broad CyBOK knowledge areas of Attacks and Defences; Human, Organisational and Regulatory Aspects; Software and Platform Security.