Semantic-Aware Parsing for Security Logs

Julien Piet
UC Berkeley, USA

Vivian Fang
UC Berkeley, USA

Rishi Khare
UC Berkeley, USA

Vern Paxson
Corelight and UC Berkeley, USA

Raluca Ada Popa
UC Berkeley, USA

David Wagner
UC Berkeley, USA

Abstract

Security analysts struggle to quickly and efficiently query and correlate log data due to the heterogeneity and lack of structure in real-world logs. Existing AI-based parsers focus on learning syntactic log templates but lack the semantic interpretation needed for querying. Directly querying large language models on raw logs is impractical at scale and vulnerable to prompt injection attacks.

In this paper, we introduce Matryoshka[1], the first end-to-end system leveraging LLMs to automatically generate semantically-aware structured log parsers. Matryoshka combines a novel syntactic parser—employing precise regular expressions rather than wildcards—with a completely new semantic parsing layer that clusters variables and maps them into a queryable, contextually meaningful schema. This approach provides analysts with queryable and semantically rich data representations, facilitating rapid and precise log querying without the traditional burden of manual parser construction. Additionally, Matryoshka can map the newly created fields to recognized attributes within the Open Cybersecurity Schema Framework (OCSF), enabling interoperability.

We evaluate Matryoshka on a newly curated real-world log benchmark, introducing novel metrics to assess how consistently fields are named and mapped across logs. Matryoshka's syntactic parser outperforms prior works, and the semantic layer achieves an F_1 score of 0.95 on realistic security queries. Although mapping fields to the extensive OCSF taxonomy remains challenging, Matryoshka significantly reduces manual effort by automatically extracting and organizing valuable fields, moving us closer to fully automated, AI-driven log analytics.

Keywords

log parsing, security operations, large language models

1 Introduction

Security operations depend on analysts' ability to rapidly search and cross-correlate vast quantities of data to detect and respond to threats. Yet, they face a fundamental challenge: their job requires correlating events from multiple log sources, but these sources are heterogeneous, massive, and most often unstructured. Notably, they rely on system log data, reporting operational events across infrastructure—from network devices and operating systems to applications and security tools—each generating logs in different formats with varying levels of detail. To leverage these diverse data sources effectively, security teams invest considerable time developing parsers and query engines. Typically, logs are converted and normalized to a structured format, often using frameworks like the Open Cybersecurity Schema Framework (OCSF), so they can be ingested into security information and event managers.

Modern AI seems ideally suited for this problem. Large language models (LLMs) can understand system logs [6, 25, 31], retrieval-augmented generation (RAG) systems can query natural language data [27], and AI systems can even generate SQL queries for structured data analysis [14, 18, 45, 61, 62]. Yet, existing methods in the literature are not adopted. Real-world systems generate millions of log lines daily—far exceeding any language model's context window. Log data combines natural language, technical identifiers, and structured delimiters, making standard text embeddings inefficient and inaccurate. Even recent advances in querying unstructured data cannot keep pace with log generation rates; such approaches require embedding each new line individually and often resort to imprecise clustering for speed [8]. Also, directly analyzing logs with LLMs exposes analyses to prompt injection attacks, potentially evading detection or introducing misleading information. Recent reports show attackers already employ AI-aware strategies to evade detection [44].

Security teams are thus left with two suboptimal approaches: either (1) write parsers by hand, requiring substantial upfront resources but enabling efficient ingestion and search over logs, or (2) use AI search tools that require embedding each line independently, a slow approach that can produce inadequate embeddings and fail to scale to security workload volumes, thereby sacrificing both ingestion and search time.

We propose Matryoshka, the first end-to-end system leveraging LLMs to automatically generate *semantically-aware structured log parsers*, offering analysts fast ingestion and search with minimal human effort. Matryoshka proceeds in three steps. First, a *syntax parser* captures the syntax of each log line and identifies

	Manual Parsing	Matryoshka	LLM Querying
Setup Cost	High (manually written parsers)	Low (auto-generates parsers)	None (no parser creation needed)
Query Cost	Fast (queries on structured data)	Fast (queries on structured data)	Slow (doesn't scale)
Security	Good (no prompt injection risk)	Good (static parsing prevents vulnerabilities)	Vulnerable (prompt injection)

Figure 1: Comparison of log parsing approaches. Matryoshka combines the advantages of both manual parsing and LLM querying.

[1]Matryoshka dolls are Russian nesting dolls where each hollow wooden figure opens to reveal a smaller similar doll inside. We named our system after these because of its nested layers of parsing, revealing more and more information about log messages.

variables. Second, a *semantic naming* step consistently names and clusters these extracted variables, producing a coherent schema suitable for structured queries. Finally, Matryoshka maps the newly created fields to standardized attributes in the OCSF taxonomy. Real-time log processing relies exclusively on static regular-expression matching, preventing prompt injection attacks and enabling efficient ingestion. The schema produced by the syntax parser and semantic naming provides analysts with a queryable structured dataset that supports fast searches. The final mapping step further enables queries using standardized variable names.

Building a reliable pipeline to transform unstructured logs into structured data is challenging. While LLMs are capable of interpreting log content, their raw outputs often lack enough consistency for high-quality parsers. Instead, Matryoshka combines LLMs with additional techniques to ensure robust results at each step.

Our approach draws on insights from log template generation, in which variable fields are captured by wildcards (e.g., "`Mar 9 12:46:17 puma25 sshd[17839]`" becomes "`<*> <*> sshd[<*>]`"). Prior work has shown that LLMs can produce such templates [21, 22], but these templates alone are insufficient to ingest log lines. Wildcards alone lack the granularity to disambiguate neighboring variables: in the example, we cannot clearly assign the date (which also contains spaces) to the first wildcard and the hostname to the second. Wildcards also risk "overcapture": the same template would match "`INFO Mar 9 12:46:17 puma25 sshd[17839]`" even though it includes more fields. To address these limitations, Matryoshka generates precise regular expressions instead of wildcards and detects overcapture with a set of heuristics, ensuring accurate variable extraction.

Next, Matryoshka groups and names the extracted variables according to their semantic roles. Naively asking an LLM to name each variable can produce inconsistent names (e.g., "`source_ip`" in one log line vs. "`src_ip`" or"`source_ip_address`" in another log line). To ensure consistency, we first have the LLM generate a description of the variable's role, and embed this descriptive text. Because this embedding emphasizes the variable's role instead of its value, we can use it to identify closely related variables. When generating names for a new variable, we provide the LLM with few-shot examples of already named variables that have similar roles. We call this technique *description embedding*, and apply it widely throughout Matryoshka to cluster objects based on their role or meaning instead of their syntax.

Finally, Matryoshka maps each named variable to OCSF fields. This step resembles prior work on schema matching [43, 46, 64], but existing approaches do not handle our setting: the OCSF spans tens of thousands of attributes, the source data is unstructured, and we lack reference mappings or documentation. Instead, Matryoshka leverages its own created schemas and their description embeddings to filter candidate OCSF fields and identify suitable matches, without relying on external documentation.

To evaluate Matryoshka, we propose a new curated dataset and metrics for structured log parsing. We create ground-truth labels for five real-world log files obtained from online bugtracker reports. We define new metrics to evaluate each step of our pipeline. We also design metrics to measure end-to-end performance, to test how well the resulting structured data supports security tasks. Our results show that while mapping fields to OCSF attributes remains challenging, the first two steps in Matryoshka produce high-quality parsers for diverse log sources. Queries over created field names achieve an average precision of 0.96 and recall of 0.95, compared to 0.94 and 0.80 for a baseline simulating existing substring-based systems (see Table 2). Moreover, on the LogPub dataset [23, 68], we surpass LILAC [21], Brain [60] and Drain [17], three wildcard-based template generation frameworks, achieving a group similarity of 0.97 on average and a template similarity of 0.91, while the best other framework, LILAC, gets 0.90 and 0.81.

Our source code and benchmark are publicly available[2]. We proceed by examining the context of log analysis and related work in Section 2, followed by a detailed description of Matryoshka's architecture in Section 3. In Section 4, we present our evaluation methodology and results using new datasets and metrics. Finally, Section 5 addresses the strengths and limitations of our approach and considers future directions.

2 Background

2.1 System logs

System logs are text-based messages that record a program's state and report events. They can cover a wide range of operations, from kernel messages to application-specific activities. Any software can produce these logs, often as single-line entries. For instance, the following DHCP log line indicates the assignment of a new IP address:

```
Mar 31 20:47:49 nfs1 dhclient[1226]: bound to 10.70.37.157
-- renewal in 40434 seconds.
```

In this example, we consider 10.70.37.157 and `40434` to be variables. The template might be "`<*> <*> dhclient[<*>]: bound to <*> – renewal in <*> seconds.`".

Because these logs are usually unstructured, querying them in a consistent manner is difficult. Most logs follow no fixed standard: thousands of applications produce logs, each with its own distinct format. Even for the same application, the format can vary between different versions or implementations. For example, one DHCP client might log "`bound to 10.70.37.157`", while another logs "`IP=10.70.37.157`". Such inconsistencies can complicate automated parsing and prevent straightforward data analysis.

2.2 Security operations

Security operations teams focus on detecting, investigating, and mitigating threats. Their daily tasks rely on correlating large amounts of data from many sources to validate alerts and defend against intrusions. Structured information is crucial for this type of analysis. While some data arrives in neat, parsed formats, much is unstructured or line-based, including system logs.

Today, analysts often search logs using substring matching. For example, to find newly assigned IPs, one might look for the substring "`bound to`" in DHCP logs. This requires knowledge of the software's exact log message. Another DHCP client version might use "`IP=`" instead, breaking simple substring search. Some search conditions are not well-suited to substring matching; for instance, restricting a query to a specific date is challenging, since date fields often follow different formats and must be extracted separately.

[2] https://github.com/julien-piet/matryoshka

Beyond substring matching, analysts write parsing scripts or regular expressions to capture log lines of interest. This can focus on one type of event (e.g., newly leased IPs), or a comprehensive parse of all log lines. In both cases, the analyst checks sample lines, crafts new rules, tests them, and repeats until every variation is covered. Every identified variable (e.g., IP addresses, ports, or timestamps) must be named consistently, or else queries may fail mysteriously. We ourselves once had to write a parser to extract successful connections from a SSH log. After multiple iterations to capture missed lines, the parser utilized a 3616-character regular expression within a 160-line python program focused solely on extracting the usernames, IPs and timestamps of successful connections. This slow, manual process can consume days or even weeks, taking up analyst time that could instead be dedicated to investigating threats.

Matryoshka's goal is to save analysts time by automating the entire parser creation process, so any log can be queried by matching the value of specific fields. While prior work explores some building blocks that we build on, we are the first to provide this capability.

2.3 Related work

Most of the literature refers to *log parsing* as the task of generating templates that identify variables using wildcard placeholders. The intuition is that applications typically generate log messages with a printf-style (format string) API, and each template should hopefully correspond to a unique format string in the source code. We call this *template generation*, as we consider it only part of the parsing problem. Template generation has been studied for decades, and existing approaches can be divided into statistics-based and semantic-based approaches [22].

Statistics-based template generation. Earlier work on template generation applies statistical methods to find variables in log messages: they identified variables based on word length, frequency, and other statistical features. Frequency-based methods [7, 39, 52, 54] rely on occurrence frequencies or n-gram counts to build templates. Clustering-based techniques [11, 15, 38, 47, 50] group similar log lines to produce templates. Heuristic-based approaches [9, 11, 16, 17, 24, 37, 49, 56, 60] employ various heuristics or rule-based methods to detect the variable parts of each line.

Semantic-based template generation. Recent work has trended towards using semantic information in logs to improve the quality of generated templates. Earlier works cast template generation as a token classification task and trained neural networks [20, 29, 33] to mine semantics from log messages. More recently, researchers have shown that LLMs are effective at template generation [3, 26, 36, 53, 63, 65], especially when leveraging in-context learning [4]. DivLog [58] selects examples for developers to label; then when generating a log template for a target log line, it includes the most similar labeled examples in the LLM prompt. LILAC [21] builds on DivLog by improving the sampling method when choosing examples for a target log line, and adds a parsing cache to reduce the number of LLM queries. Other approaches focus on reducing the number of LLM queries [19, 57, 67] or fine-tune smaller LLMs for better performance [34, 35].

We are partly inspired by LogBatcher, which clusters log lines based on statistical features and generates templates for each cluster. When testing LogBatcher on our log dataset, their templates overcaptured and performed poorly, because their log clusters were too large. We solve these problems by using a LLM to generate a description of each line, computing a vector embedding of this description, and clustering these embeddings.

All of these past works focus on generating syntactic templates, but are not enough to convert unstructured logs to a structured form. They identify the variables associated with each template, but do not map them into a common schema. Also, our experiments suggest that past work tends to overcapture and conflate consecutive variables. We improve on past work on template generation, generating a regular expression for each variable, improving the quality of templates.

Schema matching. Databases researchers have studied schema mapping, where the goal is to map data in one schema into a second schema [43, 64]. Existing methods are effective in settings where both schemas are thoroughly documented, but they are insufficient in our setting, where schemas are auto-generated, do not come with documentation of the meaning of fields in the schema, and can contain tens of thousands of fields. Recent work explores using LLMs for schema mapping [32, 42, 46, 59, 66], but it too shares some of these limitations.

Log querying. LLMs are good at analyzing data [5, 10, 28], including unstructured logs [6, 25, 31]. Structured data can be queried using text-to-SQL tools [14, 18, 45, 61, 62], while unstructured data is usually queried using RAG [12, 27]. LLMs can be used to plan out complex queries over structured data [30], unstructured data [2], or both [8]. Unfortunately, these methods are not sufficient for our use case: they typically apply a LLM to each log line, which is too expensive; query expressivity is limited, because they rely on a vector embedding of each line; and they produce approximate results.

Security logs. In the security domain, structured logs are usually ingested into log managers such as Splunk [48], Google Security Operations [13], or Elastic Security [1]. These tools support substring matching and may offer AI-assisted query engines to help analysts formulate suitable queries. Unfortunately, substring queries are limited in expressivity, and because similar events might be recorded in multiple different formats, it is often challenging to develop a substring query that matches all instances of a relevant event.

3 System architecture

We now turn to Matryoshka's architecture, which processes logs through three progressive stages of understanding. In Fig. 2, we illustrate the parser using a sample log line from an SSH server log, representing a successful authentication for the root user on an SSH connection. Matryoshka learns parsers from a set of unlabeled log lines — the generation data. These parsers are used at run-time on live data. For the purposes of our evaluation, the live data and generation data are the same. In production enviromnents, the generation data would be the set of all collected logs up until a certain time.

3.1 Anatomy of a parser

Matryoshka produces parsers that convert unstructured text-based log messages into a structured database. Parsers operate in three

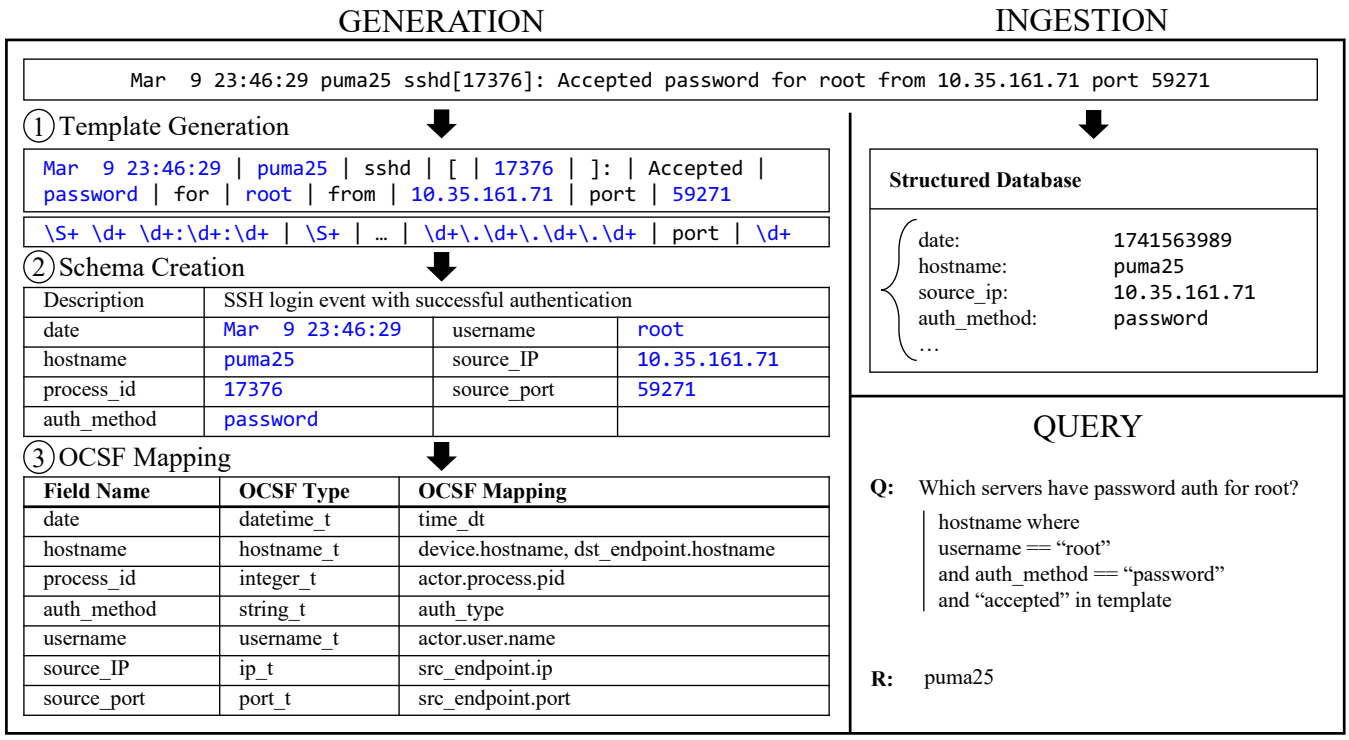

Figure 2: Matryoshka converts unstructured log lines (top) to structured data suitable for ingestion into a database (upper-right), in a sequence of three steps (①, ②, ③). This makes it easy to query the logs for events that satisfy certain conditions (lower-right).

main stages: syntax parsing, semantic parsing, and schema mapping.

Syntax parsing. Programs emit log lines by interpolating variable entities into a fixed message. Our parser infers a set of templates, each intended to represent a different fixed message. Templates are defined as sequences of tokens, where each token is either a fixed string or a variable. Variables are associated with a regular expression.

Prior work typically represents templates as text strings in which variables are replaced by wildcards. However, we observed that most variables in log messages exhibit a structured format. Rather than relying on wildcards (which tend to overcapture variables), we assign regular expressions to each variable to represent the content's expected structure. To parse a log line, we match it against the appropriate template, checking that each variable matches its corresponding regular expression.

Many templates may share a common prefix. For instance, Linux system logs often share a prefix indicating the date, hostname, process name, and process ID, followed by program-specific content. We build a tree of templates, where all templates in a subtree share a common prefix. This design helps us parse log lines consistently, as repeated prefixes are parsed identically.

For example, consider the log line in Fig. 2. It can be captured by the main branch in the parsing tree represented in Fig. 3. The start of the line (up to the process ID brackets) is present in all log lines, so it represents a branching point in the tree of templates. Each leaf represents a distinct template. Because the date and hostname are adjacent, parsing with wildcards would be unable to disambiguate where the date ends and hostname begins. In Matryoshka, we generate a regular expression for each variable (e.g., "`\S+\s+\d+\s+\d+:\d+:\d+`" for the date), which enables unique disambiguation.

At the end of this stage (step ① in Fig. 2), we have auto-generated a complete set of templates, each log line is matched to a template, and we extract the value of all variables.

Semantic parsing. While templates capture a line's syntax, they provide no information about its meaning. Therefore, the next stage maps each template to an auto-generated schema. Each schema includes a description of the template and a set of fields representing the template's variables, where each field is assigned a name and description. Names are chosen to reflect not only the data type of the original variable but also the variable's purpose within the broader log context (see Fig. 2, step ②). After this stage, each log line is mapped to a JSON object containing a description of the template and a list of named fields corresponding to the variables in the line. We also ensure that the schemas for related templates are consistent (e.g., use the same name when the "same" variable appears in multiple templates).

Schema mapping. The first two stages produce a structured representation of the log file enabled fast and efficient querying. We propose a third step, which maps this structured representation

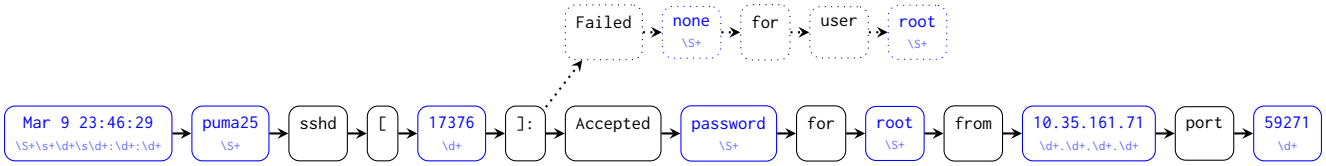

Figure 3: Example parsing tree, with two templates that share a common prefix. Each node/token represents either a string constant (black) or a variable (with associated regular expression; blue), and each leaf corresponds to a template (given by the path from the root to that leaf).

to an existing taxonomy so analysts can search over standardized attribute names (see Fig. 2, ③). We use OCSF [40], which is becoming a popular choice in many security information and event management (SIEM) systems. For each template, we:

- Identify the set of relevant OCSF events.
- Populate as many fields within each OCSF event as possible with static information from the template.
- Assign each field in the template's schema to an OCSF data type, and provide a converter to transform the field's value into the appropriate OCSF format (e.g., converting text-based dates to timestamps).
- When possible, map each field to one or more OCSF attributes. There often are multiple OCSF attributes that can represent a variable.

This final mapping step is not always feasible: OCSF is incomplete and many fields do not have a standardized attribute in OCSF, so we retain the schemas from the second stage to preserve all data.

3.2 Requirements

Building such a parser by hand requires substantial effort. Typically, templates must be inferred from example log lines. The meaning of log lines and their variables can be obscure; moreover, there are tens of thousands of OCSF fields, so creating schemas and mapping them to OCSF events is tedious.

Our system leverages LLMs' understanding of log files to automate parser construction. We designed our system around four core principles:

- **Accuracy.** The structured representation of the logs must be correct: each template should match only a single event. Variable tokens should capture parts of the line that can vary and convey some meaningful information about the event. Template schemas must accurately represent the content of log lines, with meaningful variable names. Fields must only be mapped to OCSF attributes that capture their intended role. Lastly, queries executed on the structured data should yield the same result as painstakingly searching the raw logs.
- **Completeness.** The structured representation must capture all information in the original log file. Any query that could be executed on the raw logs should be equally feasible on the structured data.
- **Consistency.** Variables fulfilling the same role across multiple log lines should be handled identically: they must have the same field names, descriptions, OCSF mappings, and data types. Similarly, templates that are syntactically different but serve the same purpose should have the same schema.
- **Run-time efficiency & security.** The parser we produce should operate efficiently: we should not need to invoke a language model on every log line, and queries against structured data should run more efficiently than a linear scan over the raw logs. The parser should not be vulnerable to prompts injection attacks at run-time.

While these four principles guide our system, it is impossible to perfectly meet them all in practice. For instance, accuracy cannot be measured without complete knowledge of the developer's original intention, and consistency involves a subjective notion of semantic similarity across fields.

3.3 Applying LLMs

LLMs are a powerful tool for log parsing, but we cannot apply a LLM to every log event. Real-world systems typically generate millions of log lines daily, and it would be far too expensive to parse each with a LLM or compute a vector embedding of each log line. Instead, we use LLMs only to create parsers, not to parse data at run time. This means while our generation process could be vulnerable to attacks against the LLM present in the generation data, Matryoshka parsers are immune to prompt injection at run-time. We use several techniques to improve accuracy:

Controlled execution. Each time we query a language model, we test its output with various heuristics to catch common errors. For instance, in Section 3.4, we verify the returned regular expressions are valid and do not over-capture — if so, we ask the model to fix its answer. In rare cases, human feedback can supplement these checks. If a particular test fails, we either directly correct the model's output when possible or interact with the model in a multi-turn exchange, explaining the inaccuracies and prompting it to self-correct.

Guided chain of thought. We frequently ask the model to outline its reasoning steps before providing the final answer. We supply a structured algorithm or checklist to steer its reasoning process, ensuring thorough analysis of the input data.

Few-shot prompting. We include a few previously generated input-output pairs in the input. We include the chain of thought for each, to show the desired reasoning style. These examples can be hard-coded, supplied by humans, or derived from the model's own past responses to maintain consistency.

Description embedding. Including few-shot examples that closely match the current prompt substantially improves both consistency and accuracy. Ordinarily, one might use text embeddings and cosine similarity to locate similar examples, but our input data often mixes structured and unstructured text (including unique identifiers),

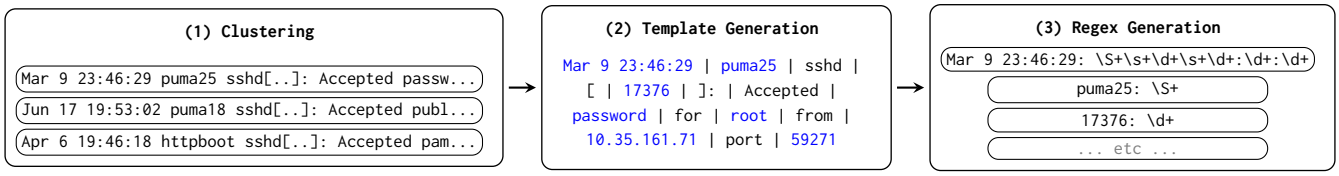

Figure 4: Overview of template creation: (1) clustering, (2) generation, and (3) regular expression assignment.

which degrades embedding performance. Instead, we devise a two-step procedure where the model first writes a descriptive summary of the object to be embedded, and we then compute an embedding of that summary. Empirically, this method works well for diverse data types (e.g., log lines, variables, templates, OCSF fields).

Consider our SSH log example in Fig. 2. Embedding this raw log line directly produces poor results because identifiers (IP addresses, ports, dates, hostnames) are not handled well by natural language tokenizers and embedding models. While removing or replacing these with placeholders improves the situation, the embedding still represents syntax rather than meaning. A better approach is to first generate a semantic description of the log line. An embedding of this description captures the meaningful content of the log line rather than its syntactic structure.

Self-consistency. Rather than having the model provide a single response, we generate multiple responses and select the most frequently produced one. We introduce slight variations in the prompt, such as randomizing the few-shot examples, to encourage diversity in the model's outputs. Self-consistency has been shown to outperform greedy decoding by allowing the model to explore multiple reasoning paths [55].

3.4 Template creation

In our system, a syntax parser is represented by a tree of templates (see Fig. 3). We use a LLM to build this tree, similar to LILAC [21] and DivLog [58], except we use regular expressions (instead of wildcards) to capture variables, yielding more precise parsers. While our overall template-generation process is largely similar to prior efforts, our specific heuristics and clustering approach improve template quality compared to existing methods.

We seed this process with a few user-provided examples of log lines and corresponding templates. Empirically, we found this helps guide the LLM in template generation. We initialize the parsing tree with these templates. Then, as we parse each log line, if it does not match any existing template, we generate a new template for it and add the new template to the parsing tree. Templates are generated by clustering unmatched log lines, using a LLM to generate a template for each cluster, and using a LLM to infer a regular expression for each variable(see Fig. 4). We further validate the result to prevent over-capturing.

Clustering. Inspired by LogBatcher [57], we observe that generated templates are of higher quality when the LLM sees multiple lines from the same template rather than a single line. Therefore, we store a streaming buffer of N unmatched log lines; when the buffer becomes full, we cluster them in three steps:

- *Coarse-grained clustering.* First, we attempt to match each unmatched line to the existing tree, and keep track of the longest matching prefix. Log lines that do not share the same prefix cannot belong to the same template. Consequently, this yields an initial partition of lines into clusters.
- *Fine-grained clustering.* We further subdivide these coarse clusters by computing description embeddings for each line and using the DBSCAN clustering algorithm[3] with cosine-similarity distance on each coarse cluster.
- *Cluster confirmation.* We select the fine-grained cluster with the highest average pairwise cosine similarity and randomly sample five log lines from it. We ask the language model—using guided chain of thought and self-consistency—to confirm if these lines come from a single format string (i.e., if they should be associated with a single template). If it rejects or partially rejects this set, we prompt it to isolate the subset of lines that may share a format string.

Template and regular expression generation. We ask the LLM to construct a template from the log lines in a cluster. First, we ask the LLM to segment the line into tokens; then we ask it to provide regular expressions for the variable tokens. We employ few-shot prompting, guided chain of thought, and self-consistency. We select the few-shot examples as the most similar templates already in the tree, selected via description embeddings.

Each template the model proposes is tested with a suite of heuristics. First, we confirm that the output is valid and can indeed serve as a template. We attempt to repair simple mistakes first using hard-coded rules, then by explaining the mistake to the model and soliciting a self-correction. From the valid templates, we choose those that (1) match as few previously parsed lines as possible, while (2) covering as many of the new cluster's lines as possible. This encourages the model to propose templates that are both precise (and do not over-capture) and suitable (capture as much of the current cluster as possible).

Validation. If a proposed new template does not match any previously captured line, it is added to the parsing tree outright. If it does match (meaning that the log line could match both the proposed new template and an existing template), heuristics are needed to decide whether to keep it or not. There are legitimate reasons for templates to have some overlap in the set of lines they can match. A previous template could be too specific, or a new template could be needed to capture malformed values. However, overlaps can also be due to the new template over-capturing, in which case we should discard it. We use heuristics to decide what to do with a new template. Let T_n be the new template and $\mathcal{T}$ the set of overlapping existing templates:

[3] In our experiments, setting $\epsilon = 0.05$ yields the best results, avoiding over-capture.

- *Capturing malformed variables.* If T_n only differs from overlapping templates by the regular expressions for variables, this new template is likely needed to capture malformed variables, thus we keep it. For example, the template for Fig. 2 will not capture "`Jun 2 05:19:38 puma15 sshd[153]: Accepted password for root from x.x.x.x port 59271`" due to the malformed IP. We need a new template with a broader regular expression for the `source_ip` field. This new template is identical to the original except for its capturing regular expressions, thus should be kept. We choose to keep both the original and new template: the difference in granularity can help distinguish valid values from invalid ones, for instance the IP in the above example.
- *Clustering heuristic.* When a new template T_n overlaps with existing templates in $\mathcal{T}$, we collect a representative sample of log lines that match each template in $\mathcal{T}$, as well as lines that only match T_n. We then query the language model, using the same query as for the clustering step of the process, to determine if these lines are part of a same cluster. If the LLM confirms that these lines are part of the same cluster, we conclude that the original templates were likely too specific. We replace the older templates with T_n.

If none of these conditions apply, we either ask the user whether to keep T_n, or if we wish to run without feedback, we automatically reject the template. Rejected templates are caused by excessively large clusters or underfitting from the generation process. We avoid repeating this mistake by rerunning the generation process on a single line of the cluster while making sure the proposed templates do not overlap with existing templates. If any overlap happens, we supply the model with the overlapping lines and prompt it for a self-correction.

3.5 Schema creation

Once the tree of templates is constructed, we map it to a schema by assigning each variable node a field name and description. This allows the logs to be ingested into a structured database and queried conveniently by field name.

We create these fields on a template-by-template basis. Templates are first sorted by descending prevalence in the log file. For each template, we ask the language model to generate a schema, passing it a few log lines matching the template and the list of tokenized template elements. The model must produce a template description and assign a name and description to each variable. Since templates reside in the same parsing tree, their tokens often share common prefixes. We give the model the names and descriptions for these existing prefix tokens so they can be reused.

To ensure consistency, we employ few-shot prompting, encouraging the model to reuse variable names and descriptions across templates when the same semantic roles are present. We do this by identifying the most similar templates that have already been mapped to a schema using description embeddings and presenting them to the model. We gather multiple candidate answers, verify that each preexisting variable's name is preserved, and then embed each description. To maximize consistency, we select the candidate whose embedding is closest to the average of the other similar templates.

Once this process is done, we ask the model to generate new descriptions for each field, using example values from the variables mapped to that field as context. Regenerating descriptions increases the quality and generality of descriptions, since the model can see more diverse usage examples.

3.6 Schema mapping

While Matryoshka can function immediately after building the schema and naming fields, using these field names directly requires familiarity with the schema. For easier interoperability with common security tools, we map the schema to a set of relevant OCSF events [40]. This lets us query logs using official OCSF attributes, which are standardized and well documented. Mapping a schema to OCSF events requires (1) inferring the type of each field, (2) identifying the relevant OCSF event(s), (3) mapping each field to one or more OCSF attributes, and (4) creating a converter for any necessary transformations.

Typing. Every field in the schema is assigned a standard OCSF type [41]. Typing enables using predicates over the type of a variable instead of over its field name (e.g., searching over all fields of type IP address instead of only those named "`source_ip`"). We determine each field's type by prompting an LLM with a list of possible types, the field's name, its description, and sample values. These sample values are taken from all the variables that share the same field name, so that the type can represent them all.

Event identification. We identify which OCSF events pertain to each template. We provide the LLM with a short description of each possible OCSF event, along with the template and its description, and ask for up to three likely event matches. We use few-shot prompting with similar templates to maintain consistency, run this process multiple times, and then pick the three most frequently identified events.

Attribute mapping. OCSF events contain hierarchically organized attributes. Mapping our fields to these attributes is challenging because: (1) the attribute tree is large (tens of thousands of leaf attributes), and (2) attributes are documented in the context of their parent. Additionally, we want a consistent mapping. Consistency involves two components:

- *Twin consistency. Twins* are two variables in the parse tree with the same role. These should map to the same OCSF attributes.
- *Sibling consistency. Siblings* are two fields that appear in the same template and describe different aspects of the same entity (e.g., source IP and source port). We prefer that such siblings map to attributes with a shared direct parent in the OCSF attribute hierarchy (e.g., `src_endpoint.ip` and `src_endpoint.port`).

First, we simplify the OCSF attribute tree by flattening it and creating a LLM–generated description for each flattened attribute. We then embed these descriptions. This process is done once and cached.

Next, we map fields sequentially. For each field in our schema, we embed the field's description and select the K OCSF attributes (of the same data type) with the highest cosine similarity. Empirically, $K = 25$ usually includes the correct attribute in the set. We give the LLM these K candidate attributes (with their descriptions), plus the field name, description, and sample log lines. The model is then

asked which attributes best match this schema field, if any. Since fields in our schema already merge variables with identical roles, twin consistency is satisfied.

In order to enforce sibling consistency, we want to ensure sibling fields appear in the list of potential candidates the LLM must decide from. For instance, if a template contains a source IP field mapped to `src_endpoint.ip`, and a source port field, we need to ensure the `src_endpoint.port` OCSF attribute is in the list of candidates. We list all potential siblings, and add them to the list of candidate OCSF attributes before querying the LLM. Let $field$ be the field we are currently mapping. (1) We list all other fields that share a template with $field$. (2) We list all OCSF attributes that are assigned to one of these fields. (3) We list all sibling OCSF attributes to an assigned attribute. (4) We add all siblings that have the same data type as $field$ to the set of candidate attributes.

We improve accuracy using guided chain of thought and self-consistency: Specifically, we ask the model to list all candidate attributes and to explain why each candidate attribute is or is not suitable, run the query multiple times, and select any attribute returned in at least half the runs.

3.7 Normalization

At this point, we have mapped logs to a structured and semantically-rich schema, and associated created fields to existing OCSF attributes. We wrote static conversion functions for dates, times and numbers, to translate them into a standard format. This is sufficient to allow users to query the data based on field values and static parts of the template, and is what our evaluation metrics focus on. However, to ensure thorough conformity with the OCSF format, we include two additional steps to help better conform to the OCSF format.

Static attributes. Some OCSF attributes can be filled statically based solely on the text of the template, and do not depend on the values of any variables. For example, for the example in Fig. 2, the "`activity_id`" field can be statically filled with the value 1, indicating that this is a new connection. For each template, we identify static values by querying a LLM with a redacted template (variables are replaced by their names) and a list of top-level OCSF attributes that remain unmapped, using guided chain of thought and self-consistency, and few-shot prompting with similar templates.

Converter creation. Some OCSF types require data to be in a specific format. Relying on our static conversion functions is insufficient. First, some OCSF attributes expect a value chosen from an enumeration. Second, some rare date, time or number formats might not be handled by our static parser. When one of these scenarios arises, we use a LLM to write Python code to convert values into the expected output format, prompting the LLM with example values from the logs and the expected output format. If the converter encounters a value at generation time that it cannot parse, it raises an error, at which point we provide the language model with this new value so it can update the converter dynamically.

4 Evaluation

Our evaluation is comprised of four parts.

(1) **End-to-end evaluation.** We measure Matryoshka's ability to convert data into a format that supports structured queries. We run a set of curated queries against datasets processed with Matryoshka, and compare the results to ground-truth answers.
(2) **Step evaluation.** We evaluate Matryoshka's step-by-step performance using metrics we developed, either refinements to existing metrics or in several cases new metrics. We compare our syntax parser to three existing works: LILAC [21], Drain [17], and Brain [60].
(3) **Ablation study: Description embedding efficacy.** We run Matryoshka twice, once with description embeddings, once with regular embeddings, and compare the performance of both using our step metrics to demonstrate the usefulness of description embeddings.
(4) **Efficiency.** We measure the time to process security logs and query them, showing that Matryoshka is efficient enough for practical use.

	Total Nodes	Variable Nodes	Templates	Unique fields	Unique OCSF attributes
SSHD	384	154	98	47	135
Cron	72	19	7	8	35
DHCP	441	153	178	41	67
Audit	6309	3000	496	150	286
Puppet	981	370	254	101	77
Total	8187	3696	1033	347	600

Table 1: Bug tracker parser summary

We ran Matryoshka on our curated dataset using one labeled example for the template parser, and zero-shot for the schema creation and OCSF mapping. Our source code, containing Matryoshka, our datasets, and our curation tools, is made public at https://github.com/julien-piet/matryoshka. We support the Gemini [51] suite of models, but the code can easily be extended to any language model. Unless stated otherwise, we mostly rely on Gemini-1.5-Pro for our experiments.

In the rest of this section, we first introduce the datasets we use, including one we created to benchmark Matryoshka. Then, we discuss the metrics we create to evaluate parsers, and for each, provide evaluation results for the experiments outlined above.

4.1 Datasets

Our evaluation relies on two datasets.

Bug tracker dataset. We curated our main dataset from Redhat Bugzilla bug reports. We discovered that many users attach system log files to public bug reports there, so we crawled the Bugzilla website and downloaded all log files attached to bug reports up to August 2023. In total, this dataset contains over 30 million log lines. We filter to logs from five applications that are particularly security-relevant and are well-enough documented that we could manually construct ground-truth parsers:

- **Linux kernel audit logs (77K lines)**: Fine-grained logs of security-relevant events in Linux systems.
- **Puppet logs (157K lines)**: Logs from Puppet, an orchestration tool for server configuration deployment.

- **SSH daemon logs (35K lines)**: Logs involving SSH authentications and connections.
- **DHCP logs (378K lines)**: DHCP client logs reporting DHCP requests and lease information.
- **Cron logs (13K lines)**: Reports of scheduled CRON jobs.

We ran Matryoshka on each log file. We also created a ground-truth parser, by manually checking and fixing every template, field name, and mapping generated by Matryoshka. The ground truth parsers are summarized in Table 1.

LogPub. Existing approaches to template generation commonly use LogPub [23, 68]. This resource contains 14 log files (over 3 million lines) annotated with ground-truth wildcard-based templates. We use it to compare the performance of our template generation step against prior methods, but we do not generate ground-truth labels for schema creation or attribute mapping because:

- Most of the data is outdated (some logs are over 20 years old).
- Several files are heavily anonymized, limiting reliable semantic extraction.
- Most are not security-focused and instead are activity logs from supercomputers or distributed systems.

4.2 End-to-end performance

We define up to ten queries for each log type (see Appendix A), chosen to represent the most common and security-relevant tasks that a security analyst might perform. For example, one of our cron log queries looks for scheduled jobs from a particular host within a specific time window, filtering on OCSF fields `scheduled_job_activity.time_dt` and `scheduled_job_activity.device.hostname`, and checking for the presence of the `executable_path` custom field.

All queries are formed by applying unions or intersections of predicates. Predicates can be over the values of the fields, or over the static part of templates. We run each query in two ways: once using OCSF attributes, and once using the field names generated by our schema mapping step.

Predicates on the static template are defined by substring matching. This limits the range of queries we can run. For example, "return all lines indicating a bind error due to an already-in-use address" is tedious to formulate in our query syntax, because it requires knowing all possible templates that could indicate such an event. A dedicated LLM-powered query planner could plausibly map queries to the relevant set of templates; we leave that to future work.

Each query effectively creates a binary classifier over the log lines, and we evaluate performance by comparing results from the parser-structured data to our ground truth data. We report the precision and recall of the results. In doing so, we measure how accurately our system can handle typical analyst queries.

Existing systems rely on substring matching on the unparsed log line. This can answer some queries, but limits the reach that an analyst can have, and many of our queries could not be expressed using substring matching alone. We illustrate this using three examples:

- Tracking the usage of a specific key based on its hash in SSH can be done efficiently using substring matching.
- Substring matching does not allow restricting results to a particular date range or time window. Analysts would have to manually write a one-off script to post-process the results of the substring search. In contrast, Matryoshka can express such a predicate natively.
- One of our queries asks to identify DHCP leases with long renewal times. This is difficult to express with substring matches, as there is a variety of log formats that contain the renewal time. With Matryoshka, this query can be expressed simply with a predicate on the "`renewal_time`" field.

Table 2: Matryoshka query precision and recall metrics

Dataset	OCSF attributes		Created attributes		Naive substring	
	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.
SSHD	0.82	**1.00**	**1.00**	**1.00**	0.90	0.80
Cron	0.40	0.40	**1.00**	**1.00**	0.92	**1.00**
DHCP	0.60	0.59	**1.00**	**0.97**	0.92	0.71
Audit	0.70	0.68	0.90	**0.88**	**0.97**	0.85
Puppet	0.80	0.80	0.90	**0.88**	**1.00**	0.63
Average	0.66	0.69	**0.96**	**0.95**	0.94	0.80

We evaluate the performance of naive substring-based systems, by translating each query into substring matching where possible. We assume an analyst will not have resources to comprehensively identify all message formats/templates that might have relevant information; instead, we simulate an analyst who looks for the first example they can find of a log message containing the necessary information and identifies a single substring from that log message to search for.

Experimental setup. We ran Matryoshka on our bug tracker dataset. We rely on Gemini-1.5-Pro and the standard Gemini embedding model as our LLM backend. We used user feedback for resolving overlaps that could not automatically be resolved — this required a total of 8 user inputs across the 5 files.

Results. We compare the output of the queries on the full pipeline versus the ground truth. We ran queries in three ways: once using OCSF attributes, once with Matryoshka-created attributes, and once using naive substring-matching. The average precision and recall of the queries is reported in Table 2. We observe that using OCSF attributes for querying performs leads to poor results compared to created attributes: This is due to the difficulty of mapping created fields to OCSF attributes. If the mapping for a high volume variable fails, this variable cannot be queried using OCSF attributes. However, if we run queries over our created schema instead, both precision and recall are consistently above 80%, with some files getting values above 95%. In fact, most queries yield perfect performance, but a few do not run properly and negatively impact the average scores.

The main error source is baseline fields being represented as multiple parsed fields. For example, the audit log has a field for the current working directory of a process. This field in the generated parser is most often called "`current_working_directory`", but sometimes abbreviated as "`cwd`". For the sake of simplicity, we chose to run queries on only one created field, even when the field is fragmented: requiring analysts to list all possible fields would be tedious. Using an LLM query engine could help here, as the model could automatically select all relevant fields.

Table 3: Comparison of Matryoshka and prior template generation on LogPub data

Dataset	Matryoshka		LILAC		Drain3		Brain		Dataset	Matryoshka		LILAC		Drain3		Brain	
	PGS	TS	PGS	TS	PGS	TS	PGS	TS		PGS	TS	PGS	TS	PGS	TS	PGS	TS
Apache	**1.00**	**1.00**	0.99	0.99	0.99	0.89	0.99	0.88	HPC	**1.00**	0.99	**1.00**	**1.00**	0.97	0.96	0.78	0.75
BGL	**0.99**	**0.98**	0.97	0.96	0.95	0.96	0.88	0.79	Linux	0.85	**0.93**	0.83	0.86	**0.87**	0.80	0.81	0.71
Hadoop	**0.97**	**0.91**	0.86	0.81	0.88	0.71	0.35	0.29	Mac	**0.96**	**0.91**	0.78	0.75	0.83	0.82	0.95	0.89
HDFS	**1.00**	**0.96**	0.96	0.84	0.97	0.93	0.97	0.84	OpenSSH	0.97	**0.97**	0.72	0.68	0.96	0.92	**0.99**	0.92
HealthApp	**1.00**	**0.94**	**1.00**	**0.94**	0.97	0.92	0.54	0.44	OpenStack	0.93	0.86	0.93	0.88	0.66	0.67	**1.00**	**0.97**
Proxifier	**1.00**	**0.50**	0.97	0.38	0.52	0.31	0.99	0.35	Spark	**1.00**	**0.97**	0.91	0.93	0.97	0.79	0.97	0.91
Thunderbird	**0.98**	**0.82**	0.82	0.68	0.87	0.68	0.79	0.69	Zookeeper	**0.99**	0.96	0.86	0.83	**0.99**	**0.98**	**0.99**	0.95

Some errors are due to overcapturing: this is when a field name in the generated parser covers multiple fields in the baseline. Unfortunately, these errors are difficult to recover from, as there is no way to separate the variables without looking at them individually. However, this type of error is rarest, as we designed the parser, schema creation and OCSF mapping to err on the side of undercapturing.

Queries over created attribute names perform better than the naive substring queries. The naive approach uses an arbitrary line as an example to model the substrings needed to run the query: this leads to missed lines when there are multiple possible templates that would match a given query, and leads to over-capture if the same substring is present in other lines. However, we want to emphasize the fact these substring queries are likely not representative of what an analyst would run: in practice, they would likely put more effort to obtain a better result. Instead, these results are representative of what could be achieved using substrings if we spend the same time writing them as writing our queries over created fields. The complete set of substring matches usSed for this task are detailed in Appendix A.

4.3 Template generation evaluation.

Prior template generation solutions measure performance using two primary metrics: *Group Accuracy (GA)* and *Parser Accuracy (PA)*.

- **GA (Group Accuracy)**: The percentage of log lines assigned to the same group as the ground truth. Each template encodes one group of lines derived from the same format string, so GA measures how well the parser separates different format strings.
- **PA (Parser Accuracy)**: The percentage of log lines whose template exactly matches the ground truth. PA focuses on whether each predicted template is identical to the ground-truth template.

Although broadly used, these metrics have limitations. Parser accuracy is too strict. If our parser splits a variable into two parts (e.g., splitting "IP:port" into separate fields when the ground truth has them merged), the template is deemed incorrect, even though it might still be perfectly useful—or even preferable—in practice. Group accuracy penalizes both overcapture and undercapture equally, although overcapture can be more problematic. Overcapture conflates data from separate format strings, making fields unusable. For example, some log parsers use wildcards in templates like:

"`Accepted <*> from <*>`"

intended to match lines such as:

"`Accepted password for john`"

However, this template also incorrectly matches:

"`Accepted password for john from 192.168.1.1`"

causing the IP address to be captured as part of the username. The parser thus fails to capture a separate IP field.

To address these issues, we propose two refined metrics. Both these metrics take the per-line average of a scoring function. If a line matches multiple templates, the score for that line is taken to be the maximum over all matches:

- **Template Similarity (TS)**: The per-line score is an edit-distance-based similarity between the predicted and ground-truth templates when all variables are replaced by "<*>". This is defined as
$$1 - \frac{\texttt{Levenshtein}(\text{Ground Truth}, \text{Predicted})}{\max(|\text{Ground Truth}|, |\text{Predicted}|)}$$
In contrast to parser accuracy, small differences (like splitting or joining variables) do not yield a zero score.
- **Parser Group Similarity (PGS)**: The per-line score is the average ratio of the predicted group size to the ground-truth group size for each line. If the parser's group is larger than the correct group (overcapture), the score is 0 for that line. Otherwise, it is
$$\frac{|\text{Parser Group}|}{|\text{Ground Truth Group}|}$$
This asymmetry acknowledges the harm of merging lines from distinct format strings, while still discouraging undercapture: A trivial parser that assigns a unique template for each line would get close to 0 score.

LogPub dataset. Templates generations has been extensively studied in prior work, and recent works such as LILAC [21], DivLog [58] or LogBatcher [57] use language models to perform this task. Although we added regular expressions for precisely matching variables, our templates can be converted back into the same wildcard templates used in the past in order to compare our approach to prior works. Many of our building blocks are similar to those found in these works. However, our approach achieves a higher accuracy with minimal labeled data (1 labeled example) by selecting the most appropriate few-shot examples dynamically using our description embeddings, and using additional heuristics to catch mistakes and self-correct, as detailed in Section 3.4.

Experimental setup. We selected three prior works to compare against: LILAC [21], one of the most promising LLM-based approaches, Drain3, the latest version of Drain [17], a popular log

Table 4: Comparison of LILAC and Matryoshka on the bug-tracker datasets

Dataset	Lines	Matryoshka		LILAC		Brain	
		PGS	TS	PGS	TS	PGS	TS
SSHD	35,329	0.99	0.99	0.92	0.81	0.88	0.73
Cron	12,547	1.00	0.99	1.00	0.80	1.00	0.6
DHCP	377,653	0.99	0.98	0.40	0.44	0.54	0.69
Audit	76,636	0.99	0.99	0.13	0.33	0.62	0.52
Puppet	156,880	0.98	0.98	0.59	0.74	0.96	0.68

parsing algorithm, and Brain [60], a lightweight parser. On the LogPub dataset, these three frameworks use a pre-parsed version of the logs and only generate templates for the suffixes. In contrast, Matryoshka is given the raw version of the logs. Drain and Brain do not use LLMs or few-shot examples. We ran LILAC and Matryoshka with Gemini-1.5-Flash and rely on a single labeled example. For fairness with prior works, we did not use user feedback to resolve overlaps, and instead automatically refused templates that overlap if the heuristics could not resolve it.

Results. We report our metrics (Parser Group Similarity and Template Similarity) in Table 3. We find that Matryoshka equals or outperforms other works on most log files. On average, we obtain a parser group similarity of 0.97 and a template similarity of 0.91, while the best of the other schemes, LILAC, obtains a PGS of 0.90 and a TS of 0.82.

Bug tracker dataset. We further compare LILAC and Brain to Matryoshka on our own dataset. In this setting, LILAC and Brain are run on the full log lines: Unlike LogPub, our dataset contains a multitude of log prefixes, because the logs originate from multiple systems. We use the same Matryoshka parsers as for the end-to-end queries. Matryoshka outperforms both other works on all five logs. We attribute this performance gap to the lack of consistency of LILAC and Brain when parsing similar structures across logs. In contrast, Matryoshka uses previously generated templates as examples for new ones, and parses shared prefixes using the same prefix branch in the parse tree, ensuring consistency. This issue does not appear as much in the LogPub dataset because the similar prefixes across lines are pre-parsed.

4.4 Schema creation evaluation.

Schema creation entails clustering variables by meaning (e.g., assigning every "source IP" variable to a field "source_ip"). For this step, we define **Schema Group Similarity (SGS)**, the analog to PGS, computed over variables. If our schema's group for a variable is larger than in the ground truth, the similarity is 0. Otherwise, it is the ratio of the parser's group size to that of the ground-truth cluster. This rewards grouping variables of the same semantic role without merging separate roles. The score over a file is computed by taking the weighted average of group similarity for each variable, so that high-volume variables are most represented in the final score than rare ones.

Results. We ran the schema creation step on the golden parsers for the five logs in our dataset to evaluate this step independently from the rest of the pipeline. Doing so allows to compute the schema group similarity across the files, as reported in the first column of Table 5. SSHD, Cron and DHCP logs all have group similarities above 0.95, indicating the created schema groups fields almost identically to the ground truth. Puppet scores a bit lower: some variables are grouped to multiple fields when the baseline grouped them as one. For example, instead of recognizing missing dependencies as a single field, it subdivided them according to the type of dependency: the baseline field "`dependency`" is split into "`missing_service`", "`missing_provider`" and "`missing_package`". This can make querying more difficult, as searching over dependencies will requires predicates over three fields instead of one. Audit scores are the worse, at 0.56.

Table 5: Matryoshka schema and mapping performance metrics

Dataset	Schema Group Similarity	Mapping Accuracy
SSHD	0.96	0.76
Cron	1.00	0.66
DHCP	0.98	0.71
Audit	0.56	0.42
Puppet	0.80	0.51

4.5 OCSF mapping evaluation.

Our baseline attribute mapping maps each field to every OCSF attribute that is relevant to that field. There are often multiple. In practice, we care more about having one correct mapping than getting all the possibilities. Thus, we design our mapping accuracy metric by defining a per-field score, equal to the proportion of assigned fields that were also present in the baseline. More precisely, if $\text{OCSF}_{\text{Baseline}}(f)$ is the set of OCSF mappings for field f in the baseline, and $\text{OCSF}_{\text{Matryoshka}}(f)$ in the generated parser, then:

$$S_f = \begin{cases} \delta_{\text{OCSF}_{\text{Baseline}}}(f) \stackrel{?}{=} \emptyset & \text{if } \text{OCSF}_{\text{Matryoshka}}(f) = \emptyset \\ \frac{|\text{OCSF}_{\text{Matryoshka}}(f) \cap \text{OCSF}_{\text{Baseline}}(f)|}{|\text{OCSF}_{\text{Matryoshka}}(f)|} & \text{otherwise} \end{cases}$$

The mapping accuracy is the weighted average of the scores.

Results. We ran mapping independently from the other steps, by using the golden schemas as a baseline, and report the mapping accuracy of each file in the second column of Table 5. Performance of this step is lower than for other steps, Matryoshka struggling most with Audit and Puppet, two rich logs containing many variables. Mapping is the most difficult step, as there are tens of thousands of candidate OCSF mapping attributes, and whether an attribute is a match for a field is often open to interpretation. For instance, the "authentication.src_endpoint" attribute is described within the taxonomy as "the source of the IAM activity.", while "`authentication.dst_endpoint`" is "the endpoint to which the authentication was targeted". The device on which an authentication happens in an authentication log message certainly maps to the "`dst_endpoint`" attribute, yet based on the definition alone, could arguably also map to "`src_endpoint`" since the authentication is happening on the device. This ambiguity is the reason we decided

to select all possible mappings for our baseline, and reward the system if it chose attributes from this set.

4.6 Ablation study

We test the usefulness of description embeddings by running Matryoshka's template generation and schema creation steps on DHCP logs with a regular embedding instead. We cannot run the mapping stage without description embeddings, as they are central to the field filtering process. We ran our standard queries using the output from this altered pipeline, obtaining a precision of 0.9 and a recall of 0.81, significantly lower than the near-perfect metrics of the regular parser. Description embeddings improve Matryoshka's performance.

4.7 Efficiency

Step	SSHD	Cron	DHCP	Audit	Puppet
Generation	2h 25m	10m	2h 56m	8h 46m	5h 18m
Creation	18m	4m	17m	2h 53m	26m
Mapping	1h 15m	12m	1h 58m	7h 24m	3h 43m
Total	3h 58m	26m	5h 10m	19h 3m	9h 27m

Table 6: Timing summary for generation steps

Our system allows running queries in a few seconds, even when the source log file is hundreds of thousands of lines. Log ingestion is also fast: we consistently parse log files at over 200 lines per second. This is possible because LLMs are only used at generation time, not at run-time: live data is statically parsed, which also prevents prompt-injection attacks. However, the process of generating parsers is slow, due to the fact most steps use previous answers of the model to few-shot prompt the next ones, thus cannot be parallelized. This only needs to be run once, but future work should look at speeding up this process.

5 Discussion

Although Matryoshka advances the state-of-the-art in end-to-end log parsing, several limitations are worth highlighting. Mapping extracted fields to standard OCSF attributes remains challenging. Reliably mapping fields into OCSF is fundamentally difficult due to the volume of target attributes and nuances in different field definitions. Future work should explore more advanced semantic techniques to further improve mapping accuracy.

Queries run against Matryoshka processed data can miss entries or report false positives. Given analysts' need for high reliability, any incorrect or missing extracted fields can significantly impact usefulness. To address this limitation, we developed a prototype user interface enabling analysts to inspect and correct automatically generated parsers. However, improving UI design for efficient parser correction remains important future work.

Matryoshka makes it easy to run queries over variables and to substring match templates. However, some queries require selecting specific templates based on the event they characterize. Using an LLM-assisted query engine could more conveniently determine relevant templates for each query.

Log ingestion and querying is fast, because we only rely on regular-expression matching. However, parser generation is slow due to sequential LLM calls used to ensure consistency. Future solutions could improve speed by parallelizing model calls or using more efficient architectures, relying on smaller domain-specific models, to speed up parser generation and improve practicality.

We assume the ingested data is the same as the generation data in Section 3. In practice, as more data gets collected, lines from unknown templates are bound to occur. Production implementations of our work would need to be online—meaning they can discover new templates at run time.

At present, we only support logs where each event is a single line. Ideally, a practical system would handle multi-line log messages and any arbitrary log format.

6 Conclusion

We presented Matryoshka, the first end-to-end system leveraging LLMs to automatically generate semantically-aware structured log parsers. Matryoshka combines a novel syntactic parser with a semantic layer that clusters variables, maps them to structured schemas, assigns contextually meaningful field names, and maps variables to attributes from the Open Cybersecurity Schema Framework. Queries ran against Matryoshka-parsed real-world logs achieve an average precision of 0.96 and recall of 0.95, which significantly higher than achievable with existing substring matching techniques. While mapping to OCSF's schema remains challenging, Matryoshka's structured representation still enables precise querying using created fields, significantly reducing security analysts' manual parsing workload. By automatically transforming unstructured logs into structured, semantically rich data, Matryoshka represents a meaningful step toward enabling security analysts to focus on threat detection rather than manual parser construction.

Acknowledgments

We thank Aashish Sharma and the cybersecurity team at Lawrence Berkeley National Laboratory for providing valuable insights into security operations and helping shape this project. We also thank Scott Coull and Sunil Vasisht for their insightful feedback that helped shape Matryoshka, and David Huang for helping implement template generation works. This research was supported by the KACST-UCB Joint Center on Cybersecurity, OpenAI, the National Science Foundation under grant numbers 2229876 (the ACTION center) and CNS-2154873, the Department of Homeland Security, IBM, C3.ai Digital Transformation Institute, Open Philanthropy, and Google. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsors.

References

[1] Elasticsearch B.V. 2025. *Elastic Security.* https://www.elastic.co/security Accessed: 2025-04-12.

[2] Eric Anderson, Jonathan Fritz, Austin Lee, Bohou Li, Mark Lindblad, Henry Lindeman, Alex Meyer, Parth Parmar, Tanvi Ranade, Mehul A. Shah, Benjamin Sowell, Dan Tecuci, Vinayak Thapliyal, and Matt Welsh. 2024. The Design of an LLM-powered Unstructured Analytics System. arXiv:2409.00847 [cs.DB] https://arxiv.org/abs/2409.00847

[3] Merve Astekin, Max Hort, and Leon Moonen. 2024. A Comparative Study on Large Language Models for Log Parsing. In *ESEM.*

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *NeurIPS* (2020).

[5] Wenhu Chen. 2023. Large Language Models are few(1)-shot Table Reasoners. arXiv:2210.06710 [cs.CL] https://arxiv.org/abs/2210.06710

[6] Tianyu Cui, Shiyu Ma, Ziang Chen, Tong Xiao, Shimin Tao, Yilun Liu, Shenglin Zhang, Duoming Lin, Changchang Liu, Yuzhe Cai, Weibin Meng, Yongqian Sun, and Dan Pei. 2024. LogEval: A Comprehensive Benchmark Suite for Large Language Models In Log Analysis. arXiv:2407.01896 [cs.CL] https://arxiv.org/abs/2407.01896

[7] Hetong Dai, Heng Li, Che-Shao Chen, Weiyi Shang, and Tse-Hsun Chen. 2020. Logram: Efficient log parsing using *n*-gram dictionaries. *IEEE TSE* (2020).

[8] Hanjun Dai, Bethany Yixin Wang, Xingchen Wan, Bo Dai, Sherry Yang, Azade Nova, Pengcheng Yin, Phitchaya Mangpo Phothilimthana, Charles Sutton, and Dale Schuurmans. 2024. UQE: A Query Engine for Unstructured Databases. arXiv:2407.09522 [cs.DB]

[9] Min Du and Feifei Li. 2019. Spell: Online Streaming Parsing of Large Unstructured System Logs. *IEEE TKDE* (2019).

[10] Xi Fang, Weijie Xu, Fiona Anting Tan, Jiani Zhang, Ziqing Hu, Yanjun Qi, Scott Nickleach, Diego Socolinsky, Srinivasan Sengamedu, and Christos Faloutsos. 2024. Large Language Models(LLMs) on Tabular Data: Prediction, Generation, and Understanding – A Survey. arXiv:2402.17944 [cs.CL] https://arxiv.org/abs/2402.17944

[11] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In *ICDM*.

[12] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. 2024. Retrieval-Augmented Generation for Large Language Models: A Survey. arXiv:2312.10997 [cs.CL] https://arxiv.org/abs/2312.10997

[13] Google LLC. 2025. *Google Security Operations*. https://cloud.google.com/security/products/security-operations Accessed: 2025-04-12.

[14] Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation. arXiv:1905.08205 [cs.CL] https://arxiv.org/abs/1905.08205

[15] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. 2016. LogMine: Fast pattern recognition for log analytics. In *CIKM*.

[16] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R. Lyu. 2018. Towards Automated Log Parsing for Large-Scale Log Data Analysis. In *IEEE TDSC*.

[17] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. 2017. Drain: An Online Log Parsing Approach with Fixed Depth Tree. In *ICWS*.

[18] Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2024. Next-generation database interfaces: A survey of llm-based text-to-sql. *arXiv preprint arXiv:2406.08426* (2024).

[19] Junjie Huang, Zhihan Jiang, Zhuangbin Chen, and Michael R Lyu. 2024. LUNAR: Unsupervised LLM-based Log Parsing. arXiv:2406.07174 [cs.SE]

[20] Yintong Huo, Yuxin Su, Cheryl Lee, and Michael R Lyu. 2023. SemParser: A semantic parser for log analytics. In *ICSE*.

[21] Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R Lyu. 2024. LILAC: Log parsing using LLMs with adaptive parsing cache. *FSE* (2024).

[22] Zhihan Jiang, Jinyang Liu, Junjie Huang, Yichen Li, Yintong Huo, Jiazhen Gu, Zhuangbin Chen, Jieming Zhu, and Michael R Lyu. 2024. A large-scale evaluation for log parsing techniques: How far are we?. In *ISSTA*.

[23] Zhihan Jiang, Jinyang Liu, Junjie Huang, Yichen Li, Yintong Huo, Jiazhen Gu, Zhuangbin Chen, Jieming Zhu, and Michael R. Lyu. 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* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 223–234. doi:10.1145/3650212.3652123

[24] Zhen Ming Jiang, Ahmed E Hassan, Parminder Flora, and Gilbert Hamann. 2008. Abstracting execution logs to execution events for enterprise applications (short paper). In *QSIC*.

[25] Egil Karlsen, Xiao Luo, Nur Zincir-Heywood, and Malcolm Heywood. 2024. Benchmarking Large Language Models for Log Analysis, Security, and Interpretation. *Journal of Network and Systems Management* (2024).

[26] Van-Hoang Le and Hongyu Zhang. 2023. Log parsing with prompt-based few-shot learning. In *ICSE*.

[27] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *NeurIPS*.

[28] Peng Li, Yeye He, Dror Yashar, Weiwei Cui, Song Ge, Haidong Zhang, Danielle Rifinski Fainman, Dongmei Zhang, and Surajit Chaudhuri. 2023. Table-GPT: Table-tuned GPT for Diverse Table Tasks. arXiv:2310.09263 [cs.CL] https://arxiv.org/abs/2310.09263

[29] Zhenhao Li, Chuan Luo, Tse-Hsun Chen, Weiyi Shang, Shilin He, Qingwei Lin, and Dongmei Zhang. 2023. Did we miss something important? studying and exploring variable-aware log abstraction. In *ICSE*.

[30] Shu Liu, Asim Biswal, Amog Kamsetty, Audrey Cheng, Luis Gaspar Schroeder, Liana Patel, Shiyi Cao, Xiangxi Mo, Ion Stoica, Joseph E. Gonzalez, and Matei Zaharia. 2025. Optimizing LLM Queries in Relational Data Analytics Workloads. arXiv:2403.05821 [cs.LG] https://arxiv.org/abs/2403.05821

[31] Yilun Liu, Yuhe Ji, Shimin Tao, Minggui He, Weibin Meng, Shenglin Zhang, Yongqian Sun, Yuming Xie, Boxing Chen, and Hao Yang. 2025. LogLM: From Task-based to Instruction-based Automated Log Analysis. arXiv:2410.09352 [cs.SE] https://arxiv.org/abs/2410.09352

[32] Yurong Liu, Eduardo Pena, Aecio Santos, Eden Wu, and Juliana Freire. 2024. Magneto: Combining Small and Large Language Models for Schema Matching. arXiv:2412.08194 [cs.DB]

[33] Yudong Liu, Xu Zhang, Shilin He, Hongyu Zhang, Liqun Li, Yu Kang, Yong Xu, Minghua Ma, Qingwei Lin, Yingnong Dang, et al. 2022. Uniparser: A unified log parser for heterogeneous log data. In *WWW*.

[34] Lipeng Ma, Weidong Yang, Sihang Jiang, Ben Fei, Mingjie Zhou, Shuhao Li, Mingyu Zhao, Bo Xu, and Yanghua Xiao. 2024. Luk: Empowering log understanding with expert knowledge from large language models. arXiv:2409.01909 [cs.SE]

[35] Zeyang Ma, An Ran Chen, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. 2024. LLMParser: An exploratory study on using large language models for log parsing. In *ICSE*.

[36] Zeyang Ma, Dong Jae Kim, and Tse-Hsun Chen. 2024. LibreLog: Accurate and Efficient Unsupervised Log Parsing Using Open-Source Large Language Models. arXiv:2408.01585 [cs.SE]

[37] Adetokunbo A.O. Makanju, A. Nur Zincir-Heywood, and Evangelos E. Milios. 2009. Clustering event logs using iterative partitioning. In *KDD*.

[38] Masayoshi Mizutani. 2013. Incremental mining of system log format. In *SCC*. IEEE.

[39] Meiyappan Nagappan and Mladen A Vouk. 2010. Abstracting log lines to log event types for mining software system logs. In *MSR*.

[40] Open Cybersecurity Schema Framework. 2025. OCSF Schema – Categories. https://schema.ocsf.io/1.4.0/.

[41] Open Cybersecurity Schema Framework. 2025. OCSF Schema – Data Types. https://schema.ocsf.io/1.4.0/data_types?extensions=.

[42] Marcel Parciak, Brecht Vandevoort, Frank Neven, Liesbet M. Peeters, and Stijn Vansummeren. 2024. Schema Matching with Large Language Models: an Experimental Study. arXiv:2407.11852 [cs.DB]

[43] Erhard Rahm and Philip A. Bernstein. 2001. A survey of approaches to automatic schema matching. *VLDB* (2001).

[44] Mikel Rodriguez, Raluca Ada Popa, Four Flynn, Lihao Liang, Allan Dafoe, and Anna Wang. 2025. A Framework for Evaluating Emerging Cyberattack Capabilities of AI. arXiv:2503.11917 [cs.CR] https://arxiv.org/abs/2503.11917

[45] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. arXiv:2109.05093 [cs.CL] https://arxiv.org/abs/2109.05093

[46] Eitam Sheetrit, Menachem Brief, Moshik Mishaeli, and Oren Elisha. 2024. ReMatch: Retrieval Enhanced Schema Matching with LLMs. arXiv:2403.01567 [cs.DB]

[47] Keiichi Shima. 2016. Length matters: Clustering system log messages using length of words. arXiv:1611.03213 [cs.OH]

[48] Splunk Inc. 2025. *Splunk*. https://www.splunk.com Accessed: 2025-04-12.

[49] Byungchul Tak and Wook-Shin Han. 2021. Lognroll: Discovering accurate log templates by iterative filtering. In *Middleware*.

[50] Liang Tang, Tao Li, and Chang-Shing Perng. 2011. LogSig: Generating system events from raw textual logs. In *CIKM*.

[51] Gemini Team and Google. 2023. Gemini: A Family of Highly Capable Multimodal Models. *arXiv preprint arXiv:2312.11805* (2023). https://arxiv.org/abs/2312.11805

[52] Risto Vaarandi. 2003. A data clustering algorithm for mining patterns from event logs. In *IPOM*.

[53] Risto Vaarandi and Hayretdin Bahsi. 2025. Using Large Language Models for Template Detection from Security Event Logs. arXiv:2409.05045 [cs.CR] https://arxiv.org/abs/2409.05045

[54] Risto Vaarandi and Mauno Pihelgas. 2015. LogCluster - A data clustering and pattern mining algorithm for event logs. In *CNSM*.

[55] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. arXiv:2203.11171 [cs.CL] https://arxiv.org/abs/2203.11171

[56] Xuheng Wang, Xu Zhang, Liqun Li, Shilin He, Hongyu Zhang, Yudong Liu, Lingling Zheng, Yu Kang, Qingwei Lin, Yingnong Dang, Saravanakumar Rajmohan, and Dongmei Zhang. 2022. SPINE: a scalable log parser with feedback guidance. In *ESEC/FSE*.

[57] Yi Xiao, Van-Hoang Le, and Hongyu Zhang. 2024. Demonstration-Free: Towards More Practical Log Parsing with Large Language Models. In *ASE*.

[58] Junjielong Xu, Ruichun Yang, Yintong Huo, Chengyu Zhang, and Pinjia He. 2024. DivLog: Log parsing with prompt enhanced in-context learning. In *ICSE*.

[59] Yongqin Xu, Huan Li, Ke Chen, and Lidan Shou. 2025. KcMF: A Knowledge-compliant Framework for Schema and Entity Matching with Fine-tuning-free LLMs. arXiv:2410.12480 [cs.CL]
[60] Siyu Yu, Pinjia He, Ningjiang Chen, and Yifan Wu. 2023. Brain: Log Parsing With Bidirectional Parallel Tree. *IEEE TSC* (2023).
[61] Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. 2018. SyntaxSQLNet: Syntax Tree Networks for Complex and Cross-DomainText-to-SQL Task. arXiv:1810.05237 [cs.CL] https://arxiv.org/abs/1810.05237
[62] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2019. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. arXiv:1809.08887 [cs.CL] https://arxiv.org/abs/1809.08887
[63] Xian Yu, Shengxi Nong, Dongbiao He, Weijie Zheng, Teng Ma, Ning Liu, Jianhui Li, and Gaogang Xie. 2024. LogGenius: An Unsupervised Log Parsing Framework with Zero-shot Prompt Engineering. In *ICWS*.
[64] Jing Zhang, Bonggun Shin, Jinho D. Choi, and Joyce C. Ho. 2021. SMAT: An Attention-Based Deep Learning Solution to the Automation of Schema Matching. In *ADBIS*.
[65] Wei Zhang, Hongcheng Guo, Anjie Le, Jian Yang, Jiaheng Liu, and Zhoujun Li. 2025. Lemur: Log parsing with entropy sampling and chain-of-thought merging. arXiv:2402.18205 [cs.SE]
[66] Yu Zhang, Mei Di, Haozheng Luo, Chenwei Xu, and Richard Tzong-Han Tsai. 2024. SMUTF: Schema Matching Using Generative Tags and Hybrid Features. arXiv:2402.01685 [cs.CL]
[67] Aoxiao Zhong, Dengyao Mo, Guiyang Liu, Jinbu Liu, Qingda Lu, Qi Zhou, Jiesheng Wu, Quanzheng Li, and Qingsong Wen. 2024. LogParser-LLM: Advancing efficient log parsing with large language models. In *KDD*.
[68] Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael R. Lyu. 2023. Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics . In *ISSRE*.

A Evaluation Queries

A.1 DHCP Log Queries

Assigned addresses for a given hostname	Description	Identifies IP addresses assigned to a specific host.
	Query	`assigned_ip exists AND log_host is laphroaig`
	Naive sub-string	`fgrep "bound to" | fgrep "laphroaig"`
List all server IPs	Description	Enumerates all DHCP server IP addresses that are not broadcast addresses.
	Query	`server_ip != 255.255.255.255`
	Naive sub-string	`fgrep "port" | fgrep -v "255.255.255.255"`
	Note	The word port seems to always be present next to server IPs
Track all log messages for a given transaction ID	Description	Follows the complete lifecycle of a specific DHCP transaction.
	Query	`transaction_id is 0x6520bf0e`
	Naive sub-string	`fgrep "xid=0x6520bf0e"`
List the MAC addresses used by a specific host	Description	Retrieves entries containing MAC address information for a particular hostname.
	Query	`mac_address exists and log_host is laphroaig`
	Naive sub-string	`fgrep "laphroaig" | fgrep "Listening"`
Entries with high renewal times	Description	Identifies DHCP leases with renewal times exceeding 1 day (86400 seconds).
	Query	`renewal_time > 86400`
	Naive sub-string	`fgrep "renewal"`
	Note	We cannot compare numbers so we can only search for lines that include renewal times.
Servers on non-standard ports	Description	Lists DHCP servers operating on ports other than the standard port 67.
	Query	`server_port is not 67`
	Naive sub-string	`fgrep "port" | fgrep -v "67"`
Specific client version usage	Description	Identifies log entries associated with a specific DHCP client version (3.0.1).
	Query	`client_version is 3.0.1`
	Naive sub-string	`fgrep "3.0.1"`
DHCPDISCOVER messages	Description	Lists clients that have sent DHCPDISCOVER messages.
	Query	`DHCP_message_type is DHCPDISCOVER`
	Naive sub-string	`fgrep "DHCPDISCOVER"`
XMT Renew messages	Description	Lists clients that have issued renewal requests.
	Query	`DHCP_message_type is Renew`
	Naive sub-string	`fgrep "Renew"`
Bad IP checksums	Description	Identifies packets with incorrect IP checksums.
	Query	`bad IP checksums`
	Naive sub-string	`fgrep "bad IP checksums"`

Table 7: DHCP Log Queries

A.2 SSHD Log Queries

Password authentication for root	Description	Identifies instances where the root account attempted to authenticate using a password.
	Query	`authentication_method is password and user_name is root`
	Naive sub-string	`fgrep "password" | fgrep "root"`
Unusual server ports	Description	Detects SSH servers operating on non-standard ports (not port 22).
	Query	`bind_port is not 22`
	Naive sub-string	`fgrep "port" | fgrep -v "22"`
Usage of specific key	Description	Tracks usage of a particular SSH key based on its fingerprint.
	Query	`key_hash is SHA256:iJC3O+heWZCsp5vkBo00csZY6bg1Ycx+VwGmmcFhEnc`
	Naive sub-string	`fgrep "SHA256:iJC3O+heWZCsp5vkBo00csZY6bg1Ycx+VwGmmcFhEnc"`
Root user SSH keys	Description	Retrieves all SSH key fingerprints associated with root user logins.
	Query	`key_hash exists and user_name is root`
	Naive sub-string	`fgrep "root" | fgrep "publickey"`
Activity from specific IP on specific date	Description	Monitors all SSH activity from IP 61.143.236.193 on September 25.
	Query	`remote_ip is 61.143.236.193 and log_timestamp > sept. 25 00:00:00 and log_timestamp < sept. 26 00:00:00`
	Naive sub-string	`fgrep "61.143.236.193" | fgrep "sept. 25"`
Non-SSH terminals for root user	Description	Identifies root logins through terminal types other than SSH.
	Query	`terminal_type is not ssh and user_name is root`
	Naive sub-string	`fgrep "root" | fgrep "tty" | fgrep -v "tty=ssh"`
Root sessions initiated by non-root accounts	Description	Detects when standard users escalate to root privileges.
	Query	`initiating_user_name is not root and user_name is root`
	Naive sub-string	`fgrep "user root" | fgrep -v "root("`
	Note	This matches the format of the main template that contains this information
“None” authentication attempts	Description	Identifies login attempts using the "none" authentication method.
	Query	`authentication_method is none`
	Naive sub-string	`fgrep "none"`
Activity for specific system and process	Description	Tracks SSH activity related to a specific process ID on a particular host.
	Query	`process_id is 4317 and log_host is LIPC003.intranet.local`
	Naive sub-string	`fgrep "4317" | fgrep "LIPC003.intranet.local"`
Host key mentions	Description	Finds log entries that reference host key files or paths.
	Query	`host_key_path exists`
	Naive sub-string	`fgrep "host key"`

Table 8: SSHD Log Queries

A.3 Audit Log Queries

Sudo/su usage by non-root users	Description	Identifies when standard users attempt to use sudo or su commands.
	Query	`(executable_path contains /sudo or executable_path contains /su) AND user_id is not 0`
	Naive sub-string	`fgrep "/su" | fgrep -v "uid=0"`
Denied write operations for rsync	Description	Detects when rsync processes are denied write permissions.
	Query	`avc_operation contains write and process_name contains rsync`
	Naive sub-string	`fgrep "write" | fgrep "rsync"`
Specific Target SELinux context	Description	Finds log entries with a specific target SELinux context.
	Query	`target_context is system_u:system_r:udev_t:s0-s0:c0.c1023`
	Naive sub-string	`fgrep "system_u:system_r:udev_t:s0-s0:c0.c1023"`
Devices that had denied calls to mount	Description	Identifies log entries related to mounting storage devices.
	Query	`device_name exists and process_name is "mount"`
	Naive sub-string	`fgrep "mount" | fgrep 'dev='`
Root user logins	Description	Captures all direct login events for the root user.
	Query	`audit_type is LOGIN and user_id is 0`
	Naive sub-string	`fgrep "LOGIN" | fgrep "uid=0"`
Audit rule removal	Description	Detects when audit rules are removed from the system.
	Query	`operation contains "remove rule"`
	Naive sub-string	`fgrep "remove rule"`
SELinux permissive mode setting	Description	Identifies when SELinux is set to permissive mode rather than enforcing.
	Query	`selinux_permissive is 1`
	Naive sub-string	`fgrep "permissive=1"`
Root directory as working directory	Description	Finds processes operating with root (/) as their current working directory.
	Query	`current_working_directory is "/" or current_working_directory is "/"`
	Naive sub-string	`fgrep "cwd=/ " and fgrep 'cwd="/"'`
Non-binary audit enabled flags	Description	Detects when the audit enabled flag is set to a value other than 0 or 1.
	Query	`audit_enabled is not 0 and audit_enabled is not 1`
	Naive sub-string	`fgrep "audit_enabled=" | fgrep -v "audit_enabled=1" | fgrep -v "audit_enabled=0"`
Remote SSH connections to specific host on specific date	Description	Tracks remote hosts that established SSH connections to a particular server on a specific date.
	Query	`audit_datetime >= Aug 3 and audit_datetime < Aug 4 and terminal contains ssh and remote_hostname exists and audit_host is perfc-380g8-01`
	Naive sub-string	`fgrep "perfc-380g8-01" | fgrep "Aug 3" | fgrep "terminal=ssh" | fgrep "hostname"`

Table 9: Audit Log Queries

A.4 Cron Log Queries

List all executed jobs on a host at a specific date	Description	Identifies entries with executable paths on a particular host within a specific date range.
	Query	`executable_path exists and log_timestamp >= 2017-07-14 and log_timestamp < 2017-07-15 and log_hostname is httpboot`
	Naive sub-string	`fgrep "CMD" | fgrep "2017-07-14" | fgrep "httpboot"`
Entries with scaling factor	Description	Locates log entries that contain scaling factor information.
	Query	`scaling_factor exists`
	Naive sub-string	`fgrep "factor"`
Specific process ID	Description	Finds entries related to a specific process ID.
	Query	`process_id is 24225`
	Naive sub-string	`fgrep "24225"`
CRON session openings for root	Description	Lists all session openings for the root user before a specific date.
	Query	`opened and username is root and log_timestamp < 2017-07-15`
	Naive sub-string	`fgrep "opened" | fgrep "root" | fgrep "2017-07-14"`
	Note	We cannot compare dates so we look for the day before
CRON session closings	Description	Lists all session closings before a specific date.
	Query	`closed and log_timestamp < 2017-07-15`
	Naive sub-string	`fgrep "closed" | fgrep "2017-07-14"`

Table 10: Cron Log Queries

A.5 Puppet Log Queries

Resource-specific failures for a host	Description	Retrieves failure reports about a specific Puppet resource on a particular host.
	Query	`puppet_resource is Service[galera] AND has failures AND log_hostname is controller1`
	Naive sub-string	`fgrep "Service[galera]" | fgrep "has failures" | fgrep "controller1"`
Revoked certificates	Description	Identifies logs reporting revoked certificates.
	Query	`certificate_common_name exists AND revoked`
	Naive sub-string	`fgrep "revoked"`
Specific error code on similar hosts	Description	Finds hosts with similar naming patterns experiencing a specific error code.
	Query	`error_code is 14 and log_hostname contains maca`
	Naive sub-string	`fgrep "14" | fgrep "maca"`
Host associated with specific request ID	Description	Identifies the host linked to a particular Puppet request ID.
	Query	`request_id is req-9ac8edb7-f81f-44a7-9f34-9a375e7df573`
	Naive sub-string	`fgrep "req-9ac8edb7-f81f-44a7-9f34-9a375e7df573"`
Interval value changes	Description	Tracks changes to interval values in Puppet configurations.
	Query	`attribute_name is interval and new_value exists`
	Naive sub-string	`fgrep "interval"`
Specific SQL password hash detection	Description	Checks if any SQL-related resources contain a specific password hash value.
	Query	`attribute_name is password_hash and new_value contains D602AB02F4227D3EBF5FE6EA0323BD6D586A7454 and reporting_resource contains sql`
	Naive sub-string	`fgrep "D602AB02F4227D3EBF5FE6EA0323BD6D586A7454" | fgrep "sql"`
Extended Puppet run durations	Description	Identifies Puppet runs that took longer than 1 hour (3600 seconds).
	Query	`run_time > 3600`
	Naive sub-string	`fgrep "catalog run"`
	Note	We cannot compare numbers without parsing
Non-localhost server connections	Description	Lists connections to non-localhost servers by a Puppet agent on a specific date.
	Query	`server_ip is not 127.0.0.1 and log_hostname is puma03 and log_timestamp >= Jan 8 and log_timestamp < Jan 9`
	Naive sub-string	`fgrep "127.0.0.1" | fgrep "puma03" | fgrep "Jan 8"`
Firewall persistence failures	Description	Identifies cases where firewall rules cannot be persisted.
	Query	`Unable to persist firewall rules`
	Naive sub-string	`fgrep "Unable to persist firewall rules"`
HTTP URL targets	Description	Lists log entries with HTTP URL targets.
	Query	`target_url contains http://`
	Naive sub-string	`fgrep "http://"`

Table 11: Puppet Log Queries