Virtual Knowledge Graphs for Federated Log Analysis

Kabul Kurniawan*
kabul.kurniawan@wu.ac.at
Vienna University of Economics and Business
Vienna, Austria

Andreas Ekelhart
aekelhart@sba-research.org
SBA Research
Vienna, Austria

Elmar Kiesling
elmar.kiesling@wu.ac.at
Vienna University of Economics and Business
Vienna, Austria

Dietmar Winkler
dietmar.winkler@tuwien.ac.at
Vienna University of Technology
Vienna, Austria

Gerald Quirchmayr
gerald.quirchmayr@univie.ac.at
University of Vienna
Vienna, Austria

A Min Tjoa
amin@ifs.tuwien.ac.at
Vienna University of Technology
Vienna, Austria

ABSTRACT

Security professionals rely extensively on log data to monitor IT infrastructures and investigate potentially malicious activities. Existing systems support these tasks by collecting log messages in a database, from where log events can be queried and correlated. Such centralized approaches are typically based on a relational model and store log messages as plain text, which offers limited flexibility for the representation of heterogeneous log events and the connections between them. A knowledge graph representation can overcome such limitations and enable graph pattern-based log analysis, leveraging semantic relationships between objects that appear in heterogeneous log streams. In this paper, we present a method to dynamically construct such log knowledge graphs at query time, i.e., without a priori parsing, aggregation, processing, and materialization of log data. Specifically, we propose a method that – for a given query formulated in SPARQL – dynamically constructs a virtual log knowledge graph directly from heterogeneous raw log files across multiple hosts and contextualizes the result with internal and external background knowledge. We evaluate the approach across multiple heterogeneous log sources and machines and see encouraging results that indicate that the approach is viable and facilitates ad-hoc graph-analytic queries in federated settings.

CCS CONCEPTS

• **Security and privacy** → *Intrusion/anomaly detection and malware mitigation*; *Vulnerability management*; • **Information systems** → *Query reformulation*; *Information extraction*; *Graph-based database models.*

KEYWORDS

Semantic Log Analysis, Virtual Log Graphs, Dynamic Log Extraction, Decentralized Log Querying, Forensics

ACM Reference Format:
Kabul Kurniawan, Andreas Ekelhart, Elmar Kiesling, Dietmar Winkler, Gerald Quirchmayr, and A Min Tjoa. 2021. Virtual Knowledge Graphs for Federated Log Analysis. In *The 16th International Conference on Availability, Reliability and Security (ARES 2021), August 17–20, 2021, Vienna, Austria.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3465481.3465767

1 INTRODUCTION

Log data analysis is a crucial task in cybersecurity, e.g., when monitoring and auditing systems, collecting threat intelligence, conducting forensic investigations of incidents, and pro-actively hunting threats [4]. Currently available log analysis solutions, such as *Security Information and Event Management (SIEM)* systems, support the process by aggregating log data as well as storing and indexing log messages in a central relational database [12]. Such databases, however, have limitations with respect to the ability to express relations between entities [16]. Without explicit links between log entries in various log sources, it is difficult to integrate the partial and isolated views on system states and activities reflected in the various logs and to contextualize, link, and query log data. In large-scale infrastructures, the central collection model is also bandwidth-intensive and computationally demanding [7, 8, 12].

In this paper, we propose a decentralized approach for log analysis that is flexible, knowledge-based, and scalable. Specifically, we introduce a method to execute federated, graph pattern-based queries on dispersed, heterogeneous raw log data by dynamically constructing virtual knowledge graphs [21, 22]. To this end, we introduce a method that *(i)* federates graph-pattern based queries across endpoints, *(ii)* extracts only potentially relevant log messages, *(iii)* integrates the dispersed log events into a common graph, and *(iv)* links them to background knowledge. All of these steps are executed at query time without any up-front ingestion and conversion of log messages.

A key advantage of the graph-based model is that it provides a concise, flexible, and intuitive abstraction for the representation of various relations – e.g., connections in networked systems, hierarchies of processes on endpoints, associations between users and services, and chains of indicators of compromise. These connections automatically link log messages that are related through

ARES 2021, August 17–20, 2021, Vienna, Austria
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9051-4/21/08…$15.00
https://doi.org/10.1145/3465481.3465767

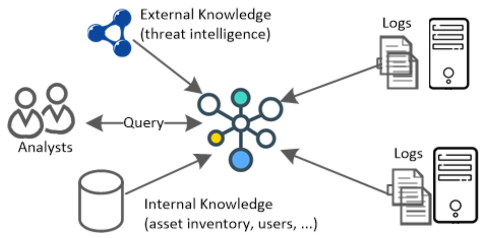


Figure 1: Concept overview

common entities (such as users, hosts, and IP addresses); they are crucial in cybersecurity investigations, as threat agent activities typically leave digital traces in various log files that are often spread across multiple endpoints in a network, particularly in discovery, lateral movement, and exfiltration stages of an attack [1].

In contrast to a traditional workflow that stores log messages in a centralized repository, the proposed approach shifts the log parsing workload from ingest to analysis time. This enables the use of the most granular, original raw log data without loss of information that would occur when pre-filtering and aggregating the logs before transferring them to a central archive, thus complementing existing log analysis approaches. Figure 1 illustrates the proposed approach for ad-hoc knowledge graph construction; the virtual log knowledge graph at the center of the figure is dynamically constructed from dispersed log sources based on analysts' queries and linked to external and internal knowledge sources.

To sum up, our contributions in this paper are as follows: We tackle current challenges in security log analysis (discussed in Section 3) by means of a Virtual Knowledge Graph (VKG) framework for federated log analysis that facilitates *(i)* ad-hoc integration and semantic analyses on raw log data without prior centralized materialization, *(ii)* the collection of evidence-based knowledge from heterogeneous log sources, *(iii)* automated linking of fragmented knowledge about system states and activities, and *(iv)* automated linking to external security knowledge (such as, e.g., attack patterns, threat implications, actionable advice).

The remainder of this paper is structured as follows: Section 2 provides an overview of related work; in Section 3, we discuss challenges in log analysis and derive requirements for our approach; Section 4 introduces the proposed architecture and components for virtual log knowledge graph construction; Section 5 describes our prototypical implementation and illustrates its use through example queries. Finally, we evaluate our approach in Section 6 and conclude with an outlook on future work in Section 7.

2 RELATED WORK

In this section, we discuss closely related work on the subjects of log management and analytics that we selected from the following categories that are particularly relevant in the context of our work[1]:

Centralized Security Log Analysis. A variety of conceptual approaches for centralized security log processing and correlation have been proposed in the literature. For forensic purposes, [2] proposes topological data analysis (TDA) to improve firewall forensics at the enterprise level. This approach employs statistical, graph-based, and visual methods for anomaly detection and interpretation. Like other similar approaches, it does not aim to integrate multiple log sources and to find links between them, but focuses on a single, highly structured, log source (i.e., firewall logs).

To learn and detect attack patterns from multiple log sources (i.e., firewall and web access log), [19] aggregates log sources into a centralized database before processing it in a rule correlation engine. Compared to our approach, the scope of existing approaches is typically limited and does not include linking to background knowledge. Contributions such as [17] propose log analysis approaches specific to cloud environments. In the latter, log events are identified and summarized before they are persisted into a centralized NoSQL database, where a Complex Event Processing (CEP) engine performs correlation and condensation. This approach specifically focuses on Syslog messages.

Somewhat closer to the approach in the present paper, [12] propose a hybrid relational-ontological architecture to overcome restriction in SIEMs (e.g., cross-domain, schema-complexity, scalability). The approach combines existing relational SIEM data repositories with external vulnerability information, i.e., Common Vulnerabilities and Exposures (CVE). The evaluation shows that the ontological approach can reduce the computation load compared to using a relational schema only.

Decentralized Security log analysis. Decentralized event correlation for intrusion detection was introduced in early work such as [13], where the authors propose a specification language to describe intrusions in a distributed pattern and use a peer-to-peer system to detect attacks. In this decentralized approach, the focus is on individual Intrusion Detection System (IDS) events only. To address scalability limitations of centralized log processing, [7] distributes correlation workloads across networks to the event-producing hosts. Similar to this approach, we aim to tackle challenges of centralized log analysis. However, we leverage semantic web technologies to also provide contextualization and linking to external background knowledge. In the cloud environment, [23] proposes a distributed and parallel security log analysis framework that provides analyses of a massive number of systems, networks, and transaction logs in a scalable manner. It utilizes the two-level master-slave model to distribute, execute, and harvest tasks for log analysis. The framework is specific to cloud-based infrastructures and lacks the graph-oriented data model and contextualization and querying capabilities of our approach.

Semantic Log Data Virtualization. As an example for semantic approaches, [10] leverages an ontology to correlate alerts from multiple IDSs with the goal to reduce the number of false-positive and false-negative alerts. It relies on a shared vocabulary to facilitate security information exchange (e.g., IDMEF, STIX, TAXII), but does not facilitate linking to other log sources that may contain indicators of attacks (e.g., authentication, file access, etc.).

To create a foundation for semantic SIEMs, [18] introduces a Security Strategy Meta-Model (SSMM) to enable interrelating information from different domains and abstraction levels in SIEMs. To facilitate log integration, contextualization and linking to background knowledge, [5] proposes a modular log vocabulary that enables log harmonization and integration between heterogeneous log

[1]Due to space restrictions, we can only make illustrative references.

sources. A recent approach proposed in [14] introduces a vocabulary and architecture to collect, extract, and correlate heterogeneous low-level file access events from Linux and Windows event logs. Using SPARQL queries, the extracted events can be constructed and linked to background knowledge. Compared to the approach in this paper, the approaches discussed so far rely on a centralized repository. A methodologically similar approach for log analysis outside of the security domain has also been introduced in [3], which leverages ontology-based data access to support log extraction and data preparation on legacy information systems for process mining. They focus on log data from legacy systems in existing relational schemas and on ontology-based query translation.

3 REQUIREMENTS

Existing log management systems typically ingest log sources from multiple log-producing endpoints and store them in a central repository for further processing. They then index and parse these logs collected from various sources before they can be analyzed. Therefore, such log management systems typically require considerable amounts of disk space to store the data as well as computational power for log analysis, which limits their scalability (due to concentrated network bandwidth, CPU, memory, and disk space requirements).

Decentralized log analysis, by contrast, (partly) shifts the computational workloads involved in log pre-processing (e.g., acquisition, extraction, and parsing) and analysis to the log-producing hosts [7]. This model has the potential for higher scalability and applicability in large-scale settings where the scope of the infrastructure prohibits effective centralization of all potentially relevant log sources in a single repository.

Existing approaches for decentralized log processing, however, primarily aim to provide correlation and alerting capabilities, rather than the ability to query dispersed log data in a decentralized manner. Furthermore, they lack effective means for semantic integration, contextualization, and linking, i.e., dynamically creating connections between entities and potentially involving externally available security information. They also typically have to ingest all log data continuously on the local endpoints, which may consume a lot of resources across the infrastructure.

In this paper, we tackle these challenges and propose a distributed approach for security log integration and analysis. Thereby, we facilitate ad hoc querying of dispersed raw log sources without prior central ingest and aggregation in order to address the following requirements (*R*):

- ***R.1 - Resource-efficiency*** avoid unnecessary log processing (acquisition, extraction, and parsing) and minimize resource requirements in terms of centralized storage space and network bandwidth.
- ***R.2 - Aggregation and integration over multiple endpoints*** ability to execute federated queries across multiple monitoring endpoints concurrently and deliver integrated results.
- ***R.3 - Integration, Contextualization & Background-Linking*** ability to contextualize disparate log information, integrate it, and link it to background knowledge and external security information.
- ***R.4 - Standards-based query language*** use of an expressive, standard based-query language for log analysis.

4 VIRTUAL LOG GRAPH ARCHITECTURE

Using the requirements set out in Section 3, we propose an approach and architecture for security log analytics based on the concept of VKGs. The proposed approach leverages Semantic Web Technologies that provide *(i)* a standardized graph-based data representation to describe data and their relationships flexibly using the Resource Description Framework (RDF)[2], *(ii)* semantic linking and alignment to integrate multiple heterogeneous log data and other resources (e.g., internal/external background knowledge), and *(iii)* a standardized semantic query language (i.e. SPARQL[3]) to retrieve and manipulate RDF data. SPARQL has a wide expressivity to perform complex querying (e.g., aggregation, subqueries, and negation).

To address **R.1**, our approach does not rely on centralized log processing, i.e., we only extract relevant log events based on the temporal scope and structure of a given query and its query parameters. Specifically, we only extract lines in a log file that *(i)* are within the temporal scope of the query, and *(ii)* may contain relevant information based on the specified query parameters and filters. The identified log lines are extracted, parsed, lifted to RDF, compressed, and temporally stored in a local cache on the respective endpoint. This approach implements the concept of data virtualization and facilitates on-demand log processing. By shifting computational load to individual monitoring agents and only extracting log entries that are relevant for a given query, this approach can significantly reduce unnecessary log data processing. Furthermore, due to the use of RDF-file compression technique, the transferred RDF data is rather small and does not require vast volumes of centralized storage. We discuss this further in Section 6.

Semantic web technologies also provide a mechanism for distributed querying through query federation[4]. We leverage this ability to query multiple log sources across distributed endpoints and to combine the results in a single integrated output, addressing **R.2**.

To address **R.3**, we interlink and contextualize our extracted log data using internal/external background knowledge (e.g., IT asset information and cybersecurity knowledge) via semantic linking and alignment.

Finally, we use SPARQL to formulate queries and perform log analysis against the processed log data, which addresses **R.4**. We show the application scenarios for SPARQL query federation and contextualization in Section 5.

Figure 2 illustrates the resulting virtual log graph and query federation architecture for log analysis; it comprises two main components: *(i)* a ***Query Processor***, which provides an interface to formulate SPARQL queries and distributes the queries among individual endpoints to retrieve, integrate, and present the resulting data sets (log graph) from each endpoint, and *(ii)* a ***Log Parser*** on each host, which receives and translates queries, extracts raw log

[2]https://www.w3.org/RDF/

[3]https://www.w3.org/TR/rdf-sparql-query/

[4]https://www.w3.org/TR/sparql11-service-description/

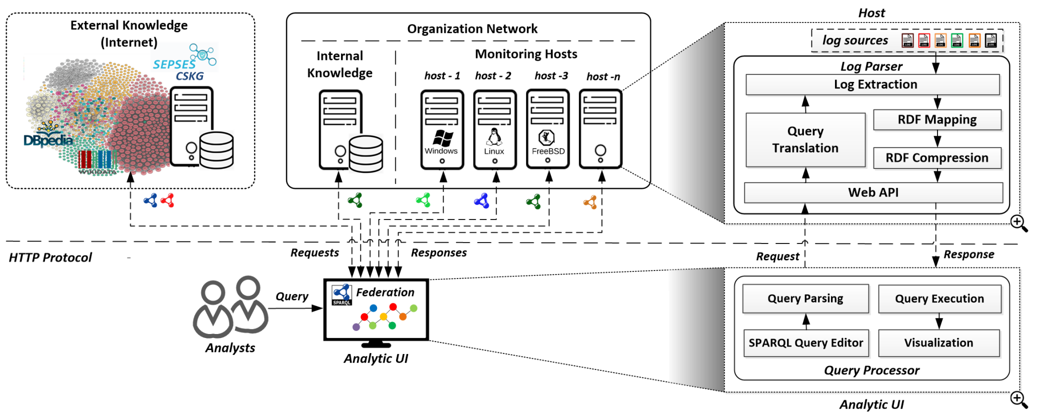


Figure 2: Virtual log graph and query federation architecture

data from hosts, parses the extracted log data into an RDF representation, compresses the resulting RDF data into a binary format, and sends the results back to the *Query Processor*. In the following, we explain the individual components in detail.

SPARQL Query Editor. This subcomponent is part of the *Query Processor* that allows users to formulate and execute SPARQL queries against hosts. The query editor allows analysts to define settings for query execution, including: *(i) Target Hosts*: a collection of endpoints which should be considered in the log analysis, *(ii) Knowledge bases*: a collection of internal and/or external sources of background knowledge that should be included in the query execution (e.g. IT infrastructure, CTI knowledge base, etc.), *(iii) Time Interval*: the time range of interest for the log analysis (i.e., start time and end time).

Query Parsing. Since the SPARQL query specification [9] provides a number of alternative syntaxes to formulate queries, we parse the raw SPARQL syntax into a structured format for easier access to the properties and variables inside the query, prior to transferring the query to the monitoring hosts. The prepared SPARQL query is then sent as a parameter to the *Query Translator* via the *Web API* in the *Log Parser* Component.

Query Translation. This subcomponent decomposes the SPARQL query to identify relevant properties for log source selection and log line matching. Algorithm 1 outlines the general query translation procedure, which identifies relevant log sources and log lines based on three criteria, i.e., *(i)* prefixes used in the query, *(ii)* triples, and *(iii)* filters.

$Prefixes(P)$ is a set of log vocabulary prefixes that appear in a given query Q. In each query, the contained prefixes will be used by the query translator to identify relevant log sources. Available prefixes can be configured to the respected log sources in the *Log Parser* configuration on each client, including, e.g., the path to the local log file location. As an example, `PREFIX auth: <http://w3id.org/authLog>` is the prefix for *AuthLog*; it's presence in a query indicates that the *AuthLog* on the selected hosts will be included in the log processing.

Algorithm 1: Query translation

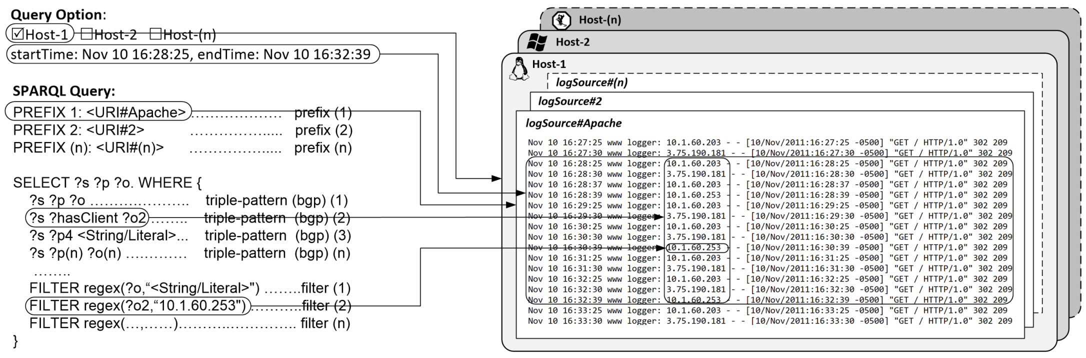
```
Input: SPARQL Query ($Q$), Vocabulary ($V$), regexPatterns ($RP$)
Output: QueryElements ($Qe$)
Prefixes $P = \{P_1,...,P_n\} \in Q$ ;
Triples $T = \{Subject, Predicate, Object\} \in Q$ ;
Filters $F = \{Variable, Value\} \in Q$;
Function translateQuery($Q$,$V$,$RP$):
    $P \leftarrow getPrefix(Q)$;
    $T \leftarrow getTriplePattern(Q)$;
    foreach Triple $T_i \in T$ do
        if $type(T_{i_{Object}})$=Literal then
            $logProperty \leftarrow getLogProperty(T_{i_{Predicate}},V,RP)$;
            $keyValue \leftarrow \{logProperty, T_{i_{Object}}\}$;
        end
        $triplesKV$ += $keyValue$;
    end
    $F \leftarrow getFilterStatement(Q)$;
    foreach Filter $F_i \in F$ do
        if $type(F_{i_{Value}})$=Literal then
            $predicate \leftarrow getPredicate(Q,F_{i_{Variable}})$;
            $logProperty \leftarrow getLogProperty(predicate,V,RP)$;
            $keyValue \leftarrow \{logProperty, F_{i_{Value}}\}$;
        end
        $filtersKV$ += $keyValue$;
    end
    $Qe \leftarrow \{P, triplesKV, filtersKV\}$;
    return $Qe$;
End Function
```

$Triples$ (T) is a set of triples that appear in a query, each represented as *Triple Pattern* or a *Basic Graph Pattern (BGP)* (i.e. `<Subject> <Predicate> <Object>`).

We match these triples to log lines (e.g., hosts and users) as follows: Function $getTriplePattern(Q)$ collects the triple patterns T contained within the query Q. For each triple statement in a query, we identify the type of `Object` $T_{i_{Object}}$. If the type is *Literal*, we identify the $T_{i_{predicate}}$ as well. For example, for the triple `{?Subject cl:originatesFrom "Host1"}`, the function $getLogProperty()$ identifies $T_{i_{Object}}$ "Host1", and additionally, looks up the property range provided in *regexPatterns* (*RP*).

regexPatterns (*RP*)[5] is the background knowledge that links property terms in a vocabulary to the terms in a log entry and

[5]https://github.com/sepses/VloGParser/blob/hdt-version/experiment/pattern/regexPattern.ttl



```
[✓ Host-1] ☐Host-2 ☐Host-(n)
startTime: Nov 10 16:28:25, endTime: Nov 10 16:32:39

PREFIX 1: <URI#Apache> ................... prefix (1)
PREFIX 2: <URI#2>      ................... prefix (2)
PREFIX (n): <URI#(n)>  ................... prefix (n)

SELECT ?s ?p ?o. WHERE {
   ?s ?p ?o .....................   triple-pattern (bgp) (1)
   ?s ?hasClient ?o2........   triple-pattern  (bgp) (2)
   ?s ?p4 <String/Literal>...   triple-pattern  (bgp) (3)
   ?s ?p(n) ?o(n) ............   triple-pattern  (bgp) (n)
    ........
   FILTER regex(?o,"<String/Literal>") ........filter (1)
   FILTER regex(?o2,"10.1.60.253")...........filter (2)
   FILTER regex(…,……)......................... filter (n)
}
```

```
Nov 10 16:27:25 www logger: 10.1.60.203 - - [10/Nov/2011:16:27:25 -0500] "GET / HTTP/1.0" 302 209
Nov 10 16:27:30 www logger: 3.75.190.181 - - [10/Nov/2011:16:27:30 -0500] "GET / HTTP/1.0" 302 209
Nov 10 16:28:25 www logger: 10.1.60.203 - - [10/Nov/2011:16:28:25 -0500] "GET / HTTP/1.0" 302 209
Nov 10 16:28:30 www logger: 3.75.190.181 - - [10/Nov/2011:16:28:30 -0500] "GET / HTTP/1.0" 302 209
Nov 10 16:28:37 www logger: 10.1.60.203 - - [10/Nov/2011:16:28:37 -0500] "GET / HTTP/1.0" 302 209
Nov 10 16:28:39 www logger: 10.1.60.253 - - [10/Nov/2011:16:28:39 -0500] "GET / HTTP/1.0" 302 209
Nov 10 16:29:25 www logger: 10.1.60.203 - - [10/Nov/2011:16:29:25 -0500] "GET / HTTP/1.0" 302 209
Nov 10 16:29:30 www logger: 3.75.190.181 - - [10/Nov/2011:16:29:30 -0500] "GET / HTTP/1.0" 302 209
Nov 10 16:30:25 www logger: 10.1.60.203 - - [10/Nov/2011:16:30:25 -0500] "GET / HTTP/1.0" 302 209
Nov 10 16:30:30 www logger: 3.75.190.181 - - [10/Nov/2011:16:30:30 -0500] "GET / HTTP/1.0" 302 209
Nov 10 16:30:39 www logger: 10.1.60.253 - - [10/Nov/2011:16:30:39 -0500] "GET / HTTP/1.0" 302 209
Nov 10 16:31:25 www logger: 10.1.60.203 - - [10/Nov/2011:16:31:25 -0500] "GET / HTTP/1.0" 302 209
Nov 10 16:31:30 www logger: 3.75.190.181 - - [10/Nov/2011:16:31:30 -0500] "GET / HTTP/1.0" 302 209
Nov 10 16:32:25 www logger: 10.1.60.203 - - [10/Nov/2011:16:32:25 -0500] "GET / HTTP/1.0" 302 209
Nov 10 16:32:30 www logger: 3.75.190.181 - - [10/Nov/2011:16:32:30 -0500] "GET / HTTP/1.0" 302 209
Nov 10 16:32:39 www logger: 10.1.60.253 - - [10/Nov/2011:16:32:39 -0500] "GET / HTTP/1.0" 302 209
Nov 10 16:33:25 www logger: 10.1.60.203 - - [10/Nov/2011:16:33:25 -0500] "GET / HTTP/1.0" 302 209
Nov 10 16:33:30 www logger: 3.75.190.181 - - [10/Nov/2011:16:33:30 -0500] "GET / HTTP/1.0" 302 209
```

Figure 3: SPARQL Query translation example

the respected regular expression pattern. For example, the property $cl : originatesFrom$ is linked to the concept `"hostname"` in $regexPattern$ (RP), which has a connected regex pattern for the extraction of host names.

The output of the $getLogProperty()$ function is a set of <$logProperty$, $T_{i_{Object}}$> key-value pairs.

Similar to triples, we also include $Filters$ (F) that appear in a query Q for log-line matching. Filter statements contain the term `FILTER` and a set of pairs (i.e., $Variable$ and $Value$), therefore each $Filter$ statement F_i has the members $Variable\ F_{i_{Variable}}$ and $Value\ F_{i_{Value}}$. Currently, we cover FILTER with simple pattern matching and regular expressions such as $FILTER$ ($?variable$ = "$StringValue$"), $FILTER\ regex(str(?variable), "StringValue")$). The function $getFilterStatement(Q)$ is used to retrieve these filter statements from the query and to identify the type of $Value\ F_{i_{Value}}$. If it is a Literal, the $getPredicate(Q)$ function will look for the connected $predicate$. Similar to the technique used in triples, we use $getLogProperty()$ to retrieve the regular expression defined in $regexPattern$ (RP).

Finally, the collected prefixes and retrieved key-value pairs, both from triples and filters, will be stored in $QueryElements$ (Qe) for further processing. Figure 3 depicts a SPARQL query translation example.

Log Extraction. This component is part of the *Log Parser* that extracts the selected raw log lines and splits them into a key-value pair representation by using predefined regular expression patterns. As outlined in Algorithm 2, Log sources (Ls) are included based on the prefixes that appear in the query.

For each log line (Ln_j) in a log source, we check whether the log timestamp ($LnO_{logTime}$) is within the defined TimeFrame (Tf).[6] If this condition is satisfied, the $matchLog()$ function checks the log-line property ($LnO_{logProperties}$) against the set of queried triples ($Qe_{triplesKV}$) and filters ($Qe_{filtersKV}$). If the log line matches the requirements, the selected log line will be parsed using $parseLine()$ based on predefined regular expression patterns. The resulting parsed queries will be accumulated and cached in a temporary file for subsequent processing.

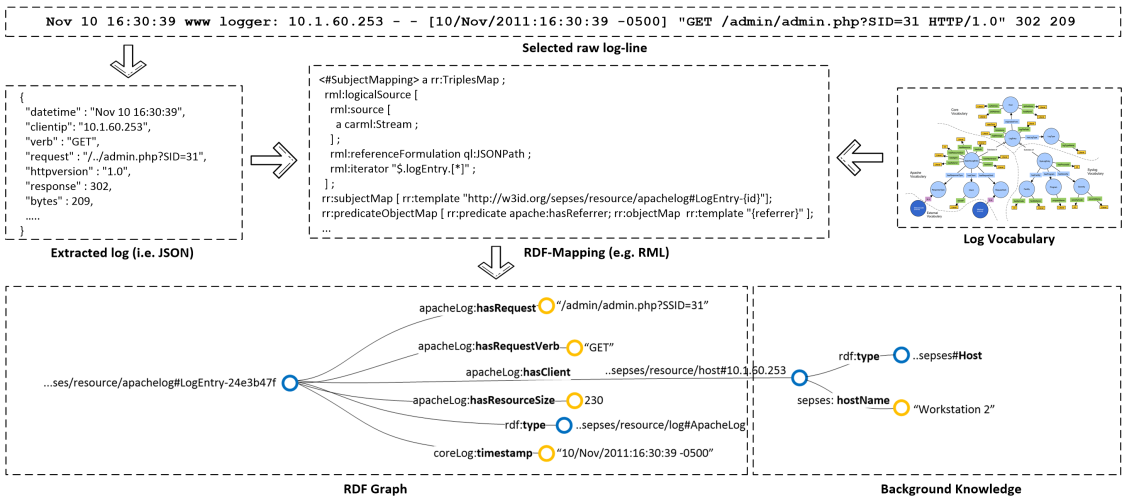
Algorithm 2: Log Extraction and RDF Mapping

```
Input: SPARQL Query ($Q$), TimeFrame ($Tf$), LogSources ($Ls$)
Output: Response ($R$)
TimeFrame $Tf = \{startT, endT\}$ ;
LogSources $Ls = \{Ls_1, \ldots, Ls_n\}$;
LogLines $Ln = \{Ln_1, \ldots, Ln_n\} \in Ls$;
LogSourceOptions $LsO = \{vocabulary, regexPatterns\} \in Ls$;
LogLineOptions $LnO = \{logTime, logProperties\} \in Ln$ ;
QueryElements $Qe = \{prefixes, triplesKV, filtersKV\}$;
$Qe \leftarrow translateQuery(Q, LsO_{vocabulary}, LsO_{regexPatterns})$;
foreach LogSource $Ls_i \in Ls$ do
    if $Qe_{prefixes}$ contains $LsO_{i_{vocabulary}}$ then
        foreach LogLines $Ln_j \in Ln$ do
            $lt \leftarrow LnO_{j_{LogTime}}$;
            if $lt$<$Tf_{endT} = False$ then
                break;
            end
            if $lt$>$Tf_{startT}$ && $lt$<$Tf_{endT}$ then
                $ml \leftarrow$
                  $matchLog(LnO_{j_{logProperties}}, Qe_{triplesKV}, Qe_{filtersKV})$;
                if $ml$=True then
                    $parsedLine \leftarrow parseLine(Ln_j)$;
                end
            end
            $parsedData$ += $parsedLine$;
        end
        $RDFData \leftarrow RDFMapping(parsedData)$;
        $result \leftarrow compressData(RDFData)$;
        if $result$=True then
            $response \leftarrow$ "$Success$";
        end
    end
    return $response$;
end
```

RDF Mapping. This subcomponent of the *Log Parser* maps and parses the extracted log data into RDF. It uses the standard RDF mapping language to map between the log data and the vocabulary. Different log sources use a common core log vocabulary (e.g., SEPSES coreLog[7]) for common terms (e.g., host, user, message) and can define extensions for specific terms (e.g., the `request` term in ApacheLog). The RDF Mapping also maps terms from a log entry to specific background knowledge (e.g., hosts in a log-entry are linked

[6] In this version of the algorithm, we leverage the monotonicity assumption in the log context by stopping the log parsing once the end of the temporal window of interest is reached in a log file (i.e., we assume that log lines do not appear out of order). This can be adapted, if required for a specific log source.

[7] https://w3id.org/sepses/vocab/log/core/



```
Nov 10 16:30:39 www logger: 10.1.60.253 - - [10/Nov/2011:16:30:39 -0500] "GET /admin/admin.php?SID=31 HTTP/1.0" 302 209
```

```
{
  "datetime" : "Nov 10 16:30:39",
  "clientip": "10.1.60.253",
  "verb" : "GET",
  "request" : "/../admin.php?SID=31",
  "httpversion" : "1.0",
  "response" : 302,
  "bytes" : 209,
  .....
}
```

```
<#SubjectMapping> a rr:TriplesMap ;
 rml:logicalSource [
  rml:source [
   a carml:Stream ;
  ] ;
  rml:referenceFormulation ql:JSONPath ;
  rml:iterator "$.logEntry.[*]" ;
 ] ;
rr:subjectMap [ rr:template "http://w3id.org/sepses/resource/apachelog#LogEntry-{id}"];
rr:predicateObjectMap [ rr:predicate apache:hasReferrer; rr:objectMap  rr:template "{referrer}" ];
...
```

Figure 4: Log graph generation overview

to their host type according to background knowledge). Figure 4 gives an overview of the log graph generation.

RDF Compression. After completion of the mapping phase, the resulting RDF data is compressed into a compact, binary format of RDF and sent to the *Query Processor* component that makes it available for querying.

Query Execution. Once the pre-processing on each target host has been completed and the compressed RDF data results have been successfully sent back to the *Query Processor*, a query engine executes the given queries against the compressed RDF data. If multiple hosts were defined in the query, the query engine will perform query federation over multiple compressed RDF data from those individual hosts and combine the query results to an integrated output.

Furthermore, due to semantic query federation, external data sources are automatically linked in the query results in case they were referenced in the query (cf. Section 5 for an example that links IDS messages to the SEPSES-CSKG[8]).

Visualization. Finally, this component presents the query results to the user; depending on the SPARQL query form[9], e.g.,: *(i)* SELECT - returns the variables bound in the query pattern, *(ii)* CONSTRUCT - returns an RDF graph specified by a graph template, and *(iii)* ASK - returns a boolean indicating whether a query pattern matches.

The returned result can be either in JSON or RDF format, and the resulting data can be presented to the user as an HTML table, chart, graph visualization, or it can be downloaded as a file.

[8]http://w3id.org/sepses/sparql

[9]https://www.w3.org/TR/sparql11-query/#QueryForms

5 IMPLEMENTATION & APPLICATION SCENARIOS

In this section, we discuss the implementation of our approach[10] and demonstrate its feasibility by means of two application scenarios: an intrusion detection scenario that integrates and links log sources with external security knowledge and a network monitoring scenario that demonstrate the use of internal background knowledge.

5.1 Implementation

We implement a *Log Parser*[11] component as a Java-based tool that is installed and run on each monitoring host. It supports log parsing from multiple heterogeneous log files (e.g. authlog, apachelog, IISlog, IDSlog) using log extraction patterns defined in *Grok Patterns*[12]. Furthermore, we used *CARML*[13] to map and parse log data into RDF and leverage the *HDT* [6] library to efficiently compress the resulting RDF data into a compact, binary format that allows query operations without prior decompression.

For the analysis interface, we implemented a *Query Processor*[14] component as a web-application that receives SPARQL queries, sends them to multiple target hosts, and presents the resulting graph to the analyst. The query execution is implemented on top of the *Comunica* [20] query engine that supports query federation over multiple linked data interface including HDT files and SPARQL endpoints.

5.2 Application Scenarios

Scenario I - Intrusion detection and background Linking. In this scenario, we illustrate the ability of our prototype to support

[10]Source code available at https://github.com/sepses

[11]https://github.com/sepses/VloGParser

[12]https://github.com/elastic/logstash/blob/v1.4.2/patterns/grok-patterns

[13]https://github.com/carml/carml

[14]https://github.com/sepses/VloGraphQueryProcessor

```
PREFIX cl: <https://w3id.org/sepses/vocab/log/core#>
  # other prefixes omitted for the sake of brevity
SELECT ?message ?sid ?cve ?impact ?cwe ?mitigation WHERE {
   ?s cl:timestamp ?timestamp. ?s cl:message ?message.
   ?s sa:signatureId ?sid. ?sid sr:hasRuleOption ?ro.
   ?ro sr:hasCVEReference ?cve. ?cve cve:hasCWE ?cwe.
   ?cve cve:hasCVSS2BaseMetric ?cbm. ?cbm cvss:baseScore ?impact.
   ?cwe cwe:hasPotentialMitigation ?cwepot.
   ?cwepot cwe:mitigationDescription ?mitigation.
} LIMIT 4
```

Listing 1: Snort alert linking query

contextualization and machine interpretation through automated interlinking between log sources and external background knowledge. This can, e.g., help security analysts to collect vulnerability information as well as actionable insights on potential mitigations.

We use an existing real-world data set of Snort alerts from MACCDC 2012[15] and place them on a monitoring host. Since individual IDS log-entries are linked to Snort rules[16](via Signature-ID), they can also be connected to vulnerability information (e.g. CVE [17]). Specifically, we link the IDS-snort alerts to the public cybersecurity knowledge graph (SEPSES-CSKG) which provides continuously updated cybersecurity information including CVE, CWE, CPE, CAPEC, etc. [11].

In this scenario, we assume that an analyst is interested in vulnerabilities and potential mitigations for all CVEs found in the IDS-snort log results. For this purpose, the analyst formulates a SPARQL query (Listing 1). Leveraging the link between the `?sid` object from IDS Snort alerts with the property `sr:hasCVEReference` and the value `?cve` from the background knowledge, the analyst can get explicit information about connected CVEs. Furthermore, they can also integrate other information such as Attack-`Impact`-Score from CVSS[18] (by using `cvss:baseScore`) and potential mitigations from CWE[19] (by using `cwe:hasPotentialMitigation`).

The integrated query results in Table 1 show the original Snort message, the included CVE number, impact scores, as well as potential mitigations from CWE, including their IDs and descriptions. Furthermore, Figure 5 shows the terms and their connection in a graph visualization.

Table 1: Snort alert linking query result (excerpt)

message	sid	cve	impact	cwe	mitigation
WEB-MISC...	2056	2004-2320	5.8	200	*Don't allow sensitive data...*
WEB-MISC...	2056	2010-0360	10	20	*Use automated static analy..*
WEB-MISC...	2056	2010-0360	10	20	*Use dynamic tools & tech...*
WEB-MISC...	2056	2010-0360	10	20	*Be especially careful to ...*

[15]https://maccdc.org/2012-agenda/

[16]https://www.snort.org/downloads/community/snort3-community-rules.tar.gz

[17]https://cve.mitre.org/

[18]https://www.first.org/cvss/

[19]https://cwe.mitre.org/

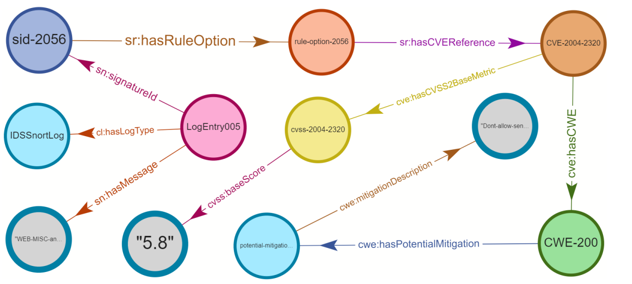


Figure 5: Visualization of Snort alert linking query results (excerpt)

```
PREFIX cl: <https://w3id.org/sepses/vocab/log/core#>
  # other prefixes omitted for the sake of brevity
SELECT ?timestamp ?user ?sourceIp ?targetHostType ?targetIp
WHERE {
   ?s cl:timestamp ?timestamp. ?s auth:hasUser ?user.
   ?s auth:hasSourceIp ?sourceIp. ?s auth:hasTargetHost ?th.
   ?s auth:hasAuthEvent ?ae. ?ae sys:partOfEvent ev:Login.
   ?th sys:hostType ?targetHostType.
   ?th cl:IpAddress ?targetIp.}
LIMIT 4
```

Listing 2: SSH connections query

***Scenario II - Network monitoring*.** In this scenario, we illustrate how our prototype provides semantic integration, generalization, and entity resolution. We simulated SSH login activities[20] across different servers (e.g., DatabaseServer, WebServer, FileServer) with multiple demo users (e.g., Bob and Alice) and then queried the authlog files with our federated approach.

Typically, atomic information on the log-entry level is not explicitly linked to semantic concepts, hence, we added extractors to e.g., detect certain log messages and map them to event types from our internal background knowledge[21] (e.g., `event:Login`, `event:Logout`). Furthermore, we added concept mappings for program names, IP addresses, etc. (cf. Section 4).

Now, an analyst can formulate a SPARQL query as shown in Listing 2 to extract successful login events from SSH connections. The query results in Table 2 and Figure 6 show successful logins via SSH over multiple hosts in the specified time range (from Dec 10 13:30:23 to Dec 10 14:53:06). The host type and target IP address come from internal background knowledge, as the host name is connected to a specific host type.

This information can be a starting point for security analysts to explore the rich context of the events in the virtual knowledge graph.

6 EVALUATION

We evaluated the scalability of our approach by means of a set of experiments in non-federated and federated settings.

[20]http://bit.ly/scenario2dataset

[21]https://w3id.org/sepses/knowledge/eventKnowledge.ttl

Table 2: SSH connections query result (excerpt)

timestamp	user	sourceIp	targetHostType	targetIp
Dec 10 13:30:23	Bob	172.24.66.19	DatabaseServer	192.168.2.1
Dec 10 13:33:31	Alice	172.24.2.1	WebServer	192.168.2.2
Dec 10 13:38:16	Alice	172.24.2.1	DatabaseServer	192.168.1.3
Dec 10 14:53:06	Bob	172.24.66.19	FileServer	192.168.2.4

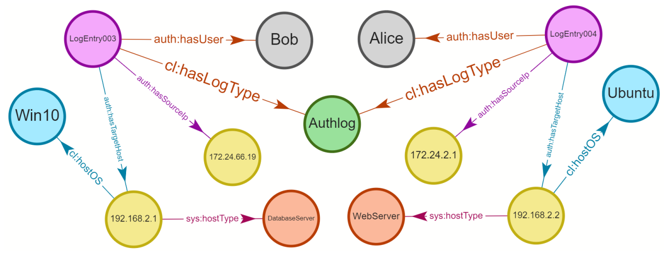


Figure 6: SSH connections query result visualization (excerpt)

6.1 Evaluation Setup

The experiments were carried out on Microsoft Azure virtual machines with seven hosts (4 Windows and 3 Linux) with 2.59 GHz vCPU and 16 GB RAM each. We reused the log vocabularies from [5] and mapped them to the log data.

Dataset Overview. We selected the systematically generated AIT log dataset (V1.1) that simulates six days of user access across multiple web servers including two attacks on the fifth day [15]. As summarized in Table 3, the dataset contains several log sources from four servers (*cup*, *insect*, *onion*, *spiral*). To reduce reading overhead and improve log processing performance, we split large log files from the data set into smaller files – this can easily be replicated in a running system using log rotation mechanisms. Specifically, we split the files into chunks of 10k–100k log lines each and annotated them with original filename and time-range information.

Table 3: Dataset description

LogType	#properties	*mail.cup.com*		*mail.insect.com*		*mail.onion.com*		*mail.spiral.com*	
		size	#lines	size	#lines	size	#lines	size	#lines
Audit	36	25 GB	123.6 M	22.7 GB	99.9 M	14.6 GB	68.8 M	12.4 GB	59.5 M
Apache	12	36.9 MB	148 K	44.4 MB	169.3 K	22.7 MB	81.9 K	24 .8 MB	100.4 K
Syslog	6	28.5 MB	158.6 K	26.9 MB	150.7 K	15 MB	86.6 K	15.1 MB	85.5 K
Exim	11	649 KB	7.3 K	567 KB	6.2 K	341 KB	3.9 K	355 KB	4 K
Authlog	11	128 KB	1.2 K	115 KB	1.1 K	102 KB	1 K	127 KB	1.2 K

6.2 Single-host evaluation

We measured the overall time for virtual log graph processing including *(i)* log reading (i.e., searching individual log lines), *(ii)* log extraction (i.e., extracting the raw log line into structured data), *(iii)* RDF Mapping (i.e., transforming json data into RDF), and *(iv)* RDF compression (i.e., compressing RDF into Header, Dictionary, Triples (HDT) format).

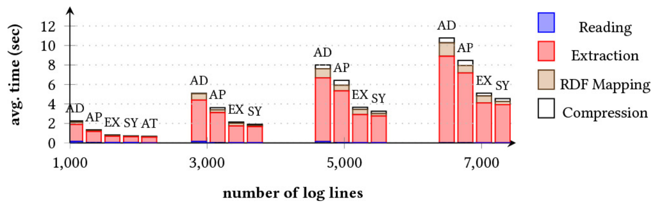


Figure 7: Average log graph generation time for n log lines with a single host (36 extracted properties)

In our scenarios, we included several log sources; for each log source, we formulated a SPARQL query[22] to extract 1k, 3k, 5k, and 7k log lines filtering by timestamp in the query option. We report the average times over five runs for experiments with several log sources – i.e., Auditlog (AD), Apache for web logs (AP), Exim for mail transfer agent logs (EX), Syslog for Linux system logs (SY), and Authlog for authentication logs (AT) – for a single host in Figure 7. We used the data set from the first web server (i.e., *mail.cup.com*) in this evaluation. Note that we only extracted 1000k log lines from Authlog due to the small original file size (less than 1.2 k log lines).

We found that the performance for log graph extraction differs across the log sources. Constructing a log graph from Auditlog (AD) data resulted in the longest processing times followed by Apache, Exim, Syslog and Authlog. The overall log processing time scales linearly with the number of extracted log lines. Typically, the log extraction phase accounts for the largest proportion (> 80%) of the overall log processing time. Furthermore, we found that the increase in log processing time with a growing number of extracted log lines is moderate, which suggests that the approach scales well to a large number of log lines.

Dynamic Log Graph Generation. As discussed in the first part of the evaluation, execution times are mainly a function of the length of text in the log source and the granularity of the extraction patterns (i.e., log properties). As can be seen in Table 3, the log sources are heterogeneous and exhibit different levels of complexity. In our setup, Auditlog, for instance, has the largest number of log properties (36), followed by Apache (12), Exim (11), Authlog (11), and Syslog (6).

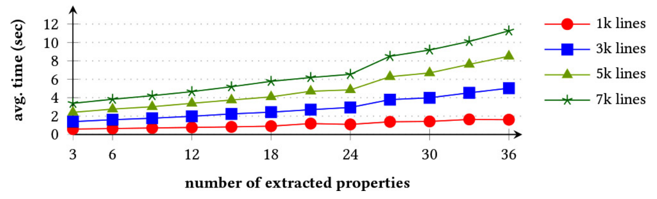


Figure 8: Dynamic log graph generation time[23]

Figure 8 shows an evaluation of log graph generation performance with respect to the complexity of the log source. We use the *Auditlog* for this evaluation as it has the highest number of log

[22]https://github.com/sepses/VloGraphQueryProcessor/tree/hdt-client-version/public/queries

[23]Experiments carried out on AuditLog data on a single host.

properties. Overall, the log graph generation performance grows linearly with the number of extracted log properties. Hence, queries that involve a smaller subset of properties (e.g., only *user* or *IP address* rather than all information that could potentially be extracted) will typically have smaller generation times.

Graph Compression. Figure 9 shows the performance for log graph compression on the *Auditlog* dataset. We performed full property extraction (i.e., all 36 identified properties) against 5k, 10k, 15k, and 20k log-lines, respectively, and compare the original size of raw log data, the generated RDF graph in TURTLE[24] format (.ttl), and the compressed graph output in HDT format.

For 5k log lines (1 MB raw log) compression results in approximately 0.4 MB compared to 5.4 MB for the uncompressed RDF graph. 20k log lines (4 MB raw log) compresses to about 1.87 MB from 21.4 MB uncompressed generated RDF graph. Overall, the compressed version is typically less than half the size of the original raw log and 10x smaller than the generated RDF graph. The resulting graph output would be even smaller for fewer extracted properties, minimizing resource requirements (i.e. storage/disk space).

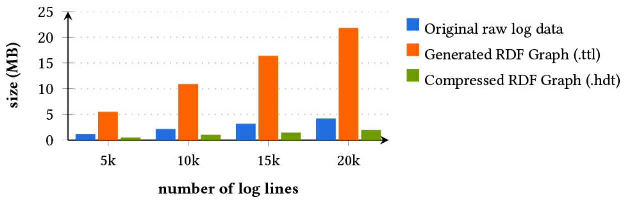


Figure 9: Graph compression

6.3 Multi-host evaluation

To evaluate the scalability of our approach, we measure the log processing time for multiple hosts on the same network. This evaluation includes not only the log processing but also the query federation performance. Federation means that the queries are not only executed concurrently, but that they involve evaluating and combining individual query results to achieve integrated results.

Table 4 summarizes the evaluation setup that consists of six experiments ranging from 30 minutes up to 5 hours. The timeframe describes the starting time and the end time of analysis; log lines per host summarizes the range of log lines per host within the timeframe. For this evaluation, we used the Apache log dataset described in Table 3 and conducted the analysis within the log timeframe of March 2nd, 2020, starting from 8pm. Host 1 to host 4 store the data from the original 4 servers in the dataset (host 1 *mail.cup.com*, host 2 *mail.insect.com*, and so on); for the 3 additional hosts in the evaluation, we replicated the log files from *mail.cup.com*, *mail.insect.com*, and *mail.spiral.com*. Similar to the single-host evaluation, for each experiment, we reported the average times over five runs.

[24]https://www.w3.org/TR/turtle/

Table 4: Multihost Experiment Timeframe

Experiment	Duration	Log lines per host	Experiment	Duration	Log lines per host
E1	30min	0.7k - 1k	E4	3h	3k - 5k
E2	1h	1k - 1.7k	E5	4h	6k - 8k
E3	2h	2.8k - 4k	E6	5h	8k - 10k

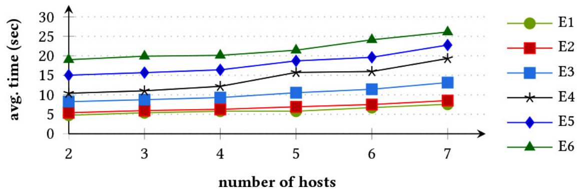


Figure 10: Query execution time in a federated setting for different time frames[25]

Figure 10 shows the average log processing times for each experiment. The 1 hour experiment shows that log processing for two hosts takes approx. 4.7 seconds on average. In the same experiment, the time slightly increases with an increasing number of hosts and reaches a max. of 7.5 seconds. The log processing time for the 5 hours experiment with two hosts takes approx. 19.01 seconds on average and reaches the max. average time of 26.10 seconds with 7 hosts. Based on these results, we conclude that the growth of the log processing time as a function of the number of hosts is moderate. Therefore, this approach scales well with a growing number of hosts to monitor, as the log processing on each host is parallelized and the query federation overhead is low.

7 CONCLUSIONS

In this paper, we presented a novel approach for distributed ad-hoc log analysis. Our approach extends the Virtual Knowledge Graph (VKG) concept – originally proposed in the context of relational data – and provides integrated access to (partly) unstructured log data. In particular, we proposed a federated method to dynamically extract, semantically lift and link named entities directly from raw log files. In contrast to traditional approaches, this method only transforms the information that is relevant for a given query, instead of processing all log data centrally in advance. Thereby, it avoids scalability issues associated with the central processing of large amounts of rarely accessed log data.

To explore the feasibility of this approach, we developed a prototype and demonstrated its application in two common log analysis tasks in security analytics. Furthermore, we conducted a performance evaluation which indicates that the total log processing time is primarily a function of the number of extracted (relevant) log lines and queried hosts, rather than the size of the raw log files. Our prototypical implementation of the approach provides scalability when facing larger log files and an increasing number of monitoring hosts.

[25]Evaluation of linking to background knowledge stored on external servers is out of scope.

Although this distributed ad-hoc querying has multiple advantages, we also identified a number of limitations. First, log files are parsed on demand and not indexed; hence, query parameters should restrict the extracted log lines to keep the processing time manageable (e.g., currently based on time frames). Second, the knowledge-based ad-hoc analysis approach presented in this paper is intended to complement, but does not replace traditional log processing techniques. A typical motivation for shipping logs to dedicated central servers, for instance, is to reduce the risk of undetected log tampering in case hosts in the network have been compromised. File integrity features could help to spot manipulations of log files, but particularly for auditing purposes, the proposed approach is not meant to replace secure log retention policies and mechanisms. Finally, while out of scope for the proof of concept implementation, the deployment of the concept in real environments requires traditional software security measures such as vulnerability testing, authentication, secure communication channels, etc.

In future work, we plan to improve the query analysis, e.g., to automatically select relevant target hosts based on the query and asset background knowledge. Furthermore, we will explore the ability to incrementally build larger knowledge graphs based on a series of consecutive queries in a step-by-step process. Finally, an interesting direction for research that would significantly extend the scope of potential use cases is a streaming mode that could execute continuous queries, e.g., for monitoring and alerting purposes. We plan to investigate this aspect and integrate and evaluate stream processing engines in this context.

REFERENCES

[1] 2019. ATT&CK Matrix for Enterprise. https://attack.mitre.org/

[2] Trevor J Bihl, Robert J Gutierrez, Kenneth W Bauer, Bradley C Boehmke, and Cade Saie. [n.d.]. Topological Data Analysis for Enhancing Embedded Analytics for Enterprise Cyber Log Analysis and Forensics. In *Cybersecurity and Privacy in Government*. 10. https://doi.org/10.24251/HICSS.2020.238

[3] Diego Calvanese, Tahir Emre Kalayci, Marco Montali, and Ario Santoso. 2017. OBDA for Log Extraction in Process Mining. In *Reasoning Web. Semantic Interoperability on the Web: 13th International Summer School 2017, London, UK, July 7-11, 2017, Tutorial Lectures*. Springer International Publishing, Cham, 292–345. https://doi.org/10.1007/978-3-319-61033-7_9

[4] Anton Chuvakin, Kevin Schmidt, and Chris Phillips. 2012. *Logging and log management: the authoritative guide to understanding the concepts surrounding logging and log management*. Newnes.

[5] Andreas Ekelhart, Elmar Kiesling, and Kabul Kurniawan. 2018. Taming the logs - Vocabularies for semantic security analysis. *Procedia Computer Science* 137, 109–119. https://doi.org/10.1016/j.procs.2018.09.011

[6] Javier D. Fernández, Miguel A. Martínez-Prieto, Claudio Gutiérrez, Axel Polleres, and Mario Arias. 2013. Binary RDF Representation for Publication and Exchange (HDT). *Web Semantics: Science, Services and Agents on the World Wide Web* 19 (2013), 22–41. http://www.websemanticsjournal.org/index.php/ps/article/view/328

[7] Michael R Grimaila, Justin Myers, Robert F Mills, and Gilbert Peterson. 2012. Design and Analysis of a Dynamically Configured Log-based Distributed Security Event Detection Methodology. *The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology* 9, 3 (July 2012), 219–241. https://doi.org/10.1177/1548512911399303

[8] Esther Palomar Guillermo Suárez de Tangil. 2013. *Advances in Security Information Management: Perceptions and Outcomes*. Nova Science Publishers, Incorporated, Commack, NY, USA.

[9] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. 2013. SPARQL 1.1 query language. *W3C recommendation* 21, 10 (2013), 778.

[10] Tayeb Kenaza and Mahdi Aiash. 2016. Toward an Efficient Ontology-Based Event Correlation in SIEM. *Procedia Computer Science* 83, 139–146. https://doi.org/10.1016/j.procs.2016.04.109

[11] Elmar Kiesling, Andreas Ekelhart, Kabul Kurniawan, and Fajar Ekaputra. 2019. The SEPSES Knowledge Graph: An Integrated Resource for Cybersecurity. In *The Semantic Web – ISWC 2019*. Vol. 11779. Springer International Publishing, Cham, 198–214. https://doi.org/10.1007/978-3-030-30796-7_13

[12] Igor Kotenko, Olga Polubelova, Andrey Chechulin, and Igor Saenko. 2013. Design and Implementation of a Hybrid Ontological-Relational Data Repository for SIEM Systems. *Future Internet* 5, 3 (July 2013), 355–375. https://doi.org/10.3390/fi5030355

[13] Christopher Krügel, Thomas Toth, and Clemens Kerer. 2002. Decentralized Event Correlation for Intrusion Detection. In *Information Security and Cryptology — ICISC 2001*, Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Kwangjo Kim (Eds.), Vol. 2288. Springer Berlin Heidelberg, Berlin, Heidelberg, 114–131. https://doi.org/10.1007/3-540-45861-1_10

[14] Kabul Kurniawan, Elmar Kiesling, Andreas Ekelhart, and Fajar Ekaputra. 2020. Cross-Platform File System Activity Monitoring and Forensics – A Semantic Approach. In *Hölbl M., Rannenberg K., Welzer T. (eds) ICT Systems Security and Privacy Protection. SEC 2020. IFIP Advances in Information and Communication Technology*. Springer, Cham.

[15] Max Landauer, Florian Skopik, Markus Wurzenberger, Wolfgang Hotwagner, and Andreas Rauber. 2021. Have it Your Way: Generating Customized Log Datasets With a Model-Driven Simulation Testbed. *IEEE Transactions on Reliability* 70, 1 (March 2021), 402–415. https://doi.org/10.1109/TR.2020.3031317

[16] Adam Oliner, Archana Ganapathi, and Wei Xu. 2012. Advances and Challenges in Log Analysis. *Commun. ACM* 55, 2 (Feb. 2012), 55–61. https://doi.org/10.1145/2076450.2076466

[17] Christian Pape, Sven Reissmann, and Sebastian Rieger. 2013. RESTful Correlation and Consolidation of Distributed Logging Data in Cloud Environments. In *The Eighth International Conference on Internet and Web Applications and Services*. 7.

[18] Julian Schütte, Roland Rieke, and Timo Winkelvos. 2012. Model-Based Security Event Management. In *Computer Network Security*. Vol. 7531. Springer Berlin Heidelberg, Berlin, Heidelberg, 181–190. https://doi.org/10.1007/978-3-642-33704-8_16

[19] Florian Skopik and Roman Fiedler. 2013. Intrusion Detection in Distributed Systems using Fingerprinting and Massive Event Correlation. In *GI-Jahrestagung*. 15.

[20] Ruben Taelman, Joachim Van Herwegen, Miel Vander Sande, and Ruben Verborgh. 2018. Comunica: A Modular SPARQL Query Engine for the Web. In *The Semantic Web – ISWC 2018*. Vol. 11137. Springer International Publishing, Cham, 239–255. https://doi.org/10.1007/978-3-030-00668-6_15

[21] Guohui Xiao, Diego Calvanese, Roman Kontchakov, Domenico Lembo, Antonella Poggi, Riccardo Rosati, and Michael Zakharyaschev. 2018. Ontology-Based Data Access: A Survey. In

Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence. International Joint Conferences on Artificial Intelligence Organization, Stockholm, Sweden, 5511–5519. https://doi.org/10.24963/ijcai.2018/777

[22] Guohui Xiao, Linfang Ding, Benjamin Cogrel, and Diego Calvanese. 2019. Virtual Knowledge Graphs: An Overview of Systems and Use Cases. *Data Intelligence* 1, 3 (2019), 201–223. https://doi.org/10.1162/dint_a_00011

[23] Xiaokui Shu, John Smiy, Danfeng Yao, and Heshan Lin. 2013. Massive distributed and parallel log analysis for organizational security. IEEE, 194–199. https://doi.org/10.1109/GLOCOMW.2013.6824985