Enabling Efficient Attack Investigation via Human-in-the-Loop Security Analysis

Saimon Amanuel Tsegai
Virginia Tech
saimon.tsegai@vt.edu

Xinyu Yang
Virginia Tech
xinyuyang@vt.edu

Haoyuan Liu
University of California, Berkeley
hy.liu@berkeley.edu

Peng Gao
Virginia Tech
penggao@vt.edu

ABSTRACT

System auditing is a vital technique for collecting system call events as system provenance and investigating complex multi-step attacks such as Advanced Persistent Threats. However, existing attack investigation methods struggle to uncover long attack sequences due to the massive volume of system provenance data and their inability to focus on attack-relevant parts. In this paper, we present Provexa, a defense system that enables human analysts to effectively analyze large-scale system provenance to reveal multi-step attack sequences. Provexa introduces an expressive domain-specific language, ProvQL, that offers essential primitives for various types of attack analyses (e.g., attack pattern search, attack dependency tracking) with user-defined constraints, enabling analysts to focus on attack-relevant parts and iteratively sift through the large provenance data. Moreover, Provexa provides an optimized execution engine for efficient language execution. Our extensive evaluations on a wide range of attack scenarios demonstrate the practical effectiveness of Provexa in facilitating timely attack investigation.

1 INTRODUCTION

Despite significant increases in spending on operations security, the frequency of modern targeted cyberattacks, such as Advanced Persistent Threats (APTs), continues to rise. Unlike traditional threats, these attacks are highly sophisticated, leveraging multiple vulnerabilities to infiltrate the system and exfiltrate sensitive data through a series of steps [50]. Consequently, many high-profile businesses have suffered massive data breaches and losses [14, 16, 25].

To counter these *intrusive multi-step attacks*, ubiquitous system auditing has emerged as a vital approach for monitoring attack footprints [12]. System auditing monitors system call events between system entities as system audit logs. Unlike application-level monitoring (e.g., Apache server logging), which only provides limited knowledge about specific applications and generates logs in different formats, system auditing is not tied to applications and generates audit logs with a consistent structure. The collected audit logs further enable the construction of a system provenance graph [37], in which nodes represent system entities (e.g., processes, files, network sockets) and edges represent their system call events (e.g., a process writes to a file). A system provenance graph provides a holistic view of all the activities in the system, which is particularly useful for investigating cyber attacks and uncovering attack steps [40, 45]. However, system auditing produces a huge amount of daily logs (e.g., 0.5 GB ~ 1 GB for one enterprise host [54]), resulting in a giant provenance graph. Additionally, the complexity of multi-step attacks poses significant challenges to existing system provenance-based defenses, which struggle to effectively uncover long attack sequences within such large provenance graphs.

Existing provenance-based attack investigation approaches often leverage causal dependency tracking [33, 40, 41, 43–45, 47]. These approaches model the control/data flow dependencies between system entities in a system event, track the dependencies from a Point-of-Interest (POI) event (e.g., an alert event like a process creating a suspicious file), and construct a dependency graph, which is a subgraph of the whole system provenance graph. Security analysts can inspect the dependency graph to reveal the attack sequence by reconstructing the chain of events that lead to the POI event. However, due to **the lack of fine-grained user control** of the tracking process, these approaches suffer from dependency explosion: the generated dependency graph is gigantic (containing >100K edges) and contains many system events that are *irrelevant* to the attack (e.g., events that load irrelevant system libraries). The problem is worse for multi-step attacks with long attack sequences, making it hard for security analysts to sift through the graph and identify the attack-relevant parts. Fig. 1 shows the dependency graph of a multi-stage data leakage attack and illustrates this problem: a small number of attack-relevant events are buried in an overwhelmingly large number of irrelevant events.

Goal and challenges. We aim to design and build a new defense system that (1) effectively filters out irrelevant system events and reveals complex attack sequences, and (2) efficiently analyzes large-scale system provenance data for a timely investigation. As reported

by many leading security vendors [21, 22], ***human-in-the-loop security*** emerges as a key paradigm for attack investigation. Humans bring unique strengths to cybersecurity, such as the ability to interpret subtle patterns, understand context, and make nuanced decisions that are difficult for automated systems alone. Such domain knowledge on expected system behaviors and malicious event patterns is crucial for filtering out irrelevant events and reducing dependency explosion as attacks become more sophisticated. Besides, attack investigation is an ***iterative process*** that involves multiple rounds of data exploration. An effective defense should provide a flexible and expressive interface for human analysts to incorporate the knowledge and customize the defenses for various attacks.

While several defenses have attempted to identify attack-relevant events from system provenance graphs, they have notable limitations. Multiple approaches seek to enhance dependency tracking fidelity through algorithms based on heuristic rules [33, 36, 45] that lead to information loss, or through binary instrumentation and kernel customization [43, 47] that introduce intrusive system changes. Several other works aim to detect malicious activities through subgraph matching, employing non-learning-based [49] and learning-based [26, 53] techniques. However, these approaches are computationally expensive, requiring extensive offline model training or incurring high runtime overhead over large provenance graphs. Learning-based methods also face generalization challenges, as their models can quickly become outdated with evolving system behaviors. Moreover, existing approaches overlook the importance of human-in-the-loop investigation, lacking flexible and efficient mechanisms to incorporate expert knowledge iteratively. Most methods assume investigations can be completed in a single round, ***providing limited support for step-by-step inquiries***—an essential process for thoroughly investigating complex attacks.

Contributions. In this work, we take an orthogonal approach to existing solutions. We introduce PROVEXA (∼15K LOC), a system that facilitates practical investigation of complex attacks through efficient human-in-the-loop provenance analysis. PROVEXA leverages system auditing frameworks and databases for provenance collection and storage. Its core contribution lies in the design of a powerful domain-specific language (DSL), called Provenance Query Language (PROVQL), which enables iterative, human-in-the-loop investigations over large-scale system provenance data. PROVQL treats system entities and events as first-class citizens and offers two primary query syntaxes for key investigative tasks: (1) *attack pattern search:* searching for complex event patterns indicative of malicious behaviors; (2) *attack dependency tracking:* tracking causal dependencies between events to uncover attack sequences and entry points. PROVQL provides rich constructs that allow users to constrain the search and tracking space to focus on critical parts while filtering out noise, mitigating dependency explosion.

PROVQL is expressive and intuitive to use. With its high-level, declarative syntax, PROVQL abstracts away the low-level complexities of data storage across different backends, enabling security analysts to focus on core attack behaviors rather than low-level details such as table joins. The two syntaxes offer complementary attack investigation capabilities. Search queries help locate potential suspects (e.g., POI events) for tracking and identify malicious events without dependencies. Tracking queries can uncover long

dependency chains that search queries cannot express. To further facilitate iterative investigations, PROVQL allows intermediate query results to be bound to variables and reused in subsequent queries, enabling analysts to refine their findings progressively.

Attack investigation is a time-critical mission to prevent further damage [45]. To efficiently execute PROVQL queries over large system provenance, PROVEXA employs a domain-aware query scheduler that decomposes the PROVQL query into small data queries and schedules their execution based on their pruning power, semantic dependencies, and domain characteristics. PROVEXA also features an in-memory management technique that maintains intermediate results bound to variables as in-memory graphs for subsequent manipulations and querying. Queries executing in this mode are much faster, facilitating iterative investigations.

Evaluation. We deployed PROVEXA on a testbed and extensively evaluated its effectiveness in reducing dependency explosion, uncovering attack sequences, and executing PROVQL queries efficiently. Additionally, we assessed PROVQL's usability and effectiveness through a user study. To conduct a thorough evaluation, we built a comprehensive benchmark by executing a wide range of attack scenarios on our testbed and collecting millions of real system events. We compared PROVEXA with multiple baselines, including state-of-the-art provenance-based attack investigation methods (BackTracker [40], PrioTracker [45], DepImpact [33]), general-purpose query languages (SQL [20], Cypher [19]), and a widely used industry attack investigation solution (Splunk [9]).

The results show that: (1) PROVEXA can accurately uncover the attack sequence in all scenarios while reducing dependency explosion, achieving 0.8766 F1-score and 58,991× graph reduction rate. This significantly outperforms existing defenses; BackTracker, PrioTracker, and DepImpact have only 0.2526, 0.2526, and 0.2604 F1-score, and 9×, 42×, and 24× reduction rate, respectively. (2) PROVEXA can efficiently execute PROVQL queries over massive provenance, running much faster than SQL and Cypher queries. (3) PROVQL features shorter query length with fewer constructs compared to SQL, Cypher, and Splunk, making investigation logic easier to express and maintain. (4) Our user study confirms PROVQL's advantages to streamline investigation workflows, reduce cognitive load, and enhance user experience. These results demonstrate that PROVEXA's superiority in combating sophisticated attacks.

2 BACKGROUND

Causal dependency tracking. Causal dependency tracking infers dependencies of system events and presents the dependencies as a directed dependency graph. In the dependency graph $G(V, E)$, a node $v \in V$ denotes a system entity (e.g., process, file, or network socket). An edge $e(u, v) \in E$ denotes a system event that involves two entities u and v (e.g., process creation, file read or write, and network access). Edge direction (from source node u to sink node v) indicates the information flow direction. For example, for a process reading data from a file, the file is the source and the process is the sink. For a process writing data to a file, the process is the source and the file is the sink. Each edge $e(u, v)$ is associated with a time window, $[ts(e), te(e)]$, where $ts(e)$ and $te(e)$ denote the start/end time of the event e. We adopt the definition consistent with previous studies [33, 40, 41, 43–45, 47] to infer edge directions for different
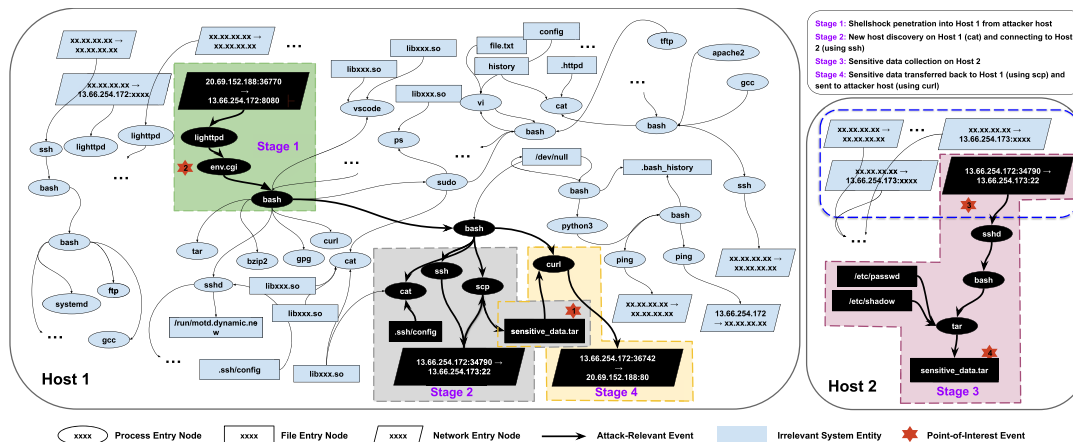
Fig. 1: System dependency graphs for a multi-stage, multi-host data leakage attack. The combined dependency graphs of the two victim hosts contain $100,524$ **nodes and** $154,353$ **edges. The attack-relevant nodes and edges, highlighted in dark black, comprise only 20 nodes and 20 edges, indicating the significant challenge of finding "a needle in a haystack".**

systems calls and event causal dependencies. Formally, for two events $e_1(u_1, v_1)$ and $e_2(u_2, v_2)$ (suppose e_1 occurs earlier than e_2), they have causal dependency if $v_1 = u_2$ and $ts(e_1) < te(e_2)$.

Causal dependency tracking, introduced in the BackTracker [40], enables two important security analyses: (1) *backward tracking* that identifies attack entry points, and (2) *forward tracking* that investigates attack ramifications. Given a POI event $e_{poi}(u, v)$, a backward tracking traces back from the source node u to find all events that have causal dependency on u, and a forward tracking traces forward from the sink node v to find all events on which v has causal dependency. The output of backward/forward tracking is a backward/forward dependency graph. Take backward tracking as an example. Starting from an empty queue Q and an empty graph G, we first add the POI event to Q and G. Then, we iteratively remove an event e from Q, find other events that have dependencies on e, and add these events to Q and G. We repeat this process until Q becomes empty, and the final G is the backward dependency graph. A major challenge in dependency tracking is dependency explosion. As shown in Section 7.3, *the dependency graphs produced remain excessively large and often miss critical attack-relevant events.*

Challenge: Needle in a haystack. Fig. 1 illustrates an example multi-stage data leakage attack across three hosts (one attacker host, two victim hosts Host 1 and 2) [2]. The attacker (20.69.152.188) identifies a victim Host 1 (13.66.254.172) in an enterprise network that is vulnerable to the lighttpd Shellshock [13] exploit. To steal data from the victim Host 1, the attacker launches a series of attack steps divided into four stages: (1) *Stage 1:* The attacker leverages Shellshock vulnerability in the lighttpd web server on Host 1 to penetrate into Host 1 from the attacker host. (2) *Stage 2:* The attacker discovers all the reachable hosts of Host 1 and connects to the discovered one, Host 2, using ssh. (3) *Stage 3:* The attacker uses tar to pack the sensitive files (e.g., /etc/passwd, /etc/shadow) on Host 2 as sensitive_data.tar. (4) *Stage 4:* The attacker uses scp on Host 1 to fetch the data sensitive_data.tar from Host 2. The attacker then uses curl on Host 1 to send the sensitive data back to the attacker host. We can observe that attack investigation faces a significant

challenge of finding "a needle in a haystack": a small number of critical attack-relevant edges (20; colored in dark black) are buried in an overwhelmingly large number (154K) of irrelevant edges. The same imbalance observation holds for attack-relevant nodes (20 vs. 101K). This makes attack investigation particularly challenging.

3 SYSTEM OVERVIEW
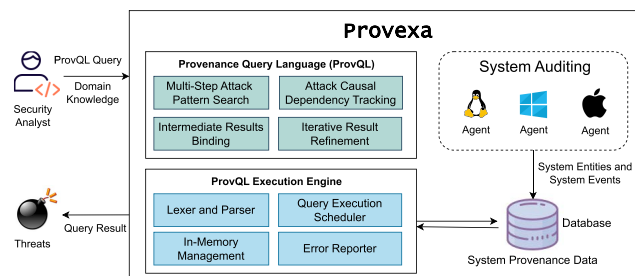
Fig. 2: Architecture of PROVEXA

Fig. 2 illustrates the system architecture. PROVEXA uses monitoring agents deployed on hosts to collect audit logs. PROVEXA then parses the logs into a sequence of system events among system entities and sends the parsed data to the database for storage. On top of the database, PROVEXA provides a DSL, PROVQL, for investigating attack behaviors. PROVQL integrates a collection of constructs for attack pattern search and attack dependency tracking analyses, as well as constructs for iteratively refining intermediate investigation results. Given a PROVQL query input by the security analyst, the language parser performs syntax analysis and semantic analysis of it. The execution scheduler then generates an execution plan based on our specialized optimizations and schedules the execution.

To enhance the user experience in iteratively exploring system provenance, we developed a user interface (UI) on top of our query execution engine. Inspired by computational notebooks like Jupyter Notebook, our UI features code cells and inline outputs. Users can construct queries within these cells. Executing a cell displays the

Table I: Representative entity attributes collected

System Entity	Attributes
File	Name, Path, User, Group
Process	PID, Executable Name, User, Group, Command Line Args
Network Socket	SRC/DST IP, SRC/DST Port, Protocol

Table II: Representative event attributes collected

Type	Operation Type (e.g., Read/Write/Execute/Start/End/Rename)
Time	Start Time, End Time, Duration
Misc.	Subject_ID, Object_ID, Data Amount

corresponding provenance graphs, and users can build upon results from previous cells to progressively deepen their investigation. Executed cells also serve as a record of the investigation process. A demo video showing how users can leverage our UI to iteratively investigate complex attacks is available at our project website [24].

Our threat model is similar to that of many previous works on system auditing [33, 40, 41, 43–45, 47]. We assume the presence of an attacker seeking to attack the system from outside: the attacker may seek to access or modify unauthorized resources, exfiltrate confidential data, or install and spread malware. Our trusted computing base includes OS kernels, host agents, data storage, and query execution engine. We assume that OS kernels and collected audit logs are secure from compromise. We do not consider malicious administrators who can disable the host agent or tamper with system audit logs, or implicit information flows like covert and timing channels which do not go through kernel-layer auditing.

4 SYSTEM AUDITING INFRASTRUCTURE

Data model and monitoring agents. System auditing collects system-level events about system calls from the OS kernel. These events describe the interactions among various system entities. As shown in previous studies [36, 38, 40, 45, 47], on mainstream OSes, system entities are primarily *files, processes, and network sockets*, and system calls are mapped to three major types of system events: (1) file access, (2) processes creation and destruction, and (3) network access. Thus, in our data model, we primarily consider these system entities. We consider a system event as the interaction between two system entities represented as ⟨subject_entity, operation_type, object_entity⟩. Subject entities are processes from software applications and objects can be files, processes, and network sockets. We categorize system events into three types according to the types of their object entities: file events, process events, and network events.

We develop monitoring agents using system auditing frameworks for different OSes: Sysdig [11] for Linux, Procmon [23] for Windows. Deployed on each host, our agent continuously monitors system activities, collects system audit logs, and extracts attributes critical for security analysis. Tables I and II show representative entity and event attributes that our agent extracts. Following previous works [33, 36, 45], to uniquely identify entities, for a process entity, we use the process executable name and PID as its identifier. For a file entity, we use the absolute path as its identifier. For a network socket entity, we use the 5-tuple (source/destination IP, source/destination port, protocol) as its identifier.

Data storage. Provexa stores the parsed system entities and system events in the databases so that the collected provenance data

can be persisted. The current implementation of Provexa supports two types of data storage backends: relational database PostgreSQL [8] and graph database Neo4j [5]. This enables Provexa to leverage the services these mature infrastructures provide, such as data management, indexing mechanism, querying, and data recovery. We perform data normalization to reduce redundancy and improve maintainability. In PostgreSQL, Provexa stores system entities and events across six separate tables, with attributes organized into columns (i.e., file, process, and network entity tables; file event, process event, and network event tables). In Neo4j, Provexa represents system entities as nodes and system events as edges (i.e., file, process, and network entity nodes; file event, process event, and network event edges). Entities can be linked to events by matching the Entity_ID attribute with the Subject/Object_ID attributes of events. Indexes are created on key attributes (e.g., file name, process executable name, source/destination IP) to speed up the search.

5 PROVQL LANGUAGE DESIGN

ProvQL integrates critical primitives for attack investigation, supporting two major types of analysis: multi-step attack pattern search and attack dependency tracking. Fig. 3 presents the grammar.

5.1 Multi-Step Attack Pattern Search

The search syntax (i.e., Rule ⟨*search_stmt*⟩ in Fig. 3) allows users to specify multiple events with constraints on entity/event attributes or event relationships. This enables the search for multi-event patterns that represent complex, multi-step attack behaviors.

Take Query ❶ in Section 7.2 as an example. First, we specify a database (e.g., db(host1)) as the data source of the search. Next, we define three system entities with constraints on their types and attributes (e.g., e1{name="curl", type=process}). Then, we define two system events using the three entities (e.g., e2[read]–>e1, where the arrow indicates information flow direction) based on Rule ⟨*search_rel_expr*⟩. Besides the *structural relationship* that two events are connected by the same entity e1, we constrain their *temporal relationship*: the two events occur within one second (e.g., &&[<1s]). Together, these three entities and two events define a subgraph pattern depicting the attack behavior: using curl to transfer a sensitive tar file to an IP. The transfer is carried out by first reading data from the file and then writing the data to the socket. After defining the multi-event pattern, we bind the results to a variable (e.g., poi1), indicating the results are retained in memory. This variable can be used in subsequent queries (e.g., results of Query ❶ are used as start entities for tracking in Query ❷) for further refinement. Users can also visualize the results in our UI by displaying this variable.

5.2 Attack Dependency Tracking

The tracking syntax (i.e., Rule ⟨*track_stmt*⟩) provides constructs for various types of fine-grained control of the causal dependency tracking process, including: tracking direction (backward/forward), POI, optional tracking depth, and entity and event constraints. This enables security analysts to effectively mitigate the dependency explosion during tracking by prioritizing attack-relevant parts.

Take Query ❷ in Section 7.2 as an example. A backward tracking query is specified with a database (e.g., db(host1)) as the data source. The tracking begins with the execution results of the previous query,

where poi1 is the starting entity, which was defined in Query ❶ and bound to its results. To constrain the process, Query ❷ excludes entities with the name vscode (i.e., Rule ⟨*track_filter*⟩), which are most likely benign. The execution results of Query ❷ produce a backward dependency graph, which is stored in the variable g1 for display in the UI and can be used in subsequent queries.

```
⟨start⟩        ::= (⟨search_stmt⟩ | ⟨track_stmt⟩ | ⟨graph_op_stmt⟩ | ⟨display_stmt⟩)*

⟨search_stmt⟩ ::= (⟨id⟩ '=')? 'search' 'from' ⟨data_source⟩ 'where' ⟨search_node⟩
                  'with' ⟨search_rel_expr⟩ ';' ;
⟨search_node⟩ ::= ⟨id⟩ '{' ⟨expr⟩ '}'
              |   ⟨search_node⟩ ',' ⟨search_node⟩
⟨data_source⟩ ::= ⟨id⟩
⟨search_rel_expr⟩ ::= ⟨search_rel⟩
              |   ⟨search_rel⟩ ⟨search_rel_op⟩ ⟨search_rel⟩
⟨search_rel⟩  ::= ⟨id⟩ '[' event_op ']')? '->' ⟨id⟩
⟨search_rel_op⟩ ::= '&&' ('[' ⟨int⟩ ('m'|'s'|'ms') ']')?
              |   '||';

⟨expr⟩         ::= '(' ⟨expr⟩ ')'
              |   '!' ⟨expr⟩
              |   ⟨expr⟩ ('&&'|'||') ⟨expr⟩
              |   ⟨binary_expr⟩
⟨binary_expr⟩ ::= ⟨string_attribute⟩ ⟨eq_op⟩ (⟨string⟩ | ⟨entity_type_id⟩)
              |   ⟨numerical_attribute⟩ ⟨numerical_op⟩ ⟨int⟩
⟨string_attribute⟩ ::= 'type' | 'name' | 'path' | 'dstip' | 'srcip' | 'exename' |
                  'exepath'|'cmdline'|'optype'
⟨numerical_attribute⟩ ::= 'id' | 'srcid' | 'dstid' | 'starttime' | 'endtime' |
                  'amount'|'pid'|'srcport'|'dstport'
⟨event_op⟩    ::= 'read'|'write'|'execve'|'readv'|'writev'|'rename'|'fork'
              |   'clone'|'recvfrom'|'sendto'
⟨entity_type_id⟩ ::= 'process' | 'file' | 'network'
⟨eq_op⟩        ::= '='|'!='|'like'
⟨numerical_op⟩ ::= ⟨eq_op⟩|'>'|'>='|'<'|'<='

⟨track_stmt⟩  ::= (⟨id⟩ '=')? ⟨track_direction⟩ 'track' ⟨track_poi⟩ 'from'
                  ⟨data_source⟩ ⟨track_filter⟩ ⟨track_limit⟩? ';' ;
⟨track_direction⟩ ::= 'back'|'forward'
⟨track_poi⟩   ::= (⟨track_constraint_expr⟩ | ⟨id⟩)
⟨track_filter⟩ ::= ('include' ⟨type_track_filter⟩)? ('exclude' ⟨type_track_filter⟩)?
⟨type_track_filter⟩ ::= ('nodes'  ⟨track_constraint_expr⟩)?  (',')?  ('edges'
                  ⟨track_constraint_expr⟩)?
⟨track_constraint_expr⟩ ::= 'where' ⟨expr⟩
⟨track_limit⟩ ::= 'limit'('step' ⟨int⟩)?(',')?('time' ⟨int⟩)'s|min')?

⟨graph_op_stmt⟩ ::= ⟨id⟩ '=' ⟨graph_expr⟩ ';'
⟨display_stmt⟩ ::= 'display' ⟨graph_expr⟩ ';'
⟨graph_expr⟩  ::= ⟨id⟩
              |   '(' ⟨graph_expr⟩ ')'
              |   ⟨graph_expr⟩ ⟨graph_op⟩ ⟨graph_expr⟩
⟨graph_op⟩    ::= '|'|'&'|'-'
```

Fig. 3: BNF grammar of PROVQL

5.3 Intermediate Results Binding

PROVQL queries return a *subgraph* of system events as results. Search queries return a subgraph that matches the specified multi-event pattern, while tracking queries return a subgraph aligned with the dependency tracking constraints. Designed to help security analysts iteratively refine their investigation results, PROVQL supports *binding query results to variables* and retaining them in memory. This enables analysts to efficiently access and modify intermediate results through subsequent queries without reprocessing the entire dataset. This approach significantly reduces query execution time. For example, Query ❹ in Section 7.2 defines the variable g3 and binds it to the tracked subgraph. Query ❺ then uses g3 as the data source for its search, i.e., search operates on the in-memory g3 rather than retrieving data from the database again.

To flexibly manipulate results, PROVQL supports different in-memory graph operations through variables. In Query ❻, a union operation (g1 | g3) merges the previously identified segments of the attack sequence, with the resulting subgraph bound to g4. Operations such as graph intersection and difference are also supported.

6 PROVQL EXECUTION ENGINE

A straightforward but inefficient approach to executing a PROVQL query is to translate the *entire* query into a single semantically equivalent SQL or Cypher query for execution. Search queries can be translated into standard SQL or Cypher queries involving multiple joins and constraints. For example, a single-step PROVQL query that finds a curl process reading from a tar file can be translated into a SELECT query joining the process, file, and process event tables, with filters on process executable and file names. Tracking queries are inherently recursive, as they traverse causality chains to find all related past or future events. These can be translated into recursive SQL queries using the WITH RECURSIVE clause, or Cypher queries that match variable-length paths with filters.

However, this approach becomes inefficient when querying complex attack patterns over large system provenance. General query optimizers are not tailored to the system provenance domain. Multi-step search queries will translate into large SQL or Cypher queries with numerous joins and constraints, which could lead to unpredictable performance. Tracking queries face similar issues: recursive SQL requires full table scans and joins, while Cypher exhaustively matches all paths before filtering, as it lacks on-the-fly constrained tracking [6]. As analyzed in Section 7.5, this single translation strategy often results in inefficient query plans.

Domain-aware query scheduler. Rather than compiling an entire PROVQL query into a single inefficient SQL or Cypher query and relying on generic, domain-agnostic schedulers provided by underlying databases, we design a custom scheduler tailored to our domain. Our scheduler decomposes the query into smaller retrieval steps, each translated into a small SQL/Cypher query. This decomposition leverages domain-specific observations, such as common patterns in system provenance data, to identify meaningful subqueries that are efficient to execute. The resulting small queries are then executed in an order informed by the semantics and structure of the provenance domain, taking into account estimated pruning power and query dependencies to maximize overall efficiency.

For a multi-step search query, each event pattern (e.g., e2[read]–> e1, e1[write]–>e3 in Query ❶, indicating a single step) is translated into a small SQL or Cypher query. The SQL version joins two entity tables and one event table with constraints in the WHERE clause; the Cypher version matches two entity nodes and one event edge with filters. To prioritize execution, the scheduler assigns each pattern a pruning score based on its semantics–a higher score indicates greater potential to reduce the search space. In our implementation, we use a coarse heuristic: patterns with more constraints receive higher scores. Additionally, inspired by prior observations that file events are more frequent than process or network events [54], we assign lower scores to file event patterns. This approach is quick to compute and proves effective in practice, as shown in Section 7.4.

When scheduling execution, the scheduler prioritizes event patterns with higher pruning scores and enforces an order based on dependencies. For instance, if two event patterns are connected through the same entity, the scheduler executes the higher-scoring data query first and incorporates the returned event IDs as additional WHERE clause constraints in the other data query. This significantly reduces the search space of the second query.

For a tracking query, the scheduler maintains a queue during recursive causality tracking. For each event dequeued, it compiles its causal dependency conditions–such as information flow, timestamps, and constraints–into small, non-recursive SQL or Cypher queries with appropriate WHERE filters. These small queries are

executed to retrieve all dependent events and recursively update the queue. A key advantage of this approach is that it enables efficient *on-the-fly* constrained tracking, avoiding the overhead of retrieving all dependencies upfront and filtering afterward.

Note that Provexa's scheduler does not replace existing database schedulers and optimizations but acts as an additional optimization layer for the system provenance domain. Each intermediate data query is executed on the databases. Due to their small sizes, their execution is typically fast, and can benefit from the indexing and database optimizations. Importantly, our query decomposition strategy leverages domain knowledge of system provenance to guide the breakdown of queries into semantically meaningful investigative steps rather than applying arbitrary splitting. This design aligns with how real-world investigations unfold: each step typically examines a localized portion of the graph (e.g., immediate predecessors or successors), making global query evaluation unnecessary. As shown in our detailed analysis of query cost in Section 7.5, executing the query in small, targeted steps avoids bloated recursive unions, reduces join and memory overhead, and enables early pruning. This step-wise traversal mirrors the investigative workflow, supports effective index reuse, and yields advantages in both performance and semantic alignment. Our extensive evaluations in Section 7.4 demonstrate that our scheduler layer consistently improves query performance in different database settings.

In-memory management. Provexa allows users to bind the execution results of a ProvQL query to a variable, which can then be used as the data source for subsequent queries, avoiding the need to repeatedly fetch data from the database. This in-memory management technique keeps the resulting graph in memory, enabling faster access when the variable is reused. Security analysts can utilize this feature to refine a previously executed query by applying additional constraints, and changes will be applied to the maintained in-memory graph, pruning unnecessary events efficiently.

7 EVALUATION

We built Provexa (~15K lines of code) upon several tools: Sysdig [11] for Linux system auditing, Procmon [23] for Windows system auditing, PostgreSQL [8] and Neo4j [5] for data storage, ANTLR 4 [1] for the DSL, React for the UI, and Java for the system.

We conduct extensive evaluations to address key research questions regarding the effectiveness in uncovering attack sequences, query performance, cost, and conciseness, and the usability of our language tool in facilitating iterative attack investigations.

(RQ1) How effective is Provexa in reducing dependency explosion and uncovering attack sequences in diverse scenarios?

(RQ2) How efficient is Provexa's scheduler in executing ProvQL queries over large system provenance data?

(RQ3) How does Provexa's scheduler optimize query cost?

(RQ4) How concise are ProvQL queries?

(RQ5) How does ProvQL's usability compare to general languages?

7.1 Evaluation Setup

We deployed Provexa on a server equipped with an AMD EPYC 7313 (3.00GHz) CPU and 1TB of RAM, running 64-bit Ubuntu 20.04. We deployed monitoring agents on nine hosts (eight Linux and one Windows machines), which, together with the server, form a controlled local network. All machines are connected through a high-speed Ethernet switch, simulating a typical enterprise network topology with the server acting as a centralized log aggregation and analysis node. Attacks were carried out on these hosts, and the agents collected system events that were transmitted to the server.

We constructed a benchmark of 30 attack cases covering diverse strategies and complexity levels, including 23 executed in our own network and 7 drawn from widely used system audit logs datasets. During the attacks, hosts continued to support benign user activities (e.g., file editing, software development), ensuring that realistic background noise was captured alongside malicious activities.

- **Multi-step attacks** (8 cases, "*multistep_*"): We carried out these attacks across multiple Linux hosts. Each attack begins with an initial penetration from an external host, followed by malware infection, lateral movement across hosts, and data exfiltration. The attacks exploit different vulnerabilities (e.g., Netcat backdoors, Shellshock, and EternalBlue) based on known CVEs [2].
- **Malicious exploits** (10 cases, "*malicious_*"): We carried out these attacks on a Linux host based on widely used exploits reported in previous studies [36, 45].
- **Malware samples** (5 cases, "*malware_*"): We selected five malware samples from VirusSign and executed them on our Windows host to infect the system or create a backdoor.
- **DARPA TC cases** (3 cases, "*tc_*"): We selected three APT attack cases from the DARPA TC Engagement 5 data release [18]. These attacks, conducted by a red team in a controlled environment, span multiple hosts and consist of multiple stages. Each case involves three hosts, with system audit logs collected during the engagement to capture both benign and malicious activities.
- **ATLASv2 cases** (4 cases, "*atlasv2_*"): We selected four cases from the recent ATLASv2 Attack Engagement dataset [51], which provides realistic system logs with high-quality background benign noise. Two hosts were involved. Our cases comprise multiple steps and exploit various vulnerabilities in widely used software (e.g., Adobe Flash Player and Microsoft Word).

Table III lists all 30 cases. The collected logs contain 1,548,202 system entities and 58,388,158 events. Our evaluations span 20 hosts in total: 9 from our own setup, 9 from DARPA TC, and 2 from ATLASv2. Detailed case descriptions are available at [24].

7.2 Case Study: APT Attack Investigation

We demonstrate how ProvQL is used to investigate the data leakage case (i.e., multistep_data_leakage in Table III) described in Section 2. To avoid bias, we assume no prior knowledge of the attack: an author, uninvolved in constructing the attack, independently used ProvQL to iteratively uncover the attack sequence.

- *Query ❶:* We search for a malicious pattern on Host 1, where a sensitive tar file is transferred to an unknown IP. The results confirm the presence of this pattern and identify the file as sensitive_data.tar, confirming data exfiltration. We bound these events to an in-memory variable poi1 for further analysis.

```
1 poi1 = search from db(host1) where e1{name="curl",
      type=process}, e2{name="*.tar", type=file}, e3{
      type=network} with e2[read]->e1 &&[<1s] e1[write
      ]->e3;
2 display poi1;
```

Table III: 30 attack cases in our evaluation benchmark

Case ID	Case Name	OS
multistep_cmd_injection	Command-line injection	Linux
multistep_data_leakage	Data leakage after penetration	Linux
multistep_netcat_backdor	Leaving a netcat backdoor after penetration	Windows
multistep_password_crack	Password crack after penetration	Linux
multistep_penetration	Shellshock penetration	Linux
multistep_phishing_email	Phishing email	Windows
multistep_supply_chain	Supply chain attack	Linux
multistep_wannacry	WannaCry attack	Windows
malicious_wget	Using wget to download and execute script	Linux
malicious_illegal_store	Storing in another user's home directory	Linux
malicious_hide_file	Downloading and hiding malicious file	Linux
malicious_backdoor_dl	Downloading backdoor malware with noise	Linux
malicious_server_usr	Server user performing malicious actions	Linux
malicious_ssh_theft	Adding public key to another user's profile	Linux
malicious_gcc_crash	Using gcc to compile and run code that crashes the system.	Linux
malicious_scan_login	Nmap scan and login for ssh	Linux
malicious_pwd_reuse	Password decode with John the Ripper password cracker	Linux
malicious_student	Student changing midterm score on server	Linux
malware_autorun	Trojan.Autorun	Windows
malware_danger	Trojan.Danger	Windows
malware_hijack	Virus.Hijack	Windows
malware_infector	Virus.Infector	Windows
malware_sysbot	Virus.Sysbot	Windows
tc_fivedirections_1	05092019 1326 - FiveDirections 2 - Firefox Drakon APT Elevate Copykatz Sysinfo	Linux
tc_fivedirections_2	05172019 1226 - FiveDirections 3 - Firefox DNS Drakon APT FileFilter-Elevate	Linux
tc_theia	05152019 1448 - THEIA 1 - Firefox Drakon APT BinFmt-Elevate Inject	Linux
atlasv2_s1	Adobe Flash Exploit - CVE-2015-5122	Windows
atlasv2_s2	Adobe Flash Exploit - CVE-2015-3105	Windows
atlasv2_s3	Microsoft Word Exploit - CVE-2017-11882	Windows
atlasv2_s4	Microsoft Word Exploit - CVE-2017-0199	Windows

- ***Query ❷:*** We investigate the origin of the sensitive_data.tar file by tracing the dependencies of poi1 on Host 1, excluding benign entities to narrow the scope. This backward dependency graph reveals that an scp process created this file after reading data from an IP associated with Host 2. This confirms that the attacker used scp on Host 1 to transfer the sensitive file from Host 2.

```
1  g1 = back track poi1 from db(host1) exclude nodes
       where name like "vscode"; display g1;
```

- ***Query ❸:*** We further investigate how sensitive_data.tar was created on Host 2 by issuing a backward tracing query. The results show that a tar process read the password files /etc/passwd and /etc/shadow, then wrote the data to sensitive_data.tar, confirming the sensitive data collection step of the attack.

```
1  g2 = back track "sensitive_data.tar" from db(host2)
2      exclude nodes where name like "vscode";
3  display g2;
```

- ***Query ❹:*** We want to further identify the attack's entry point on Host 1 by tracing the backward dependencies of the malicious curl process, while filtering out non-critical processes such as ping. The backward dependency graph is bound to g3.

```
1  g3 = back track where exename like "curl" from db(
       host1) include nodes where not path like "ping";
2  display g3;
```

- ***Query ❺:*** Since the attacker is likely operating from a remote server using the same IP (i.e., 20.69.152.188, as revealed in Query ❶), we search for this IP in g3 (in-memory). This reveals a lighttpd process, which we bind to poi2. The results suggest that the attacker exploited a vulnerability in the lighttpd web server to initially compromise Host 1 from the attacker's host 20.69.152.188.

```
1  poi2 = search from g3 where e1{srcip="20.69.152.188"
       }, e2{type=process} with e1[read]->e2;
2  display poi2;
```

- ***Query ❻:*** With the entry points identified, we reconstruct the full attack sequence on Host 1. We merge g1 and g3 via a union operation, storing the result in g4. Forward dependencies from

Table IV: Effectiveness in uncovering attack sequences

Approach	F1 Score	Miss Ratio (MR)	Noise Ratio (NR)
BackTracker	0.2526	0.1855	0.7731
PrioTracker	0.2526	0.1855	0.7731
DepImpact	0.2604	0.3999	0.6578
PROVEXA	**0.8766**	**0.0525**	**0.1658**

the entry point poi2 are then traced within g4 and saved as g5. Intersecting forward and backward traces helps significantly reduce the backward dependency graph.

```
1  g4 = g1 | g3;
2  g5 = forward track poi2 from g4 exclude nodes where
       name like "cat"; display g5;
```

The dark black paths in Fig. 1 show the results of Query ❻, highlighting the critical attack steps with significantly reduced graph size. This demonstrates the effectiveness of our PROVQL in supporting iterative investigation of complex, multi-step attacks. Following a similar procedure, we construct PROVQL queries for each attack case and use them in evaluations.

7.3 RQ1: Attack Investigation Effectiveness

We evaluate PROVEXA's effectiveness in mitigating dependency explosion and uncovering attack sequences, by comparing it with one seminal method (BackTracker [40]) and two state-of-the-art techniques (Priotracker [45] and DepImpact [33]). BackTracker reconstructs event sequences leading to POIs by tracing dependencies, but suffers from explosion. PrioTracker improves this by prioritizing abnormal dependencies based on event rareness within a time limit. DepImpact assigns weights to edges using features like data flow and temporal relevance, propagates impact scores from POIs to entry nodes, and identifies critical paths by intersecting forward traces from top-ranked entry nodes with backward traces. We followed the original parameter settings in these papers and validated our reimplementations with the respective authors.

Mitigating dependency explosion. As in PrioTracker and DepImpact, we compute the ***Graph Reduction Rate (GRR)***, defined as the ratio of edges in the original provenance graph to those in the graph produced by the investigation. PROVEXA achieves the highest *GRR* of 58, 990.7×, significantly outperforming BackTracker (9.4×), PrioTracker (41.6×) and DepImpact (24.3×). Moreover, the graphs generated by PROVEXA are closest in size to the attack ground truth. BackTracker produces much larger graphs due to its lack of control. PrioTracker performs better than BackTracker in large TC cases due to its prioritization of events. For small cases, the graph generated by PrioTracker is the same as BackTracker, as the tracking can be finished within the time limit. However, it is hard to select a proper time limit across scenarios. Similarly, DepImpact offers limited user control. Its performance heavily depends on the choice of entry nodes: too few may miss critical paths, while too many introduce noise. This lack of flexibility limits DepImpact's effectiveness.

Uncovering attack sequences. We use metrics adopted from prior work [33, 45]: ***miss rate (MR)***, ***noise ratio (NR)***, and ***F1-score***. Miss rate, defined as $MR = FN/E_c$, measures the proportion of attack-relevant events that are not identified. Noise ratio, defined as $NR = FP/E_{total}$, measures the proportion of irrelevant events

included. F1-score is computed as $F1 = \frac{TP}{(TP+0.5\times(FP+FN))}$. TP is true positives. FP is false positives. FN is false negatives. E_c is the attack-relevant event count. E_{total} is the total event count.

Table IV presents the aggregated results. PROVEXA significantly outperforms existing approaches, achieving higher $F1$ and lower MR and NR, indicating its effectiveness in accurately revealing attack sequences while filtering out irrelevant events. BackTracker suffers from high NR due to dependency explosion, producing large graphs (averaging 2,022 nodes and 230,253 edges). It also has a higher MR because it excludes events occurring after the POI in its backward analysis. PrioTracker shows higher MR and NR than PROVEXA. While it generates smaller graphs than BackTracker in large TC cases (4.42× fewer edges), the results are still noisy (NR near 1). In smaller cases, it produces identical graphs to BackTracker. PROVEXA also outperforms DepImpact on both MR and NR, benefiting from PROVQL's customizable queries. DepImpact's performance is sensitive to the choice of entry nodes, which is difficult to optimize across cases. For instance, in *multistep_penetration*, it misses many initial access attempts, resulting in a high MR (0.93). While PROVEXA's performance is also influenced by parameter choices, its human-in-the-loop refinement through a DSL enables adaptive, case-specific tuning that reduces noise and improves accuracy.

7.4 RQ2: Query Execution Efficiency

We evaluate the **execution time** of PROVEXA through two complementary analyses. First, we perform an ablation study to quantify the performance gains from PROVEXA's in-memory management. Second, we compare query execution times under different database optimization settings to understand the performance gains from PROVEXA's domain-aware scheduler.

For each PROVQL query, we construct semantically equivalent SQL and Cypher queries. As detailed in Section 6, for search queries, we use SQL CROSS JOIN and Cypher MATCH to specify identical subgraph patterns. For tracking queries, we use SQL WITH RECURSIVE and Cypher's variable-length path matching with filters.

7.4.1 Impact of In-Memory Management. We evaluate PROVEXA's runtime performance with and without in-memory management using a benchmark inspired by DepImpact. Each attack case consists of a backward query from POI alerts, a search for key entry nodes, a forward query from those nodes, and an intersection of forward and backward traces to reduce the graph size. We compare four settings: *provexa-in*, *provexa-db*, *sql*, and *cypher*. *provexa-in* loads data from the database for the initial backward query and executes subsequent queries in memory. To ensure a fair comparison, we store intermediate results in temporary databases. *provexa-db* loads data from the original and temporary databases for each query. SQL and Cypher do not support in-memory execution on prior results, so we run semantically equivalent SQL and Cypher queries on the original and temporary databases (i.e., the same data as *provexa-db*).

We report the total runtime across all queries: *provexa-in* significantly outperforms *provexa-db*, *sql*, and *cypher*, with 37×, 41×, 68× speedups, respectively. This highlights the performance gains of PROVEXA's in-memory management.

Table V: PostgreSQL optimization configurations

Setting	Categories	Representative Parameters
Default	None	No changes
Light	Planner, memory	random_page_cost = 1.1 work_mem = 128 MB effective_cache_size = 12 GB default_statistics_target = 200
Moderate	+ parallelism and collapse	*Light settings, plus:* join_collapse_limit = 12 from_collapse_limit = 12 parallel_setup_cost = 100 parallel_tuple_cost = 0.01
Advanced	+ I/O, JIT	*Moderate settings, plus:* shared_buffers = 4 GB temp_buffers = 64 MB jit = on autovacuum_vacuum_scale_factor = 0.1
Final	Planner, memory, collapse	random_page_cost = 1.1 work_mem = 64 MB effective_cache_size = 8 GB join_collapse_limit = 8

7.4.2 Impact of Database Optimizations. PostgreSQL's default configuration is conservative, prioritizing stability and broad compatibility across low-resource environments. While not optimized for performance, we include this *Default* setup as a baseline to measure tuning benefits. In contrast, PROVQL's tracking and search queries are memory-intensive and involve complex joins. To assess how tuning alone can affect performance, we tested three additional configurations—*Light*, *Moderate*, *Advanced*—as shown in Table V. These incrementally adjust memory parameters, planner cost settings, join-collapsing thresholds, parallelism, and I/O behaviors.

We evaluated these configurations with our benchmark queries. The results show that while each tuning level offered incremental gains over the baseline, improvements were modest and inconsistent across workloads. This variability reflects the structural diversity of our benchmark queries. No single configuration consistently dominated. Therefore, we combined parameters to derive a configuration that achieved the lowest average query cost and execution time across all queries, denoted as the *Final* configuration. Detailed results for all tuning levels are available at [24].

7.4.3 Impact of Domain-Aware Scheduler. We evaluate the execution time gains from PROVEXA's scheduler layer across four configurations: *vanilla*, *optimized*, *vanilla-provexa*, and *optimized-provexa*. *vanilla* and *optimized* correspond to the *Default* and *Final* database configurations described in Section 7.4.2. *vanilla* reflects the out-of-the-box setup commonly seen in many deployments, while *optimized* represents a best-effort tuning. *vanilla-provexa* layers PROVEXA's domain-aware scheduler on top of the default engine, and *optimized-provexa* combines the scheduler with the tuned engine. We followed a similar tuning approach to Neo4j. PROVQL queries are executed in the *vanilla-provexa* and *optimized-provexa* setups, whereas semantically equivalent SQL and Cypher queries are used for *vanilla* and *optimized*. Each query is executed 10 times per configuration, with caches cleared and the databases restarted between runs to ensure fairness and eliminate residual effects.

Table VI shows the results. PROVEXA's scheduler consistently improves performance across all database configurations (comparing *vanilla-provexa* to *vanilla*, and *optimized-provexa* to *optimized*). For tracking queries, optimized PostgreSQL outperforms the vanilla setup, while Neo4j's tuning yields mixed results (improving in some cases, degrading in others). This highlights the inherent difficulty of crafting tuning strategies that generalize across all attack cases.

Table VI: Execution time (in milliseconds) for PostgreSQL and Neo4j with various optimization settings, both with and without Provexa's scheduler. Each query is executed for 10 rounds. The time-out threshold is 30 minutes. We exclude DARPA TC and ATLASv2 cases (marked with "-") from the Neo4j database evaluation due to their time-out in data loading. V - Vanilla DB, O - Optimized version of DB, VR - Vanilla with Provexa's scheduler, OR - Optimized version of DB with Provexa's scheduler.

Case	Tracking PostgreSQL V	O	VR	OR	Tracking Neo4j V	O	VR	OR	Search PostgreSQL V	O	VR	OR	Search Neo4j V	O	VR	OR
multistep_cmd_injection	176	236	2	2	2,879	2,538	852	769	46	50	52	56	671	576	579	569
multistep_data_leakage	684	631	6	3	18,781	18,104	2093	1733	323	265	301	286	802	701	795	694
multistep_netcat_backdor	524	570	5	5	28,513	28,247	3,323	2,898	17	17	18	18	473	400	448	414
multistep_password_crack	232	312	5	4	4,649	4,542	1,636	1,374	86	80	97	94	711	651	671	647
multistep_penetration	1,011	922	2	1	2,054	1,672	689	674	488	429	531	447	720	643	765	665
multistep_phishing_email	6,394	5,750	10	10	Time-out	Time-out	3,558	3,101	1,996	1,693	68	68	Time-out	Time-out	559	467
multistep_supply_chain	22	112	1	1	2,737	2,430	500	487	10	9	9	10	463	434	487	433
multistep_wannacry	318	416	1	2	1,248,315	1,301,313	2,194	1,904	159	161	157	166	506	457	631	450
malicious_wget	495	513	2	1	5,494	5,190	775	609	27,490	21,565	173	164	739	617	626	610
malicious_illegal_store	287	349	2	1	4,424	4,101	826	729	146	155	150	171	544	452	489	447
malicious_hide_file	515	528	1	1	2,224	1,965	675	522	139	150	144	159	428	443	489	452
malicious_backdoor_bl	558	564	2	1	5,577	5,062	1,038	857	149	150	156	162	554	443	517	443
malicious_server_usr	492	462	3	1	13,843	13,408	961	883	2,530	1,967	659	594	891	825	944	860
malicious_ssh_theft	287	323	1	1	2,166	1,893	810	687	130	104	130	112	737	717	836	683
malicious_gcc_crash	338	336	6	5	3,039	2,807	964	801	139	108	135	124	819	683	736	677
malicious_scan_login	77	166	4	3	738,173	713,940	1,230	1,081	8,682	7,287	10,602	9,354	686	617	719	616
malicious_pwd_reuse	78	173	5	5	54,134	54,025	786	667	62	64	56	65	509	461	504	437
malicious_student	12	105	3	2	1,801	1,520	908	771	10	11	11	11	526	431	527	439
malware_autorun	2,682	2,544	5	5	23,004	21,658	1,707	1,545	29	31	30	33	464	435	488	434
malware_danger	201,698	186,305	11	12	1,006,627	1,037,522	16,302	15,355	124	642	120	667	707	565	725	571
malware_hijack	112,522	102,836	18	11	1,141,741	1,162,804	6,983	6,395	154	147	160	156	592	537	682	523
malware_infector	10,268	9,569	9	9	545,064	536,059	3,231	2,761	102	96	77	69	621	599	724	587
malware_sysbot	5,958	5,367	9	8	181,274	277,693	2,958	2,546	40	42	47	46	594	450	527	440
tc_fivedirections_1	113,757	113,140	98	93	-	-	-	-	28,699	40,467	27,667	41,124	-	-	-	-
tc_fivedirections_2	128,568	134,492	304	264	-	-	-	-	52,200	68,633	41,702	42,454	-	-	-	-
tc_theia	75,956	71,837	301	245	-	-	-	-	11,682	14,216	11,307	14,577	-	-	-	-
atlasv2_s1	3,129	2,569	2	1	-	-	-	-	1,507	1,106	1,487	1,070	-	-	-	-
atlasv2_s2	3,531	3,110	2	1	-	-	-	-	1,182	909	1,198	840	-	-	-	-
atlasv2_s3	2,796	2,653	1	1	-	-	-	-	1,195	926	1,218	988	-	-	-	-
atlasv2_s4	1,565	1,412	3	3	-	-	-	-	1,233	931	1,091	919	-	-	-	-
Average	22,498	21,610	27	24	228,932	236,295	2,391	2,137	4,692	5,414	3,319	3,833	625	551	629	546

In contrast, the integration of Provexa's domain-aware scheduling yields significant speedups, reaching up to 900× in PostgreSQL and 110× in Neo4j when layered atop optimized engines.

For search queries, improvements are more modest. In PostgreSQL, the optimized setup slightly underperforms the vanilla version, further illustrating the challenge of identifying optimization parameters that consistently improve performance across diverse workloads. The introduction of Provexa nonetheless results in measurable gains–1.43× in PostgreSQL and 1.01× in Neo4j. These modest gains stem from the simplicity of some search queries, where native planners already perform well and the benefits of additional processing by Provexa's scheduler are less pronounced.

DuckDB database. We extend evaluation to DuckDB [3], a fast, in-process analytical database for single-node OLAP workloads. Unlike PostgreSQL and Neo4j, DuckDB offers a lightweight, embedded alternative. To assess Provexa's scheduler on DuckDB, we compare vanilla SQL against ProvQL queries executed using the scheduler. Queries ran on DuckDB's native file format, leveraging its columnar storage and vectorized execution engine. Caching was disabled, and results were averaged over 10 trials. The results show that Provexa delivers substantial performance gains for tracking queries and complex search queries in DuckDB (averaging 148ms vs. 1.4s with vanilla SQL, a 9.4× speedup), mirroring improvements seen on other backends. For simpler search queries, Provexa's scheduler provides less speedup, as direct SQL execution is already efficient.

Despite its fast performance, DuckDB exhibits limitations under investigative workloads involving large provenance data. In such settings, particularly during evaluation of DARPA TC scenarios (e.g., *tc_fivedirections_1*, including 225K nodes and 23M edges), DuckDB frequently ran out of memory and crashed during tracking queries.

In contrast, PostgreSQL remained stable and completed the same queries, albeit with higher execution times. Notably, Provexa's scheduler reduced PostgreSQL's runtime from 113 seconds to 93 milliseconds for this case, demonstrating both its scalability and resilience under heavier workloads.

Summary: The results show that across all cases, query types, database systems (PostgreSQL, Neo4j, DuckDB), and configuration settings (vanilla, optimized), adding Provexa's scheduling layer consistently improves performance. These gains are particularly evident for complex queries, where conventional database tuning often yields limited or inconsistent benefits. In contrast, Provexa provides reliable gains through a system provenance–aware decomposition and execution strategy that prioritizes high-pruning, semantically meaningful query steps. Overall, these results highlight Provexa's generalizability and robustness as a complementary optimization layer tailored to the system provenance domain.

7.5 RQ3: Cost Estimation

To further assess the improvements introduced by Provexa's scheduler upon the database planner, we analyze ***cost estimates*** across the same four configurations from Section 7.4.3: *vanilla*, *optimized*, *vanilla-provexa*, and *optimized-provexa*. We use EXPLAIN ANALYZE to extract PostgreSQL's internal cost estimates, which approximate expected resource consumption–lower values indicate more efficient plans. As Neo4j's PROFILE command does not expose comparable cost metrics, it is excluded from this experiment.

Table VII presents the cost estimates. While manual tuning of PostgreSQL yields modest performance gains, Provexa's scheduler achieves significantly greater improvements. Across both query types, Provexa consistently lowers cost estimates relative to the

Table VII: Cost estimation of PostgreSQL with and without Provexa's scheduler. Each query is executed for 10 rounds.

Case	Tracking				Search			
	Vanilla	Optimized	Vanilla-Provexa	Optimized-Provexa	Vanilla	Optimized	Vanilla-Provexa	Optimized-Provexa
multistep_cmd_injection	1.7E+14	6.1E+13	436	169	3,225	2,524	3,138	2,436
multistep_data_leakage	8.3E+18	2.8E+18	1,315	507	38,978	10,782	38,765	10,568
multistep_netcat_backdor	3.3E+8	1E+8	486	258	902	725	873	696
multistep_password_crack	2E+16	7.7E+15	1,096	419	5,225	3,938	5,139	3,851
multistep_penetration	1E+20	3.9E+19	1,045	375	71,395	19,587	71,004	19,195
multistep_phishing_email	1.6E+11	6E+10	1,389	769	13,074	4,504	10,455	2,958
multistep_supply_chain	7.4E+7	2.4E+7	163	102	596	464	582	450
multistep_wannacry	3.7E+6	2E+6	93	71	354	285	343	274
malicious_backdoor_bl	2.2E+15	7.2E+14	251	98	24,142	6,665	23,863	6,386
malicious_gcc_crash	5.9E+14	2.2E+14	710	523	18,197	4,988	18,079	4,870
malicious_hide_file	2.2E+15	7.4E+14	167	58	24,124	6,663	23,844	6,384
malicious_illegal_store	2.4E+15	8.1E+14	134	46	24,693	6,900	24,410	6,617
malicious_pwd_reuse	4.8E+11	1.67E+11	391	271	3,456	2,692	3,376	2,612
malicious_scan_login	6E+11	2E+11	352	244	5,634	3,968	3,481	2,693
malicious_server_usr	1.3E+16	5.5E+15	552	190	46,488	14,832	452,117	269,703
malicious_ssh_theft	7.5E+16	2.9E+16	148	53	18,351	5,010	18,233	4,891
malicious_student	1.3E+6	3.5E+5	214	76	258	204	252	197
malicious_wget	1.2E+15	4.2E+14	225	85	28,900	9,550	24,043	6,815
malware_autorun	7.8E+9	3.8E+9	668	367	1,998	1,601	1,936	1,539
malware_danger	1.9E+12	6.7E+11	2,017	1,149	22,319	6,108	22,057	5,846
malware_hijack	1.5E+14	5.5E+13	2,826	1,220	22,571	6,817	22,320	6,566
malware_infector	2.1E+10	7.8E+9	1,361	699	4,955	3,526	3,331	2,576
malware_sysbot	9E+9	4.4E+9	1,260	648	2,596	2,092	2,511	2,006
tc_fivedirections_1	3.3E+23	1.3E+23	61,785	25,527	8.9E+6	3.4E+6	8.8E+6	3.3E+6
tc_fivedirections_2	1.5E+12	5.1E+11	148,555	60,180	1.2E+7	5.1E+6	1.1E+7	3.9E+6
tc_theia	1.5E+11	5.7E+10	136,603	49,401	2.4E+6	1.2E+6	2.3E+6	1.2E+6
atlasv2_s1	1.3E+9	4E+8	285	98	169,574	37,084	168,362	35,871
atlasv2_s2	2.6E+21	9.9E+20	253	85	168,665	36,924	167,459	35,717
atlasv2_s3	2.7E+21	1E+21	361	133	169,837	37,124	168,622	35,908
atlasv2_s4	6.4E+20	2.4E+20	725	254	174,938	42,181	173,731	40,973
Average	**1.1E+22**	**4.3E+21**	**12,196**	**4,791**	**795,132**	**331,835**	**773,928**	**298,299**

vanilla setup, with stronger effects when paired with the optimized configuration. The most substantial gains occur when Provexa is layered atop the tuned PostgreSQL engine, where its decomposition and scheduling strategies complement the tuned planner.

To better understand why our multi-query execution strategy leads to lower planner-estimated costs, we analyze representative tracking and searching queries. As an example, consider the tracking query for the *multistep_supply_chain* case shown below.

```
1 bg = back track where (cmdline like "freemem.sh", type=
    process) from db(case2_supply_chain); display bg;
```

This query looks for processes whose command line contains freemem.sh and tracks all events that causally precede them, in the case2_supply_chain database. Without our scheduler, this requires a recursive SQL query that generates a monolithic execution plan processing over 144 million intermediate rows, with an estimated cost of approximately 74 million. In contrast, our approach starts with a simple query that selects processes matching cmdline LIKE 'freemem.sh' and then iteratively retrieves parent processes through separate follow-up queries. Each step operates over a narrow set of candidates (3 to 140 PIDs), with individual query costs between 22 and 73, adding up to just 163. This strategy avoids the overhead of unconstrained recursive joins and large materialized intermediaries.

A similar benefit appears for search queries. Consider the query for the *malware_infector* case shown below.

```
1 poi = search from bg where e1{path like "Virus.Infector
    ", type=file}, e2{pid=2688, exename like "exe",
    type=process}, e3{type=file} with e1->e2 &&[<100s]
    e2->e3; display poi;
```

The goal is to find a file named Virus.Infector read by a process with pid=2688 and exename extension 'exe', which then writes to another file within 100 seconds. This query runs over a prior result

bg from a backtracking query. Instead of executing the entire pattern in a single query with multiple joins and constraints, we split it into two subqueries: the first computes e1->e2 to find file-to-process reads, and the second computes e2->e3 to find subsequent writes from those processes, applying the temporal constraint (<100s) in the second step. This decomposition reduces the planner-estimated cost by 1.5× compared to the monolithic version.

Summary: These improvements reflect a key planning advantage: query optimizers are more effective at handling small, localized subqueries than deep recursive plans or multi-join blocks. By isolating semantically meaningful constraints early and narrowing each step's scope, our strategy enables selective index scans and faster execution. This approach is particularly effective when queries contain selective constraints that sharply reduce the search space, for example, filtering by a specific script name, IP address, or file path. Such constraints are common in real investigative workflows, where analysts typically start from a point of interest and explore outward. In cases where constraints are overly general (e.g., searching for any process with cmdline LIKE 'exe'), the initial query may return a large result set, reducing the pruning benefits of decomposition. Nevertheless, because real-world investigations often start with targeted hypotheses or known indicators, our multi-query decomposition strategy remains well-suited in practice and consistently delivers lower cost and better performance. By decomposing queries in the application layer while accounting for domain semantics, we play to the strengths of the database engine, which can efficiently execute simple, targeted queries using indexes and optimized access paths. The database still applies its own planner optimizations, but we effectively reduce its burden by structuring the workload to align with its strengths. Detailed query splitting examples and their cost analyses are available at [24].

7.6 RQ4: Query Length

We compare the *query length* of PROVQL with SQL, Cypher, and Splunk SPL, an industry-standard security DSL. Query length is a standard metric in DSL research, as demonstrated in previous works [28, 42], where shorter queries have been shown to improve efficiency and reduce cognitive overhead. This metric highlights how PROVQL simplifies query formulation, reduces verbosity and cognitive load, and facilitates easier iterative attack investigations. Splunk [9], a widely used security log analysis solution, employs its own DSL, Splunk SPL, for investigating system logs. However, Splunk SPL lacks native support for recursive queries needed for tracking [15], so our comparison is limited to search queries.

Example queries are available at [24]. PROVQL tracking queries are more concise, averaging 19 words vs. 421 in SQL and 183 in Cypher. While Cypher is more concise than SQL due to its direct modeling of graph, it becomes verbose when using multiple FOREACH loops for traversal. PROVQL also leads in search query conciseness, averaging 15 words, compared to 192 in SQL, 25 in Cypher, and 62 in Splunk SPL. SQL and Splunk SPL require multiple joins to express entity relationships, making complex queries verbose. Cypher, despite modeling relationships directly, still needs verbose MATCH and WHERE clauses for complex constraints, resulting in lengthy queries for expressing complex relationships.

7.7 RQ5: User Study

We further conducted a user study to evaluate PROVEXA's usability and adoption. The study compared PROVQL with SQL and Cypher, two widely used query languages, to assess efficiency, learning curve, and practicality for attack investigation. We recruited 10 participants with SQL and Cypher experience, and a security background; several also had data science expertise and familiarity with exploratory analysis, providing broader perspectives. Each participant completed six investigation tasks using SQL, Cypher, and PROVQL, divided into two complexity levels. *Simple tasks* involved single-host attacks (*malicious_ssh_theft, malicious_illegal_store, malicious_server_usr*), while *complex tasks* spanned multiple hosts and required more investigative queries (*multistep_supply_chain, multistep_penetration, multistep_password_crack*). Task difficulty was balanced within each category. Participants used SQL for *malicious_ssh_theft* and *multistep_supply_chain*, Cypher for *malicious_server_usr* and *multistep_password_crack*, and PROVQL for *malicious_illegal_store* and *multistep_penetration.*

To evaluate ease of transitioning between languages, we introduced a transition task focused on complex cases. Participants reattempted tasks using a different language, switching from PROVQL to SQL/Cypher or vice versa. This design tests whether PROVQL's high-level constructs simplify query formulation and reduce cognitive load. Each participant completed four additional transition tasks: *multistep_supply_chain* and *multistep_password_crack* with PROVQL, and *multistep_penetration* with SQL and Cypher.

Each participant signed a consent form (IRB-approved) and received PostgreSQL and Neo4j databases with provenance data, schema details, and case descriptions outlining investigation goals (e.g., identifying attacker IPs and key PIDs). Simple tasks had a 30-minute limit; complex ones, 45 minutes. Studies were conducted individually on our laptops using PROVEXA's UI for PROVQL,

pgAdmin for SQL, and the Neo4j Dashboard for Cypher. Screen activity (no audio/video) was recorded to track completion time and query count. Afterward, participants completed a QuestionPro survey rating each language on ease of use, syntax simplicity, and overall experience, using a 5-point Likert scale.

User study results. For simple cases, *malicious_ssh_theft* using SQL averaged 19 minutes and 3 queries; *malicious_server_usr* using Cypher took 16 minutes and 3.8 queries. In contrast, *malicious_illegal_store* using PROVQL was completed in just 7 minutes with 1.7 queries. For complex cases, PROVQL showed clear efficiency gains. In *multistep_supply_chain*, users improved from 6 minutes with SQL to 4 minutes using PROVQL. In *multistep_password_crack*, Cypher users averaged 14 minutes and 4.8 queries, but only 6 minutes with PROVQL. In *multistep_penetration*, PROVQL users completed the task in 16 minutes with 3 queries. By comparison, only 33% of SQL users finished within the 45-minute limit (average 30 minutes, 5.6 queries). 30% of Cypher users timed out, and the rest averaged 12 minutes and 3.8 queries. Despite familiarity, SQL's verbosity led to longer times and more queries, while Cypher showed inconsistent performance. In contrast, PROVQL was more reliable, requiring fewer queries, and delivering steady performance.

We also analyzed post-survey feedback; detailed questions and results are available at [24]. Participants rated four metrics (ease of query writing, result interpretation, event search, and tracking sequences) on a 5-point scale (Very Complex, Complex, Neutral, Easy, and Very Easy). PROVQL consistently received higher ratings, with the most rating it "Easy" or "Very Easy" across all metrics. Specifically, 100% rating it "Easy" or better for event search, and 70% rating it "Very Easy" for tracking. In contrast, SQL and Cypher received mixed feedback, with many users marking them as "Complex"or "Very Complex". Overall, 95% of participants found PROVQL easier to use than SQL or Cypher. For language transitions, 90% rated switching from SQL to PROVQL as "Easy" or better, while 70% found the reverse "Complex" or "Very Complex". A similar pattern emerged for Cypher. 80% preferred PROVQL for attack investigations, and 80% rated transitioning to it as "Easy" or better. However, returning to Cypher was harder; 40% rated it "Complex" and 40% "Neutral". These findings confirm PROVQL's usability advantages to streamline attack investigations and reduce cognitive load.

8 DISCUSSION

Query optimizations. PROVEXA can benefit from built-in database query optimization strategies while adding its own domain-specific enhancements. Traditional cost-based optimization selects query plans based on estimated resource costs (e.g., CPU, I/O, memory). PROVEXA provides a domain-aware scheduler that decomposes complex provenance queries based on pruning power, ensuring that high-impact constraints are applied early to reduce the search space. PROVEXA's scheduler operates as an additional domain-aware optimization layer atop the database, dispatching its decomposed subqueries to the database's native scheduler. This dual-layer optimization ensures that PROVEXA benefits from both its own optimization strategies and the database's native performance-enhancing techniques. Future work could explore integrating cost-based decision-making into the decomposition process and extending lazy evaluation to operate across decomposed subqueries.

Data science notebooks. While computation notebooks like Jupyter are excellent for general-purpose exploration, they are not ideally suited for system provenance analysis in security contexts. Audit logs are massive in volume, span large time windows, and are stored in databases to support scalable, persistent querying. Notebooks would still require integration with these backends, introducing overhead and setup complexity. Raw logs must be parsed and transformed into structured provenance graphs for identifying complex patterns–capabilities that Provexa provides natively. Nevertheless, Provexa offers a notebook-like UI featuring code cells to support iterative exploration.

9 RELATED WORK

Attack investigation using audit logs. System audit logs are vital resources for investigating complex attacks. Causal dependency tracking methods trace root causes. To mitigate dependency explosion, some works use heuristics to prioritize dependencies [33, 36, 45], but risk information loss. Others partition process execution into finer units [43, 47], but require intrusive instrumentation and kernel modifications, limiting their practical adoption. Another class of techniques uses subgraph matching over provenance graphs. Poirot [49] uses heuristics to score node and path alignments, but subsequent studies [26, 35, 53] report high computational costs and missed attack steps due to its early termination at the first valid alignment. Learning-based methods like ProvG-Searcher [26] and DeepHunter [53] embed graphs using neural models but only determine whether a query graph is entailed, without reconstructing attack steps. They also require costly offline training.

Provexa takes a different path. Emphasizing human-in-the-loop analysis, it offers a DSL for flexible, iterative investigation. Unlike prior one-shot approaches, Provexa supports incremental refinement and makes analyst intent explicit through DSL query construction. It avoids early stopping, needs no offline training, and uniquely supports native dependency tracking–capabilities missing from prior subgraph matching approaches.

Database query languages. Existing query languages like SQL and Cypher are not designed for attack investigation. They lack native support for causality tracking, which is essential for tracing multi-step attacks, and their general-purpose execution models perform poorly on system provenance workloads. Queries in these languages often become verbose and error-prone when investigating complex attacks. SPARQL [52] and MongoDB's JSON-based query language [30] share similar limitations: SPARQL adopts SQL-like syntax for RDF data, and MongoDB's model is tailored for document retrieval. Neither supports iterative workflows, efficient modeling of multi-stage attacks, or variable binding across queries, making them inefficient for complex, evolving investigations. Provexa addresses these limitations through a domain-specific abstraction tailored to system provenance and attack investigation. Its DSL enables intuitive event pattern search, chaining, and iterative human-in-the-loop workflows. A domain-aware scheduler leverages event dependencies and pruning to optimize execution, adding a layer atop existing backends for performance enhancement. Queries in ProvQL are more concise, making it easier to incorporate analyst knowledge and refine iteratively. These design choices enable Provexa to deliver faster, more intuitive, and more robust forensic analysis than traditional query languages.

Security analysis languages. Splunk [9] and Elasticsearch [4] are among the most widely used log analysis tools for system performance monitoring and security diagnostics. Splunk offers a Search Processing Language (SPL) that combines keyword and regex-based search with shell-like piping for extracting insights from logs. Elasticsearch provides a JSON-based DSL for text search and filtering. OSQuery [7], an open-source tool, uses a SQL-like syntax to query the operating system state. Unlike Provexa, these languages do not support complex dependency tracking queries, which are essential for tracing multi-step attack event chains. Other specialized languages have been developed for different security tasks, such as network intrusion detection and prevention [27, 29, 32], secure overlay networks [39, 46], and threat intelligence representation [10, 17]. Each is tailored to its own domain and use case. In contrast, ProvQL is specially designed for the iterative investigation of complex, multi-stage attacks over system provenance data. In addition, the two query syntaxes supported by ProvQL cannot be easily expressed in these languages.

Future directions. Several future directions can be explored. First, while our current approach prioritizes split queries based on constraints, this can sometimes lead to high intermediate costs when that constraint is not selective. In such cases, a monolithic query planned holistically by the database might be more efficient. One extension is to add a lightweight cost-based fallback mechanism that compares the estimated cost of a monolithic plan with the split plan and selects the cheaper option dynamically. Second, although we apply best-effort tuning to the underlying database engine, Provexa achieves the largest gains when paired with an optimized configuration. Since Provexa's decomposition is orthogonal to engine-level tuning, future work could explore how more adaptive database optimizations further amplify its benefits. Third, while Provexa largely reduces the effort of query formulation compared to general-purpose languages, analysts still have to write queries manually. Leveraging cyber threat intelligence [31, 34, 48], which captures common attack patterns and indicators, along with large language models offers a potential path toward automatically generating ProvQL queries aligned with investigative goals.

10 CONCLUSION

We presented Provexa, a defense system that empowers human analysts to effectively analyze large-scale system provenance to investigate complex multi-step attacks. Provexa introduces an intuitive and expressive domain-specific language, ProvQL, for capturing diverse attack patterns and analytical workflows. It also features an efficient, domain-aware execution engine that enhances query performance across different database backends.

ACKNOWLEDGMENTS

REFERENCES

[1] [n.d.]. ANTLR. http://www.antlr.org/ Accessed: July 8, 2025.
[2] [n.d.]. Common Vulnerabilities and Exposures. https://cve.org/ Accessed: July 8, 2025.
[3] [n.d.]. DuckDB. https://duckdb.org/ Accessed: July 8, 2025.
[4] [n.d.]. Elasticsearch. https://www.elastic.co/ Accessed: July 8, 2025.
[5] [n.d.]. Neo4j. http://neo4j.com/ Accessed: July 8, 2025.
[6] [n.d.]. Neo4j: Conditional Cypher Execution. https://neo4j.com/developer/kb/conditional-cypher-execution/ Accessed: July 8, 2025.
[7] [n.d.]. osquery. https://osquery.io/ Accessed: July 8, 2025.
[8] [n.d.]. PostgreSQL. http://www.postgresql.org/ Accessed: July 8, 2025.
[9] [n.d.]. Splunk. http://www.splunk.com/ Accessed: July 8, 2025.
[10] [n.d.]. Structured Threat Information eXpression. http://stixproject.github.io/ Accessed: July 8, 2025.
[11] [n.d.]. Sysdig. http://www.sysdig.com/ Accessed: July 8, 2025.
[12] [n.d.]. The Linux Audit Framework. https://github.com/linux-audit/ Accessed: July 8, 2025.
[13] 2014. CVE-2014-6271. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271 Accessed: July 8, 2025.
[14] 2014. Target Data Breach Incident. http://www.nytimes.com/2014/02/27/business/target-reports-on-fourth-quarter-earnings.html Accessed: July 8, 2025.
[15] 2016. Splunk: How to Write a Recursive Search to Build a Tree Structure? https://community.splunk.com/t5/Splunk-Search/How-to-write-a-recursive-search-to-build-a-tree-structure/m-p/255819 Accessed: July 8, 2025.
[16] 2019. Case Equifax Data Breach. https://www.ftc.gov/legal-library/browse/cases-proceedings/172-3203-equifax-inc Accessed: July 8, 2025.
[17] 2020. Cyber Observable eXpression. https://cyboxproject.github.io/ Accessed: July 8, 2025.
[18] 2020. Transparent Computing Engagement 5 Data Release. https://github.com/darpa-i2o/Transparent-Computing/blob/master/README.md Accessed: July 8, 2025.
[19] 2021. Cypher Query Language. https://neo4j.com/docs/cypher-manual/4.4/introduction/ Accessed: July 8, 2025.
[20] 2023. Information technology — Database languages SQL. https://www.iso.org/standard/76583.html Accessed: July 8, 2025.
[21] 2024. Crowdstrike 2024 Global Threat Report. https://go.crowdstrike.com/rs/281-OBQ-266/images/GlobalThreatReport2024.pdf Accessed: July 8, 2025.
[22] 2024. IBM X-Force Threat Intelligence Index 2024. https://newsletter.radensa.ru/wp-content/uploads/2024/03/IBM-XForce-Threat-Intelligence-Index-2024.pdf Accessed: July 8, 2025.
[23] 2024. ProcMon. https://learn.microsoft.com/en-us/sysinternals/downloads/procmon Accessed: July 8, 2025.
[24] 2025. Provexa Project Website. https://provexa-app.github.io/.
[25] 2025. The 20 biggest data breaches of the 21st century. https://www.csoonline.com/article/534628/the-biggest-data-breaches-of-the-21st-century.html Accessed: July 8, 2025.
[26] Enes Altinisik, Fatih Deniz, and Hüsrev Taha Sencar. 2023. ProvG-Searcher: A Graph Representation Learning Approach for Efficient Provenance Graph Search. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2247–2261.
[27] Osama Bajaber, Bo Ji, and Peng Gao. 2024. P4Control: Line-Rate Cross-Host Attack Prevention via In-Network Information Flow Control Enabled by Programmable Switches and eBPF. In *IEEE Symposium on Security and Privacy (SP)*. 4610–4628.
[28] Ankica Barisic, Vasco Amaral, and Miguel Goulão. 2012. Usability Evaluation of Domain-Specific Languages. In *2012 Eighth International Conference on the Quality of Information and Communications Technology*. 342–347. https://doi.org/10.1109/QUATIC.2012.63
[29] Kevin Borders, Jonathan Springer, and Matthew Burnside. 2012. Chimera: A Declarative Language for Streaming Network Traffic Analysis. In *USENIX Security Symposium (USENIX Security)*. 365–379.
[30] Shannon Bradshaw, Eoin Brazil, and Kristina Chodorow. 2019. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly Media.
[31] Yutong Cheng, Osama Bajaber, Saimon Amanuel Tsegai, Dawn Song, and Peng Gao. 2025. CTINexus: Automatic Cyber Threat Intelligence Knowledge Graph Construction Using Large Language Models. In *IEEE European Symposium on Security and Privacy (EuroS&P)*.
[32] Frédéric Cuppens and Rodolphe Ortalo. 2000. LAMBDA: A Language to Model a Database for Detection of Attacks. In *Recent Advances in Intrusion Detection (RAID)*. 197–216.
[33] Pengcheng Fang, Peng Gao, Changlin Liu, Erman Ayday, Kangkook Jee, and Ting Wang. 2022. Back-Propagating System Dependency Impact for Attack Investigation. In *USENIX Security Symposium (USENIX Security)*. 2461–2478.

[34] Peng Gao, Xiaoyuan Liu, Edward Choi, Sibo Ma, Xinyu Yang, and Dawn Song. 2023. ThreatKG: An AI-Powered System for Automated Open-Source Cyber Threat Intelligence Gathering and Management. In *ACM Workshop on Large AI Systems and Models with Privacy and Safety Analysis (LAMPS)*. 1–12.
[35] Peng Gao, Fei Shao, Xiaoyuan Liu, Xusheng Xiao, Zheng Qin, Fengyuan Xu, Prateek Mittal, Sanjeev R Kulkarni, and Dawn Song. 2021. Enabling Efficient Cyber Threat Hunting with Cyber Threat Intelligence. In *International Conference on Data Engineering (ICDE)*. 193–204.
[36] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. 2019. NODOZE: Combatting Threat Alert Fatigue with Automated Provenance Triage. In *Network and Distributed System Security Symposium (NDSS)*.
[37] Muhammad Adil Inam, Yinfang Chen, Akul Goyal, Jason Liu, Jaron Mink, Noor Michael, Sneha Gaur, Adam Bates, and Wajih Ul Hassan. 2023. SoK: History is a Vast Early Warning System: Auditing the Provenance of System Intrusions. In *IEEE Symposium on Security and Privacy (S&P)*. 2620–2638.
[38] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. 2017. RAIN: Refinable Attack Investigation with On-demand Inter-Process Information Flow Tracking. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 377–390.
[39] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. 2007. Mace: Language Support for Building Distributed Systems. In *Programming Language Design and Implementation (PLDI)*. 179–188.
[40] Samuel T. King and Peter M. Chen. 2003. Backtracking intrusions. In *ACM Symposium on Operating Systems Principles (SOSP)*. 223–236.
[41] Samuel T. King, Zhuoqing Morley Mao, Dominic G. Lucchetti, and Peter M. Chen. 2005. Enriching Intrusion Alerts Through Multi-Host Causality. In *Network and Distributed System Security Symposium (NDSS)*.
[42] Tomaž Kosar, Marjan Mernik, and Jeff Carver. 2012. Program Comprehension of Domain-Specific and General-Purpose Languages: Comparison Using a Family of Experiments. *Empirical Software Engineering* 17 (2012), 276–304.
[43] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela F Ciocarlie, et al. 2018. MCI: Modeling-based Causality Inference in Audit Logging for Attack Investigation.. In *Network and Distributed System Security Symposium (NDSS)*.
[44] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition. In *Network and Distributed System Security Symposium (NDSS)*.
[45] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. 2018. Towards a Timely Causality Analysis for Enterprise Security. In *Network and Distributed System Security Symposium (NDSS)*.
[46] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2006. Declarative Networking: Language, Execution and Optimization. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 97–108.
[47] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In *Network and Distributed System Security Symposium (NDSS)*.
[48] Rob McMillan. 2013. Definition: Threat Intelligence. https://www.gartner.com/en/documents/2487216 Accessed: July 8, 2025.
[49] Sadegh M Milajerdi, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. 2019. POIROT: Aligning Attack Behavior with Kernel Audit Records for Cyber Threat Hunting. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1795–1812.
[50] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, R Sekar, and VN Venkatakrishnan. 2019. HOLMES: Real-Time APT Detection through Correlation of Suspicious Information Flows. In *IEEE Symposium on Security and Privacy (S&P)*. 1137–1152.
[51] Andy Riddle, Kim Westfall, and Adam Bates. 2023. ATLASv2: ATLAS Attack Engagements, Version 2. arXiv:2401.01341 [cs.CR] https://arxiv.org/abs/2401.01341
[52] W3C. 2008. SPARQL Query Language for RDF. https://www.w3.org/TR/rdf-sparql-query/ Accessed: July 8, 2025.
[53] Renzheng Wei, Lijun Cai, Lixin Zhao, Aimin Yu, and Dan Meng. 2021. DeepHunter: A Graph Neural Network Based Approach for Robust Cyber Threat Hunting. In *EAI International Conference on Security and Privacy in Communication Networks (SecureComm)*. 3–24.
[54] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. 2016. High Fidelity Data Reduction for Big Data Security Dependency Analyses. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 504–516.