PROGQL: A Provenance Graph Query System for Cyber Attack Investigation

Technical Report

Fei Shao[†], Jia Zou[*], Zhichao Cao[*], Xusheng Xiao[*]
[*]Arizona State University. Email: {jia.zou, zhichao.cao, xusheng.xiao}@asu.edu
[†]Case Western Reserve University. Email: fxs128@case.edu

***Abstract*—Provenance analysis (PA) has recently emerged as an important solution for cyber attack investigation. PA leverages system monitoring to monitor system activities as a series of system audit events and organizes these events as a *provenance graph* to show the dependencies among system activities, which can reveal steps of cyber attacks. Despite their potential, existing PA techniques face two critical challenges: (1) they are inflexible and non-extensible, making it difficult to incorporate analyst expertise, and (2) they are memory-inefficient, often requiring >100GB of RAM to hold entire event streams, which fundamentally limits scalability and deployment in real-world environments. To address these fundamental limitations, we propose the PROGQL framework, which provides a domain-specific graph search language with a well-engineered query engine, allowing PA over system audit events and expert knowledge to be jointly expressed as a graph search query and thereby facilitating the investigation of complex cyberattacks. In particular, to support dependency searches from a starting edge required in PA, PROGQL introduces new language constructs for constrained graph traversal, edge weight computation, value propagation along weighted edges, and graph merging to integrate multiple searches. Moreover, the PROGQL query engine is optimized for efficient incremental graph search across heterogeneous database backends, eliminating the need for full in-memory materialization and reducing memory overhead. Our evaluations on real attacks demonstrate the effectiveness of the PROGQL language in expressing a diverse set of complex attacks compared with the state-of-the-art (SOTA) graph query language Cypher, and the comparison with the SOTA PA technique DEPIMPACT further demonstrates the significant improvement of the scalability brought by our PROGQL framework's design (8× saving of memory consumption without penalty on runtime performance).**

***Index Terms*—Graph Query Language, System Auditing, Cyber Threat Investigation**

I. INTRODUCTION

Large enterprises are increasingly plagued by cyber-attacks, causing significant financial losses [1]–[7]. These attacks often exploit multiple types of vulnerability to infiltrate target systems in multiple stages, posing challenges for detection and investigation. To counter these attacks, recent approaches based on *ubiquitous system monitoring* have emerged as an important approach that monitors system activities and assists attack investigation [8]–[15]. System monitoring collects kernel auditing events about system calls as system audit logs. The collected data enables techniques based on causality analysis to perform *provenance analysis* (referred to as PA) on attack-related events [8], [9], [12], [13], [16], which provides the contextual information about the attacks to identify entry points of invasions (i.e., backward PA) and ramifications of attacks (i.e., forward PA).

PA assumes causal dependencies between system entities (e.g., files, processes, and network connections) involved in the same system call event (e.g., a process reading a file). Based on this assumption, given a Point-Of-Interest (POI) event (e.g., an alert reported by intrusion detection), these techniques search for the system call events that have dependencies on the POI event and organize these events as a *provenance graph* (referred to as PG), a data structure that models dependencies between system activities [8], [9], [17], with nodes representing system entities (e.g., processes and files) and edges representing events (e.g., file creation by a process). A PG can provide contextual information for the POI event by reconstructing a chain of events that lead to the POI event. It has been proven effective in reducing false alerts of intrusions [18], [19] and assisting timely system recoveries [20], [21]. For example, as ransomware and compression programs (e.g., `bzip2`) both read and write many files in a short period of time, many ransomware detectors that check only the behavior of a single process will likely classify `bzip2` as ransomware; with the dependency graph provided by causality analysis, the detector can distinguish `bzip2` from ransomware, as the entry point of ransomware (e.g., email attachment) is often different from the `bzip2` program.

While PA makes promising progress in attack investigation, existing techniques [8], [9], [13], [20], [21] suffer from two fundamental limitations. First, existing techniques **lack *extensibility* to incorporate expert knowledge and *flexibility* to integrate multiple PA results.** PA is prone to dependency explosion since existing techniques mainly use happen-before relationships to identify dependencies [8], [22]. Consequently, the resulting PG often exceeds one million edges. To mitigate this, recent techniques define edge weights based on edge attributes and apply optimization techniques to filter irrelevant dependencies [12], [13], [16], [20], [21], [23]. However, these techniques are not universally effective across all attack scenarios, and investigations often require applying multiple PA across several alerts [24] to determine which are related

and which are irrelevant. This process is similar to interactive queries in database systems, where results are progressively refined to support data-driven decisions. To facilitate this process, expert knowledge must be incorporated into provenance analysis for choosing weight functions or specifying domain-specific filters, and the results from multiple analyses need to be integrated seamlessly. Unfortunately, existing techniques provide neither effective expert knowledge incorporation nor efficient integration of results. Second, **existing techniques are memory-hungry and inefficient.** For example, they often load all the events within a period of time into memory and perform analysis on these events to filter irrelevant ones. This strategy requires a significant amount of memory to hold all the events (often $> 100GB$) [17], [24], [25] and greatly limit the scalability of PA and prevent it from being used in resource-constrained environments.

Contribution. To address these fundamental limitations of existing PA techniques, in this paper, we propose a novel query framework, called PROGQL, which provides (1) a domain-specific graph search language that can jointly express PA over system audit events and expert knowledge as a graph search query, and (2) a query engine that optimizes query execution by combining domain-specific optimizations with database backends. The language design of PROGQL is motivated by the following insights.

- *Dependency-based Graph Search*: Fundamentally, PA is a graph search technique, finding edges that act as dependencies of a specific edge (i.e., POI event) in a PG derived from system audit logs. For example, backward PA finds the edges whose start time is before the end time of the POI event, and then recursively applies the same search constraints on the last found edges until no more edges can be found. Also, computing certain edge properties requires *value propagation*, such as finding all the ancestors for an edge. While existing graph search languages (e.g., Cypher [26], SPARQL [27], [28], and CRPQ [29], [30]) and domain-specific languages (e.g., AIQL [31] and SAQL [32]) support finding subgraphs or paths expressible through regular expressions and anomaly patterns, they lack the ability to model dependency-based searches that involve iteratively extending from existing edges and checking edge-dependent constraints. For example, Cypher can match edges such as $e(u, v)$ using attribute-based conditions (e.g., `MATCH (c:node name: Chrome)`), but it cannot express queries that depend on the properties of adjacent edges. Specifically, it cannot retrieve incoming edges of u whose start time is earlier than the maximum end time among v's outgoing edges.
- *Edge Weights and Value Propagation*: Edge weights are used by various PA techniques to filter out irrelevant dependencies [13], [17], [24], and impact scores propagated from the POI event through the weighted edges are also used to identify attack-related entry nodes [24], [33], [34]. Unfortunately, existing languages also lack the capabilities to express edge weights and impact scores computed based on value propagation through weighted edges. Additionally, they do not support combining the results of PA applied to multiple POI events, a capability essential for retaining attack-related information while filtering irrelevant edges.

To address these issues, we propose a novel graph query language, PROGQL, built upon the syntax of Cypher [26], which is a popular graph query language for Neo4j [35]. PROGQL provides novel language constructs for (1) recursive constrained graph search, (2) edge weight assignment based on arbitrary feature projections, and (3) impact score propagation across the graph, and graph merging:

- *Constrained Graph Search*: PROGQL extends the `BFS` and `DFS` constructs to support Breadth-First and Depth-First Search with user-defined constraints, allowing dependency edges to be identified based on the properties of adjacent edges.
- *Edge Weight Computation*: PROGQL extends the `UNWIND` and `SET` operators of Cypher to support the edge weight computation based on adjacent edges' properties such as time.
- *Value Propagation*: PROGQL extends the Cypher `MATCH` and `SET` operators to support the value propagation through the weighted edges.
- *Graph Merge*: PROGQL provides the language constructs (`UNION` and `INTERSECT`) to merge multiple PGs by combining all edges or retaining only the edges common to each graph.

To support the query execution of PROGQL language, our framework provides a data importer that parses the events recorded in system auditing logs, builds the data model of the parsed events, and performs batch insertion of the modeled events into a database backend with high efficiency. In particular, the database backend can be built using different types of databases such as relational databases (e.g., PostgreSQL [36] and MyRocks [37]) and graph databases (e.g., Neo4j [35] and Nebula [38]). Different types of databases support security-related searches with varying efficiency, each excelling at different search types, as shown in recent studies [31], [32], [39], [40]. For example, graph databases can efficiently support finding neighboring edges given a POI event, while relational databases can leverage join to efficiently find out events satisfying specific constraints even when the edges are not connected. Building upon a database backend enables our PROGQL framework to leverage the critical services provided by the mature infrastructures, such as data management, indexing mechanisms, recovery, and security.

In addition, our framework provides a novel query engine that efficiently executes PROGQL queries. As PA usually needs to process a colossal amount of log data [22]–[24], [41]–[44], it is critical to optimize the performance of graph search when building PGs from system auditing logs. Thus, our query engine optimizes the proposed graph search and merging through an incremental graph search by performing fine-grained edge fetch with the help of the database backend rather than loading the whole graph into memory, greatly improving memory efficiency and search performance. Moreover, the query engine performs edge merge based on time

TABLE I: Representative system calls

Event Category	System Call
File event	read, write, execute, rename
Process event	execve, execute, clone
Network event	read, write, sendto, sendmsg, recvfrom

differences to minimize duplicate information, performs edge weight assignment based on arbitrary feature projections, and supports efficient graph merge through edge signatures.

Evaluation. We conduct comprehensive evaluations of PROGQL by composing PROGQL queries for 14 real attacks that represent a diversified set of attack scenarios. The number of the system events to search for each attack case is about 19 million on average. The results show that PROGQL can execute these queries using averagely 5GB memory and finish the search within 224 seconds, and the output PGs preserve all the attack steps for each attack. To demonstrate the expressiveness of PROGQL, we compare PROGQL with Cypher in expressing attack behaviors, and the results show that due to the lack of expressiveness in constrained graph search, weight computation, and value propagation, Cypher needs to use 32 separate queries to achieve the same search results as a PROGQL query. On average, when expressing queries for the same attacks, PROGQL uses 9× fewer constraints, 15× fewer words, and 17× fewer characters than Cypher. Furthermore, PROGQL queries are executed 20.7 times more efficiently (20s v.s. 414s) and requires only about half (57.6%) of the memory used by Cypher running in Neo4j.

We also compared PROGQL with the state-of-the-art (SOTA) PA: DEPIMPACT [24]. The results show that PROGQL can achieve similar runtime performance as DEPIMPACT with 8× smaller of memory consumption. This is a great improvement on scalability, as DEPIMPACT loads all events into memory and performs the in-memory graph search, while PROGQL leverages the database backends. Even though the queries with the database may cause performance overhead compared to executing the queries in memory, PROGQL's edge indexes and the incremental graph search employed by PROGQL speed up the subsequent search with substantially lower memory consumption, while achieving similar runtime performance. We also showed that PROGQL can easily express PA techniques with different weight computations and compare their performance. Finally, we compared the performance of using different types of database backends, the results show that overall Neo4j achieves the best runtime performance and PostgreSQL achieves the lowest memory consumption. Our code and data are publicly available at our project website [45].

II. BACKGROUND AND MOTIVATION

A. System Audit Logs

System audit logs record the kernel-level audit events about system calls and are crucial for cyber attack investigation [8]–[15]. These audit events provide detailed information on monitored system calls, describing how system entities interact with system resources and other system entities in a monitored computer system. Formally, a system audit event is modeled as a directed edge between subject and object or vice-versa, represented as ⟨sub, op, obj⟩ or ⟨obj, op, sub⟩, where sub represents a process entity, obj represents different types of system entities (e.g., process, file, or network entities), and op represents the system activity performed by the system call (e.g., reading file or spawning a new process). Based on the types of the objects, system audit events are categorized as *process events*, *file events*, and *network events*. *Process events* record the operations of processes, such as execve. *File events* record the operations on files, such as files read, write, and rename. *Network events* record the operations of network accesses, such as sending and receiving messages from sockets. (See Table I for more details.)

B. Provenance Analysis (PA)

PA [8], [9], [13], [20], [21] analyzes the auditing events to infer their dependencies and present the dependencies as a directed graph, called a provenance graph (PG). In a PG, a node is a system entity, such as a process, a file, or a network connection. An edge represents a system audit event, and its direction indicates the direction of data flow (from sub to obj or vice-versa). An edge is associated with event properties that are critical for security analysis (e.g., data amount) and a time window that indicates the start time and the end time of the event. Given a POI event, PA starts from the POI event, and searches for other qualified events to form the PG. Formally, in the PG $G(E, V)$, a node $v \in V$ represents a system entity, i.e., a process, a file, or a network connection. An edge $e(u, v) \in E$ indicates a system call event involving two entities u and v (e.g., file read), and its direction (from u to v) indicates the direction of data flow. Each edge is associated with a time window, and $ts(e)$ and $te(e)$ are used to represent the start time and the end time of e, respectively. Formally, in the PG, for two events $e_1(u_1, v_1)$ and $e_2(u_2, v_2)$, there exists dependency between e_1 and e_2 if $v_1 = u_2$ and $ts(e_1) < te(e_2)$. PA performs two types of search on the system audit events: (1) *backward PA* that searches backward in time to find all the events that have causal dependencies on the POI event, and (2) *forward PA* that searches forward in time to find all the events that the POI event has causal dependencies with.

III. OVERVIEW

Our PROGQL framework consists of 3 major modules, as shown in Figure 1. The data importer module takes system auditing logs as input, and performs batch insertion into the database backend. The language parsing module takes a PROGQL query as input, parses the query text, and extracts the query context that contains all the required information for query execution. The query engine is the core module of the PROGQL system, which executes the PROGQL query based on the extracted query context and searches the system auditing logs stored in the database backend to generate the desired PG. The query engine consists of four components: ① The graph traversal component finds the POI record in the database backend and performs graph search based on

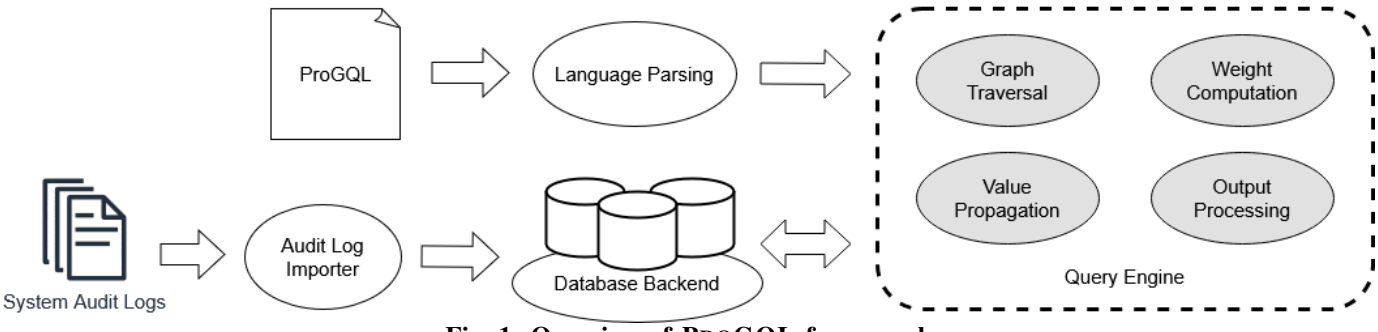


Fig. 1: Overview of PROGQL framework

TABLE II: Representative attributes of system entities

Entity	Attributes
File	id, path/name
Process	id, name, pid
Network	id, src_ip, src_port, dst_ip, dst_port

TABLE III: Representative attributes of system events

Operation	read, write, rename, create_object, execute, clone, recvmsg, sendmsg
Time	starttime, endtime
Misc.	ID, src_entity_ID, dst_entity_ID, amount

the query context. In particular, incremental graph search is adopted to optimize the memory footprint and minimize search scope. ② The weight computation component applies the weight computation function defined in the query context for each edge. ③ The value propagation component propagates the values defined in the query context for each node. ④ The graph merge component performs union or intersection of the graphs defined in the query context and outputs the processed graph as the PG.

IV. DESIGN OF PROGQL

In this section, we present the design details of each module shown in Figure 1.

A. Audit Log Importer

This module accepts system audit logs as input, constructs the data model for the events recorded in the logs, and performs batch insertion to import the modeled system events into multiple databases.

Data Model. Existing works [8], [9], [12]–[15] have indicated that on most modern operating systems (Windows, Linux, and OS X), system entities in most cases are files, processes, and network connections. Thus, in our data model, we consider system entities as files, processes, and network connections. We consider a system event as the interaction between two system entities represented by the tuple ⟨subject, operation, object⟩ where subject represents a process entity, objects represent different types of system entities, and operation represents the system activity performed by the system call (e.g., reading a file or spawning a new process). We categorize system events into three types based on their object entities, namely file events, process events, and network events. Both entities and events have critical security-related attributes shown in Tables II and III. The attributes of entities include the properties to support various security analyses (e.g., file path/name, process name, and ip addresses). The attributes of events include the source entity and the target entity, event origins (i.e., event id, starttime, endtime), and operations (e.g., read/write).

Batch Insertion. As system audit logs contain a huge amount of events [22], [41]–[44], performing a PA often requires scanning hundreds of millions of events. Thus, our importer adopts batch insertion to improve the performance of data insertion. By doing so, we avoid lots of individual network message round-trips and other per-statement inefficiencies. To support near-real-time processing, we adopt a tiered-storage solution in our experiments. For hot data (most recent 1-3 days), we use SSD to support efficient batch insertion and gradually move the cold data to HDD. As system audit logs exhibit strong temporal/spatial properties [14], [15], the data can be easily partitioned across different days and hosts. Such partitions automatically separate hot data and cold data and enable efficient data moving.

B. PROGQL Language

PROGQL introduces novel language constructs, which can express (1) dependency-based graph search that iteratively expands from already discovered edges by evaluating edge-dependent constraints to identify qualified adjacent edges, (2) assign weights to edges based on dynamically found edges and their adjacent edges, (3) propagate scores through weighted edges, and (4) merge multiple PGs directly within a query using union and intersection, enabling analysts to combine backward and forward analyses or cross-host results without external post-processing. These are required features that are not supported by existing languages but necessary for PA on system audit logs.

Grammar 1 shows the representative BNF grammar of PROGQL langauge.

The grammar specifies both the adapted constructs (e.g., MATCH, WHERE, RETURN) and the new operators introduced in PROGQL, which include:

Constrained Graph Search. PA constructs tailored graphs based on the constraints (e.g., recursively filtering edges based on timestamps). To express a constrained graph search, PROGQL defines rules ⟨*Bfs*⟩ and ⟨*Dfs*⟩ combined with ⟨*Backward*⟩ and ⟨*Forward*⟩ to define traversal strategy and direction. For example, `backward(f)` traces back from the

⟨*ProGQL*⟩ ::= sp? ⟨*ProGQLQuery*⟩ (sp (‘union’ | ‘intersect’) sp ‘(’ ⟨*ProGQLQuery*⟩ ‘)’)* (sp? ‘;’)? sp? EOF
⟨*ProGQLQuery*⟩ ::= sp? ⟨*SingleQuery*⟩ (sp (‘intersect’ | ‘union’) sp ⟨*WithQuery*⟩)?
⟨*SingleQuery*⟩ ::= ((sp? ⟨*Match*⟩) (sp (⟨*Bfs*⟩ | ⟨*Dfs*⟩))? sp ⟨*Yield*⟩ sp ⟨*Return*⟩) | (((sp? ⟨*Match*⟩) (sp (⟨*Bfs*⟩ | ⟨*Dfs*⟩)) sp ⟨*Yield*⟩ sp ⟨*Unwind*⟩)? sp ⟨*Set*⟩ (sp ⟨*WithQuery*⟩)?)+ (sp ⟨*Yield*⟩)? SP ⟨*Return*⟩)
⟨*Match*⟩ ::= ‘match’ sp? ⟨*Pattern*⟩ (sp? ⟨*Where*⟩)?
⟨*Pattern*⟩ ::= (⟨*PatternPart*⟩ (sp? ‘,’ sp? ⟨*PatternPart*⟩)*) | ⟨*IdInColl*⟩ | ⟨*Expr*⟩
⟨*Expr*⟩ ::= ‘NOT’ sp? ⟨*CompExpr*⟩ | (sp? ⟨*CompExpr*⟩ (‘AND’ | ‘OR’) sp? ⟨*CompExpr*⟩);
⟨*CompExpr*⟩ ::= sp? ⟨*ArithmeticExpr*⟩ (sp? PartialCompExpr)*
⟨*PartialCompExpr*⟩ ::= (‘=’ ⟨*ArithmeticExpr*⟩ | sp? ⟨*TraversalExpr*⟩) | (‘<>’ ⟨*ArithmeticExpr*⟩ | sp? ⟨*TraversalExpr*⟩) | (‘<’ ⟨*ArithmeticExpr*⟩ | sp? ⟨*TraversalExpr*⟩) | (‘>’ ⟨*ArithmeticExpr*⟩ | sp? ⟨*TraversalExpr*⟩) | (‘<=’ ⟨*ArithmeticExpr*⟩ | sp? ⟨*TraversalExpr*⟩) | (‘>=’ ⟨*ArithmeticExpr*⟩ | sp? ⟨*TraversalExpr*⟩)
⟨*TraversalExpr*⟩ ::= sp? (⟨*Max*⟩ | ⟨*Min*⟩)
⟨*ArithmeticExpr*⟩ ::= sp? ⟨*PropExpr*⟩ (((‘*’ | ‘/’ | ‘+’ | ‘−’) ⟨*PropExpr*⟩)+)?;
⟨*Max*⟩ ::= ‘max’ sp? ‘(’ sp? ⟨*Collect*⟩ sp? ‘)’
⟨*Min*⟩ ::= ‘min’ sp? ‘(’ sp? ⟨*Collect*⟩ sp? ‘)’
⟨*Collect*⟩ ::= ‘collect’ sp? ‘(’ sp? ⟨*IdInColl*⟩ sp? ‘|’ sp ⟨*PropExpr*⟩ sp? ‘)’
⟨*PropExpr*⟩ ::= sp? ⟨*Atom*⟩ (sp? ⟨*PropLookup*⟩)*
⟨*PropLookup*⟩ ::= ‘.’ sp? ⟨*PropKeyName*⟩
⟨*IdInColl*⟩ ::= sp? ⟨*Var*⟩ sp ‘in’ ⟨*Expr*⟩
⟨*Bfs*⟩ ::= ‘bfs’ sp? ‘(’ sp? ⟨*Var*⟩ sp ‘in’ (sp? ⟨*Backward*⟩|sp? ⟨*Forward*⟩) sp? ‘|’ sp ⟨*Match*⟩ sp? ‘)’
⟨*Backward*⟩ ::= ‘backward’ sp? ‘(’ ⟨*Var*⟩ ‘)’
⟨*Forward*⟩ ::= ‘forward’ sp? ‘(’ ⟨*Var*⟩ ‘)’
⟨*Dfs*⟩ ::= ‘dfs’ sp? ‘(’ sp? ⟨*Var*⟩ sp ‘in’ (sp? ⟨*Backward*⟩|sp? ⟨*Forward*⟩) sp? ‘|’ sp ⟨*Match*⟩ sp? ‘)’
⟨*Where*⟩ ::= ‘where’ sp ⟨*Expr*⟩ (sp ⟨*Order*⟩)? (sp ⟨*Limit*⟩)?
⟨*Order*⟩ ::= ‘order’ sp ‘by’ sp ⟨*SortItem*⟩ (‘,’ sp? ⟨*SortItem*⟩)*
⟨*SortItem*⟩ ::= sp? ⟨*PropExpr*⟩ (sp? (‘asc’ | ‘desc’))?
⟨*Limit*⟩ ::= ‘limit’ sp ⟨*NumberLiteral*⟩
⟨*Unwind*⟩ ::= ‘uwind’ sp ⟨*Var*⟩ sp ‘as’ sp ⟨*Var*⟩
⟨*Yield*⟩ ::= ‘yield’ sp ⟨*Var*⟩
⟨*Atom*⟩ ::= sp? (⟨*Literal*⟩ | ⟨*Var*⟩ | ⟨*FuncInvoc*⟩ | ⟨*ParenExpr*⟩)
⟨*ParenExpr*⟩ ::= ‘(’ sp? ⟨*Expr*⟩ sp? ‘)’
⟨*Set*⟩ ::= ‘set’ sp? ⟨*SetItem*⟩ (‘,’ ⟨*SetItem*⟩)*
⟨*SetItem*⟩ ::= (sp? ⟨*PropExpr*⟩ sp? ‘=’ (sp? ⟨*Expr*⟩ | sp? ⟨*Projection*⟩))|(sp? ⟨*PropExpr*⟩ sp? ‘=’ (sp? ⟨*Expr*⟩ | sp? ⟨*Reduce*⟩))
⟨*Projection*⟩ ::= ‘projection’ sp? ‘(’ (sp? ⟨*Expr*⟩ (‘,’)?)+ sp? ‘)’
⟨*Reduce*⟩ ::= ‘reduce’ sp? ‘(’ ⟨*Var*⟩ sp? ‘=’ (sp? ⟨*NumberLiteral*⟩) sp? ‘,’ sp? ⟨*IdInColl*⟩ sp? ‘|’ (sp? ⟨*Expr*⟩) sp? ‘)’
⟨*Return*⟩ ::= ‘return’ sp ⟨*Var*⟩
⟨*WithQuery*⟩ ::= ‘with’ sp ⟨*Var*⟩ ((sp? ‘=’ sp? ‘(’ sp? ⟨*Match*⟩ sp? ‘)’ sp (⟨*Bfs*⟩|⟨*Dfs*⟩) sp ⟨*Yield*⟩ sp ⟨*Return*⟩) | (sp ⟨*Where*⟩))
⟨*FuncInvoc*⟩ ::= ⟨*FuncName*⟩ sp? ‘(’ ⟨*Expr*⟩ ‘)’
⟨*FuncName*⟩ ::= ‘count’ | ‘dst’ | ‘src’ | ‘out’ | ‘in’ | ‘abs’ | ‘ln’ |...

Grammar 1: Representative BNF grammar of ProGQL

node `f`, and `forward(entry)` traces forward from the node `entry`. The search can be constrained using the rules ⟨*Match*⟩ and ⟨*Where*⟩, which filter candidate edges or nodes at each expansion step.

The rule ⟨*Match*⟩ uses ⟨*Pattern*⟩, which can be a combination of multiple ⟨*PatternPart*⟩ separated by commas to specify an edge pattern for finding qualified edges (Line 1). Or it can be a single identifier within a collection (⟨*IdInColl*⟩) (Line 8: `n in nodes(r)`), or an expression ⟨*Expr*⟩ (Line 2: `v=dst(r)`) to specify qualified nodes. The rule ⟨*Where*⟩ specifies the search conditions using the rule ⟨*Expr*⟩, which is defined to express logical and arithmetic operations, including negation, conjunction (AND), disjunction (OR), comparison operations, traversal expressions (⟨*TraversalExpr*⟩), and arithmetic expressions (⟨*ArithmeticExpr*⟩) on edge properties (⟨*PropExpr*⟩).

The rule ⟨*TraversalExpr*⟩ further enhances expressiveness by allowing aggregation over traversals. Specifically, ⟨*TraversalExpr*⟩ uses the rule ⟨*Max*⟩ or the rule ⟨*Min*⟩ to compute the maximum or minimum value of the selected property, and ⟨*Collect*⟩ to gather sets of properties, enabling iterative constrained search. For example, the condition (Line 2: `r.starttime < max(collect(vout IN out(v) | vout.endtime))`) is used to retain only the edges whose start time are earlier than the max end time of all the outgoing edges. The rules ⟨*Atom*⟩, ⟨*PropLookup*⟩, ⟨*FuncInvoc*⟩, ⟨*FuncName*⟩ and ⟨*ParenExpr*⟩ allow flexible combinations of edge properties and functions.

Weight Computation. The graph generated by the constraint graph search contains the contextual information of an attack. However, this graph can be gigantic, typically containing > 100,000 edges. As a result, it is difficult for security analysts to find the edges that are critical to the attack. To address this issue, ProGQL extends the rules ⟨*Unwind*⟩, ⟨*Set*⟩ and defines a new rule ⟨*Projection*⟩ to compute edges weights so that edge weights based on critical security properties can be defined to filter irrelevant edges [17], [24]. The rules ⟨*Unwind*⟩ and ⟨*Set*⟩ are used to specify edge weight computation based on edge properties such as data amount, in-degrees, out-degrees, and time, which are specified in the rule ⟨*Projection*⟩ (Lines 4). Recent studies [13], [17], [24] show that combinations of multiple edge properties work well for a wide range of attacks. In particular, the rule ⟨*Projection*⟩ indicates to use a projection function that projects the selected property values into a single-dimensional weight such as LDA [46] to obtain the edge weight score. An edge with a higher dependency weight score implies more relevance to the POI event, and is more likely to be a critical edge.

Value Propagation. To reveal attack entries, ProGQL extends the rules ⟨*Match*⟩, ⟨*Set*⟩ and ⟨*Reduce*⟩ to compute nodes' values based on edge weights, which is used to model the node's impact on the POI event. In Query 1, the score propagation scheme computes the impact score of a node as a weighted sum of its children's impact scores.

Graph Merge. Existing studies [9], [24] show that multiple PA can be used to connect attack behaviors across hosts, or perform more effective filtering on irrelevant edges. ProGQL

defines graph merge composition through the rules ⟨*ProGQL*⟩ and ⟨*ProGQLQuery*⟩. The rule ⟨*ProGQL*⟩ specifies that a ProGQL program may consist of one or more queries, optionally combined with the keywords `union` or `intersect`. Each merge operation takes as input a ⟨*ProGQLQuery*⟩ enclosed in parentheses, allowing the results of multiple queries to be unified into a single output graph. The rule ⟨*ProGQLQuery*⟩ further refines this by allowing each query to contain one or more ⟨*SingleQuery*⟩ statements, optionally followed by additional merge clauses (`union` or `intersect`) together with a ⟨*WithQuery*⟩. This enables analysts to express multi-stage query pipelines where intermediate graphs are named and then combined. ⟨*WithQuery*⟩ provides a hook to bind intermediate results for further processing. Its full semantics are illustrated in the Query 1 (e.g., ranking and selecting entry nodes before forward search).

Query Example. Query 1 shows a PROGQL query for investigating password crack. After successfully penetrating host1, an attacker downloads and executs malicious scripts to identify additional victim hosts, such as host2. The attacker then cracks host2's password data, copys it back to host1, and compresses it for further malicious activity.

```
1  MATCH (p:Process)-[st:FileEvent{optype:"write"}]->(f:File{
       name:"/tmp/passwords.tar.bz2", hostid:"1"})
2      BFS (r IN backward(f) | MATCH v=dst(r) WHERE r.
       starttime<max(collect(vout IN out(v) | vout.endtime)))
       YIELD g1
3      UNWIND g1 AS e
4      SET e.weight=projection(1/(abs(r.amount-st.amount)
       +0.0001),ln(1+1/abs(r.endtime-st.endtime)),count(out(v)
       )/count(in(v)))
5      MATCH u=src(e) SET u.rel=reduce(sum = 0, o IN out(u
       ) | sum+o.weight*dst(o).rel)
6      RETURN g1
7  intersect
8  WITH entry = (MATCH n in nodes(r) WHERE count(in(n))=0
       ORDER BY n.rel DESC LIMIT 15)
9      BFS (re IN forward(entry) | MATCH u=src(re) WHERE
       re.endtime>min(collect(uin IN in(u) | uin.starttime))
       and re.starttime<1724731846719889370) yield g2
10     RETURN g2
11
12 UNION
13 (MATCH (p:Process)-[st:NetworkEvent{id:100005}]->(f:Network
       {srcip:"192.168.1.128/32",dstip:"192.168.1.131/32",
       hostid:"2"})
14     BFS (r IN backward(f) | MATCH v=dst(r) WHERE r.
       starttime<max(collect(vout IN out(v) | vout.endtime)))
       YIELD g3
15     UNWIND g3 AS e
16     SET e.weight=projection(1/(abs(r.amount-st.amount)
       +0.0001),ln(1+1/abs(r.endtime-st.endtime)),count(out(v)
       )/count(in(v)))
17     MATCH u=src(e) SET u.rel=reduce(sum = 0, o IN out(u
       ) | sum+o.weight*dst(o).rel)
18     RETURN g3
19 intersect
20 WITH entry = (MATCH n in nodes(r) WHERE count(in(n))=0
       ORDER BY n.rel DESC LIMIT 15)
21     BFS (re IN forward(entry) | MATCH u=src(re) WHERE
       re.endtime>min(collect(uin IN in(u) | uin.starttime))
       and re.starttime<1724731846712161377) yield g4
22     RETURN g4)
```

Query 1: PROGQL Query for Password Crack

In Query 1, for instance, the union of two hosts' output PGs is computed (Line 12) to help reassemble an attack scenario that involves password cracking and data exfiltration across hosts. For each host, a backward graph search and a forward graph search are employed and their output graphs are intersected to form the host's PG. Specifically, PROGQL uses the rule ⟨*WithQuery*⟩ to bind intermediate subgraphs and identify entry nodes for further exploration. In Query 1, `WITH entry = (...)` names a subgraph expression that can be reused in later traversals or merge operations. Entry nodes are selected with `nodes(r)` and filtered using `count(in(n))=0`, which captures provenance boundaries where external influence first enters the system. Candidate entry nodes are then ranked by their propagated impact scores using ⟨*Order*⟩, and ⟨*Limit*⟩ is applied to restrict forward exploration to the top-k nodes. Once these entry nodes are chosen, a forward ⟨*Bfs*⟩ is performed from them to construct a new provenance subgraph. This forward PG is then combined with the backward PG using `intersect`, which captures the overlap between the two searches and yields a focused explanation of the attack chain.

C. Query Execution Engine

The query engine executes the query context generated by the language parser and outputs a PG. It consists of four components: ① graph traversal, ② weight computation, ③ value propagation, ④ graph merge.

In the graph traversal component, the engine synthesizes database queries to perform graph searches with the help of the database backend. In the weight computation component, when a query specifies edge re-materialization and weighting (e.g., `UNWIND g1 AS e` followed by `SET e.weight=projection(...)`), the engine first applies an edge merge technique to reduce the size of the PG that is generated by the graph search. Then it will compute the edge's weight score based on the edge features defined in the projection function and employ a discriminate feature projection scheme based on LDA to project the selected features into a single-dimensional weight, so that critical edges can be better revealed. In the value propagation component, when a query specifies score assignment and propagation (e.g., `MATCH u = src(e)` followed by `SET u.rel = reduce(...)`), the engine applies a weighted score propagation scheme to propagate the impact score from the POI node to all other nodes in the PG. In the output processing component, if `union` or `intersect` is declared in the query, the engine will merge multiple generated PGs generated and output a final PG. We next describe each phase in detail.

Component ①: Graph Traversal. Based on the PROGQL query semantics, the engine synthesizes database queries (SQL for the relational database, nGQL for NebulaGraph, and traversal API calls for Neo4j), which first identify the starting node of the traversal (POI node that is extracted from the POI event defined in the graph search query, or attack entry node defined in the `WITH` query) and add it to a queue. Then PROGQL performs incremental backward/forward search from the starting node, without the need to load all events that happen before the starting node. Such incremental search greatly reduces the number of events to be checked as the number of events to be loaded is usually huge (easily exceeding 10 million events) and many events do not have dependency paths that lead to

the starting node. Note that this incremental search will not be possible if a database backend is not adopted and the events are not indexed.

With the starting node in the queue, the engine pop ups the node from the queue and constructs database queries based on the search constraints (e.g., the `MATCH` query in Line 2 in Query 1) to find eligible incoming/outgoing edges. If any edges are found, their source/sink nodes are added to the queue for further search, and the engine continues to pop up the node from the queue. This process repeats until the queue is empty or the search exceeds the resource limits. [47] shows the detailed implementation of a backward BFS search from a POI node.

Component ②: Weight Computation. Weight computation consists of three major steps:

- *Edge Merge*: the engine reduces the PG size by merging parallel edges between two nodes in the PG. PA often results in a PG with numerous parallel edges connecting two nodes, which arises from the operating system's tendency to distribute data across several system calls upon completing read/write tasks, such as file operations [8], [42]. The engine addresses this issue by merging edges between nodes when the time differences are below a specified threshold, following existing works [22], [24].
- *Weight Evaluation*: the engine computes the weights of the edges using the features defined in the `projection` function. The engine extracts features for each edge from the `projection` function, along with the corresponding arithmetic expressions for weight calculation. Subsequently, an arithmetic evaluator is employed to calculate the weight of each individual feature. For example, in Query 1, `ln(1+1/abs(r.endtime-st.endtime))` represents a temporal weight that models the temporal relevance of an edge r to the POI event st. Similarly, `1/(abs(r.amount-st.amount)+0.0001` represents a data flow weight that models the data flow relevance of r to st.
- *Weight Normalization*: the engine normalizes the weights of all outgoing edges for each node, and thus the final weights are local weights for each node. It is an optional step, and it can potentially mitigate the weight degradation for the edges that are far from the POI event. To do so, the engine applies a clustering algorithm such as KMeans [48] or DBScan [49] to separate edges into critical edges group and non-critical edges group. After the clustering, the engine employs a discriminative feature projection scheme (e.g., LDA) to compute a final weight based on the individual features' weights. Finally, the engine normalizes an edge $e(u, v)$'s final weight by the sum of weights of all outgoing edges of the source node u.

Component ③: Value Propagation. The engine propagates values from the POI node to all other nodes backward along the weighted edges following the `reduce` function definition in the query. For example, according to `reduce(sum = 0, o IN out(u) | sum+o.weight*dst(o).rel)`, an arithmetic evaluator is used to calculate a node's impact score by taking the weighted sum of impact scores of its child nodes. Note that the value propagation is recursive. In each iteration, the node's impact score is updated based on the impact scores of its child nodes. The iterative process continues until the impact scores converge to stable values. The engine will terminate the propagation when the aggregate difference between the current iteration and the previous iteration is smaller than a threshold (e.g., $1e$-13), as it indicates that the propagated values become stable after iterations.

Component ④: Graph Merge. This component merges the PGsproduced by each sub-query and supports query-level merge constructs such as `union(g1, g2)`, which combines the nodes and edges of two subgraphs, and `intersect(g1, g2)`, which retains only the overlapping portions. To perform merging efficiently, the engine constructs an edge signature by leveraging the unique identifiers of both the source and sink nodes; this signature enables fast decisions on whether an edge should be included, greatly improving performance when combining multiple PGs. Finally, the merged PG is transformed into the output format specified in the ProGQL query.

V. Evaluation

We built ProGQL (∼32K lines of code for the language and ∼10K lines of code for log parser) on top of both relational database PostgresSQL 9.5 [36], MyRocks (a.k.a, Facebook MySQL 5.6) [37], Mariadb 10.4.19 [50] and graph database Neo4j 3.5.11 [35], Nebula 3.3.0 [38] and evaluate ProGQL using both the attack cases constructed based on the known exploits [13], [22], [51], [52] and the attack cases collected by the DARPA Transparent Computing (TC) program [53]. We constructed 14 ProGQL queries to investigate these attacks, demonstrating the effectiveness of ProGQL in searching attack behaviors for diverse attacks. In the evaluations, we aim to answer the following research questions:

- **RQ1**: How effective is ProGQL in expressing different PA to generate PGs for finding attack behaviors?
- **RQ2**: How does ProGQL improve the efficiency and accuracy of identifying attack patterns in audit logs compared to the most popular graph query language, Cypher?
- **RQ3**: How effective is ProGQL in revealing attack steps for advanced cyberattacks? How does ProGQL compare with other state-of-art (SOTA) techniques in attack investigation?
- **RQ4**: How flexible can ProGQL support different weight computations?
- **RQ5**: How do different backend databases impact ProGQL's runtime performance?

A. Evaluation Setup

We deployed ProGQL on a server with an Intel(R) Xeon(R) CPU E5-2637 v4 (3.50GHz), 256GB RAM running 64bit Ubuntu 16.04.7. Neo4j and Nebula databases are configured by importing system entities as nodes and system events as relationships. To evaluate the effectiveness and performance of ProGQL, we deployed Sysdig [54] on Linux hosts to

collect system auditing events and performed a broad set of attack behaviors including 6 attacks based on commonly used exploits and 3 multi-host intrusive attacks based on the Cyber Kill Chain framework [55] and CVE reports [56]. The DARPA datasets include system audit logs collected from 4 hosts with different OS systems.

Batch Insertion. Batch insertion can support 3000 events/second (supporting 50-100 monitored hosts) using HDD, and improve to 8000 events/second using SSD. Modern DBs support incremental index building and query speed is improved significantly by 2-3 times using SSD. Thus, PROGQL can support near-real-time queries.

Database Backend Setup. We developed a tool to parse Sysdig logs and DARPA released logs and loaded them into five databases: PostgresSQL, MyRocks, Mariadb, Neo4j, and Nebula. We then deploy PROGQL to use these five databases as five types of database backends. For evaluation, we used datasets containing a total of 140,523 entities and 28,088,979 events, in addition to the separate DARPA datasets with 46,756,662 and 78,219,245 events, respectively. We next describe these attacks in detail.

1) Attacks Based on Commonly Used Exploits: These 6 attacks are used in prior work's evaluations [13], [22], [51], [52], which consist of the following scenarios:

- *Wget Executable*: A vulnerable server allows the attacker to download executable files using wget. The attacker downloads Python scripts and executes the scripts to write some garbage data to a user's home directory.
- *Illegal Storage*: A server administrator uses wget to download suspicious files to a user's directory.
- *Hide File*: The goal of the attacker is to hide malicious files among a user's normal files. The attacker downloads the malicious file and hides it by changing its file name and moving it to a user's home directory.
- *Steal Information*: The attacker steals the user's sensitive information and writes the information to a hidden file.
- *Backdoor Download*: A malicious insider uses the ping command to connect to the malicious server, and then downloads the backdoor script from the server and renames the script to hide it.
- *Annoying Server User*: The annoying user logs into other users' directories on a vulnerable server, and runs the backdoor script downloaded from the malicious server using ping command to write some garbage data to other users' directories.

2) Multi-host Intrusive Attacks: This multi-host intrusive attack encapsulates key characteristics found in the Cyber Kill Chain framework [55] and Common Vulnerabilities and Exposures (CVE) [56]. In these three attack scenarios, the attacker leverages an external host, designated as the C2 (Command and Control) server, to execute penetration, disseminate malware, steal data, and establish the persistent connection. The target host compromised by the attack is referred to as the Victim host.

Attack1: Password Crack After Shellshock Penetration. The Shellshock vulnerability was a critical flaw discovered in the GNU Bash shell. It allowed attackers to execute arbitrary commands on a targeted system through specially crafted environment variables. When Bash processes these variables, the attacker's code gets executed. This attack exploited the Shellshock vulnerability to infiltrate a vulnerable host (host1). Upon successful compromise, the attacker establishes a reverse shell connection to remotely control host1. In this stage, the attacker generally takes a series of stealthy reconnaissance maneuvers. Among those, we emulate the password cracking attack. The attacker downloads and executes a malicious script `gather_password.sh`. This script identifies victim hosts (i.e., host2) and downloads another malicious script `crack_passwd.sh`, transfers it to host2 and executes it. `crack_passwd.sh` then downloads a series of files, including a malicious payload `libfoo.so` from the attack server. `libfoo.so` cracks passwords on the victim host. The resulting `password_crack.txt` file contains plaintext passwords, which is then transferred to host1 and compressed as the `/tmp/passwords.tar.bz2`. This file serves as a consolidated package of sensitive information, ready for exfiltration.

Attack 2: Data Leakage. After Shellshock Penetration, the attacker attempts to steal all the valuable assets from the host. This stage mainly involves the behaviors of local and remote file system scanning activities, copying and compressing of important files. The attacker initiated the second stage of the attack by downloading the script `leak_data.sh` from their server. The script was then transferred to a compromised host (host2) using scp and executed. The `leak_data.sh` script bundled specific files, including hidden files and sensitive system files, into a tarball. This tarball was then compressed as `leaked.tar.bz2` and exfiltrated to the attacker's designated server for further use.

Attack 3: VPN Filter. At this stage, the attacker targeted to establish a persistent connection between the victim hosts and the C2 server. The attacker utilized VPNFilter malware, which infected millions of IoT devices by exploiting a number of known or zero-day vulnerabilities. The attacker first downloaded `vpn_filter.sh` script to host1, the script then was transferred to host2 and executed. `vpn_filter.sh` changed to the /tmp directory on host2, downloaded `vpnfilter` from the attacker's server, made it executable, and then ran it with C2 server IP address. This established a persistent connection between host2 and the C2 server.

3) DARPA TC Attack Cases: The datasets provided by the DARPA TC program feature instances of attacks conducted on various operating systems. Our selection process, guided by DARPA's ground truth document, involved excluding failed attack cases and those targeting the Android system. The latter was omitted due to the constrained behaviors of mobile applications within the Android sandbox, rendering them unsuitable for our analysis. Additionally, we opted to exclude phishing email attacks, given their reliance on browser interactions, which result in limited traces within system audit logs.

Ultimately, our focus narrowed to five distinct attacks that

TABLE IV: Statistics of all the 14 attacks

Attack	Critical Event	System Entity	System Event
Wget Executable	12	134,066	27,836,522
Illegal Storage	6	134,066	27,836,522
Hide File	8	134,066	27,836,522
Steal Information	7	134,066	27,836,522
Backdoor Download	6	134,066	27,836,522
Annoying Server User	12	134,066	27,836,522
Password Crack	37	6,457	252,457
Data Leakage	17	6,457	252,457
Vpn Filter	16	6,457	252,457
Theia Case 1	7	1,487,424	8,704,250
Theia Case 3	4	1,487,424	8,704,250
Fivedirections Case 1	4	814,091	5,092,216
Fivedirections Case 3	2	814,091	5,092,216
Trace Case 5	3	44,455,147	64,422,779
Average	10	3,562,996	18,556,587

target diverse operating systems such as Linux and Windows, exploiting vulnerabilities such as the Firefox backdoor and browser extensions. Notably, these attack cases unfold over extended periods, exemplified by Theia data encompassing logs spanning 8 days.

4) Obtaining Ground Truth for the Attacks: Within our best efforts, we manually ensured that the critical events that represent attack steps were identified based on the knowledge of the attacks performed by us and DARPA attack descriptions in these PGs. Table IV shows the statistics of all the 14 attacks. Columns "Critical Event" shows the critical edges that represent attack steps of the PGs. "System Entity" and "System Event" show the number of entities and events that were imported into the database by our Data Importer. On average there are 3,562,996 entities and 18,556,587 events (with the max being 44,455,147 entities and 64,422,779 events). Loading them into the memory and performing the PA are super expensive, which motivates the database approach provided by ProGQL.

Evaluation Metrics. To measure the effectiveness of ProGQL, we counted false positives (detected edges that are not critical edges) and false negatives (missing edges that are critical edges) and showed them in Table VII Columns "FP" and "FN".

B. RQ1: Expressing PA for Attacks

We compose 14 ProGQL queries for the 14 attacks and execute them in our deployed ProGQL framework to generate PGs for performing attack investigation. Table V presents statistical insights into the generated PGs. Based on the existing studies [12], [24], [34], ProGQL queries are formulated to comprise of 1∼3 sub-queries. In each sub-query, The first nested sub-query initiates from a given POI event and conducts backward PA with the specified qualification constraints, aiming to identify all the potentially attack-related edges. This nested sub-query also computes edge weights based on three features (data amount, relative time, and ratio of incoming and outgoing edges) as suggested by the existing studies, and propagates values to each identified nodes as each node's impact score. Note that the ProGQL language can easily incorporate any new edge-related feature as long as the system auditing events can provide the required data. Subsequently, in the second nested sub-query, we introduce a condition to choose the top N entry nodes as the starting nodes to perform the subsequent forward PA. Finally, the ProGQL query merges the output PGs from the all sub-queries using `INTERSECT` / `UNION`. The merged PG preserves the nodes and edges that are highly relevant to both the POI event and the attack entries. All queries are available in our project website [45].

Table V presents the statistical insights of the generated PGs, where the Column "# Edges of Backward PA" shows the number of edges in the PG generated by the first sub-query, Column "# Edges of Forward PA" shows the number of edges in the PG generated by the second sub-query, and Column "# Edges of Output PG" shows the number of edges in the final merged PG. We can observe that the PG generated by the backward sub-queries is very large (averagely 172,097), while the final merged PG has about 176 edges (merely 0.10%). This shows that *multiple PA collaborates with each other with the help of weight computation and value ranking is very effective in reducing irrelevant edges, which is consistent with the previous studies [9], [24].*

Execution Performance and Memory Consumption. Table IV shows that the number of events to search is about 19 million on average, while the output PG that are useful for attack investigation has the average size of 176, which is like "finding needle in haystack". With the edge indexing and the incremental graph search, when deploys ProGQL on Neo4j, it can conduct such search using averagely 5GB memory and finish the search within 224 seconds, while existing works such as DepImpact [24] that load all the data into the memory can easily consume up to 100GB memory (see Section V-C) or cause time out. These results demonstrate the optimization brought by our query engine design and the improved scalability.

Top N Ranked Nodes for Forward PA. Column "Top Ranked Nodes" shows the number of selected entry nodes for performing the forward PA. With the help of the ProGQL language, security analysts can effortlessly customize this number by adapting the query (`WITH entry = (MATCH n in nodes(r) Where count(in(n))=0 order by n.rel desc limit "N")`) to align with their domain knowledge and the observed outputs. In our evaluations, we keep increasing the value of N until all critical edges are found or a larger N will include too much irrelevant edges into the final PG.

Customized Filtering Constraints. Unlike existing PA techniques, as a domain-specific language, the ProGQL language allows security analysts to effortlessly incorporate additional constraints into the query for filtering irrelevant edges based on their domain knowledge or the observed outputs. In our evaluations, we added customized filters to filter out irrelevant system libraries files that are read-only and are not tampered with. While these libraries may vary in different systems, our queries can easily adapt to different systems by adjusting the filtering constraints. Furthermore, we added temporal constraints specifically in the analysis

TABLE V: PA executed by PROGQL for 14 Attacks

Attack	# Sub-Q	# E of Backward PA	Top Nodes	# E of Forward PA	# E of PG	Memory (GB)	Exe. Time (s)
Wget Executable	1	46,798	5	209	65	1.61	99
Illegal Storage	1	34,242	3	46,956	212	1.94	128
Hide File	1	43,444	2	20,726	46	1.63	90
Steal Information	1	52,292	6	26,773	50	2.01	140
Backdoor Download	1	46,454	2	128	47	1.44	93
Annoying Server User	1	2,046,735	3	158	46	15.07	629
Password Crack	2	20,349	30	27,797	282	1.94	23
Data Leakage	2	24,177	10	23,254	281	1.45	16
Vpn Filter	3	50,935	18	30,631	637	1.83	23
Theia Case 1	1	27,397	2	1,398	339	3.00	242
Theia Case 3	1	8	3	1,039,978	6	8.30	320
Fivedirections Case 1	1	319	3	32	14	1.46	56
Fivedirections Case 3	1	17	1	604,836	17	5.32	210
Trace Case 5	1	827	3	1,028,353	474	24.76	1,069
Average							

TABLE VI: PGs produced by PROGQL and Cypher (Neo4j)

Attack	Cypher (Neo4j)			ProGQL			Cypher (Neo4j) Execution Time(s)	ProGQL Execution Time(s)	Cypher (Neo4j) Memory Consumption(GB)	ProGQL Memory Consumption(GB)
	FP	FN	Edges	FP	FN	Edges				
Password Crack	21,481	0	21,518	245	0	282	612	23	5.53	1.94
Data Leakage	21,539	0	21,556	264	0	281	185	16	1.26	1.45
Vpn Filter	24,817	0	24,827	627	0	637	638	23	2.09	1.91
Average										

of DARPA TC attacks, given the substantial size of their datasets. For instance, the Theia dataset spanning logs over 8 days. Without imposing time constraints, conducting a search from a POI event would undoubtedly result in an overwhelmingly large PG. To manage this, we apply time constraints (`MATCH v=dst(r) Where r.starttime<max(collect(vout IN out(v) | vout.endtime)) and r.starttime>=timestamp1 and r.endtime<=timestamp2`), restricting our search to the specific day when a particular attack occurred based on the attack descriptions provided by the datasets.

Case Study. We use password crack attack as a case study, Figure 2 shows critical edges of it. We illustrate, step-by-step, how ProGQL queries can be formulated in an interactive query style to perform provenance analysis for attack investigation.

- *Step 1 - Backward BFS from the POI*: Starting from the point of interest (POI), `/tmp/passwords.tar.bz2` on host1, ProGQL performs a backward BFS with temporal constraints:

```
MATCH (p:Process)-[st:FileEvent{id:15035}]->(f:File{name:
    "/tmp/passwords.tar.bz2", hostid:"1"})
       BFS (r IN backward(f) | MATCH v=dst(r) WHERE r.
    starttime<max(collect(vout IN out(v) | vout.endtime))
    ) YIELD g1
       RETURN g1
```

This query produces a PG with 154 vertices and 3117 edges.

- *Step 2 - Edge weighting and impact score propagation*: We next assign weights and propagate impact scores across the backward graph:

```
MATCH (p:Process)-[st:FileEvent{id:15035}]->(f:File{name:
    "/tmp/passwords.tar.bz2", hostid:"1"})
       BFS (r IN backward(f) | MATCH v=dst(r) WHERE r.
    starttime<max(collect(vout IN out(v) | vout.endtime))
    ) YIELD g1
       UNWIND g1 AS e
       SET e.weight=projection(1/(abs(r.amount-st.amount
    )+0.0001),ln(1+1/abs(r.endtime-st.endtime)),count(out
    (v))/count(in(v)))
       MATCH u=src(e) SET u.rel=reduce(sum = 0, o IN out
    (u) | sum+o.weight*dst(o).rel)
       RETURN g1
```

This step highlights critical dependencies by assigning high scores to influential nodes in PG.

- *Step 3 - Backward + forward analysis with entry selection*: PROGQL then selects the top-ranking candidate entry nodes and performs forward BFS from them. Finally, it intersects the backward and forward graphs to isolate the most relevant PG:

```
MATCH (p:Process)-[st:FileEvent{id:15035}]->(f:File{name:
    "/tmp/passwords.tar.bz2", hostid:"1"})
       BFS (r IN backward(f) | MATCH v=dst(r) WHERE r.
    starttime<max(collect(vout IN out(v) | vout.endtime))
    ) YIELD g1
       UNWIND g1 AS e
       SET e.weight=projection(1/(abs(r.amount-st.amount
    )+0.0001),ln(1+1/abs(r.endtime-st.endtime)),count(out
    (v))/count(in(v)))
       MATCH u=src(e) SET u.rel=reduce(sum = 0, o IN out
    (u) | sum+o.weight*dst(o).rel)
       RETURN g1
intersect
WITH entry = (MATCH n in nodes(r) WHERE count(in(n))=0
    ORDER BY n.rel DESC LIMIT 15)
       BFS (re IN forward(entry) | MATCH u=src(re) WHERE
     re.endtime>min(collect(uin IN in(u) | uin.starttime)
    ) and re.starttime<1724731846719889370) yield g2
       RETURN g2
```

The resulting PG has 39 vertices and 86 edges.

- *Step 4 - Multi-host correlation*: Repeating the same analysis on host2 and unioning the graphs from host1 and host2 yields a concise multi-host attack PG.

```
MATCH (p:Process)-[st:FileEvent{id:15035}]->(f:File{name:
    "/tmp/passwords.tar.bz2", hostid:"1"})
       BFS (r IN backward(f) | MATCH v=dst(r) WHERE r.
    starttime<max(collect(vout IN out(v) | vout.endtime))
    ) YIELD g1
       UNWIND g1 AS e
       SET e.weight=projection(1/(abs(r.amount-st.amount
    )+0.0001),ln(1+1/abs(r.endtime-st.endtime)),count(out
    (v))/count(in(v)))
       MATCH u=src(e) SET u.rel=reduce(sum = 0, o IN out
    (u) | sum+o.weight*dst(o).rel)
       RETURN g1
intersect
WITH entry = (MATCH n in nodes(r) WHERE count(in(n))=0
    ORDER BY n.rel DESC LIMIT 15)
       BFS (re IN forward(entry) | MATCH u=src(re) WHERE
     re.endtime>min(collect(uin IN in(u) | uin.starttime)
    ) and re.starttime<1724731846719889370) yield g2
       RETURN g2
UNION
(MATCH (p:Process)-[st:NetworkEvent{id:100005}]->(f:
    Network{srcip:"192.168.1.128/32",dstip:"
    192.168.1.131/32",hostid:"2"})
       BFS (r IN backward(f) | MATCH v=dst(r) WHERE r.
    starttime<max(collect(vout IN out(v) | vout.endtime))
    ) YIELD g1
       UNWIND g1 AS e
```

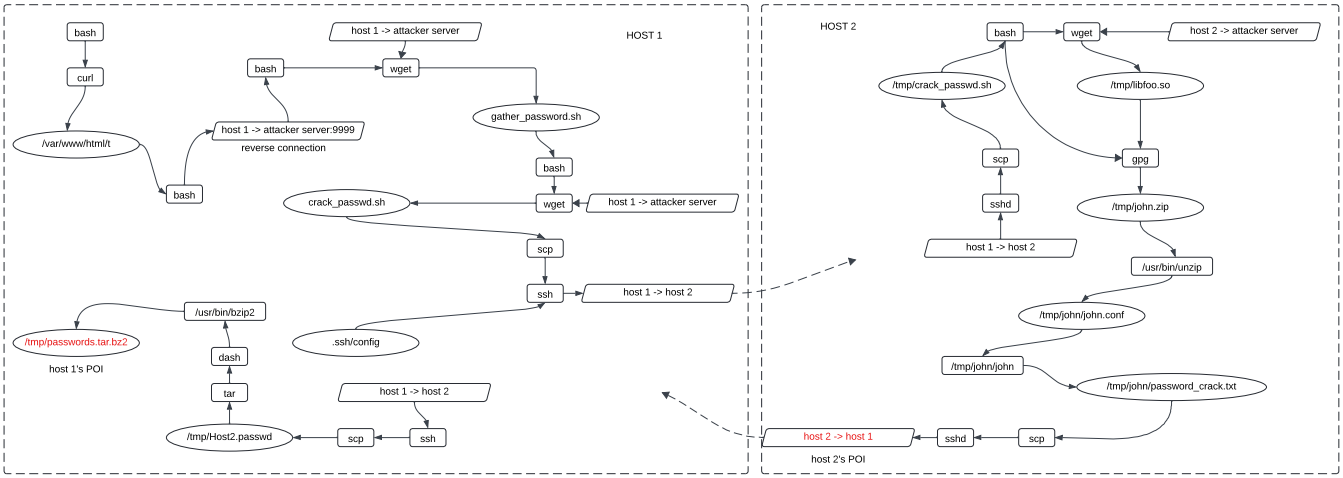


Fig. 2: PG identified by PROGQL for Password Crack attack (non-critical edges are omitted)

```
15        SET e.weight=projection(1/(abs(r.amount-st.amount
      )+0.0001),ln(1+1/abs(r.endtime-st.endtime)),count(out
      (v))/count(in(v)))
16        MATCH u=src(e) SET u.rel=reduce(sum = 0, o IN out
      (u) | sum+o.weight*dst(o).rel)
17        RETURN g1
18  intersect
19  WITH entry = (MATCH n in nodes(r) WHERE count(in(n))=0
      ORDER BY n.rel DESC LIMIT 15)
20        BFS (re IN forward(entry) | MATCH u=src(re) WHERE
       re.endtime>min(collect(uin IN in(u) | uin.starttime)
      ) and re.starttime<1724731846712161377) yield g2
21        RETURN g2)
```

The union PG has 124 vertices and 282 edges.

C. RQ2: Comparison with Cyher

We choose to compare against Cypher because other domain-specific languages either share its level of expressiveness (e.g., AIQL [31]) or lack graph search capabilities altogether (e.g., SAQL [32]). To evaluate language effectiveness, we conduct a series of experiments that assess both expressiveness and the ability to accurately reconstruct attack sequences. Our results show that only PROGQL can generate concise graphs that effectively capture attack patterns from audit logs. Cypher, despite its flexibility, requires overly verbose queries and fails to match PROGQL 's precision and performance. We next discuss the complexity of Cypher's query construction and its execution performance.

1) Complexity of Query Construction: PROGQL is specifically designed to streamline the process of querying audit logs for attack detection by efficiently performing constrained graph search, computing edge weights, and propagating values through the graph. In contrast, Cypher lacks the expressiveness for these critical functions, making it incapable of identifying attack entry points without extensive manual intervention. To compensate for this limitation in our evaluations, we manually identified the top ranked nodes and constructed corresponding Cypher queries to assess their ability to generate a concise graph that reveals the attack sequences. Unlike PROGQL, Cypher necessitates the construction of multiple, complex queries to achieve the same goal. For example, rewriting PROGQL Query 1 in Cypher requires 32 separate Cypher queries. Specifically, the forward BFS in PROGQL Query 1 (Line 8) needs to be repeated 15 times using queries like Query [57]. This not only increases the complexity of the query construction but also makes the process more error-prone and less maintainable.

TABLE VII: PGs produced by PROGQL and DEPIMPACT

Attack	DEPIMPACT			PROGQL		
	FP	FN	Edges	FP	FN	Edges
Wget Executable	56	0	68	53	0	65
Illegal Storage	206	0	212	206	0	212
Hide File	141	0	149	38	0	46
Steal Information	328	0	335	43	0	50
Backdoor Download	128	0	134	41	0	47
Annoying Server User	254	0	266	34	0	46
Password Crack	257	1	293	245	0	282
Data Leakage	268	0	285	264	0	281
Vpn Filter	575	4	581	627	0	637
Theia Case 1	12,734	0	12,741	332	0	339
Theia Case 3	797,509	0	797,513	2	0	6
Fivedirections Case 1	119,861	0	119,865	10	0	14
Fivedirections Case 3	173,996	0	173,998	15	0	17
Trace Case 5		OOM		471	0	474
Average						

TABLE VIII: Statistics of the execution time (second)

Attack	DEPIMPACT	PROGQL
Wget Executable	232	24
Illegal Storage	237	50
Hide File	349	36
Steal Information	268	55
Backdoor Download	317	22
Annoying Server User	631	629
Password Crack	9	23
Data Leakage	9	16
Vpn Filter	10	23
Theia Case 1	2,995	155
Theia Case 3	2,906	320
Fivedirections Case 1	242	8
Fivedirections Case 3	245	210
Trace Case 5	OOM	1,069
Average		

2) Inefficiency and Inaccuracy in Cypher: **False Positives.** Table VI shows Cypher generates a significant larger number of false positives in the output graphs (21,481 for Password Crack, 21,539 for Data Leakage, and 24,827 for VPN Filter). Cypher (even with APOC) is fundamentally path-

TABLE IX: Results of RQ1 variants with various weight filtering conditions

Attack	CPR (# E)	ProGQL $_w$ (# E)	# CritEdges	# CritKept	% CritLost
Wget Executable	1,967	167	12	6	50%
Illegal Storage	880	153	6	4	33%
Hide File	1,420	173	8	6	25%
Steal Information	1,650	168	7	4	43%
Backdoor Download	1,773	175	6	4	33%
Annoying Server User	6,901	762	12	9	25%
Password Crack	995	281	37	22	41%
Data Leakage	1,676	234	17	8	53%
Vpn Filter	12,062	1,344	16	10	38%
Theia Case 1	27,397	6,525	7	4	43%
Theia Case 3	8	6	4	4	0%
Fivedirections Case 1	319	244	4	2	50%
Fivedirections Case 3	17	2	2	0	100%
Trace Case 5	827	41	3	2	33%
Average	4,135	734	10	6	41%

oriented: it enumerates complete paths from the start node. Filters like `WHERE incoming.starttime < updatedMaxEndTime` are evaluated after a path has been generated, based only on the nodes and relationships in that path. This means Cypher cannot enforce recursive, edge-dependent constraints during the traversal itself. Instead, it allows traversal to continue along edges that should already have been pruned, and only later removes them from the returned results. As a consequence, Cypher produces additional edges (false positives) because it lacks ProGQL 's ability to prune dynamically at each recursion step. In contrast, ProGQL is subgraph-oriented: its BFS operator applies edge-dependent constraints at every step, ensuring that only qualified edges are included.

Case Study - Controlled depth comparison with Cypher. To further highlight ProGQL 's precision, we conducted a controlled depth comparison by running backward BFS from the POI on host1 with a maximum depth of nine. We then compared the provenance graphs produced by ProGQL and by a best-effort Cypher approximation of the same query. Below we show the Cypher query we constructed as a best-effort approximation of the ProGQL backward BFS; this highlights the structural similarities but also makes evident where Cypher fails to enforce recursive, edge-dependent constraints.

At depths (1–8), the difference is not immediately visible. At depth 9, the provenance graph returned by ProGQL has 119 vertices and 2998 edges PG, while the PG returned by Cypher has 138 vertices and 6600 edges. At shallow depths (1–8), the difference is not immediately visible, because most paths near the POI happen to overlap with causally valid edges, so Cypher's outputs look superficially similar. However, starting at depth 9, the divergence becomes pronounced. Cypher continues expanding along many irrelevant paths - such as PAM libraries, NSS lookups, and background socket activity - because it evaluates constraints only after paths are generated, rather than pruning edges recursively during traversal. ProGQL, by contrast, consistently excludes these false positives, maintaining a concise subgraph that reflects only attack-relevant dependencies.

```
MATCH (p:Process)-[st:FileEvent{optype:"write"}]->(start:File{name:"/tmp/passwords.tar.bz2", hostid:"1"})

// Carry the POI and initialize a max end-time constant. This matches the same starting condition as in ProGQL.
WITH start, 1724731846719889370 AS initialMaxEndTime

// APOC expands all backward paths up to maxLevel. Here, YIELD path always means the entire path from start to the current node, not just the most recent hop.
// This whole-path semantics is where Cypher diverges from ProGQL. ProGQL expands step by step with constraints applied at each hop; APOC dumps a superset of all traversed edges into path.

CALL apoc.path.expandConfig(start, {
    relationshipFilter: "<",
    minLevel: 0,
    maxLevel: 9,
    bfs: true,
    uniqueness: "NODE_GLOBAL",
    filterStartNode: false
}) YIELD path

// Label the path (visitedPath) and extract its current endpoint (currentNode).
WITH start, path AS visitedPath, last(nodes(path)) AS currentNode, initialMaxEndTime

// Intended to prune traversal depth.
WHERE length(visitedPath) < 9
// Tries to restrict expansion to edges that are in the current path. But because visitedPath is the whole path, this condition still admits edges from earlier steps - including ones that would have been pruned if filtering happened inline (like in ProGQL). This is the main FP source.
OPTIONAL MATCH (currentNode)-[outgoing]->()
WHERE outgoing IN relationships(visitedPath) OR currentNode = start

// Aggregate outgoing edges to compute the temporal bound. But since outgoing edges include superset ones, the max can be inflated.
WITH start, currentNode, outgoing, max(outgoing.endtime) AS maxEndtime, initialMaxEndTime, visitedPath

// Set the temporal constraint. At the POI, use the initial cutoff; otherwise, inherit the max endtime.
WITH start, currentNode, outgoing, initialMaxEndTime, visitedPath,
    CASE
   WHEN currentNode = start THEN initialMaxEndTime
   ELSE COALESCE(maxEndtime, initialMaxEndTime)
    END AS updatedMaxEndTime

// Apply causal pruning (only accept edges that end before the cutoff). But pruning happens after expansion, so invalid edges were already collected into visitedPath.
MATCH (currentNode)<-[incoming]-()
WHERE incoming.starttime < updatedMaxEndTime

// Gather pruned incoming edges and project source/sink sets. This simulates causal filtering, but again false positives that slipped into visitedPath remain in play.
WITH start, currentNode, COLLECT(incoming) AS updatedVisitedRels, updatedMaxEndTime, visitedPath
WITH start, currentNode, updatedVisitedRels, updatedMaxEndTime,
    [r IN updatedVisitedRels | startNode(r)] AS sourceNodes,
    [r IN updatedVisitedRels | endNode(r)] AS sinkNodes, visitedPath

// Output the traversal frontier.
RETURN DISTINCT currentNode, updatedVisitedRels AS visitedRels, updatedMaxEndTime, sourceNodes, sinkNodes;
```

Conciseness Evaluation. We evaluated query conciseness using three metrics: the number of constraints, words, and characters. A constraint is defined as any atomic restriction that filters, bounds, or enforces semantics in a query, including `WHERE` clauses (temporal, structural, or equality conditions), typed edge/node restrictions (e.g., `ptype:"write"FileEvento`), aggregations for pruning (e.g., `max(...)`, `min(...)`), and bounds such as `LIMIT` or depth cutoffs. Words are counted as tokens separated by whitespace, and characters as any non-whitespace

symbols. As shown in Figure 3, across three multi-host attack cases, PROGQL consistently outperforms Cypher in brevity, requiring on average 9× fewer constraints, 15× fewer words, and 17× fewer characters.

Execution Time and Memory Consumption. The experiments reveal a stark contrast in performance between the two languages. PROGQL not only executes significantly faster (e.g., 20s vs. 414s on average) but also consumes less memory (1.74GB vs. 3.02GB on average). For other larger datasets used in the experiments, Cypher's performance deteriorates severely, as it was unable to complete the queries within a reasonable timeframe (over 2 hours), making it impractical for attack investigation in large-scale environments.

D. RQ3: Comparison with SOTA PA Technique

To demonstrate the effectiveness of PROGQL in revealing the attack sequence, we compare PROGQL with the state-of-the-art (SOTA) technique: DEPIMPACT [24]. For each attack, PROGQL ranks the nodes based on their impact scores and chooses the entry nodes in each of the three system entity categories to perform forward PA analysis from the nodes in the order of decreasing impact scores. PROGQL stops choosing a new node once all the critical events that represent attack steps have been identified in the PG or if the newly chosen node causes the output PG to include significantly more edges. We apply the same entry nodes picking logic to DEPIMPACT to make a fair comparison.

1) Effectiveness Comparison: Table VII shows the comparison of the PGs produced by PROGQL and DEPIMPACT. The results show that although both techniques do not miss any critical edge that represents attack steps, the PGs produced by PROGQL are much smaller on the DARPA datasets. This is because DARPA attack cases span multiple days (e.g., the Theia dataset contains logs for 8 days), PROGQL can target the attacks accurately with less processing time by adding temporal constraints or system library constraints in the `Where` clause, which can filter out more irrelevant edges. On the other side, DEPIMPACT produced 484× larger graphs on average since it does not have the flexibility to add more filters. Moreover, DEPIMPACT encountered **out-of-memory errors (OOM)** when running on the DARPA datasets even if we increased the heap size to 100GB. This shows the major limitation of DEPIMPACT that loads all the event data into memory.

2) Efficiency Comparison: To evaluate the efficiency of PROGQL, we conducted experiments for each case on each database, repeating each test three times and calculating the average execution time. We picked the optimal performances among the six types of database backends (See Section V-A) and compared them with DEPIMPACT. To make the evaluation fair, we disabled/cleared database caches for PROGQL after finishing running each PROGQL query, and we excluded logs loading and parsing time from DEPIMPACT. The results are shown in Table VIII. Overall, PROGQL executes faster than DEPIMPACT (189s vs. 651s). These results indicate that ***even though* PROGQL *executes search through queries with databases, the index of the edges and the incremental graph search make up for the loss of performing search in the memory***. On the other side, DEPIMPACT spent much more time on the DarpaTC datasets because it lacks the flexibility to add the temporal filter.

3) Memory Consumption Comparison: Table X compares the memory consumption of DEPIMPACT with PROGQL with all the six types of database backends. We can observe that the memory consumption of DEPIMPACT is 8× greater than PROGQL, which shows the superiority of PROGQL that utilizes a database backend. Note that building a large memory pool is usually impractical for many companies due to the much higher costs. We can also observe that all databases require much less memories compared to DEPIMPACT.

E. RQ4: Flexibility in Edge Weight Computation

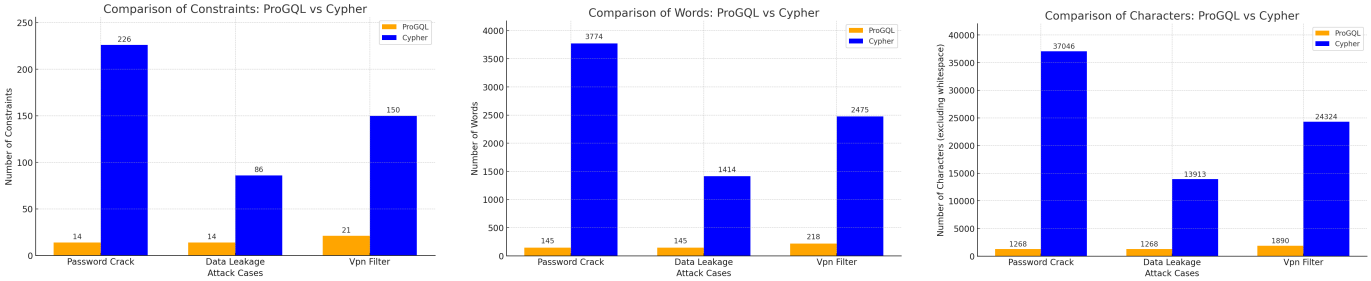
To demonstrate the flexibility of PROGQL, we implement variants of the PA used in RQ1 by adopting different weight-based mechanisms to filter out irrelevant edges. Specifically, we compose a variant for CPR [22] (without edge filtering based on weights) and a variant (PROGQL $_w$) that performs backward PA and retains edges with weights greater than or equal to 0.5 (Query 2). This shows that by easily changing the filtering condition specified in the ⟨*Where*⟩ rule, PROGQL can easily re-implement any existing weight-based PA [13], [17], [22].

```
MATCH (p:Process)-[st:FileEvent{optype:"write"}]->(f:File{
    name:"/home/fs/sysrep_random"})
  BFS (r IN backward(f) | MATCH v=dst(r) WHERE r.starttime<
    max(collect(vout IN out(v) | vout.endtime))) YIELD g1
  UNWIND g1 AS e
  SET e.weight=projection(1/(abs(r.amount-st.amount)
    +0.0001),ln(1+1/abs(r.endtime-st.endtime)),count(out(v)
    )/count(in(v)))
  WITH e WHERE e.weight >=0.5
  RETURN g1
```

Query 2: PROGQL Query with weight filtering

Table IX shows the results for the two variants of PROGQL. Column "CPR (# E)" shows the number of edges after applying CPR's edge merge (Section IV-C). Column "PROGQL (# E)" shows the number of edges output by PROGQL $_w$. Column "# CritEdges" displays the critical edges that represent attack steps of the PGs, while Column "# CritKept" shows how many of these critical edges are preserved by PROGQL $_w$. Finally, Column "% CritLost" quantifies the percentage of critical edges missed by PROGQL $_w$.

As we can see, although edge merging reduces parallel edges between nodes within a given time threshold, the resulting graph still contains a significant number of irrelevant edges (averaging 4,135) compared to the ground truth, which typically includes only 10 edges. PROGQL $_w$ greatly improves over CPR by removing edges using computed weights (from an average of 4,135 to 734), but it also causes significant loss of critical edges. Notably, some scenarios, such as "Fivedirections Case 3" experienced a complete loss of critical edges (100%). These results emphasize the need to carefully choosing weight computations for edge filering, and demonstrate



(a) Comparison of Constraints **(b) Comparison of Words** **(c) Comparison of Characters**

Fig. 3: Conciseness evaluation of queries written in PROGQL and Cypher

TABLE X: Statistic of memory consumption (GB)

Attack	DEPIMPACT	PROGQL				
		PostgreSQL	MyRocks	Mariadb	Neo4j	Nebula
Wget Executable	66.26	0.82	0.84	0.81	1.61	0.84
Illegal Storage	71.08	0.97	0.98	0.94	1.94	0.95
Hide File	74.76	0.89	0.93	0.97	1.63	0.93
Steal Information	70.70	1.05	1.43	1.03	2.01	1.01
Backdoor Download	72.19	0.79	0.89	0.82	1.44	0.87
Annoying Server User	79.48	15.14	15.38	13.32	15.07	19.82
Password Crack	1.67	0.80	0.80	0.89	1.94	0.82
Data Leakage	1.61	0.88	0.91	0.83	1.45	0.79
Vpn Filter	2.25	0.86	1.34	0.85	1.91	0.86
Theia Case 1	26.17	2.72	1.78	2.72	3.00	1.54
Theia Case 3	24.71	4.38	3.49	NA	8.30	3.26
Fivedirections Case 1	5.98	0.36	0.36	0.35	1.46	0.30
Fivedirections Case 3	7.21	3.25	1.64	1.75	5.32	1.71
Trace Case 5	OOM	7.92	6.95	7.48	24.76	NA
Average						

the flexibility of PROGQL in expressing PA techniques with different weight computations.

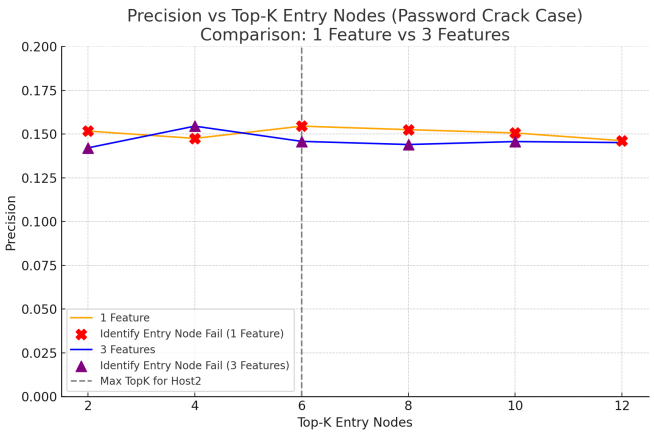
Supporting Different Edge Weights with Variable K in Top-K Entry Node Selection. As shown in a recent study [24], the choice of edge-weight assignments and the selection of K in top-K entry node selection are critical for preserving attack-related edges while filtering out irrelevant ones. Thus, we further conduct experiments on the choices of edge weights combined with different K in top-K entry node selection across three multi-host attack cases: Password Crack, Data Leakage, and Vpn Filter. PROGQL can naturally express both feature-based edge weighting and entry node candidate selection. For example, top-K entry node selection can be directly expressed using `LIMIT` with proper sorting, making it straightforward to select the most likely attack entry points. Similarly, feature selection can be declaratively controlled via projection functions in the query. For example, `SET e.weight = projection(1/(abs(r.amount - st.amount) + 0.0001))` uses only the data flow feature (amount) for weight computation, rather than the 3-feature expressed in Query 1.

As shown in Figure 4, we observed a natural trade-off between recall and precision in top-K entry node selection: a larger K improves recall but also introduces more false positives, since additional non-critical edges are included. The benefit of 3-feature weighting is evident in this trade-off. While both 1-feature and 3-feature weight computations eventually achieve high recalls, the 3-feature weight computation often reaches comparable recall earlier and maintains consistently higher precision by suppressing spurious edges. Moreover, 1-feature weight computation fails to identify ground-truth entry nodes for Password Crack and Data Leakage, whereas 3-feature weight computation avoids such failures, demonstrating greater robustness. Together, these results show that combining temporal, structural, and data-flow features yields more efficient prioritization, higher precision, and more reliable recall than relying on a single feature.

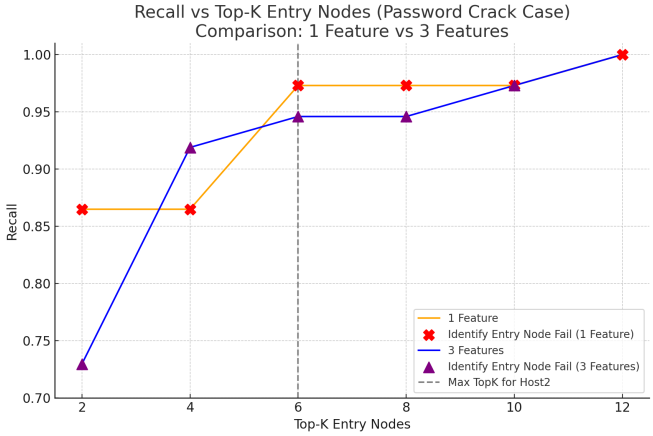
F. RQ5: Database Backend Comparison

We compare the performance of PROGQL using 5 types of database backends: PostgreSQL, MyRocks, MariaDB, Neo4j, and Nebula. The comparison of these databases is based on measurements of memory consumption (Table X) and running performance (Table XI). Since some database backends may run for a long time for certain queries, we terminated the query execution if no results are obtained within two hours.

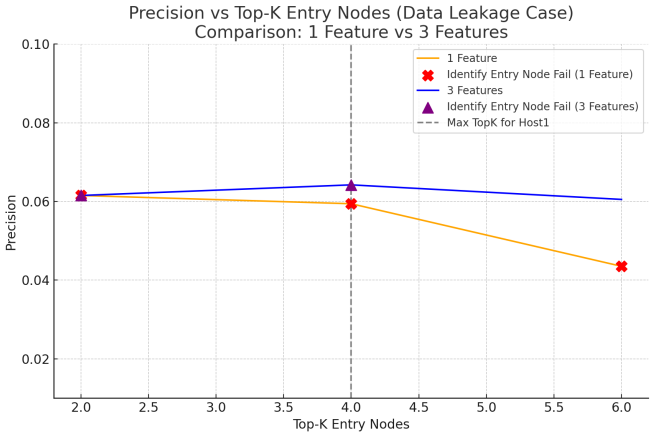
Runtime Performance Comparison. Based on Table XI, it becomes evident that among the databases completing all experiments within two hours, Neo4j demonstrates superior running performance (with an average of 224 seconds) over other databases, particularly when traversing through a substantial volume of nodes/edges. This performance superiority can be attributed to Neo4j's optimized graph traversal mechanism: Neo4j traversal API for graph search, which offers a flexible and expressive approach to define graph traversal logic. A interesting observation is that in scenarios where extensive volume is not a prerequisite, MyRocks and Mariadb surpass



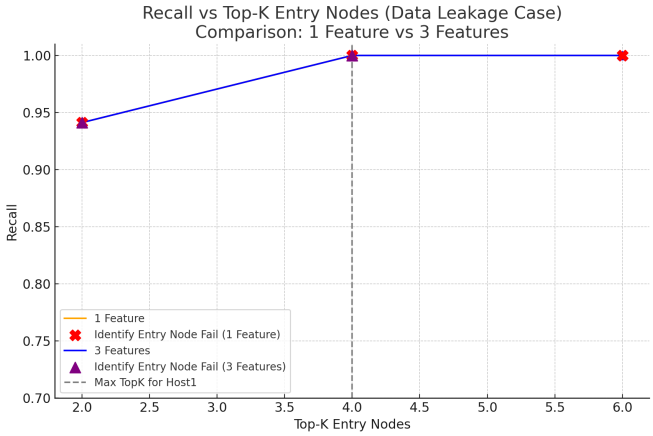
(a) Precision vs Top-K (Password Crack)



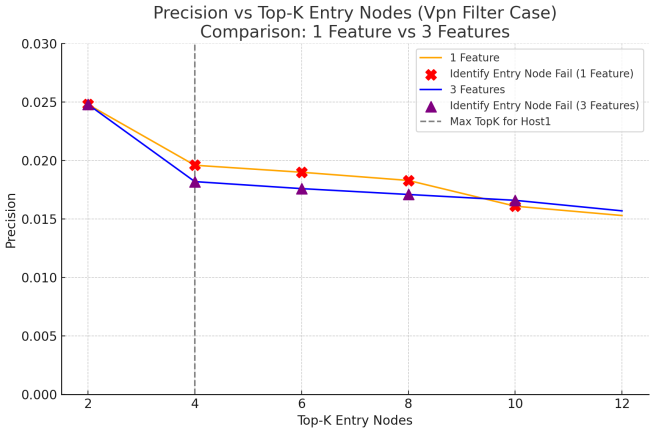
(b) Recall vs Top-K (Password Crack)



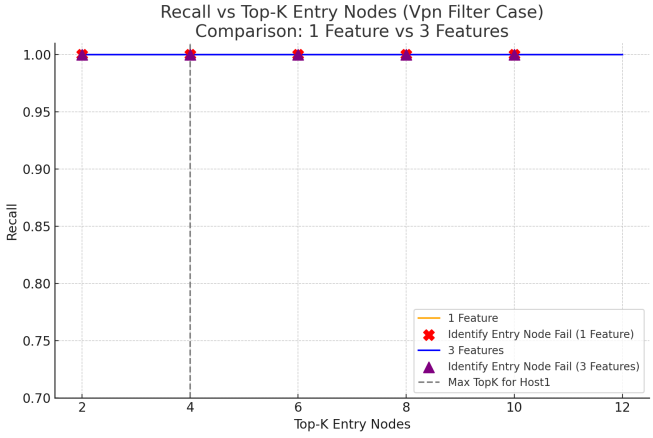
(c) Precision vs Top-K (Data Leakage)



(d) Recall vs Top-K (Data Leakage)



(e) Precision vs Top-K (VPN Filter)



(f) Recall vs Top-K (VPN Filter)

Fig. 4: Feature and TopK entry nodes ablation analysis across three attack cases (Password Crack, Data Leakage, VPN Filter)

TABLE XI: Statistics of database execution time (second)

Attack	PostgreSQL	MyRocks	Mariadb	Neo4j	Nebula
Wget Executable	60	26	24	99	236
Illegal Storage	82	59	50	128	226
Hide File	57	36	36	90	155
Steal Information	94	57	55	140	394
Backdoor Download	57	22	22	93	88
Annoying Server User	680	810	734	629	3,533
Password Crack	80	134	122	23	564
Data Leakage	27	41	35	16	151
Vpn Filter	72	116	112	23	469
Theia Case 1	155	230	1,370	242	180
Theia Case 3	491	684	$>$2 hr	320	4,821
Fivedirections Case 1	26	13	8	56	29
Fivedirections Case 3	222	435	365	210	1,358
Trace Case 5	1,271	2,823	2,396	1,069	$>$2 hr
Average					

other databases in running performance. It's interesting to observe that when Nebula is tasked with traversing a substantial volume of nodes/edges, it tends to persist indefinitely with relatively low memory consumption. For instance, it consistently consumes about 24GB even after running for 3 hours, and the mechanism behind this prolonged runtime may be caused by the employed memory consumption cap. In contrast, Neo4j automatically allocates more memory to ensure expedited traversals.

Memory Consumption Comparison. According to Table X, it is evident that among the databases where all experiments are successfully executed within two hours, MyRocks exhibits the lowest memory usage (average of 2.73GB), while Neo4j demonstrates the highest memory consumption (average of 5.13GB). The memory consumption for the executions that are longer than 2 hours are marked as "NA". This discrepancy can be attributed to the utilization of the Neo4j traversal API for graph search. Nevertheless, in comparison to other databases, Neo4j's higher memory usage can be linked to additional memory requirements for indexing, aiming at optimizing the graph traversal speed. The variance in memory requirements is also influenced by the unique indexing structures employed by relational databases for similar operations.

Recommended Database Backends. While no database is the clear winner for both memory consumption and execution time, we recommend using either Neo4j or PostgresSQL. In scenarios where memory consumption is not an issue (up to 25GB), Neo4j's fast search offers superior efficiency, benefiting from today's cheaper memory. Otherwise, PostgreSQL provides slightly slower search performance but operates within a smaller memory footprint (up to 15GB).

VI. Discussion

Data Compression. In our implementation, we employ some existing data compression techniques such as CPR [22]. There are several more recent works [23], [41], [43], [44] that achieve even better data compression rates. These compression approaches preserve the dependencies used by PA, and thus will not affect our search results and can be directly integrated with our ProGQL framework.

Mixed Database Backend. Our evaluation shows that relational databases are faster in searching for specific events while graph databases are more efficient in edge traversal tasks. One potential optimization is to combine the strengths of these two types of databases: using relational databases' indexes to find specific starting nodes and then using graph databases for graph traversal. Although Neo4j supports third-party indexes, the lack of native integration leads to additional overhead. Native index support in graph databases remains an area in need of further research.

Extensibility for Dependency-Centric Graph Domains. Beyond APT attack investigation, ProGQL's operators naturally generalize to any dependency-centric graph domain. For instance, constrained recursive search has also been applied to knowledge graphs or RDF database to discover multi-hop relationships under semantic or temporal constraints. Similarly, feature-driven edge weighting generalizes to applications in program analysis, supply-chain dependency tracking, or data lineage systems, where analysts must flexibly choose relevant edge features (degree, timestamps, amounts, etc.) to emphasize in traversal. ProGQL 's weight computation operator is designed to support this flexibility: it allows arbitrary combinations of neighbor-based features, encompassing prior designs such as the 3-feature and the 1-feature weight computations described in Section V-E.

Parallelism for Further Scaling. Our current implementation considers relational databases and graph databases as our database backend. As shown in the recent studies [31], [39], [58], [59], parallelism, which is orthogonal to what we propose in this paper, can further speed up the query search if the auditing event data is stored based on its domain-specific properties such as hosts and time. Furthermore, if the PGs built from different sub-queries in a ProGQL query has no data dependence on each other, which requires program analysis on the ProGQL query, then we can leverage "map-reduce" strategy to execute multiple search in parallel and then performs graph merge in the end. We also would like to explore the direction on adopting Hadoop [60] and MapReduce [61] to further scale up the data storage and speed up query search.

VII. Related Work

Domain-Specific Query Language for Security Applications. There exist domain-specific languages in a variety of

security fields that have a well-established corpus of low level algorithms, such as threat descriptions [62]–[64], secure overlay networks [65], [66], and network intrusions [67]–[70]. These languages provide specialized constructs for their particular problem domain. Recent works [31], [32], [39], [40], [58], [59] also provide domain-specific languages to express various patterns to detect attack behaviors from system audit logs. In contrast to these languages, the novelty of PROGQL focuses on provenance analysis by providing specialized constructs for recursive graph search and value propagation, which existing graph query languages [26], [29], [30] also do not support.

Provenance Analysis. King et al. [8], [9] proposed a backward causality analysis technique to perform intrusion analysis by automatically reconstructing a series of events that are dependent on a user-specified POI event. Following this research, recent efforts have been made to mitigate the dependency explosion problem [12], [13], [16], [17], [71]–[73]. In Section V, we have shown that PROGQL can express complex algorithm like DEPIMPACT [24]. Our proposed PROGQL system can well support different PA algorithms by expressing their search constraints except for intrusive system modifications like binary instrumentation [12], [16], and can also seamlessly integrate their optimizations by consuming the optimized events produced by these techniques.

Database Query Language. Database query languages are designed for general-purpose data search. Relational databases based on SQL and SPARQL [28], [36], [74], [75] provide language constructs for joins, facilitating specification of relationships among activities. Graph databases such as Neo4j [35] provide language constructs in their query language Cypher [26] for finding paths or nodes in graphs. NoSQL tools such as DynamoDB [76] and MongoDB [77] provide simpler language for fast data fetches based on keys. There are also other query languages for spatio-temporal databases [78]–[80]. None of these languages can express customized graph search and support value propagation like PROGQL, which is critical for PA. Based on prior languages (e.g., Cypher), CQL and SQL/PGQ [81] are proposed towards a standard query language on property graphs. Our proposed language structs built on top of Cypher comply with the standard and can be easily integrated.

System Analysis Language. Besides academia, industry has recently released several query languages designed for fine-grained system analysis. OSQuery [82], [83] lets analysts use SQL queries to probe the real-time system status. Elasticsearch [84] and Splunk [85] are log-analysis platforms that index general application logs, and provide a keyword-based search language to perform data search. Similar to database query languages, these languages lack of the capability to express PA.

VIII. CONCLUSION

We propose PROGQL, a framework that supports customizable PA on system audit logs and improves the scalability of PA through incremental graph search. Specifically, the PROGQL framework provides a domain-specific language that includes novel language constructs for constrained graph search, edge weight computation, value propagation, and graph merge. To scale up the search over a colossal amount of system audit logs, the PROGQL framework imports the logs into a database backend formed by either relational databases or graph databases, and provides a query engine to achieve incremental graph search with the help of the database backend. Our evaluations show that our PROGQL language provides better expressiveness than SOTA graph query languages such as Cypher in expressing a diverse set of attack behaviors, and the comparison with the state-of-the-art PA demonstrates the PROGQL framework's significant scalability improvement (8 times saving of memory consumption without penalty on runtime performance).

REFERENCES

[1] Ebay, "Ebay Inc. to ask Ebay users to change passwords," 2014, http://blog.ebay.com/ebay-inc-ask-ebay-users-change-passwords/.

[2] "OPM government data breach impacted 21.5 million," 2015, http://www.cnn.com/2015/07/09/politics/office-of-personnel-management-data-breach-20-million.

[3] N. Y. Times, "Target data breach incident," 2014, http://www.nytimes.com/2014/02/27/business/target-reports-on-fourth-quarter-earnings.html?_r=1.

[4] "Home Depot Confirms Data Breach At U.S., Canadian Stores," 2014, http://www.npr.org/2014/09/09/347007380/home-depot-confirms-data-breach-at-u-s-canadian-stores.

[5] "Yahoo discloses hack of 1 billion accounts," 2016, https://techcrunch.com/2016/12/14/yahoo-discloses-hack-of-1-billion-accounts/.

[6] "The Equifax data breach," 2020, https://www.ftc.gov/equifax-data-breach.

[7] "The Marriott data breach," 2018, https://www.consumer.ftc.gov/blog/2018/12/marriott-data-breach.

[8] S. T. King and P. M. Chen, "Backtracking intrusions," in *ACM Symposium on Operating systems principles (SOSP)*. ACM, 2003, pp. 223–236.

[9] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen, "Enriching intrusion alerts through multi-host causality," in *Network and Distributed System Security Symposium (NDSS)*, 2005.

[10] X. Jiang, A. Walters, D. Xu, E. H. Spafford, F. Buchholz, and Y.-M. Wang, "Provenance-aware tracing of worm break-in and contaminations: A process coloring approach," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2006, pp. 38–38.

[11] S. Ma, J. Zhai, Y. Kwon, K. H. Lee, X. Zhang, G. F. Ciocarlie, A. Gehani, V. Yegneswaran, D. Xu, and S. Jha, "Kernel-supported cost-effective audit logging for causality tracking," in *USENIX Annual Technical Conference (ATC)*, 2018, pp. 241–254.

[12] Y. K., F. W., W. W., K. H. L., W. L., S. M., X. Z., D. X., S. J., G. F. C., A. G., and V. Y., "MCI : Modeling-based causality inference in audit logging for attack investigation," in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[13] Y. Liu, M. Zhang, D. Li, K. Jee, Z. Li, Z. Wu, J. Rhee, and P. Mittal, "Towards a timely causality analysis for enterprise security," in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[14] P. Gao, X. Xiao, Z. Li, F. Xu, S. R. Kulkarni, and P. Mittal, "AIQL: Enabling efficient attack investigation from system monitoring data," in *USENIX Annual Technical Conference (ATC)*, 2018, pp. 113–126.

[15] P. Gao, X. Xiao, D. Li, Z. Li, K. Jee, Z. Wu, C. H. Kim, S. R. Kulkarni, and P. Mittal, "SAQL: A stream-based query system for real-time abnormal system behavior detection," in *USENIX Security Symposium*, 2018, pp. 639–656.

[16] S. Ma, X. Zhang, and D. Xu, "Protracer: towards practical provenance tracing by alternating between logging and tainting," in *Network and Distributed System Security Symposium (NDSS)*, 2016.

[17] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "Nodoze: Combatting threat alert fatigue with automated provenance triage," in *Network and Distributed System Security Symposium (NDSS)*, 2019.

[18] T. Pietraszek, "Using adaptive alert classification to reduce false positives in intrusion detection," in *Recent Advances in Intrusion Detection (RAID)*, 2004, pp. 102–124.

[19] A. Kharraz, S. Arshad, C. Mulliner, W. K. Robertson, and E. Kirda, "UNVEIL: A large-scale, automated approach to detecting ransomware," in *USENIX Security Symposium*, 2016, pp. 757–772.

[20] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara, "The taser intrusion recovery system," in *ACM Symposium on Operating systems principles (SOSP)*, 2005, pp. 163–176.

[21] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Intrusion recovery using selective re-execution," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010, pp. 89–104.

[22] Z. Xu, Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang, "High fidelity data reduction for big data security dependency analyses," in *ACM Conference on Computer and Communications Security (CCS)*, 2016, pp. 504–516.

[23] Z. Xu, P. Fang, C. Liu, X. Xiao, Y. Wen, and D. Meng, "Depcomm: Graph summarization on system audit logs for attack investigation," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 540–557.

[24] P. Fang, P. Gao, C. Liu, E. Ayday, K. Jee, T. Wang, Y. F. Ye, Z. Liu, and X. Xiao, "DEPIMPACT: back-propagating system dependency impact for attack investigation," in *2022 USENIX Security Symposium, Boston, MA, USA, Aug, 2022.*, 2022.

[25] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "HOLMES: real-time APT detection through correlation of suspicious information flows," in *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2019, pp. 1137–1152.

[26] A. Taylor, A. Green, T. Lindaaker, S. Plantikow, M. Rydberg, P. Selmer, and M. Junghanns, "Cypher: An evolving query language for property graphs," in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 2018, pp. 1365–1370.

[27] W3C, "SPARQL Query Language for RDF," 2008, https://www.w3.org/TR/rdfsparqlquery/.

[28] C. Brodie, C.-M. Karat, J. Karat, and J. Feng, "Usable Security and Privacy: A Case Study of Developing Privacy Management Tools," in *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*, 2005.

[29] P. T. Wood, "Query languages for graph databases," *SIGMOD Record*, vol. 41, no. 1, pp. 50–60, 2012.

[30] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč, "Foundations of modern query languages for graph databases," *ACM Computing Surveys (CSUR)*, vol. 50, no. 5, pp. 1–40, 2017.

[31] P. Gao, X. Xiao, Z. Li, F. Xu, S. R. Kulkarni, and P. Mittal, "AIQL: enabling efficient attack investigation from system monitoring data," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.*, 2018, pp. 113–126.

[32] P. Gao, X. Xiao, D. Li, Z. Li, K. Jee, Z. Wu, C. H. Kim, S. R. Kulkarni, and P. Mittal, "SAQL: A stream-based query system for real-time abnormal system behavior detection," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, 2018, pp. 639–656.

[33] M. N. Hossain, S. Sheikhi, and R. Sekar, "Combating dependence explosion in forensic analysis using alternative tag propagation semantics," in *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2020.

[34] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. D. Stoller, and V. N. Venkatakrishnan, "SLEUTH: real-time attack scenario reconstruction from COTS audit data," in *USENIX Security Symposium*, 2017, pp. 487–504.

[35] Neo4j, "Neo4j: The World's Leading Graph Database," 2017, http://neo4j.com/.

[36] P. G. D. Group, "PostgreSQL," 2017, http://www.postgresql.org/.

[37] Myrocks, "MyRocks - facebook/mysql-5.6," 2015, https://github.com/facebook/mysql-5.6.

[38] NebulaGraph, "NebulaGraph," 2022, https://docs.nebula-graph.io/3.3.0/.

[39] P. Gao, X. Xiao, Z. Li, K. Jee, F. Xu, S. R. Kulkarni, and P. Mittal, "A query system for efficiently investigating complex attack behaviors for enterprise security," *Very Large Data Base Endowment (VLDB Endowment)*, vol. 12, no. 12, pp. 1802–1805, 2019.

[40] P. Gao, X. Xiao, D. Li, K. Jee, H. Chen, S. R. Kulkarni, and P. Mittal, "Querying streaming system monitoring data for enterprise system anomaly detection," in *IEEE International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1774–1777.

[41] Y. T. Tang, D. Li, Z. C. Li, M. Zhang, K. Jee, X. S. Xiao, Z. Y. Wu, J. Rhee, F. Y. Xu, and Q. Li, "Nodemerge: Template based efficient data reduction for big-data causality analysis," in *ACM Conference on Computer and Communications Security (CCS)*, 2018, pp. 1324–1337.

[42] K. H. Lee, X. Zhang, and D. Xu, "Loggc: garbage collecting audit log," in *ACM Conference on Computer and Communications Security (CCS)*, 2013, pp. 1005–1016.

[43] M. N. Hossain, J. Wang, R. Sekar, and S. D. Stoller, "Dependence-preserving data compaction for scalable forensic analysis," in *USENIX Security Symposium*, 2018, pp. 1723–1740.

[44] P. Fei, Z. Li, Z. Wang, X. Yu, D. Li, and K. Jee, "{SEAL}: Storage-efficient causality analysis on enterprise logs with query-friendly compression," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2987–3004.

[45] "Progql project website," 2025, https://github.com/ProGQL/ProGQL.

[46] S. Mika, G. Ratsch, J. Weston, B. Scholkopf, and K.-R. Muller, "Fisher discriminant analysis with kernels," in *IEEE Signal Processing Society Workshop*, 1999, pp. 41–48.

[47] "Progql backward bfs in pa," 2025, https://github.com/ProGQL/ProGQL/tree/main/appendix/backward%20bfs%20in%20PA.

[48] D. Arthur and S. Vassilvitskii, "K-means++: The advantages of careful seeding," in *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2007, pp. 1027–1035.

[49] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, "Dbscan revisited, revisited: why and how you should (still) use dbscan," *ACM Transactions on Database Systems (TODS)*, vol. 42, no. 3, pp. 1–21, 2017.

[50] M. Foundation, "Mariadb," 2023, https://mariadb.org/.

[51] E. Database, "Exploit Database," 2017, https://www.exploit-db.com/.

[52] Y. Kwon, F. Wang, W. Wang, K. H. Lee, W.-C. Lee, S. Ma, X. Zhang, D. Xu, S. Jha, G. F. Ciocarlie *et al.*, "Mci: Modeling-based causality inference in audit logging for attack investigation." in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[53] DARPA, "Transparent Computing, Defense Advanced Research Projects Agency," 2014, http://www.darpa.mil/program/transparent-computing.

[54] Sysdig, "Sysdig," 2017, https://sysdig.com/.

[55] "Cyber kill chain," 2021, https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html.

[56] MITRE, "Common Vulnerabilities and Exposures (CVE)," 2020, https://cve.mitre.org/.

[57] "Forward bfs in cypher," 2025, https://github.com/ProGQL/ProGQL/blob/main/appendix/forwardBFS_Cypher.png.

[58] J. Gui, X. Xiao, D. Li, C. H. Kim, and H. Chen, "Progressive processing of system behavioral query," in *Annual Computer Security Applications Conference (ACSAC)*, 2019, pp. 378–389.

[59] J. Gui, D. Li, Z. Chen, J. Rhee, X. Xiao, M. Zhang, K. Jee, Z. Li, and H. Chen, "APTrace: A responsive system for agile enterprise level causality analysis," in *IEEE International Conference on Data Engineering (ICDE), Industry and Application Track*, 2020, pp. 1701–1712.

[60] M. R. Ghazi and D. Gangodkar, "Hadoop, mapreduce and hdfs: a developers perspective," *Procedia Computer Science*, vol. 48, pp. 45–50, 2015.

[61] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[62] O. C. T. I. C. TC, "Cyber Observable eXpression (CybOX^{TM})," 2017, https://cyboxproject.github.io/.

[63] ——, "Trusted Automated eXchange of Indicator Information (TAXII^{TM})," 2017, https://taxiiproject.github.io/.

[64] ——, "Structured Threat Information eXpression (STIX^{TM})," 2017, http://stixproject.github.io/.

[65] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat, "Mace: Language support for building distributed systems," in *PLDI*, 2007, pp. 179–188.

[66] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica, "Declarative networking: Language, execution and optimization," in *SIGMOD*, 2006, pp. 97–108.

[67] K. Borders, J. Springer, and M. Burnside, "Chimera: A declarative language for streaming network traffic analysis," in *USENIX Security*, 2012.

[68] F. Cuppens and R. Ortalo, "LAMBDA: A language to model a database for detection of attacks," in *RAID*, 2000, pp. 197–216.

[69] R. Sommer, M. Vallentin, L. De Carli, and V. Paxson, "Hilti: An abstract execution environment for deep, stateful network traffic analysis," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ser. IMC '14. New York, NY, USA: ACM, 2014, pp. 461–474. [Online]. Available: http://doi.acm.org/10.1145/2663716.2663735

[70] M. Vallentin, V. Paxson, and R. Sommer, "Vast: A unified platform for interactive network forensics," in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 345–362. [Online]. Available: http://dl.acm.org/citation.cfm?id=2930611.2930634

[71] K. H. Lee, X. Zhang, and D. Xu, "High accuracy attack provenance via binary-based execution partition," in *Network and Distributed System Security Symposium (NDSS)*, 2013.

[72] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee, "Rain: Refinable attack investigation with on-demand inter-process information flow tracking," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017, pp. 377–390.

[73] Y. Ji, S. Lee, M. Fazzini, J. Allen, E. Downing, T. Kim, A. Orso, and W. Lee, "Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking," in *Proceedings of the USENIX Security Symposium*, 2018, pp. 1705–1722.

[74] ISO, "SQL: Structured Query Language," 2008, http://www.iso.org/iso/catalogue_detail.htm?csnumber=45498.

[75] "Database performance tuning guide," 2017, https://docs.oracle.com/cd/B19306_01/server.102/b14211/.

[76] A. AWS, "Amazon dynamodb," 2023, https://aws.amazon.com/dynamodb/.

[77] K. Chodorow, *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. " O'Reilly Media, Inc.", 2013.

[78] F. Geerts, S. Haesevoets, and B. Kuijpers, "First-order complete and computationally complete query languages for spatio-temporal databases," *ACM Transactions of Computational Logic*, vol. 9, no. 2, pp. 13:1–13:51, 2008.

[79] R. Snodgrass, "The temporal query language tquel," *TODS*, vol. 12, no. 2, pp. 247–298, 1987.

[80] R. T. Snodgrass, *Developing Time-oriented Database Applications in SQL*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000.

[81] A. Deutsch, N. Francis, A. Green, K. Hare, B. Li, L. Libkin, T. Lindaaker, V. Marsault, W. Martens, J. Michels *et al.*, "Graph pattern matching in gql and sql/pgq," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 2246–2258.

[82] "osquery," https://osquery.io/.

[83] C. Long, "osquery for security," 2016, https://medium.com/@clong/osquery-for-security-b66fffdf2daf.

[84] elastic, "Elasticsearch," 2017, https://www.elastic.co/.

[85] Splunk, "Splunk," 2017, http://www.splunk.com/.