AIQL: Enabling Efficient Attack Investigation from System Monitoring Data

Peng Gao, *Princeton University;* Xusheng Xiao, *Case Western Reserve University;*
Zhichun Li and Kangkook Jee, *NEC Laboratories America, Inc.;*
Fengyuan Xu, *National Key Lab for Novel Software Technology, Nanjing University;*
Sanjeev R. Kulkarni and Prateek Mittal, *Princeton University*

https://www.usenix.org/conference/atc18/presentation/gao

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

AIQL: Enabling Efficient Attack Investigation from System Monitoring Data

Peng Gao[1] Xusheng Xiao[2] Zhichun Li[3] Kangkook Jee[3] Fengyuan Xu[4]
Sanjeev R. Kulkarni[1] Prateek Mittal[1]
[1]*Princeton University* [2]*Case Western Reserve University* [3]*NEC Laboratories America, Inc.*
[4]*National Key Lab for Novel Software Technology, Nanjing University*
[1]{*pgao,kulkarni,pmittal*}*@princeton.edu* [2]*xusheng.xiao@case.edu* [3]{*zhichun,kjee*}*@nec-labs.com* [4]*fengyuan.xu@nju.edu.cn*

Abstract

The need for countering Advanced Persistent Threat (APT) attacks has led to the solutions that ubiquitously monitor system activities in each host, and perform timely attack investigation over the monitoring data for analyzing attack provenance. However, existing query systems based on relational databases and graph databases lack language constructs to express key properties of major attack behaviors, and often execute queries inefficiently since their semantics-agnostic design cannot exploit the properties of system monitoring data to speed up query execution.

To address this problem, we propose a novel query system built on top of existing monitoring tools and databases, which is designed with novel types of optimizations to support timely attack investigation. Our system provides (1) domain-specific *data model and storage* for scaling the storage, (2) a domain-specific query language, *Attack Investigation Query Language (*AIQL*)* that integrates critical primitives for attack investigation, and (3) an optimized query engine based on the characteristics of the data and the semantics of the queries to efficiently schedule the query execution. We deployed our system in NEC Labs America comprising 150 hosts and evaluated it using 857 GB of real system monitoring data (containing 2.5 billion events). Our evaluations on a real-world APT attack and a broad set of attack behaviors show that our system surpasses existing systems in both efficiency (124x over PostgreSQL, 157x over Neo4j, and 16x over Greenplum) and conciseness (SQL, Neo4j Cypher, and Splunk SPL contain at least 2.4x more constraints than AIQL).

1 Introduction

Advanced Persistent Threat (APT) attacks are sophisticated (involving many individual attack steps across many hosts and exploiting various vulnerabilities) and stealthy (each individual step is not suspicious enough), plaguing many well-protected businesses [9, 11, 15, 18, 27, 30]. A recent massive Equifax data breach [11] has exposed the sensitive personal information of 143 million US customers. In order for enterprises to counter advanced attacks, recent approaches based on *ubiquitous system monitoring* have emerged as an important solution for monitoring system activities and performing attack investigation [37,42,47–49,54,57,58]. System monitoring observes *system calls* at the kernel level to collect system-level events about system activities. Collection of system monitoring data enables security analysts to investigate these attacks by *querying risky system behaviors* over the historical data [71].

Although attack investigation is performed after the attacks compromise enterprises' security, it is a considerably time-sensitive task due to two major reasons. First, advanced attacks include a sequence of steps and are performed in multiple stages. A timely attack investigation can help understand all attack behaviors and prevent the further damage of the attacks. Second, understanding the attack sequence is crucial to correctly patch the systems. A timely attack investigation can pinpoint the vulnerable components of the systems and protect the enterprises from future attacks of the same types.

Challenges: However, there are two major challenges for building a query system to support security analysts in efficient and timely attack investigation.

Attack Behavior Specification: The system needs to provide a query language with specialized constructs for expressing various types of attack behaviors using system monitoring data: (1) **Multi-Step Attacks**: risky behaviors in advanced attacks typically involve activities that are related to each other based on either specific attributes (e.g., the same process reads a sensitive file and accesses the network) or temporal relationships (e.g., file read happens before network access), which requires language constructs to easily specify *relationships among activities*. In Fig. 1, the attacker runs `osql`

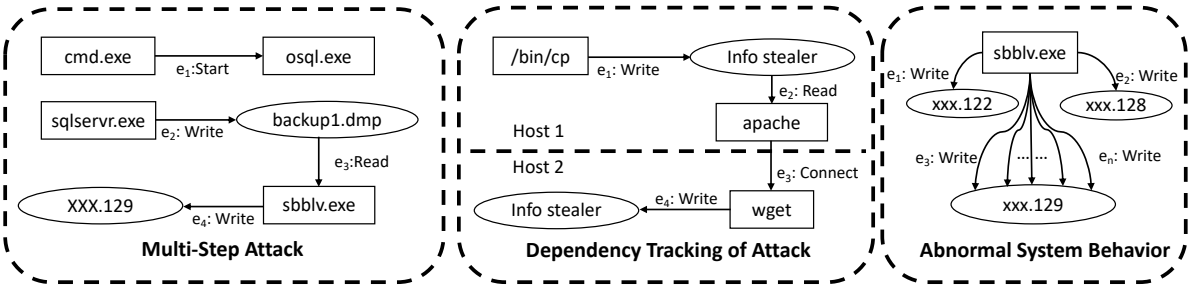


Figure 1: Major types of attack behaviors (events $e_1, \ldots, e_n$ are shown in ascending temporal order)

`.exe` to cause the database `sqlservr.exe` to dump its data into a file `backup1.dmp`. Later (i.e., `e3` happens after `e2`; temporal relationship), a malicious script `sbblv.exe` reads from the dump `backup1.dmp` (i.e., the same dump file in `e2` and `e3`; attribute relationship) and sends the data back to the attacker. (2) **Dependency Tracking of Attacks**: dependency analysis is often applied to track causality of data for discovering the "attack entry" (i.e., provenance) [48,49,61], which requires language constructs to *chain constraints among activities*. In Fig. 1, a malicious script `info_strealer` in Host 1 infects Host 2 via *network communications between* `apache` *and* `wget`. (3) **Abnormal System Behaviors**: frequency-based behavioral models are often required to express abnormal system behaviors, such as network access spikes [20, 29]. Investigating such spikes requires the system to support *sliding windows and statistical aggregation* of system activities, and compare the aggregate results with either *fixed thresholds (in absolute sense)* or the *historical results (in relative sense)*. In Fig. 1, a malicious script `sbblv.exe` sends a *large* amount of data to a particular destination `xxx.129`.[1]

Big-Data Security Analysis: System monitoring produces a huge amount of daily logs [55,69] ($\sim$ 50 GB per day for 100 hosts), and the investigation of these attacks typically requires enterprises to keep at least a 0.5 $\sim$ 1 year worth of data [32]. Such *a big amount of security data* poses challenges for the system to meet the requirements of *timely* attack investigation.

Limitations of Existing Systems: Unfortunately, existing query systems do not address both of these inherent challenges in attack investigation. First, existing query languages in relational databases based on SQL and SPARQL [19,22,25] lack constructs for easily chaining constraints among relations. Graph databases such as Neo4j [16] and NoSQL tools such as MongoDB [38], Splunk [23], and ElasticSearch [10] are ineffective in expressing event relationships where two events have no common entities (e.g., e_1 and e_2 in Fig. 1). More importantly, none of these languages provide language constructs to express behavioral models with historical results. Second, system monitoring data is generated with a timestamp on a specific host in the enterprise, exhibiting strong *spatial and temporal properties*. However, none of these systems provide optimizations that exploit the domain specific characteristics of the data, missing opportunities to optimize the system for supporting timely attack investigation and often causing queries to run for hours (e.g., performance evaluation results in Sec. 6.2.2).

Contributions: We design and build a novel system for efficient attack investigation from system monitoring data. We build our system ($\sim$ 50,000 lines of Java code) on top of existing system-level monitoring tools (i.e., auditd [28] and ETW [13]) for data collection and relational databases (i.e., PostgreSQL [19] and Greenplum [14]) for data storage and query. This enables our system to leverage the services provided by these mature infrastructures, such as data management, indexing mechanisms, recovery, and security. In particular, our system is designed with three novel types of optimizations. First, our system provides a domain-specific query language, *Attack Investigation Query Language (*AIQL*)*, which is optimized to express the three aforementioned types of attack behaviors. Second, our system provides domain-specific *data model and storage* for scaling the storage. Third, our system optimizes the query engine based on the characteristics of the monitoring data and the semantics of the queries to efficiently schedule the query execution. To the best of our knowledge, we are *the first to accelerate attack investigation via optimizing storage and query of system monitoring data*.

```
1 agentid = 1 // host id; spatial constraints
2 (at "01/01/2017") // temporal constraints
3 proc p1 start proc p2["%telnet%"] as evt1
4 proc p3 start ip ipp[dstport = 4444] as evt2
5 proc p4["%apache%"] read file f1["/var/www%"] as evt3
6 with p2 = p3,  // attribute relationship
7 evt1 before evt2, evt3 after evt2 // temporal
      relationships
8 return p1, p2, p4, f1
```

Query 1: AIQL Query for CVE-2010-2075 [5]

Domain-Specific Query Language (Sec. 4): Our AIQL language is designed for specifying the attack behaviors shown in Fig. 1 (i.e., Query 7 in Sec. 6.2.1, Query 3 in Sec. 4.2, and Query 5 in Sec. 6.2.1, respectively). Specifically, AIQL provides language constructs to specify re-

[1] While existing complex event processing systems [3, 12, 21] support similar features, they operate over stream rather than historical data stored in databases.

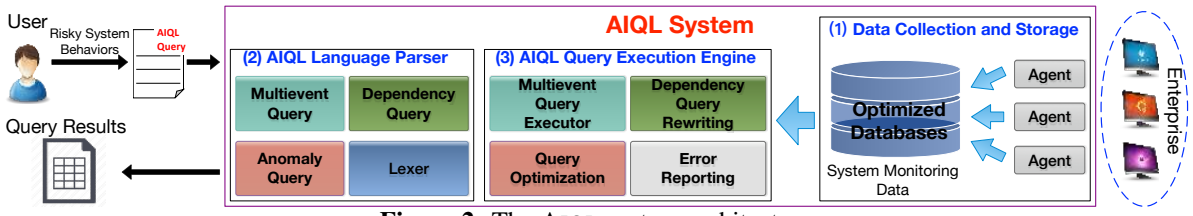


Figure 2: The AIQL system architecture

lationships among system activities (Sec. 4.1), chain constraints among activities (Sec. 4.2), and compute aggregate results in sliding time windows (Sec. 4.3). AIQL adopts the {*subject-operation-object*} syntax to represent system behavior patterns as events (e.g., `proc p1 write file f1`) and supports *attribute relationships* and *temporal relationships* of multiple events, as well as syntax shortcuts based on context-aware inference (Sec. 4.1). As shown in Query 1, AIQL can relate multiple system activities using spatial/temporal constraints and attribute/temporal relationships.

Data Model and Storage (Sec. 3.2): Our system models system monitoring data as a sequence of events, where each event describes how a process interacts with a system resource, such as writing to a file. More importantly, our system clearly identifies the spatial and temporal properties of the events, and leverages these properties to partition the data storage in both *spatial and temporal dimensions*. Such partitioning presents opportunities for parallel processing of query execution (Sec. 5.2).

Query Scheduling (Sec. 5): Our system identifies both *spatial* and *temporal* constraints in AIQL queries, and optimizes the query execution in two aspects: (1) for AIQL queries that involve multiple event patterns, our system prioritizes the search of event patterns with high pruning power, maximizing the reduction of irrelevant events as early as possible; (2) our system breaks down the query into independent sub-queries along temporal and spatial dimensions and executes them in parallel.

Evaluation: We deployed the AIQL system in NEC Labs America comprising 150 hosts. We performed a broad set of attack behaviors in the deployed environment, and evaluated the query performance and conciseness of AIQL against existing systems using 857 GB of real system monitoring data (16 days; 2.5 billion events): (1) our end-to-end efficiency evaluations on an APT attack case study (27 queries) show that AIQL surpasses both PostgreSQL (124x) and Neo4j (157x); (2) our performance evaluations show that the query scheduling employed by AIQL is efficient in both single-node databases (40x over PostgreSQL scheduling) and parallel databases (16x over Greenplum scheduling); (3) our conciseness evaluations on four major types of attack behaviors (19 queries) show that SQL, Neo4j Cypher, and Splunk SPL contain at least 2.4x more constraints, 3.1x more words, and 4.7x more characters than AIQL. All queries and a demo video are available on our *project website [1]*.

2 System Overview and Threat Model

Fig. 2 shows the AIQL system architecture: (1) we deploy monitoring agents across servers, desktops and laptops in the enterprise to monitor system activities by collecting information about system calls from kernels. The collected system monitoring data is then sent to the central server and stored in our optimized data storage (Sec. 3); (2) the language parser, implemented using ANTLR 4 [2], analyzes input queries and generates query contexts. A query context is an object abstraction of the input query that contains all the required information for the query execution. Multievent syntax, dependency syntax, and anomaly syntax are supported (Sec. 4); (3) the query execution engine executes the generated query contexts to search for the desired attack behaviors. Based on the data storage and the query semantics, domain-specific optimizations, such as relationship-based scheduling and temporal & spatial parallelization, are adopted to speedup the query execution (Sec. 5).

Threat Model: Our thread model follows the threat model of previous work [34, 48, 49, 54, 55]. We assume that kernel is trusted, and the system monitoring data collected from kernel is not tampered with [13, 28]. Any kernel-level attack that deliberately compromises security auditing systems is beyond the scope of this work.

3 Data Model and Storage

3.1 Data Model and Collection

System monitoring data records the interactions among system resources as system events [48]. Each of the recorded event occurs on a particular host at a particular time, thus exhibiting strong spatial and temporal properties. Existing works have indicated that on most modern operating systems (Windows, Linux and OS X), system resources (system entities) in most cases are files, processes, and network connections [42, 45, 48, 49]. Thus, in our data model, we consider system *entities* as *files*, *processes*, and *network connections*. We define a system event as the interaction among two system entities represented using the triple ⟨*subject, operation, object*⟩, which consists of the initiator of the interaction, the type

Table 1: Representative attributes of system entities

Entity	Attributes
File	Name, Owner/Group, VolID, DataID, etc.
Process	PID, Name, User, Cmd, Binary Signature, etc.
Network Connection	IP, Port, Protocol

of the interaction, and the target of the interaction. Subjects are processes originating from software applications such as Firefox, and objects can be files, processes and network connections. We categorize system events into three types according to their object entities, namely *file events*, *process events*, and *network events*.

Both entities and events have critical security-related attributes (Tables 1 and 2). The attributes of entities include the properties to support various security analyses (e.g., file name, process name, and IP addresses), and the unique identifiers to distinguish entities (e.g., file data ID and process ID). The attributes of events include event origins (i.e., agent ID and start time/end time), operations (e.g., file read/write), and other security-related properties (e.g., failure code). Agent ID refers to the unique ID of the host where the entity/event is observed.

Data Collection: We implement data collection agents for Windows and Linux based on ETW event tracing [13] and the Linux Audit Framework [28]. Tables 1 and 2 show representative attributes of our collected data.

3.2 Data Storage

After the modeling, we store the data in relational databases powered by PostgreSQL [19]. Relational databases come with mature indexing mechanisms and are scalable to massive data. However, even with indexes for speeding up queries, relational databases still face challenges in handling high ingest rates of massive system monitoring data. We next describe how we address these challenges to optimize the database storage.

Time and Space Partitioning: System monitoring data exhibits strong *temporal and spatial properties*: the data collected from different agents is independent from each other, and the timestamps of the collected data increase monotonically. Queries of the data are often specified with a specific time range or a host, or across many hosts within some time interval. Therefore, when storing the data, we partition the data in both the time and the space dimensions: separating groups of agents into table partitions and generating one database per day for the data collected on that day. We build various types of indexes on the attributes that will be queried frequently, such as executable name of process, name of file, source/destination IP of network connection.

Hypertable: For large organizations with hundreds or thousands of machines, we scale the data storage using MPP (massively parallel processing) databases Greenplum [14]. These databases intelligently distribute the storage and search of events and entities based on the spatial and temporal properties of our data model.

Table 2: Representative attributes of system events

Operation	Read/Write, Execute, Start/End, Rename/Delete
Time/Sequence	Start Time/End Time, Event Sequence
Misc.	Subject ID, Object ID, Failure Code

Time Synchronization: We correct potential time drifting of events on agents by applying synchronization protocols like Network Time Protocol (NTP) [17] at the client side, and checking with the clock at the server side.

4 Query Language Design

AIQL is designed to specify three types of attack behaviors: multi-step attacks, dependency tracking of attacks, and abnormal system behaviors. In contrast to previous query languages [7, 22, 23, 25] that focus on the specification of relation joins or graph paths, AIQL uniquely integrates the critical primitives for attack investigation, providing explicit constructs for spatial/temporal constraints, relationship specifications, constraint chaining among system events, and the access to aggregate and historical results in sliding time windows. Grammar 1 shows the representative rules of AIQL.

4.1 Multievent AIQL Query

For multievent queries, AIQL provides explicit language constructs for system events (in a natural format of {*subject-operation-object*}), spatial/temporal constraints, and event relationships.

A Running Example: Query 2 specifies an example system behavior that probes user command history files. Multiple context-aware syntax shortcuts (illustrated in comments) are used, such as attribute inference and omitting unreferenced entity IDs (details are given later).

```
1 agentid = 1 // unique id of the enterprise host
2 (at "01/01/2017") // time window
3 proc p2 start proc p1 as evt1
4 proc p3 read file[".viminfo" || ".bash_history"] as
      evt2 // .viminfo -> name=.viminfo; omit file ID
5 with p1 = p3, evt1 before evt2
6 return p2, p1 //p2 -> p2.exe_name, p1 -> p1.exe_name
7 sort by p2, p1
```

Query 2: Command history probing

Global Constraints: The global constraint rule (⟨*global_cstr*⟩) specifies the constraints for all event patterns (e.g., *agentid* and *time window* in Query 2).

Event Pattern: The event pattern rule (⟨*evt_patt*⟩) specifies an event pattern that consists of the subject/object entity (⟨*entity*⟩), operation (⟨*op_exp*⟩), and optional event ID (⟨*evt*⟩). The entity rule (⟨*entity*⟩) consists of entity type, optional entity ID, and optional attribute constraints (⟨*attr_cstr*⟩). Logical operators ("&&" for AND,

"||" for OR, "!" for NOT) can be used in ⟨*op_exp*⟩ and ⟨*attr_cstr*⟩ to form complex expressions. The optional time window rule (⟨*twind*⟩) further narrows down the search for the event pattern. Common time formats (US formats and ISO 8601) and granularities are supported.

```
⟨aiql⟩ ::= ⟨multievent⟩ | ⟨dependency⟩
⟨multievent⟩ ::= (⟨global_cstr⟩)* (⟨m_query⟩)+
⟨dependency⟩ ::= (⟨global_cstr⟩)* ⟨d_query⟩
⟨global_cstr⟩ ::= ⟨cstr⟩ | '(' ⟨twind⟩ ')' | ⟨slide_wind⟩
⟨twind⟩ ::= 'from' ⟨datetime⟩ 'to' ⟨datetime⟩ | ...
⟨slide_wind⟩ ::= ⟨wind_length⟩ ⟨wind_step⟩
Multi-event query:
⟨m_query⟩ ::= ⟨evt_patt⟩+ ⟨evt_rel⟩? ⟨return⟩ ⟨filter⟩?
⟨evt_patt⟩ ::= ⟨entity⟩ ⟨op_exp⟩ ⟨entity⟩ ⟨evt⟩? ('(' ⟨twind⟩ ')')?
⟨entity⟩ ::= ⟨entity_type⟩ ⟨e_id⟩ ? ('[' ⟨attr_cstr⟩']')?
⟨attr_cstr⟩ ::= ⟨cstr⟩
  | '!'⟨attr_cstr⟩
  | ⟨attr_cstr⟩ ('&&' | '||') ⟨attr_cstr⟩
  | '(' ⟨attr_cstr⟩ ')'
⟨cstr⟩ ::= ⟨attr⟩ ⟨bop⟩ ⟨val⟩
  | '!'? ⟨val⟩
  | ⟨attr⟩ 'not'? 'in' '(' ⟨val⟩ (',' ⟨val⟩)* ')'
⟨op_exp⟩ ::= ⟨op⟩
  | '!'⟨op_exp⟩
  | ⟨op_exp⟩ ('&&' | '||') ⟨op_exp⟩
  | '(' ⟨op_exp⟩ ')'
⟨evt⟩ ::= 'as' ⟨evt_id⟩ ('[' ⟨attr_cstr⟩']')?
⟨evt_rel⟩ ::= 'with' ⟨rel⟩ (',' ⟨rel⟩)*
⟨rel⟩ ::= ⟨attr_rel⟩ | ⟨temp_rel⟩
⟨attr_rel⟩ ::= ⟨e_id⟩'.'⟨attr⟩ ⟨bop⟩ ⟨e_id⟩'.'⟨attr⟩
  | ⟨e_id⟩ ⟨bop⟩ ⟨e_id⟩
⟨temp_rel⟩ ::= ⟨evt_id⟩ ('before' | 'after' | 'within') ('[' ⟨val⟩'-'⟨val⟩ ⟨timeunit⟩']')? ⟨evt_id⟩
⟨return⟩ ::= 'return' 'count'? 'distinct'? ⟨res⟩ (',' ⟨res⟩)*
⟨res⟩ ::= ⟨e_id⟩('.'⟨attr⟩)?
  | ⟨agg_func⟩'(' ⟨res⟩ ')'
  | 'as' ⟨rename_id⟩
⟨group_by⟩ ::= 'group by' ⟨res⟩ (',' ⟨res⟩)*
⟨filter⟩ ::= 'having' ⟨expr⟩
  | 'sort by' ⟨attr⟩ (',' ⟨attr⟩)* ('asc' | 'desc')?
  | 'top' ⟨int⟩
Dependency query:
⟨d_query⟩ ::= (('forward' | 'backward') ':')? (⟨entity⟩ ⟨op_edge⟩)+ ⟨entity⟩ ⟨return⟩ ⟨filter⟩?
⟨op_edge⟩ ::= ('->' | '<-') '[' ⟨op_exp⟩ ']'
```

Grammar 1: Representative BNF grammar of AIQL

Event Attribute and Temporal Relationships: The event relationship rule (⟨*evt_rel*⟩) specifies how multiple event patterns are related. The attribute relationship rule (⟨*attr_rel*⟩) uses attribute values of event patterns to specify their relationships. In Query 2, `p1=p3` (inferred as `p1.id=p3.id`) indicates that two event patterns `evt1` and `evt2` are linked by the same entity. The temporal relationship rule (⟨*temporal_rel*⟩) specifies temporal order ("before", "after", "within") of event patterns. For example, `evt1 before[1-2 minutes] evt2` specifies that `evt1` occurred 1 to 2 minutes before `evt2`.

Event Return and Filters: The event return rule (⟨*return*⟩) retrieves the attributes of the matched events. Constructs such as "count", "distinct", "top", "having", and "sort by" are provided for result manipulation and filtering.

Context-Aware Syntax Shortcuts: AIQL includes language syntax shortcuts to make queries more concise.

- *Attribute inference*: (1) default attribute names will be inferred if users specify only attribute values in an event pattern, or specify only entity IDs in the return clause. We select the most commonly used attributes in security analysis as the default attributes: `name` for files, `exe_name` for processes, and `dst_ip` for networks; (2) `id` will be used as the default attribute if users specify only entity IDs in attribute relationships.
- *Optional ID*: the ID of entity/event can be omitted if it is not referenced in the event relationship clause or the event return clause.
- *Entity ID reuse*: reusing entity IDs in multiple event patterns implicitly means that these event patterns share the same entity.

For example, in Query 2, `".viminfo"`, `return p2`, and `p1 = p3` will be inferred as `name = ".viminfo"`, `return p2.exe_name`, and `p1.id = p3.id`, respectively. Query 2 also omits the file ID in *evt2* since it is not referenced. We can also replace *p3* with *p1* in *evt2* and omit `p1 = p3`.

4.2 Dependency AIQL Query

AIQL provides the dependency syntax that chains constraints and specifies temporal relationships among event patterns, facilitating the specification of dependency tracking of attacks. The syntax specifies a sequence of event patterns in the form of a path, where nodes in the path represent system entities and edges represent operations. The `forward` and `backward` keywords can be used to specify the temporal order of the events on the path: `forward` (`backward`) means the events found by the left-most event pattern occurred earliest (latest).

```
1 (at "01/01/2017")
2 forward: proc p1["%/bin/cp%", agentid = 2] ->[write]
      file f1["/var/www/%info_stealer%"]
3 <-[read] proc p2["%apache%"]
4 ->[connect] proc p3[agentid=3] // tracking across
      host
5 ->[write] file f2["%info_stealer%"]
6 return f1, p1, p2, p3, f2
```

Query 3: Forward tracking for malware ramification

Query 3 shows a forward dependency query in AIQL that investigates the ramification of malware (`info_stealer`), which originates from host h_a (`agentid = 2`) and affects host h_b (`agentid = 3`) through an Apache web server. Lines 2-3 specify that `p1` writes to `f1`, and then `f1` is read by `p2`. Such syntax eliminates the need to repetitively specify the shared entity (i.e., `f1`) in each

event pattern. An example result may show that `p3` is the `wget` process that downloads the malicious script from host h_b. The operation `->[connect]` at Line 4 indicates the search will track dependencies of events across hosts.

4.3 Anomaly AIQL Query

AIQL provides the constructs of *sliding time window* with common aggregation functions (e.g., `count`, `avg`, `sum`) to facilitate the specification of frequency-based system behavioral models. Besides, AIQL provides the construct of *history states*, allowing queries to compare frequencies using historical information.

```
1 (at "01/01/2017")
2 window = 1 min
3 step = 10 sec
4 proc p read ip ipp
5 return p, count(distinct ipp) as freq
6 group by p
7 having freq > 2 * (freq + freq[1] + freq[2]) / 3
```

Query 4: Simple moving average for network frequency

Query 4 shows an anomaly query that specifies a 1-minute sliding time window and computes the moving average [44] to detect network spikes (Line 7). AIQL supports the common types of moving averages through built-in functions (SMA, CMA, WMA, EWMA [44]). For example, the computation of EWMA for network frequency with normalized deviation can be expressed as: `(freq - EWMA(freq, 0.9)) / EWMA(freq, 0.9) > 0.2`.

5 Query Execution Engine

The AIQL query execution engine executes the query context generated by the parser and optimizes the query execution by leveraging domain-specific properties of system monitoring data. Optimizing a query with many constraints is a difficult task due to the complexities of joins and constraints [8]. AIQL addresses these challenges by providing explicit language constructs for spatial/temporal constraints and temporal relationships, so that the query engine can directly optimize the query execution by: (1) using event patterns as a basic unit for generating data queries and leveraging attribute/temporal relationships to optimize the search strategy; (2) leveraging the spatial and temporal properties of system monitoring data to partition the data and executing the search in parallel based on the spatial/temporal constraints.

5.1 Query Execution Pipeline

Fig. 3 shows the execution pipeline of a multievent query. Based on the query semantics, for every event pattern, the engine synthesizes a *SQL data query*, which searches the optimized relational databases (Sec. 3.2) for

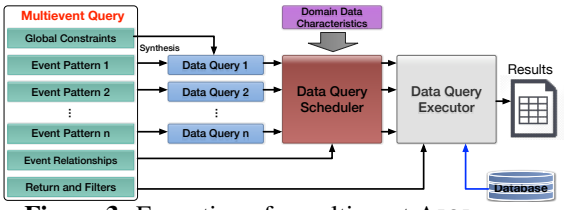


Figure 3: Execution of a multievent AIQL query

the matched events. The data query scheduler prioritizes the execution of data queries to optimize execution performance (Sec. 5.2). Execution results of each data query are further processed by the executor to perform joins and filtering to obtain the desired results. Note that by weaving all these join and filtering constraints together, the engine could generate a large SQL with many constraints mixed together. Such strategy suffers from indeterministic optimizations due to the large number of constraints and often causes the execution to last for minutes or even hours (Sec 6.2.2). For an input dependency query, the engine compiles it to an equivalent multievent query for execution. For an anomaly query, the engine maintains the aggregate results as historical states and performs the filtering based on the historical states.

5.2 Data Query Scheduler

The data query scheduler in Fig. 3 schedules the execution of data queries. A straightforward scheduling strategy (*fetch-and-filter*) is to: (1) execute data queries separately and store the results of each query in memory; (2) leverage event relationships to filter out results that do not satisfy the constraints. However, this strategy incurs non-trivial computation costs and memory space if some data queries return a large number of results.

Relationship-Based Scheduling: To optimize the execution scheduling of data queries, we leverage two insights based on event relationships: (1) event patterns have different levels of pruning power, and the query engine can prioritize event patterns with more pruning power to narrow the search; (2) if two event patterns are associated with an event relationship, the query engine can execute the data query for the pattern that has more constraints first (likely having more pruning power), and use the execution results to constrain the execution of the other data query.

Algorithm 1 gives the *relationship-based* scheduling:

1. A pruning score is computed for every event pattern based on the number of constraints specified.
2. Event relationships are sorted based on the relationship type (process events and network events are sorted in front of file events) and the sum of the involved event patterns' pruning scores.
3. The main loop processes event relationships returned from the sorted list, executes data queries, and gener-

ates result tuples. The engine executes the data query whose associated event pattern has a higher pruning score first, and leverages existing results to narrow the search scope. To facilitate tuple management, we maintain a map M that stores the mapping from the event pattern ID to the set of event ID tuples that its execution results belong to. As the loop continues, new tuple sets are created and put into M, and old tuple sets are updated, filtered, or merged.

4. After analyzing all event relationships, if there remain unexecuted data queries, these queries are executed and the corresponding results are put into M.
5. The last step is to merge tuple sets in M, so that all event patterns are mapped to the same tuple set that satisfy all constraints.

Algorithm 1: Relationship-based scheduling

Input: n data queries: $Q = \{q_i \mid i \leq n, i \in \mathbb{N}^+\}$
n event patterns: $E = \{e_i \mid i \leq n, i \in \mathbb{N}^+\}$
m event relationships: $R = \{rel(e_i, e_j)\}$
Output: Event ID tuples that satisfy all constraints

1. $\forall e_i \in E, score(e_i) \xleftarrow{compute} e_i$;
2. $R_{sorted} \xleftarrow{sort} R$;
3. Initialize empty set $Exec$, empty map M;

for $rel(e_i, e_j)$ *in* R_{sorted} **do**
  if e_i *not in Exec* ***and*** e_j *not in Exec* **then**
    // Suppose $score(e_i) \geq score(e_j)$
    $S_i \xleftarrow{execute} q_i$; $Exec.add(e_i)$; // S_i:event ID set
    $S_j \xleftarrow{execute}_{S_i} q_j$; $Exec.add(e_j)$;
    $T \leftarrow S_i \times S_j \mid_{rel(e_i,e_j)}$; // **create** tuple set from S_i and S_j, then filter by $rel(e_i,e_j)$
    $M.put(e_i, T)$; $M.put(e_j, T)$;
  else if *Either of* $\{e_i, e_j\}$ *in Exec* **then**
    // Suppose e_i in $Exec$
    $S_j \xleftarrow{execute}_{S_i} q_j$; $Exec.add(e_j)$;
    $T \leftarrow M.get(e_i)$; $T' \leftarrow T \times S_j \mid_{rel(e_i,e_j)}$; // **update** tuple set using S_j and $rel(e_i,e_j)$
    $replaceVals(M, T, T')$; $M.put(e_j, T')$;
  else
    $T_i \leftarrow M.get(e_i)$; $T_j \leftarrow M.get(e_j)$;
    if $T_i = T_j$ **then**
      $T' \leftarrow T_i \mid_{rel(e_i,e_j)}$; // **filter** tuple set
      $replaceVals(M, T_i, T')$;
    else
      $T' \leftarrow T_i \times T_j \mid_{rel(e_i,e_j)}$; // **merge** tuple sets
      $replaceVals(M, T_i, T')$; $replaceVals(M, T_j, T')$;

4. **for** $e_i \in E$ ***and*** e_i *not in Exec* **do**
  $S_i \xleftarrow{execute} q_i$; $Exec.add(e_i)$; $M.put(e_i, S_i)$;
5. **while** $unique(M.values()) > 1$ **do**
  Pick T_i, T_j from $M.values()$, such that $T_i \neq T_j$;
  $T' \leftarrow T_i \times T_j$; // **merge** tuple sets
  $replaceVals(M, T_i, T')$; $replaceVals(M, T_j, T')$;
6. Return $unique(M.values())$;

Function *replaceVals (M, T, T')*
  Replace all values T stored in M with T';

Our empirical results (Sec. 6.3.2 and 6.3.3) demonstrate that the number of constraints work well in approximating the pruning power of event patterns in a broad set of queries, even though they may not accurately represent the size of the results returned by event patterns.

Time Window Partition: The AIQL query engine leverages temporal properties of the data to further speed up the execution of synthesized data queries: the engine partitions the time window of a data query into sub-queries with smaller time windows, and executes them in parallel. Currently, our system splits the time window into days for a query over a multi-day time window.

6 Deployment and Evaluation

We deployed the AIQL system in NEC Labs America comprising 150 hosts (10 servers, 140 employee stations). We performed a series of attacks based on known exploits in the deployed environment and constructed 46 AIQL queries to investigate these attacks, demonstrating the expressiveness of AIQL. To evaluate the effectiveness of AIQL in supporting timely attack investigation, we evaluate the query *efficiency* and *conciseness* against existing systems: PostgreSQL [19], Neo4j [16], Splunk [23]. We also evaluate the efficiency offered by our data query scheduler (Sec. 5.2) in both storage settings: PostgreSQL and Greenplum. In total, our evaluations use 857GB of real system monitoring data (16 days; 2.5 billion events).

6.1 Evaluation Setup

The evaluations are conducted on a database server with an Intel(R) Xeon(R) CPU E5-2660 (2.20GHz), 64GB RAM, and a RAID that supports four concurrent reads/writes. Neo4j databases are configured by importing system entities as nodes and system events as relationships. Greenplum databases are configured to have 5 segment nodes that can effectively leverage the concurrent reads/writes of RAID. For each AIQL query (except anomaly queries), we construct semantically equivalent SQL, Cypher, and Splunk SPL queries. We measure the execution time and the conciseness of each query. Note that we omit the performance evaluation of Splunk since the community version is limited to 500MB per day and the enterprise version is prohibitively expensive ($1,900 per GB). Nevertheless, Splunk's limited support for joins [24] makes it inappropriate for investigating multi-step attack behaviors. Due to the limited expressiveness of SQL and Cypher, we cannot compare the anomaly queries (e.g., Query 5). All queries are available on our *project website [1]*.

6.2 Case Study: APT Attack Investigation

We conduct a case study by asking white hat hackers to perform an APT attack in the deployed environment, as

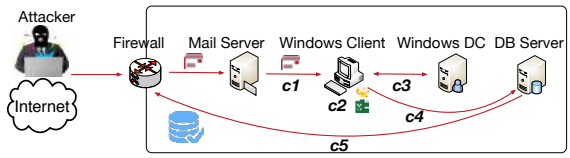


Figure 4: Environmental setup for the APT attack

shown in Fig. 4. Below are the attack steps:

c1 Initial Compromise: The attacker sends a crafted email to the victim. The email contains an Excel file with a malicious macro embedded.

c2 Malware Infection: The victim opens the Excel file through the Outlook mail client and runs the macro, which downloads and executes a malware (CVE-2008-0081 [4]) to open the backdoor to the attacker.

c3 Privilege Escalation: The attacker enters the victim's machine through the backdoor, scans the network ports to discover the IP address of the database, and runs the database cracking tool (`gsecdump.exe`) to obtain the credentials of the user database.

c4 Penetration into Database Server: Using the credentials, the attacker penetrates into the database server and delivers a VBScript to drop another malware, which creates another backdoor to the attacker.

c5 Data Exfiltration: With the access to the database server, the attacker dumps the database content using `osql.exe` and sends the data dump back.

Anomaly Detectors: We deployed two anomaly detectors based on existing solutions [36, 52, 66]. The first detector is deployed on the database server, which monitors network data transfer and emits alerts when the transfer amount is abnormally large. The second detector is deployed on the Windows client, which monitors process creation and emits alerts when a process starts an unexpected child process. These detectors may produce false positives, and we need tools like AIQL to investigate the alerts before taking any further action.

6.2.1 Attack Investigation Procedure

Our investigation assumes no prior knowledge of the detailed attack steps but merely the detector alerts. We start with these alerts and iteratively compose AIQL queries to investigate the entire attack sequence.

Step c5: We first examine the alerts reported by the database server detector, and identify a suspicious external IP "XXX.129" (obfuscated for privacy). Existing network traffic detectors usually cannot capture the precise process information [50, 64]. Thus, we first compose an anomaly AIQL query that computes moving average (SMA3) to find processes which transfer a large amount of data to this suspicious IP.

```
(at "mm/dd/2017") // date (obfuscated)
agentid = xxx // SQL database server (obfuscated)
window = 1 min, step = 10 sec
proc p write ip i[dstip="XXX.129"] as evt
return p, avg(evt.amount) as amt
group by p
having (amt > 2 * (amt + amt[1] + amt[2]) / 3)
```

Query 5: AIQL anomaly query for large file transfer

Query 5 finishes execution within 4 seconds and identifies a suspicious process "sbblv.exe". We then compose a multievent AIQL query to find the data sources for this process (Query 6).

```
(at "mm/dd/2017")
agentid = xx // SQL database server (obfuscated)
proc p1["%sbblv.exe"] read || write file f1 as evt1
proc p1 read || write ip i1[dstip="XXX.129"] as evt2
with evt1 before evt2
return distinct p1, f1, i1, evt1.optype, evt1.access
```

Query 6: Starter AIQL query for *c5*

We identify a suspicious file "BACKUP1.DMP" for `f1` out of the other normal DLL files. We investigate its creation process and find "sqlservr.exe", which is a standard SQL server process with verified signature. Thus, we speculate that the attacker penetrates into the SQL server, dumps the data ("BACKUP1.DMP"), and sends the data back to his host ("XXX.129"). We verify this by checking that "osql.exe" process is started by "cmd.exe" (OSQL utility is often involved in many SQL database attacks). Query 7 gives the complete query for investigating the step *c5*.

```
(at "mm/dd/2017")
agentid = xxx // SQL database server (obfuscated)
proc p1["%cmd.exe"] start proc p2["%osql.exe"] as
    evt1
proc p3["%sqlservr.exe"] write file f1["%backup1.dmp"
    ] as evt2
proc p4["%sbblv.exe"] read file f1 as evt3
proc p4 read || write ip i1[dstip="XXX.129"] as evt4
with evt1 before evt2, evt2 before evt3, evt3 before
    evt4
return distinct p1, p2, p3, f1, p4, i1
```

Query 7: Complete AIQL query for *c5*

Steps c4-c1: The investigation for *c4-c1* is similar to *c5*, including iterative query execution and editing. In total, we constructed 26 multievent queries and 1 anomaly query to successfully investigate the APT attack, touching 119GB of data/422 million events.

6.2.2 Evaluation Results

As we can see, attack investigation is an iterative process that revises queries: (1) latter iterations add more event patterns based on the selected results from the former queries, and (2) 4-5 iterations are needed before finding a complete query with 5-7 event patterns. Thus, *slow response* and *verbose specification* could greatly impede the effectiveness and efficiency of the investigation.

End-to-End Execution Efficiency: Fig. 5 shows the execution time of AIQL queries, SQL queries in PostgreSQL, and Cypher queries in Neo4j. For evaluation

Table 3: Aggregate statistics for case study

Attack Step	# of Queries	# of Evt Patterns	AIQL (s)	PostgreSQL (s)	Neo4j (s)
c1	1	3	3.8	3.1	10.8
c2	8	27	31.0	8038.7	10981.7
c3	2	4	15.9	15.3	3615.6
c4	8	35	61.0	10906.7	8150.6
c5	7	18	58.8	2166.5	4285.4
All	26	87	170.5	21130.3	27044.1

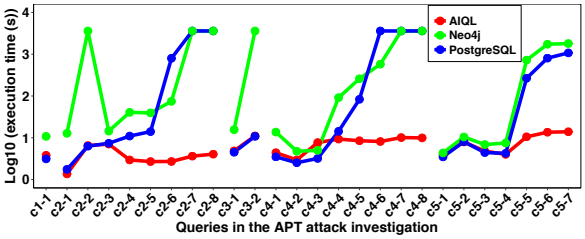


Figure 5: Log10-transformed query execution time

fairness, PostgreSQL and Neo4j databases store the same copies of data and employ the same schema and index designs as AIQL, but they do not employ our domain-specific data storage optimizations such as spatial and temporal partitioning, nor our scheduling optimizations.[2] Table 3 shows aggregate statistics for investigating each attack step, including the number of queries, the number of event patterns, and the total investigation time (second). We observe that: (1) Neo4j generally runs slower than PostgreSQL, due to the lack of support for efficient joins; (2) PostgreSQL and Neo4j become very slow when the query becomes complex and the number of event patterns (hence the required table joins) becomes large. Many large queries in PostgreSQL and Neo4j cannot finish within 1 hour (e.g., c2-7, c2-8, c4-7, c4-8); (3) all AIQL queries finish within 15 seconds, and the performance of the queries grows linearly with the number of event patterns (rather than the exponential growth in PostgreSQL and Neo4j), demonstrating the effectiveness of our domain-specific storage optimizations and query scheduling. (4) the total investigation time is ∼5.9 hours for PostgreSQL and ∼7.5 hours for Neo4j, which is a significant bottleneck for a timely attack investigation. In contrast, the total investigation time for AIQL is within 3 minutes (124x speedup over PostgreSQL and 157x speedup over Neo4j).

Conciseness: The largest AIQL query is c4-8 with 7 event patterns, 25 query constraints, 109 words, and 463 characters (excluding spaces). The corresponding SQL query contains 77 constraints (3.1x larger), 432 words (4.0x larger), and 2792 characters (6.0x larger). The corresponding Cypher query contains 63 constraints (2.5x larger), 361 words (3.3x larger), and 2570 characters (5.6x larger). As the attack behaviors become more complex, SQL and Cypher queries become verbose with many joins and constraints, posing challenges for constructing the queries for timely attack investigation.

[2]Fine-grained evaluations of the AIQL scheduling are in Sec. 6.3.

Table 4: Selected malware samples from Virussign

ID	Name	Category
v1	7dd95111e9e100b6243ca96b9b322120	Trojan.Sysbot
v2	425327783e88bb6492753849bc43b7a0	Trojan.Hooker
v3	ee111901739531d6963ab1ee3ecaf280	Virus.Autorun
v4	4e720458c357310da684018f4a254dd0	Virus.Sysbot
v5	7dd95111e9e100b6243ca96b9b322120	Trojan.Hooker

6.3 Performance Evaluation

We evaluate the performance of AIQL in both storage settings (PostgreSQL and Greenplum) by constructing 19 AIQL queries for a broad set of attack behaviors, touching 738GB/2.1 billion events. Particularly, we are interested in the efficiency speedup provided by the AIQL scheduling (Sec. 5.2) in comparison with PostgreSQL scheduling and Greenplum scheduling.

6.3.1 Attack Behaviors

Multi-Step Attack Behaviors: We asked white hat hackers to launch another APT attack using different exploits (details available on [1]). We then constructed 5 AIQL queries for investigating the attack steps (*a1-a5*).

Dependency Tracking Behaviors: We performed causal dependency tracking of origins of Chrome update executables (*d1*) and Java update executables (*d2*). We performed forward dependency tracking of the ramification malware `info_stealer` (*d3*).

Real-World Malware Behaviors: We obtained a dataset of free malware samples from VirusSign [33]. We then randomly selected 5 malware samples (Table 4) from the 3 largest categories: *Autorun*, *Sysbot*, and *Hooker*. We executed the 5 selected samples in the deployed environment and constructed AIQL queries by analyzing the accompanied behavior reports [33] (*v1-v5*).

Abnormal System Behaviors: We evaluated 6 abnormal system behaviors based on security experts' knowledge: (1) *s1*: command history probing; (2) *s2*: suspicious web service; (3) *s3*: frequent network access; (4) *s4*: erasing traces from system files; (5) *s5*: network access spike; (6) *s6*: abnormal file access. Note that for *s5* and *s6*, we did not construct SQL, Cypher, or Splunk queries, due to their lack of support for sliding window and history state comparison.

6.3.2 Efficiency in PostgreSQL

We select two baselines: (1) PostgreSQL databases that *employ our data storage optimizations (Sec. 3.2)*. Note that this setting is different from the end-to-end efficiency evaluation in Sec. 6.2.2, because here we want to rule out the speedup offered by the data storage component; (2) AIQL with fetch-and-filter scheduling (denoted as AIQL_FF; Sec. 5.2). We measure the execution time of the 19 queries in Sec. 6.3.1.

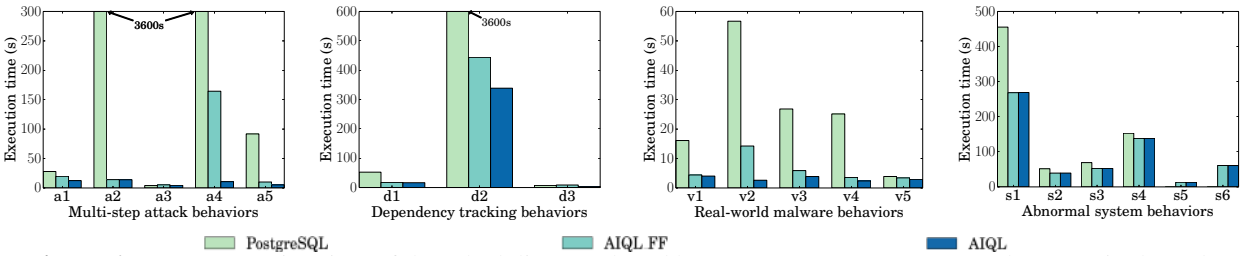


Figure 6: Query execution time of the scheduling employed by PostgreSQL, AIQL_FF, and AIQL (single-node)

Table 5: Conciseness improvement statistics

Metrics	AIQL/SQL	AIQL/ Cypher	AIQL/Splunk SPL
# of constraints	3.0x	2.4x	4.2x
# of words	3.9x	3.1x	3.8x
# of characters	5.3x	4.7x	4.7x

Evaluation Results: Fig. 6 shows the execution time of queries in PostgreSQL, AIQL_FF, and AIQL. We observe that: (1) the scheduling employed by PostgreSQL is inefficient in executing complex queries. In particular, PostgreSQL cannot finish executing *a2*, *a4*, and *d2* within 1 hour; (2) the scheduling employed by AIQL_FF and AIQL is more efficient than PostgreSQL, with 19x and 40x speedup, respectively; (3) the relationship-based scheduling employed by AIQL is more efficient than the fetch-and-filter scheduling employed by AIQL_FF.

6.3.3 Efficiency in Parallel Databases

We compare the performance of AIQL scheduling in the Greenplum storage with the Greenplum scheduling (i.e., running SQLs). As in Sec. 6.3.2, the Greenplum databases also *employ our data storage optimizations*.

Evaluation Results: Fig. 7 shows the execution time of queries in Greenplum and AIQL. We observe that: (1) in most cases, our scheduling in parallel settings achieves a comparable performance as Greenplum scheduling; (2) in certain cases (e.g., *a4*, *d3*), our scheduling is significantly more efficient than Greenplum scheduling; (3) the average speedup over Greenplum is 16x. The results show that without our semantics-aware model, Greenplum distributes the storage of events based on their incoming orders (which is arbitrary). On the contrary, our data model allows Greenplum to evenly distribute events in a host, and achieves more efficient parallel search.

6.4 Conciseness Evaluation

We evaluate the conciseness of queries that express the 19 attack behaviors in Sec. 6.3.1 in three metrics: the number of query constraints, the number of words, and the number of characters (excluding spaces).

Evaluation Results: Fig. 8 shows the conciseness metrics of AIQL, SQL, Neo4j Cypher, and Splunk SPL queries. Table 5 shows the average improvement of AIQL queries over other queries. We observe that AIQL *is the most concise query language* in terms of all three metrics and all attack behaviors: SQL, Neo4j Cypher, and Splunk SPL contain at least 2.4x more constraints, 3.1x more words, and 4.7x more characters than AIQL. In contrast to SQL, Cypher, and SPL which employ lots of joins on tables or nodes, AIQL provides high-level constructs for spatial/temporal constraints, relationship specifications, constraints chaining, and context-aware syntax shortcuts, making the queries much more concise.

7 Discussion

Query Scheduler: Our data query scheduler estimates the pruning score of an event pattern based on its number of constraints. This can be improved by (1) considering the number of records in different hosts and different time periods and (2) constructing a statistical model of constraint pruning power. Additionally, the query scheduler may partition the time window uniformly based on the data volume. Such strategies require further analysis of the domain data statistics to infer the proper data volume for splitting, which we leave for future work.

System Entities and Data Reduction: In the future work, we plan to add registry entries in Windows and pipes in Linux to expand the monitoring scope. We also plan to incorporate more finer granularity system monitoring, such as execution partition [58, 59] and in-memory data manipulations [40, 43]. To handle the increase of data size, we plan to explore more aggressive data reduction techniques in addition to existing solutions [55, 69] to make the system more scalable.

8 Related Work

Security-Related Languages: There also exist domain-specific languages in a variety of security fields that have a well-established corpus of low level algorithms, such as threat descriptions [6, 26, 31], secure overlay networks [46, 56], and network intrusions [35, 39, 65, 68]. These languages provide specialized constructs for their particular problem domain. In contrast to these languages, the novelty of AIQL focuses on querying attack behaviors, including (a) providing specialized constructs

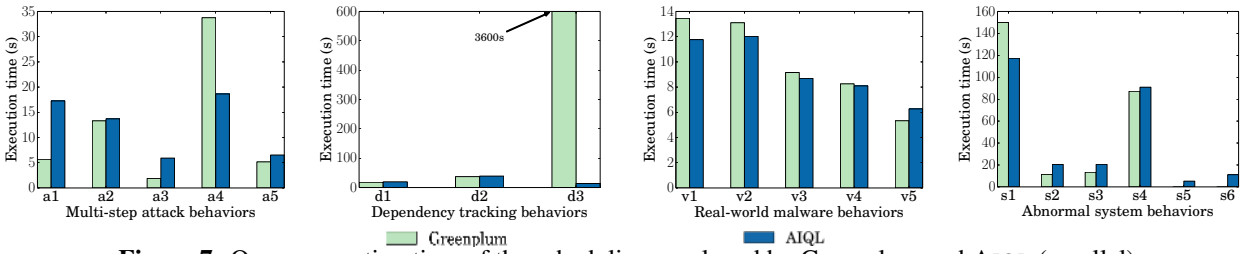
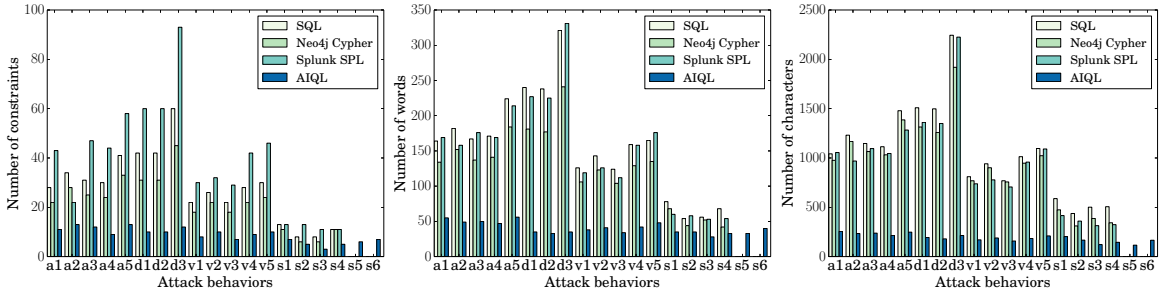


Figure 7: Query execution time of the scheduling employed by Greenplum and AIQL (parallel)



(a) Number of constraints **(b)** Number of words **(c)** Number of characters

Figure 8: Conciseness evaluation of queries written in AIQL, SQL, Neo4j Cypher, and Splunk SPL

for system interaction patterns/relationships and abnormal behaviors; (b) optimizing query execution over system monitoring data. Splunk [23] and Elasticsearch [10] are distributed search and analytics engine for application logs, which provide search languages based on keywords and shell-like piping. However, these systems lack efficient supports for joins and their languages cannot express abnormal behaviors with history states as AIQL. Furthermore, our AIQL can be used to investigate the real-time anomalies detected on the stream of system monitoring data, complementing the stream-based anomaly detection systems [41] for better defense.

Database Query Languages: Relational databases based on SQL [19, 25] and SPARQL [22] provide language constructs for joins, facilitating the specification of relationships among events, but these languages lack constructs for easily chaining constraints among relations (i.e., tables). Graph databases [16] provide language constructs for chaining constraints among nodes in graphs, but these databases lack efficient support for joins. Similarly, NoSQL tools [38] lack efficient supports for joins. Temporal expressions are also introduced to databases [62], and various time-oriented applications are explored [63]. Currently, AIQL focuses on the set of temporal expressions that are frequently used in expressing attack behaviors, which is a subset of the temporal expressions proposed in [62]. More importantly, none of these languages provide constructs to express frequency-based behavioral models with historical results.

System Defense Based on Behavioral Analytics: Existing malware detection has looked at various ways to build behavioral models to capture malware, such as sequences of system calls [67], system call patterns based on data flow dependencies [51], and interactions between benign programs and the operating system [53]. Behavioral analytics have also shown promising results for network intrusion [70, 72] and internal threat detection [60]. These works learn models to detect anomaly or predict attacks, but they do not provide mechanisms for users to perform attack investigation. Our AIQL system fills such gap by allowing security analysts to query historical events for investigating the reported anomalies.

9 Conclusion

We have presented a novel system for collecting attack provenance using system monitoring and assisting timely attack investigation. Our system provides (1) domain-specific data model and storage for scaling the storage and the search of system monitoring data, (2) a domain-specific query language, *Attack Investigation Query Language (*AIQL*)* that integrates critical primitives for attack investigation, and (3) an optimized query engine based on the characteristics of the data and the queries to better schedule the query execution. Compared with existing systems, our AIQL system greatly reduces the cycle time for iterative and interactive attack investigation.

Acknowledgement: This work was partially supported by the National Science Foundation under grants CNS-1553437 and CNS-1409415, Microsoft Research Asia, Jiangsu "Shuangchuang" Talents Program, CCF-NSFOCUS "Kunpeng" Research Fund, and Alipay Research Fund. Any opinions, findings, and conclusions made in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] AIQL: Enabling efficient attack investigation from system monitoring data. https://sites.google.com/site/aiqlsystem/.

[2] ANTLR. http://www.antlr.org/.

[3] Apache Flink. https://flink.apache.org/.

[4] CVE-2008-0081. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0081.

[5] CVE-2010-2075. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2075.

[6] Cyber Observable eXpression (CybOX^{TM}). https://cyboxproject.github.io/.

[7] Cypher Query Language. http://neo4j.com/developer/cypher/.

[8] Database performance tuning guide. https://docs.oracle.com/cd/B19306_01/server.102/b14211/.

[9] eBay Inc. To Ask eBay Users To Change Passwords. http://blog.ebay.com/ebay-inc-ask-ebay-users-change-passwords/.

[10] Elasticsearch. https://www.elastic.co/.

[11] The Equifax data breach. https://www.ftc.gov/equifax-data-breach.

[12] Esper. http://www.espertech.com/products/esper.php.

[13] ETW events in the common language runtime. https://msdn.microsoft.com/en-us/library/ff357719(v=vs.110).aspx.

[14] Greenplum. http://greenplum.org/.

[15] Home Depot confirms data breach at U.S., Canadian stores. http://www.npr.org/2014/09/09/347007380/home-depot-confirms-data-breach-at-u-s-canadian-stores.

[16] Neo4j: The world's leading graph database. http://neo4j.com/.

[17] Network Time Protocol (version 3) specification, implementation and analysis. https://tools.ietf.org/html/rfc1305.

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

[19] PostgreSQL. http://www.postgresql.org/.

[20] Protecting against potentially unwanted programs. https://portal.mcafee.com/documents/Show/2096.

[21] Siddhi. https://github.com/wso2/siddhi.

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

[23] Splunk. http://www.splunk.com/.

[24] Splunk: joining two searches with common field. https://answers.splunk.com/answers/105469/joining-two-searches-with-common-field.html.

[25] SQL. http://www.iso.org/iso/catalogue_detail.htm?csnumber=45498.

[26] Structured Threat Information eXpression (STIX^{TM}). http://stixproject.github.io/.

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

[28] The Linux audit framework. https://github.com/linux-audit/.

[29] Top 5 causes of sudden network spikes. https://www.paessler.com/press/pressreleases/top_5_causes_of_sudden_spikes_in_traffic.

[30] Transparent computing. http://www.darpa.mil/program/transparent-computing.

[31] Trusted Automated eXchange of Indicator Information (TAXII^{TM}). https://taxiiproject.github.io/.

[32] Trustwave global security report 2015. https://www2.trustwave.com/rs/815-RFM-693/images/2015_TrustwaveGlobalSecurityReport.pdf.

[33] Virussign. http://www.virussign.com/.

[34] Bates, A., Tian, D., Butler, K. R. B., and Moyer, T. Trustworthy whole-system provenance for the linux kernel. In *USENIX Security* (2015).

[35] Borders, K., Springer, J., and Burnside, M. Chimera: A declarative language for streaming network traffic analysis. In *USENIX Security* (2012).

[36] Chandola, V., Banerjee, A., and Kumar, V. Anomaly detection: A survey. *CSUR 41*, 3 (2009), 15:1–15:58.

[37] Chandra, R., Kim, T., Shah, M., Narula, N., and Zeldovich, N. Intrusion recovery for database-backed web applications. In *SOSP* (2011).

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

[39] Cuppens, F., and Ortalo, R. Lambda: A language to model a database for detection of attacks. In *RAID* (2000).

[40] Dolan-Gavitt, B., Hodosh, J., Hulin, P., Leek, T., and Whelan, R. Repeatable reverse engineering with panda. In *PPREW* (2015).

[41] Gao, P., Xiao, X., Li, D., Li, Z., Jee, K., Wu, Z., Kim, C. H., Kulkarni, S. R., and Mittal, P. SAQL: A stream-based query system for real-time abnormal system behavior detection. In *USENIX Security* (2018).

[42] Goel, A., Po, K., Farhadi, K., Li, Z., and de Lara, E. The taser intrusion recovery system. In *SOSP* (2005).

[43] Guo, Z., Wang, X., Tang, J., Liu, X., Xu, Z., Wu, M., Kaashoek, M. F., and Zhang, Z. R2: An application-level kernel for record and replay. In *OSDI* (2008).

[44] Hamilton, J. D. *Time series analysis*, vol. 2. Princeton University Press, 1994.

[45] Jiang, X., Walters, A., Xu, D., Spafford, E. H., Buchholz, F., and Wang, Y.-M. Provenance-aware tracing of worm break-in and contaminations: A process coloring approach. In *ICDCS* (2006).

[46] Killian, C. E., Anderson, J. W., Braud, R., Jhala, R., and Vahdat, A. M. Mace: Language support for building distributed systems. In *PLDI* (2007).

[47] Kim, T., Wang, X., Zeldovich, N., and Kaashoek, M. F. Intrusion recovery using selective re-execution. In *OSDI* (2010).

[48] King, S. T., and Chen, P. M. Backtracking intrusions. In *SOSP* (2003).

[49] King, S. T., Mao, Z. M., Lucchetti, D. G., and Chen, P. M. Enriching intrusion alerts through multi-host causality. In *NDSS* (2005).

[50] Ko, C., Ruschitzka, M., and Levitt, K. N. Execution monitoring of security-critical programs in distributed systems: a specification-based approach. In *IEEE S&P* (1997).

[51] Kolbitsch, C., Comparetti, P. M., Kruegel, C., Kirda, E., Zhou, X., and Wang, X. Effective and efficient malware detection at the end host. In *USENIX Security* (2009).

[52] Kruegel, C., Valeur, F., and Vigna, G. *Intrusion Detection and Correlation - Challenges and Solutions*, vol. 14 of *Advances in Information Security*. Springer, 2005.

[53] LANZI, A., BALZAROTTI, D., KRUEGEL, C., CHRISTODORESCU, M., AND KIRDA, E. Accessminer: Using system-centric models for malware protection. In *CCS* (2010).

[54] LEE, K. H., ZHANG, X., AND XU, D. High accuracy attack provenance via binary-based execution partition. In *NDSS* (2013).

[55] LEE, K. H., ZHANG, X., AND XU, D. Loggc: Garbage collecting audit log. In *CCS* (2013).

[56] LOO, B. T., CONDIE, T., GAROFALAKIS, M., GAY, D. E., HELLERSTEIN, J. M., MANIATIS, P., RAMAKRISHNAN, R., ROSCOE, T., AND STOICA, I. Declarative networking: Language, execution and optimization. In *SIGMOD* (2006).

[57] MA, S., LEE, K. H., KIM, C. H., RHEE, J., ZHANG, X., AND XU, D. Accurate, low cost and instrumentation-free security audit logging for windows. In *ACSAC* (2015).

[58] MA, S., ZHAI, J., WANG, F., LEE, K. H., ZHANG, X., AND XU, D. MPI: Multiple perspective attack investigation with semantic aware execution partitioning. In *USENIX Security* (2017).

[59] MA, S., ZHANG, X., AND XU, D. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS* (2016).

[60] SENATOR, T. E., GOLDBERG, H. G., MEMORY, A., YOUNG, W. T., REES, B., PIERCE, R., HUANG, D., REARDON, M., BADER, D. A., CHOW, E., ESSA, I., JONES, J., BETTADAPURA, V., CHAU, D. H., GREEN, O., KAYA, O., ZAKRZEWSKA, A., BRISCOE, E., MAPPUS, R. I. L., MCCOLL, R., WEISS, L., DIETTERICH, T. G., FERN, A., WONG, W.-K., DAS, S., EMMOTT, A., IRVINE, J., LEE, J.-Y., KOUTRA, D., FALOUTSOS, C., CORKILL, D., FRIEDLAND, L., GENTZEL, A., AND JENSEN, D. Detecting insider threats in a real corporate database of computer usage activity. In *KDD* (2013).

[61] SITARAMAN, S., AND VENKATESAN, S. Forensic analysis of file system intrusions using improved backtracking. In *IWIA* (2005).

[62] SNODGRASS, R. The temporal query language tquel. *TODS 12*, 2 (1987), 247–298.

[63] SNODGRASS, R. T. *Developing Time-oriented Database Applications in SQL*. Morgan Kaufmann Publishers Inc., 2000.

[64] SOMMER, R., AND PAXSON, V. Outside the closed world: On using machine learning for network intrusion detection. In *IEEE S&P* (2010).

[65] SOMMER, R., VALLENTIN, M., DE CARLI, L., AND PAXSON, V. Hilti: An abstract execution environment for deep, stateful network traffic analysis. In *IMC* (2014).

[66] STOLFO, S. J., HERSHKOP, S., BUI, L. H., FERSTER, R., AND WANG, K. Anomaly detection in computer security and an application to file system accesses. In *ISMIS* (2005).

[67] SUNG, A. H., XU, J., CHAVEZ, P., AND MUKKAMALA, S. Static analyzer of vicious executables (SAVE). In *ACSAC* (2004).

[68] VALLENTIN, M., PAXSON, V., AND SOMMER, R. Vast: A unified platform for interactive network forensics. In *NSDI* (2016).

[69] XU, Z., WU, Z., LI, Z., JEE, K., RHEE, J., XIAO, X., XU, F., WANG, H., AND JIANG, G. High fidelity data reduction for big data security dependency analyses. In *CCS* (2016).

[70] YEN, T.-F., AND REITER, M. K. Traffic aggregation for malware detection. In *DIMVA* (2008).

[71] YU, S. Understanding the security vendor landscape using the cyber defense matrix, 2016. RSA Conferences.

[72] ZHANG, H., YAO, D. D., AND RAMAKRISHNAN, N. Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery. In *ASIA CCS* (2014).