A Query System for Efficiently Investigating Complex Attack Behaviors for Enterprise Security

Peng Gao[1] Xusheng Xiao[2] Zhichun Li[3] Kangkook Jee[4] Fengyuan Xu[5]
Sanjeev R. Kulkarni[6] Prateek Mittal[6]
[1]UC Berkeley [2]Case Western Reserve University [3]NEC Labs America [4]UT Dallas [5]Nanjing University
[6]Princeton University

penggao@berkeley.edu xusheng.xiao@case.edu zhichun@nec-labs.com kangkook.jee@utdallas.edu fengyuan.xu@nju.edu.cn
kulkarni@princeton.edu pmittal@princeton.edu

ABSTRACT

The need for countering Advanced Persistent Threat (APT) attacks has led to the solutions that ubiquitously monitor system activities in each enterprise host, and perform timely attack investigation over the monitoring data for uncovering the attack sequence. However, existing general-purpose query systems lack explicit language constructs for expressing key properties of major attack behaviors, and their semantics-agnostic design often produces inefficient execution plans for queries. To address these limitations, we build Aiql, a novel query system that is designed with novel types of domain-specific optimizations to enable efficient attack investigation. Aiql provides (1) a domain-specific data model and storage for storing the massive system monitoring data, (2) a domain-specific query language, *Attack Investigation Query Language (*Aiql*)* that integrates critical primitives for expressing major attack behaviors, and (3) an optimized query engine based on the characteristics of the data and the semantics of the query to efficiently schedule the execution. We have deployed Aiql in NEC Labs America comprising 150 hosts. In our demo, we aim to show the complete usage scenario of Aiql by (1) performing an APT attack in a controlled environment, and (2) using Aiql to investigate such attack by querying the collected system monitoring data that contains the attack traces. The audience will have the option to perform the APT attack themselves under our guidance, and interact with the system and investigate the attack via issuing queries and checking the query results through our web UI.

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 with significant losses [6, 4]. In order for enterprises to counter APT attacks, recent approaches based on *ubiquitous system monitoring* have emerged as an important solution for *monitoring system activities and performing attack investigation* [9, 10, 8, 7]. System monitoring observes system calls at the kernel level to collect system-level events that record system interactions among system entities (e.g., processes, files, and network sockets). Collection of system monitoring data enables security analysts to investigate these attacks by *querying attack behaviors* over the historical data.

Attack investigation is a time-sensitive task. However, there are two major challenges for building a query system to support efficient and timely attack investigation:

(1) Attack Behavior Specification: The system needs to provide a query language with specialized constructs for expressing major attack behaviors: **a. Multi-step attacks:** complex attacks such as APTs typically involve multiple system activities connected by specific attribute relationships (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*; **b. Dependency tracking of attacks**: dependency tracking is widely used in investigation to track causality of data for discovering the attack entry [9], which requires language constructs to easily *chain constraints among activities*; **c. Abnormal system behaviors**: frequency-based anomaly models are widely used to investigate abnormal system behaviors, such as network access spikes, which requires language constructs to easily specify *sliding windows* and *statistical aggregation* of system activities and compare the aggregate results in the current window with the results in previous windows.

(2) Timely Big-Data Analysis: System monitoring produces a huge amount of daily logs [9, 10] ($\sim$ 50 GB per day for 100 hosts), and the investigation of APT attacks typically requires the enterprise to keep at least a $0.5 \sim 1$ year worth of data. Such *a huge amount of data* poses challenges for a timely investigation: the system needs to provide efficient data storage and query execution engine.

Unfortunately, existing query systems do not address both of these inherent challenges: (1) Existing query languages in relational databases (e.g., PostgreSQL), graph databases
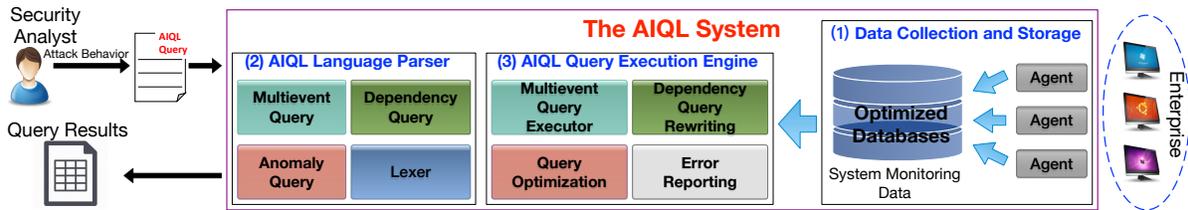
Figure 1: Architecture of the AIQL system

(e.g., Neo4j), and other NoSQL databases (e.g., MongoDB, SPARQL) lack explicit constructs to chain constraints among system activities and specify their relationships. To specify an attack behavior with multiple steps, these languages often lead to large queries with many joins and constraints mixed together, posing great challenges for performance tuning. Constructing such queries correctly is also time consuming and error-prone. Moreover, none of these languages provide explicit constructs for expressing behavioral models with accesses to historical aggregate results; (2) System monitoring data is generated with a timestamp on a specific host in the enterprise, exhibiting strong *spatial and temporal properties*. However, existing query systems are designed to work with general-purpose data thus missing opportunities for optimizations based on the domain data characteristics, which might lead to some queries executing very inefficiently.

To address these challenges, we build AIQL [8], a system that enables security analysts to perform efficient attack investigation via querying system monitoring data. AIQL employs three novel types of optimizations: (1) AIQL provides a domain-specific query language, *Attack Investigation Query Language (AIQL)*, which is optimized to express the three aforementioned types of attack behaviors; (2) AIQL provides a domain-specific *data model and storage* for scaling the storage; (3) AIQL optimizes the query engine based on the domain-specific characteristics of the system monitoring data and the semantics of the query for efficient execution.

We have deployed the AIQL system in NEC Labs America comprising 150 hosts and made a demo video [3]. In our demo, we aim to show the complete usage scenario of AIQL. We first perform an APT attack in a controlled environment (for protecting the normal business) that exfiltrates sensitive data from database server by exploiting multiple vulnerabilities in multiple steps. The system monitoring data that contains the attack traces is collected by our data collection agents and stored in our optimized databases. Then, we use AIQL to investigate the attack by querying the collected data. The audience will have the option to perform the APT attack themselves under our guidance, and interact with the system and investigate the attack via issuing queries and checking the query results through our web UI. The audience will also experience the superiority of AIQL by comparing the conciseness and performance of AIQL queries with SQL queries executed in PostgreSQL databases.

2. THE AIQL SYSTEM ARCHITECTURE

Figure 1 shows the architecture of the AIQL system. AIQL takes an input query from the user (e.g., security analyst) that specifies certain attack behaviors to be investigated, executes the query, and retrieves the matched results.

2.1 Data Collection and Storage

Data Model. System monitoring data records the interactions among system entities as system events. Each of the recorded event occurs on a particular host at a particular time, thus exhibiting strong spatial and temporal properties. 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 as ⟨*subject, operation, object*⟩ (SVO). Subjects are processes originating from software applications (e.g., Firefox), and objects can be files, processes, and network connections. We categorize system events into three types according to their objects, namely *file events*, *process events*, and *network events*.

Data Collection. We develop data collection agents based on mature system monitoring frameworks: auditd for Linux, ETW for Windows, and DTrace for MacOS. Our agents are deployed across servers, desktops, and laptops in the enterprise and collect critical security-related attributes (e.g., file name, process executable name, IP, port, etc.; details in [8]).

Data Storage. Querying complex attack behaviors typically requires the efficient support for joins. Compared to graph databases and other NoSQL databases, relational databases come with mature indexing mechanisms and are more scalable to the massive data in our context. Thus, in AIQL, we store the collected system monitoring data in relational databases (PostgreSQL and Greenplum). We further optimize the write throughput and the data storage using techniques such as data deduplication and in-memory indexes, batch commit, time and space partitioning, and hypertable (details in [8]).

2.2 AIQL Query Language

We build the AIQL language using ANTLR 4. Our language uniquely integrates a series of critical primitives for concisely expressing three major types of attack behaviors.

2.2.1 Multievent AIQL query

AIQL provides explicit constructs for system events, spatial/temporal constraints, and event temporal/attribute relationships, which facilitates the specification of multi-step attack behaviors. Query 1 shows a multievent AIQL query that investigates the data exfiltration from database server: the attacker leverages OSQL utility (`osql.exe`) to dump the database content (`backup1.dmp`) and runs a malware (`sbblv.exe`) to send the dump back to his host (`XXX.129`). Four event patterns are declared (Lines 3-6) with two global constraints (Lines 1-2), a temporal relationship (Line 7), and an implicit attribute relationship (Lines 4-5 specify the same `f1` in both events). Desired attributes of matched events are returned (Line 8) with context-aware syntax shortcuts adopted (i.e., `p1 → p1.exe_name`, `f1 → f1.name`, `i1 → i1.dst_ip`).

```
1  (at "mm/dd/2018") // time window (obfuscated)
2  agentid = xxx // SQL database server (obfuscated)
3  proc p1["%cmd.exe"] start proc p2["%osql.exe"] as evt1
4  proc p3["%sqlservr.exe"] write file f1["%backup1.dmp"] as
     evt2
```
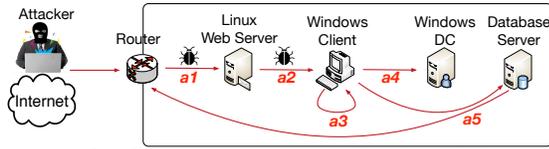
Figure 2: Demonstration setup for the APT attack

```
5 proc p4["%sbblv.exe"] read file f1 as evt3
6 proc p4 read || write ip i1[dstip="XXX.129"] as evt4
7 with evt1 before evt2, evt2 before evt3, evt3 before evt4
8 return distinct p1, p2, p3, f1, p4, i1
```

Query 1: Data exfiltration from database server

2.2.2 Dependency AIQL query

AIQL provides explicit constructs for chaining constraints among system events in the form of event path, which facilitates the dependency tracking of attacks. Query 2 shows a forward dependency AIQL query that investigates the ramification of a malware (`info_stealer`), which originates from Host 1 (`agentid = 1`) and affects Host 2 (`agentid = 2`) through an Apache web server. An example execution result may show that `p3` is the `wget` process that downloads the malicious script from Host 2. The `forward` keyword (Line 2) specifies the temporal order of the events: left event occurs earlier. The operation `connect` (Line 4) indicates that the tracking is across different hosts.

```
1 (at "mm/dd/2018") // time window (obfuscated)
2 forward: proc p1["%/bin/cp%", agentid = 1] ->[write] file
        f1["/var/www/%info_stealer%"]
3 <-[read] proc p2["%apache%"]
4 ->[connect] proc p3[agentid=2] // tracking across hosts
5 ->[write] file f2["%info_stealer%"]
6 return f1, p1, p2, p3, f2
```

Query 2: Forward tracking for malware ramification

2.2.3 Anomaly AIQL query

AIQL provides explicit constructs for sliding windows, aggregation functions, and accesses to historical aggregate results, which facilitates the specification of frequency-based anomaly models. Query 3 shows an anomaly AIQL query that specifies a 1-minute sliding window (Line 3) and computes a moving average (Line 7) to investigate processes on the database server (Line 2) that transfer a large amount of data to a suspicious IP (`XXX.129`). An example execution result may show that the process `p` is `sbblv.exe`, which is suspicious and deserves further investigation.

```
1 (at "mm/dd/2018") // time window (obfuscated)
2 agentid = xxx // SQL database server (obfuscated)
3 window = 1 min, step = 10 sec
4 proc p write ip i[dstip="XXX.129"] as evt
5 return p, avg(evt.amount) as amt
6 group by p
7 having (amt > 2 * (amt + amt[1] + amt[2]) / 3)
```

Query 3: Large data transfer from database server

2.3 AIQL Query Execution Engine

Our query execution engine leverages domain-specific characteristics of the data and the semantics of the query to efficiently schedule the execution. Optimizing a query with many constraints is a difficult task due to the complexities of joins and constraints. For a multievent query, AIQL addresses this challenge by synthesizing a *SQL data query* for *every* event pattern and schedules the execution of these data queries using our *optimized scheduling strategy*, rather than weaving all the joins and constraints together in a large SQL query and relying on the inefficient default SQL engine scheduling. Our *optimized scheduling strategy* (details in [8]) has two key insights: (1) for a query with multiple event patterns, we prioritize the search of event patterns with higher pruning power, maximizing the reduction of irrelevant events as early as possible; (2) we partition the query into independent sub-queries along the temporal (i.e., time window) and spatial (i.e., agent ID) dimensions and execute these sub-queries in parallel. For a dependency query, the parser compiles it to a semantically equivalent multievent query for execution. For an anomaly query, the engine partitions the events into sliding windows by the timestamp, computes the aggregate results, and enforces the filters.

3. DEMONSTRATION OUTLINE

Demonstration Setup. We have deployed AIQL in NEC Labs America comprising 150 hosts. The purpose of our demo is to illustrate the complete usage scenario of AIQL and showcase its superiority in enabling efficient attack investigation. To achieve this goal, we perform an APT attack in a controlled environment (Figure 2) using a set of known exploits. The APT attack consists of five steps as follows:

a1 *Initial Compromise*: The attacker first exploits the UnReal IRC server remote code execution vulnerability [1] to create a telnet connection to his host.

a2 *Malware Infection*: The attacker uploads a malware via the connection and waits for the malware to infect other hosts to gain access to the intranet.

a3 *Privilege Escalation*: With the access to the intranet, the attacker leverages other vulnerabilities [2] to escalate his privilege and executes memory dumping tools (Mimikatz, Kiwi) to obtain administrator credentials.

a4 *Obtain User Credentials*: The attacker penetrates into the domain controller and executes password dumping tools (PwDump7.exe, WCE.exe) to obtain the credentials of all users.

a5 *Data Exfiltration*: Finally, the attacker penetrates into the database server and dumps the data back to his host.

Live End-to-End Investigation Procedure. After performing the attack, the system monitoring data that contains the attack traces is collected by our data collection agents and stored in our optimized databases. Next, we begin the attack investigation process by constructing and iteratively revising AIQL queries. Assuming no prior knowledge of the attack, we start the investigation by first constructing an anomaly AIQL query and identify a process "powershell.exe" transferring large data to a suspicious external IP "XXX.129" from the database server. We then construct a multievent AIQL query to investigate the files read by this process and identify a database dump file "db.bak". We further investigate the creation process of this dump file and identify "sqlservr.exe", which is a standard SQL server process with verified signature. We also confirm that the process "powershell.exe" creates a connection to the IP "XXX.129" before the data transfer. This confirms the existence of data exfiltration from the database server and completes the investigation of the step *a5*. We follow a similar procedure for investigating the steps *a1-a4*. Please refer to [5] for more investigation details and all AIQL queries.
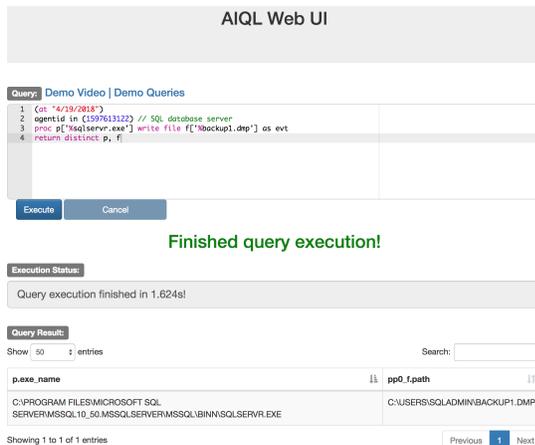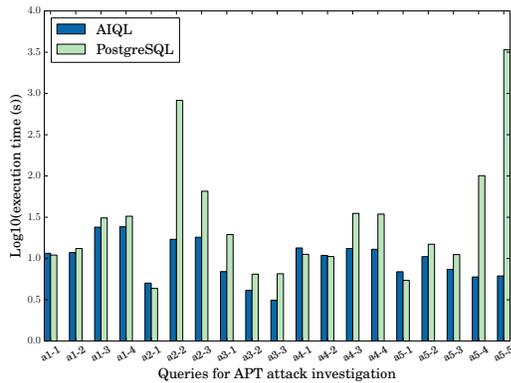
Figure 3: Web UI of the AIQL system

Figure 4: Log10-transformed query execution time

Web UI. In our demo, the audience will have the option to perform the attack under our guidance and do the investigation themselves by interacting with AIQL. To facilitate such interaction, we built a web UI (Figure 3) upon Apache Tomcat. Our web UI consists of (1) an input box for entering AIQL queries, (2) an execution status area to show the query execution time, and (3) an interactive table that visualizes and manages the execution results. Furthermore, our web UI provides query editing and result analysis features to facilitate efficient investigation: (1) syntax highlighting for query construction, (2) syntax checking for query debugging, and (3) sorting and searching for result management. To get a better sense of how to use the web UI and interpret the query results, please refer to our demo video [3].

Post-Demo Evaluation: AIQL vs. PostgreSQL (w/ Our Optimized Storage). Our investigation used 19 multievent queries and 1 anomaly query, touching 85 GB of data/257 million events. Figure 4 shows the log10-transformed execution time of AIQL queries and the semantically equivalent SQL queries executed in PostgreSQL. Note that both AIQL and PostgreSQL employ our data storage optimizations. We observe clear superiority of AIQL in scheduling the execution of complex queries (e.g., *a2-2, a5-5*). The total execution time of AIQL is 3.6 minutes, achieving 21× performance speedup over PostgreSQL (77 minutes). For the query conciseness, SQL queries contain at least 3.0× more constraints, 3.5× more words, and 5.2× more characters (excluding spaces) than AIQL queries.

Post-Demo Evaluation: AIQL vs. PostgreSQL (w/o Our Optimized Storage) vs. Neo4j. In another case
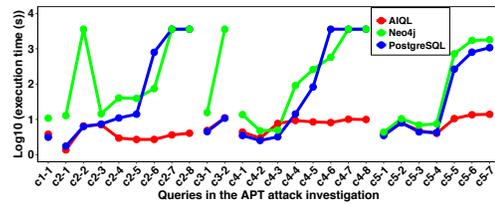

Figure 5: Log10-transformed query execution time for another APT attack in [8]

study of APT attack [8], we evaluated the performance of AIQL against PostgreSQL w/o our optimizations and Neo4j. As shown in Figure 5, the AIQL system as a whole is much faster than PostgreSQL (124× speedup) and Neo4j (157× speedup). In particular, Neo4j runs generally slower than PostgreSQL since it lacks support for efficient joins, which are required in expressing attack behaviors with multiple steps. As the attack behaviors become more complex, besides performance degradation, both SQL and Cypher queries become quite verbose with many joins and constraints, making it labor-intensive and error prone in constructing queries for timely attack investigation [8].

4. CONCLUSION

We have presented AIQL, a novel system for efficient attack investigation via querying system monitoring data. Compared with existing solutions, AIQL significantly reduces the cycle time for interactive investigation and improves the query conciseness.

5. REFERENCES

[1] CVE-2010-2075. https://goo.gl/MmskVz.

[2] CVE-2015-1701. https://goo.gl/MmDcJx.

[3] Demo video of AIQL.
https://youtu.be/2dDVngg0UN8.

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

[5] Project website.
https://sites.google.com/site/aiqldemo/.

[6] Target data breach. https://goo.gl/2awnKE.

[7] 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*, pages 639–656, 2018.

[8] P. Gao, X. Xiao, Z. Li, K. Jee, F. Xu, S. R. Kulkarni, and P. Mittal. AIQL: Enabling efficient attack investigation from system monitoring data. In *USENIX ATC*, pages 113–126, 2018.

[9] S. T. King and P. M. Chen. Backtracking intrusions. In *SOSP*, pages 223–236, 2003.

[10] 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 *CCS*, pages 504–516, 2016.