Abstract

Advanced Persistent Threat (APT) attacks and data breaches are sophisticated and stealthy, plaguing many well-protected businesses (e.g., Target, Yahoo, Home Depot, eBay, Equifax, Marriott, etc.) with significant losses. To counter these advanced attacks, approaches based on ubiquitous system monitoring have emerged as an important solution for monitoring system activities from enterprise hosts and performing forensic analysis. System monitoring audits system calls at the kernel level to collect information about system activities, providing a global view of interactions among applications and system resources. Collection of system monitoring data enables security analysts to identify the root causes and the ramifications of attacks (i.e., attack investigation) and to detect the abnormal behaviors of attacks (i.e., attack detection). However, the daunting amount of system monitoring data and the complexity of advanced attacks pose significant challenges for designing solutions for effective and efficient forensic analysis.

In this thesis, we propose novel approaches for effective and efficient forensic analysis (attack investigation and attack detection) via system monitoring. First, we propose Aiql, a system that enables efficient post-mortem attack investigation via querying the historical system monitoring data. Second, we propose Saql, a system that enables real-time abnormal system behavior detection via querying the stream of system monitoring data. Both Aiql and Saql provide (1) domain-specific languages that uniquely integrate critical primitives for easily incorporating the domain knowledge of security experts to express a wide range of attack behaviors, and (2) query execution engines that employ novel optimizations based on the domain-specific characteristics of the system monitoring data and the semantics of the query for efficient query execution. Finally, we propose SysRep, a system that facilitates automatic attack investigation via (1) propagating reputation from seed sources (can be trusted or suspicious) along system dependency paths to infer the reputation of POI (point
of interest) entities (e.g., files, network sockets), and (2) automatically reconstructing the attack sequence from POI entities. Together, AIQL, SAQL, and SysRep work seamlessly for effective and efficient forensic analysis of APT attacks.
Acknowledgements

This thesis would not be possible without the help of my research advisors, professor Prateek Mittal and professor Sanjeev R. Kulkarni. Their approaches to problem discovery, problem solving, and student mentoring have been a great learning experience, which has enlightened my research career. I would like to thank Prateek and Sanjeev for their great mentorship, long-term support, and encouragement throughout my doctoral study.

I would like to thank my doctoral committee members, professor Niraj K. Jha and professor Paul R. Prucnal for their feedback and support. I would like to thank professor Xusheng Xiao for being a reader of my thesis and for being a great mentor of my two internships at NEC Labs America. Xusheng introduced me to the wonderful world of systems security and forensic analysis, which has helped shape my research as well as this thesis. I would like to thank professor Neil Zhenqiang Gong for being the first mentor of my research outside Princeton. Neil introduced me to the wonderful world of social networks, which has helped me publish two papers in my early study.

I would like to thank all of my collaborators. During my doctoral study, I have collaborated with Neil Zhenqiang Gong, Binghui Wang, Prateek Mittal, Sanjeev R. Kulkarni, Changchang Liu, Matthew Wright, Marius Jakubowski, Xusheng Xiao, Zhichun Li, Ding Li, Kangkook Jee, Fengyuan Xu, Zhenyu Wu, Pengcheng Fang, Changlin Liu, Zhuotao Liu, Xiaoyin Wang, and others. I was fortunate to work with them and thanks for many insightful discussions.

I would like to thank my fellow students at Princeton INSPIRE Research Group and friends at Princeton University - Changchang Liu, Sameer Wagh, Yushan Liu, Yixin Sun, Thee Chanyaswad, Liwei Song, Zhixing Xu, Xinyu He, Wei-Han Lee, Yechi Ma, Jie Lu, and others. I am also grateful to the teachers of the courses that I have taken. A special thank goes to professor Ruby Lee, who was the committee member of my General Examination in year two and gave me a lot of feedback. Another
special thank goes to professor Sun-Yuan Kung, who was my academic advisor in the first semester of year one and introduced me to Princeton. Thanks for the great memory and learning experience.

I would like to thank professor Dawn Song, who offered me the position of Visiting Student Researcher in the Department of Electrical Engineering and Computer Sciences at University of California, Berkeley, and supported me in completing this thesis. I am also grateful to my officemates and colleagues at UC Berkeley - Jian Liu, Min Du, Ruoxi Jia, and Lun Wang, for their insightful discussions.

I would like to thank the funding agencies for supporting my doctoral study. My work was supported by the National Science Foundation under grants CNS-1553437 and CNS-1409415. Parts of this thesis have been published in USENIX ATC 2018 and USENIX Security 2018. The SysRep system is currently under submission at ACM CCS 2019.

Being able to pursue a doctoral degree at Princeton University was in large part due to excellent teachers and mentors that I encountered in my life. I am grateful to professor Horst Hohberger, professor Xinen Zhu, and professor Peisen Huang at Shanghai Jiao Tong University. Without their mentorship and support during my undergrad study, it would be unlikely for me to be accepted by Princeton as a Ph.D. student and to pursue a research career.

Finally, I would like to thank my wife and my parents. This thesis would not be possible without their love and support. I am especially grateful to my wife, Yiqi Su, who has supported me through many challenges in life.
To my family
Contents

Abstract .............................................................. iii
Acknowledgements ..................................................... v
List of Tables .......................................................... xii
List of Figures .......................................................... xiii

1 Introduction ............................................................. 1
  1.1 Contributions ...................................................... 3
    1.1.1 Efficient Attack Investigation via Querying the Historical System Monitoring Data ........................................... 3
    1.1.2 Real-Time Abnormal System Behavior Detection via Querying the Stream of System Monitoring Data ......................... 4
    1.1.3 Automatic Attack Investigation via Weight-Aware Reputation Propagation from System Monitoring ......................... 4
    1.1.4 Organization ..................................................... 5

2 Literature Review ..................................................... 6
  2.1 System Audit Logging and Forensic Analysis ......................... 6
    2.1.1 Causality Analysis and Runtime Analysis ......................... 6
    2.1.2 Reduction of System Monitoring Data .............................. 8
  2.2 Domain-Specific Languages ........................................ 8
    2.2.1 Security-Related Languages ...................................... 8
2.2.2 System Analysis Languages ........................................ 9
2.3 Database Management Systems and Stream Processing Systems 9
  2.3.1 Database Query Languages ....................................... 9
  2.3.2 Complex Event Processing Platforms & Data Stream Management Systems .......................... 10
  2.3.3 Stream Computation Systems ................................. 11
2.4 Other Aspects of Security Defenses ............................... 11
  2.4.1 System Defenses Based on Behavioral Analytics ........... 11
  2.4.2 Security Anomaly Detection ................................... 12
2.5 Reputation Propagation ............................................. 12
2.6 Threat Model .......................................................... 13
3 AIQL: Enabling Efficient Attack Investigation from System Monitoring Data ................................. 14
  3.1 Overview ............................................................... 18
  3.2 Design of The AIQL System ........................................ 18
    3.2.1 Data Collection ............................................... 18
    3.2.2 Data Storage ................................................ 20
    3.2.3 Query Language Design .................................... 22
    3.2.4 Query Execution Engine .................................... 27
    3.2.5 Web UI ....................................................... 34
  3.3 Evaluation ............................................................ 35
    3.3.1 Evaluation Setup .............................................. 35
    3.3.2 Case Study: APT Attack Investigation ..................... 36
    3.3.3 Performance Evaluation ..................................... 43
    3.3.4 Conciseness Evaluation ..................................... 48
  3.4 Discussion ............................................................. 48
  3.5 Summary .............................................................. 49
4 SAQL: A Stream-Based Query System for Real-Time Abnormal System Behavior Detection 50

4.1 Overview .......................................................... 54
4.2 Design of The SAQL System .................................. 56
   4.2.1 Data Collection .............................................. 56
   4.2.2 Query Language Design ................................. 57
   4.2.3 Example SAQL Queries .................................. 63
   4.2.4 Query Execution Engine ................................ 66
4.3 Evaluation ........................................................ 70
   4.3.1 Evaluation Setup .......................................... 70
   4.3.2 Case Study of Four Major Types of Attacks ......... 71
   4.3.3 Pressure Test ................................................ 76
   4.3.4 Performance Evaluation of Concurrent Query Execution .... 78
4.4 Discussion .......................................................... 81
4.5 Summary ............................................................ 82

5 SysRep: Towards Automatic Attack Investigation via Weight-Aware Reputation Propagation from System Monitoring 84

5.1 Background and Motivating Example ....................... 90
   5.1.1 System Monitoring ....................................... 90
   5.1.2 Causality Analysis ....................................... 91
   5.1.3 Motivating Example ..................................... 91
5.2 Overview .......................................................... 93
5.3 Design of The SysRep System ................................. 94
   5.3.1 Phase I: Dependency Graph Generation ............... 94
   5.3.2 Phase II: Graph Preprocessing ......................... 95
   5.3.3 Phase II: Feature Extraction ............................ 97
   5.3.4 Phase II: Weight Computation ......................... 99
# List of Tables

3.1 Representative attributes of system entities .......................................... 20
3.2 Representative attributes of system events ........................................... 20
3.3 Aggregate statistics for case study ....................................................... 43
3.4 Selected malware samples from Virussign ........................................... 44
3.5 Conciseness improvement statistics ...................................................... 48

4.1 Representative attributes of system entities ........................................... 57
4.2 Representative attributes of system events ........................................... 57
4.3 Execution statistics of 17 SAQL queries for four major types of attacks ...... 75

5.1 Representative attributes of system entities ........................................... 90
5.2 Representative attributes of system events ........................................... 90
5.3 System calls processed by SysRep ....................................................... 94
5.4 POI reputations of benign payloads through key system interfaces (expected 1.0) ................................................................. 110
5.5 POI reputations of malicious payloads through key system interfaces (expected: 0.0) ................................................................. 110
5.6 POI reputations of APT attacks (expected: 0.0) ..................................... 111

A.1 List of vulnerabilities and tools for the attack ....................................... 121
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Major types of attack behaviors (events $e_1, \ldots, e_n$ are shown in ascending temporal order)</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>Architecture of the AiQL system</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>Execution pipeline of an AiQL query</td>
<td>28</td>
</tr>
<tr>
<td>3.4</td>
<td>Execution pipeline of a multievent AiQL query</td>
<td>29</td>
</tr>
<tr>
<td>3.5</td>
<td>Web UI of the AiQL system</td>
<td>34</td>
</tr>
<tr>
<td>3.6</td>
<td>Environmental setup for the APT attack</td>
<td>36</td>
</tr>
<tr>
<td>3.7</td>
<td>Log10-transformed query execution time</td>
<td>41</td>
</tr>
<tr>
<td>3.8</td>
<td>Query execution time of the scheduling employed by PostgreSQL, AiQL FF, and AiQL (single-node)</td>
<td>45</td>
</tr>
<tr>
<td>3.9</td>
<td>Query execution time of the scheduling employed by Greenplum and AiQL (parallel)</td>
<td>46</td>
</tr>
<tr>
<td>3.10</td>
<td>Conciseness evaluation of queries written in AiQL, SQL, Neo4j Cypher, and Splunk SPL</td>
<td>47</td>
</tr>
<tr>
<td>4.1</td>
<td>Major types of abnormal system behaviors ($e_1, \ldots, e_n$ are shown in ascending temporal order)</td>
<td>51</td>
</tr>
<tr>
<td>4.2</td>
<td>The architecture of the SaQL system</td>
<td>55</td>
</tr>
<tr>
<td>4.3</td>
<td>Stream replayer</td>
<td>71</td>
</tr>
<tr>
<td>4.4</td>
<td>Environmental setup for the APT attack</td>
<td>71</td>
</tr>
<tr>
<td>4.5</td>
<td>Throughput of the SaQL system under different CPU utilizations</td>
<td>76</td>
</tr>
</tbody>
</table>
5.1 Partial dependency graph of a motivating data leakage attack case.

Rectangles denote processes, ovals denote files, and parallelograms denote network sockets. Yellow ovals are the files leaked in this attack.

Nodes in the attack path have red frames while normal activities and system libraries are in dashed green rectangles. Critical edges are represented with solid red arrows. Non-critical edges are represented with dashed arrows. The complete graph contains 5,817 nodes and 215,380 edges. Note that the reputation of the suspicious source on the top left propagates mainly through the critical edges.

5.2 The architecture of the SysRep system.

5.3 Effectiveness of filtering. The percentage of edges remaining after filtering drops significantly at $T = 0.10$ and remains stable below 10%.

5.4 Critical edge loss from filtering. All missing points are distributed between $T = 0.25$ and $T = 1.0$.

A.1 Environmental setup for the APT attack in Sections 3.3.3 and 3.3.4.
Chapter 1

Introduction

Advanced Persistent Threat (APT) attacks have plagued many well-protected businesses with significant financial losses [23, 35, 29, 18, 25, 60, 31]. For example, the massive Equifax data breach [25] in 2017 has exposed the sensitive personal information of 143 million US customers, affecting nearly half of the U.S. population. Unlike convention attacks, these advanced attacks are sophisticated (involving many individual attack steps across many hosts and exploiting various vulnerabilities) and stealthy (each individual step is not suspicious enough) [52, 3, 1], posing significant challenges for designing efficient approaches for countering these attacks.

In order for enterprises to counter advanced attacks, it is crucial to understand the activities of hosts at a fine-grained level. Recent approaches based on ubiquitous system monitoring have emerged as an important solution for monitoring system activities and performing forensic analysis [111, 112, 96, 110, 78, 110, 125, 123, 117, 124, 115, 121, 102]. System monitoring audits system calls at the kernel level to collect system-level events that record interactions among applications and system resources (e.g., files, processes, and network sockets), providing a global view of system behaviors across all hosts in the enterprise. Collection of system monitoring data enables
security analysts to identify the root causes and the ramifications of attacks (i.e., attack investigation), and to detect attacks in progress (i.e., attack detection).

While the research on forensic analysis via system monitoring shows great potential, there are three major challenges for designing solutions that demonstrate the practical efficacy in the forensic analysis of real-world, large cyber attacks:

(1) Expert Knowledge Incorporation: Advanced attacks such as APT typically involve multiple steps across many hosts, exploiting various types of vulnerabilities. Besides, models derived from data have been increasingly used in detecting and investigating various types of attack behaviors [114]. System administrators, security analysts, and security experts have extensive domain knowledge about the enterprise, such as the expected system behaviors and the suspicious behaviors. Fighting against such complex attacks requires the solution to identify a wide range of attack behaviors while incorporating the domain knowledge from experts.

(2) Timely Big-Data Analytics: System monitoring produces a huge amount of daily logs (~50GB for 100 hosts per day) [153, 148]. Furthermore, since advanced attacks such as APT often infiltrate into the target system in multiple stages, spanning a long duration of time with a low profile, the investigation of these attacks typically requires the enterprise to keep at least 0.5 ~ 1 year worth of data [54]. Such a huge amount of system monitoring data poses challenges for the solution to provide timely big-data analytics for efficient forensic analysis. Nevertheless, fighting against advanced attacks such as APT is a time-critical mission. As such, we need a timely solution to search for a “needle in a haystack” for preventing additional damage and for system recovery.

(3) Automatic Attack Investigation: Attack investigation aims to investigate the POI (point of interest) events to confirm whether the events indicate attacks, and if so to identify the root causes and the ramifications of the attacks. The investigation of advanced attacks such as APT is a tedious process, given the complexity of the
attacks and the daunting amount of system monitoring data. As such, it is desirable to have a solution that automates the attack investigation process to a certain level, which facilitates the reduction of efforts in manual inspection by security analysts.

Unfortunately, none of the existing approaches propose practical solutions that address these challenges. In this thesis, we investigate the design of novel approaches that demonstrate the practical efficacy in effective and efficient forensic analysis of advanced cyber attacks.

1.1 Contributions

1.1.1 Efficient Attack Investigation via Querying the Historical System Monitoring Data

First, we propose AiQL, a novel system that enables efficient post-mortem attack investigation via querying the historical system monitoring data. The AiQL 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 expressing three major types of attack behaviors: multi-step attacks, dependency tracking of attacks, and abnormal system behaviors, and (3) an optimized query engine based on the characteristics of the data and the semantics of the query to efficiently schedule the query execution. We demonstrate the practical efficacy of AiQL on a wide range of attacks, and show that AiQL surpasses existing systems in both query efficiency (124x over PostgreSQL, 157x over Neo4j, and 16x over Greenplum) and query conciseness (SQL, Neo4j Cypher, and Splunk SPL contain at least 2.4x more constraints than AiQL queries).
1.1.2 Real-Time Abnormal System Behavior Detection via Querying the Stream of System Monitoring Data

Next, we propose SaQL, a novel system that enables real-time abnormal system behavior detection via querying the stream of system monitoring data. The SaQL system takes as input a real-time event feed aggregated from multiple hosts in an enterprise, and provides an anomaly query engine that queries the event feed to identify abnormal behaviors based on the specified anomaly models. To facilitate the specification of a variety of anomaly models based on the expert knowledge, the SaQL system provides a domain-specific query language, *Stream-Based Anomaly Query Language (SaQL)*, that integrates critical primitives for expressing four major types of anomaly models: rule-based anomaly models, time-series anomaly models, invariant-based anomaly models, and outlier-based anomaly models. Our evaluation on a wide range of attacks and micro-benchmarks demonstrates that SaQL has a low alert detection latency and a high system throughput, and is more efficient in memory utilization than existing stream-based complex event processing systems.

1.1.3 Automatic Attack Investigation via Weight-Aware Reputation Propagation from System Monitoring

Finally, we propose SysRep, a novel system that facilitates automatic attack investigation. SysRep assigns discriminative weights to edges in a dependency graph built from system monitoring data and propagates reputation scores in the weighted dependency graph from seed sources to infer the reputation of POI (point of interest) entities (e.g., files, network sockets). The reputation propagation paradigm enables SysRep to automatically determine the suspiciousness/trustworthiness of POI entities based on whether the system entities originate from suspicious sources or trusted sources. The discriminative edge weights enable SysRep to reveal critical edges for
automatically reconstructing the attack sequence. Our evaluation on a wide range of attacks demonstrates the practical efficacy of SysRep in reputation inference of POI and attack sequence reconstruction.

1.1.4 Organization

In Chapter 2, we review the related work on forensic analysis as well as other related domains. We also present our threat model. In Chapter 3, we present the AiQL system for attack investigation via querying the historical system monitoring data. In Chapter 4, we present the SaQL system for abnormal system behavior detection via querying the stream of system monitoring data. In Chapter 5, we present the SysRep system for automatic attack investigation via weight-aware reputation propagation from system monitoring. In Chapter 6, we conclude this thesis.
Chapter 2

Literature Review

In this section, we review the related work on system audit logging/system monitoring and forensic analysis, security-related languages and system analysis languages, database management systems and stream processing systems, system defenses based on behavioral analytics, security anomaly detection, as well as reputation propagation mechanisms. We also describe our threat model.

2.1 System Audit Logging and Forensic Analysis

2.1.1 Causality Analysis and Runtime Analysis

Significant progress has been made to leverage system audit logs for forensic analysis. Among the proposed approaches, causality analysis based on provenance graphs generated from system monitoring data plays a critical role. King et al. [111] proposed a backward causality analysis technique to perform intrusion analysis by automatically reconstructing a series of events that are dependent on a user-specified event. King et al. [112] further improved the approach to track dependencies cross different hosts. Goel et al. [96] proposed a technique that recovers from an intrusion based on forensic analysis. Sitaraman et al. [139] proposed a technique that leverages forensic
analysis to detect file system intrusion. Further efforts have been made to mitigate the dependency explosion problem by performing fine-grained causality analysis. Lee et al. [117] proposed a technique to identify the event handling loops in programs and enable selective logging for unit boundaries and unit dependencies through instrumentation. Ma et al. [126] alternated between logging and taint tracking to derive more accurate dependencies. Kwon et al. [115] further leveraged context-sensitive causal models to improve the analysis. Liu et al. [121] proposed to learn a reference model of normal activities to prioritize the tracking of abnormal activities.

In addition to causality analysis, there also exist approaches that focus on other aspects of forensic analysis. Hassan et al. [102] proposed to rely on the contextual and historical information of generated threat alerts to reduce the false alarms of threat detection. Pasquier et al. [134] proposed a runtime analysis technique of provenance by combining runtime kernel-layer reference monitor with a query module mechanism. Milajerdi et al. [129] proposed to rely on the correlation of suspicious information flows to detect ongoing attack campaigns.

A major limitation of these approaches is that they fail to provide an efficient backend solution for managing the massive system monitoring data and performing forensic analysis of complex attack behaviors while incorporating the domain knowledge of experts. Our AiQL system (described in Chapter 3) and SaQL system (described in Chapter 4) address this limitation by (1) providing scalable backend solutions for managing the data stored in either databases (for historical data) or streams (for data collected in real time), and (2) providing domain-specific query languages for querying a wide range of attack behaviors from the data while incorporating the domain knowledge of experts. Furthermore, though efforts have been made to mitigate the dependency explosion problem of causality analysis approaches, the generated dependency graphs are still quite large (usually 100,000+ edges) and these approaches require non-trivial efforts of the security analyst to manually inspect the
large graphs for identifying the paths that are related to the attack. Our SysRep system (described in Chapter 5) addresses this limitation by automating the attack investigation process via (1) propagating reputation from seed sources (much fewer than the number of edges) on the weighted dependency graph to infer the reputation of POI (point of interest) entities, and (2) leveraging discriminative edge weights to automatically reconstruct the attack sequence.

2.1.2 Reduction of System Monitoring Data

To mitigate the burden of storing large amount of system monitoring data, Lee et al. [118] proposed to leverage garbage collection to remove temporary files. Xu et al. [153] proposed to merge dependencies while still preserving high-fidelity causal dependencies. Tang et al. [148] used templates to summarize the set of system libraries loaded at the process initiation period. Hassan et al. [101] investigated how to reduce the storage overheads of provenance graphs generated in distributed systems such as data centers. Our approaches proposed in this thesis (i.e., AiQl, SaQL, SysRep) can be integrated with these systems to achieve better scalability.

2.2 Domain-Specific Languages

2.2.1 Security-Related Languages

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 [16, 53, 49], cryptographic systems [69, 70, 119], secure overlay networks [109, 122], and network intrusions [72, 82, 143, 151] and obfuscations [86]. These languages are explicitly designed to solve domain-specific problems, providing specialized constructs for their particular problem domain and eschewing irrelevant features. In contrast, the domain-specific languages provided by our AiQl and SaQL systems are specially designed
to query complex attack behaviors from system monitoring data, with a focus on enabling effective and efficient forensic analysis of advanced cyber attacks.

2.2.2 System Analysis Languages

Splunk [46] and Elasticsearch [24] are distributed search and analytics engine for application logs, which provide search languages based on keywords and shell-like piping. OSQuery [36, 37] allows analysts to use SQL-like queries to probe the real-time system status. Other languages, such as Weir [73] and StreamIt [149], focus on monitoring the system performance. However, these systems lack efficient supports for joins, which are essential for efficiently querying complex attack behaviors. These systems also lack essential language constructs for expressing advanced anomaly detection models which require the support for stateful computation in sliding windows (described in Section 4.2.2). In contrast, the domain-specific languages provided by our AIQL and SAQL systems address these limitations.

2.3 Database Management Systems and Stream Processing Systems

2.3.1 Database Query Languages

Database query languages are designed for managing data held in various types of databases. SQL [47] is a classic query language designed for manipulating data stored in relational databases [38]. Cypher [17] is a declarative, SQL-like language for describing patterns in graph databases [33]. SPARQL [45] is a semantic query language that is able to retrieve and manipulate data stored in Resource Description Framework (RDF) format [40]. MongoDB [32] is a popular NoSQL database for document search, which supports JSON-like queries. ProQL [108] is a language for querying
tuple-based data provenance. Logic programming languages such as Prolog \cite{74} can also be used as a database query language, which is equivalent to a subset of SQL. There also exist works that introduce temporal expressions to databases \cite{140}, and explore various time-oriented applications \cite{141}. These languages are designed to query general data stored in databases, thus missing opportunities for optimizations based on the domain-specific characteristics of system monitoring data. Furthermore, these languages lack constructs for easily chaining constraints among relations (i.e., database tables; described in Section 3.2.3) and for expressing advanced anomaly detection models which require the support for stateful computation in sliding windows (described in Section 4.2.2). In contrast, the domain-specific languages provided by our AiQL and SaQL systems are specially designed to query complex attack behaviors from system monitoring data, and the query execution engines leverage domain-specific characteristics of the data and the semantics of the query to efficiently schedule the execution.

### 2.3.2 Complex Event Processing Platforms & Data Stream Management Systems

Complex Event Processing (CEP) platforms and data stream management systems, such as Esper \cite{26}, Siddhi \cite{44}, Apache Flink \cite{5}, and Aurora \cite{62}, match continuously incoming events against a pattern. Wukong+S \cite{156} builds a stream querying platform that can query both the stream data and stored data. CQL \cite{95} manages multiple data streams and provide a query language to process the data over the stream. Unlike traditional database management systems where a query is executed on the stored data, Stream and CEP queries are applied on a potentially infinite stream of data, and all data that is not relevant to the query is immediately discarded. These platforms provide their own domain-specific languages that can compose patterns of complex events with the support of sliding windows. These systems are useful in
managing large streams of data. However, these systems do not provide specialized language constructs for specifying stateful anomaly models for the purpose of forensic analysis, as our SAQL system does (described in Section 4.2.2). Nevertheless, these systems can be used as a stream management infrastructure for our approach. In our proposed design, we leverage Siddhi [44] to build the query execution engine of our SAQL system (described in Section 4.2.4).

2.3.3 Stream Computation Systems

Stream computation systems allow users to compute various metrics based on the stream data. These systems include Microsoft StreamInsight [65], MillWheel [63], Naiad [131], and Puma [79]. These systems normally provide a good support for stateless computation (e.g., data aggregation). However, they do not support stateful anomaly models as our SAQL system does, which are far more complex than data aggregation (described in Section 4.2.2).

2.4 Other Aspects of Security Defenses

2.4.1 System Defenses 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 [147], system call patterns based on data flow dependencies [61], and interactions between benign programs and the operating system [116]. Behavioral analytics have also shown promising results for network intrusion [154, 155] and internal threat detection [137]. These works learn models to detect anomalies or predict attacks, but they do not provide mechanisms for users to perform attack investigation. Our AiQL system fills this gap by allowing the security analyst to query historical system monitoring data for investigating the reported anomalies.
2.4.2 Security Anomaly Detection

Anomaly detection techniques have been widely used in detecting malware [103, 147, 61, 116], preventing network intrusion [154, 155, 135], internal threat detection [137], and attack prediction [152]. Rule-based detection techniques characterize normal behaviors of programs through analysis and detect unknown behaviors that have not been observed during the characterization [88, 103]. Outlier-based detection techniques [154, 155, 135] detect unusual system behaviors based on clustering or other machine learning models. Unlike these techniques, which focus on finding effective features and building specific models under different scenarios, our SAQL system provides a unified interface to express a variety of anomaly models based on the domain knowledge of experts.

2.5 Reputation Propagation

The reputation propagation model in our SysRep system (described in [Section 5.3.5]) was inspired by the TrustRank algorithm [99], which was originally designed to separate spam and reputable web pages: it first selects a small set of reputable seed pages, then propagates the trust scores following the link structures using the PageRank algorithm [133], and identifies spam pages as those with low scores. Similar ideas have been applied in security and privacy application scenarios including Sybil detection [75, 97, 90, 120], fake review detection [136], and attribute inference attacks [106]. The model adopted in the SysRep system differs in that it propagates the reputation score in an inheritance fashion rather than a distribution fashion, which avoids the serious degradation of reputation scores after propagating on long dependency paths.
2.6 Threat Model

Our threat model follows the threat model of previous work \cite{111,112,118,117,68,121,153,148,121,102}. We assume that the kernel is trusted, and the system monitoring data collected from kernel space is not tampered with \cite{30,27,22}. Any kernel-level attack that deliberately compromises security auditing systems is beyond the scope of this thesis.
Chapter 3

AIQL: Enabling Efficient Attack Investigation from System Monitoring Data

We propose a novel system, AIQL, that enables efficient and timely attack investigation via querying the historical system monitoring data. We build our system (~50K lines of Java code) on top of existing system-level monitoring tools (i.e., auditd [30], ETW [27], DTrace [22]) for data collection and relational databases (i.e., PostgreSQL [38] and Greenplum [28]) for data storage. 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 three major 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 system monitoring data and the semantics of the query to efficiently schedule the query execution.
Figure 3.1: Major types of attack behaviors (events $e_1, \ldots, e_n$ are shown in ascending temporal order)

Domain-Specific Query Language: Our AIQL language uniquely integrates a series of critical primitives for expressing three major types of attack behaviors:

(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). To investigate such attacks, AIQL provides specialized language constructs to easily specify *multiple system activities* in the form of event patterns based on the \{subject-operation-object\} syntax (e.g., `proc p1 write file f1`), as well as *attribute relationships and temporal relationships among activities*.

For example, in Figure 3.1, the attacker runs `osql.exe` to cause the database `sqlservr.exe` to dump its data into a file `backup1.dmp`. Later (i.e., $e_3$ happens after $e_2$; temporal relationship), a malicious script `sbblv.exe` reads from the dump `backup1.dmp` (i.e., the same dump file in $e_2$ and $e_3$; attribute relationship) and sends the data back to the attacker. Query 3.1 in Section 3.2.3 shows the corresponding AIQL query for expressing this attack behavior.

(2) **Dependency Tracking of Attacks**: dependency analysis is often used to track the causality of data for discovering the “attack entry” (i.e., provenance) [111, 112, 139]. To investigate such attacks, AIQL provides specialized language constructs to easily *chain constraints among activities*. For example, in Figure 3.1, a malicious script `info_stealer` in Host 1 infects Host 2 via *network communications between apache*
and `wget`. Query 3.2 in Section 3.2.3 shows the corresponding AIQL query for expressing this attack behavior.

(3) Abnormal System Behaviors: frequency-based behavioral models are often required to express abnormal system behaviors, such as network access spikes [51, 39]. To investigate such attacks, AIQL provides specialized language constructs to easily specify 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). For example, in Figure 3.1 a malicious script `sbblv.exe` sends a large amount of data to a particular destination `XXX.129`. Query 3.3 in Section 3.2.3 shows the corresponding AIQL query for expressing this attack behavior.

Data Model and Storage: Our AIQL 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 execution of a query.

Query Scheduling: Our AIQL system identifies both spatial and temporal constraints in AIQL queries, and optimizes the query execution in two aspects: (1) for an AIQL query that involves 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 an AIQL query into independent sub-queries along temporal and spatial dimensions and executes these sub-queries in parallel.

Deployment and Evaluation: We deployed the AIQL system in NEC Labs America comprising 150 hosts. We performed a wide range 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 (collected for 16 days;
containing 2.5 billion events): (1) our end-to-end efficiency evaluation on an APT attack case study (27 queries) shows that AiQL surpasses both PostgreSQL (124x) and Neo4j (157x); (2) our performance evaluation shows that the query scheduling employed by AiQL is efficient in both single-node databases (40x over PostgreSQL scheduling) and parallel processing databases (16x over Greenplum scheduling); (3) our conciseness evaluation on four major types of attack behaviors (19 queries) shows 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. In addition to the conciseness and efficiency, our evaluations also demonstrate the efficacy of AiQL in investigating a wide range of attacks.

The AiQL system proposed in this thesis was published and presented at The 2018 USENIX Annual Technical Conference (ATC) \[94\]. Furthermore, we made a demo video \[20\] for showcasing the key functionalities of our AiQL system, and wrote a demonstration proposal \[93\]. All the queries in our case study (Section 3.3.2), performance evaluation (Section 3.3.3), and conciseness evaluation (Section 3.3.4) are available on our project website \[2\]. To the best of our knowledge, AiQL is the first work that facilitates effective and efficient attack investigation via querying system monitoring data.

This chapter is organized as follows. In Section 3.1 we present an overview of the AiQL system. In Section 3.2 we present the design details of the data collection, data storage, domain-specific query language, and query execution engine. In Section 3.3 we present an extensive evaluation of the effectiveness of AiQL in terms of query performance and query conciseness. We discuss the aspects of future improvement in Section 3.4 and summarize this chapter in Section 3.5.
3.1 Overview

Figure 3.2 shows the architecture of the AIQL system: (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 (Section 3.2.1). The collected system monitoring data from each host is then sent to the central server and stored in our optimized data storage (Section 3.2.2); (2) the language parser, implemented using ANTLR 4 [4], analyzes the input queries and generates query contexts. A query context is an object abstraction of an input query that contains all the required information for the query execution. Multievent syntax, dependency syntax, and anomaly syntax are supported (Section 3.2.3); (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 (Section 3.2.4).

3.2 Design of The AIQL System

3.2.1 Data Collection

System monitoring data records the interactions among system resources as system events [111]. 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 \cite{111, 112, 96, 107}. 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 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.

We develop data collection agents based on mature system monitoring frameworks: auditd \cite{30} for Linux, ETW \cite{27} for Windows, and DTrace \cite{22} for MacOS. Our agents are deployed across servers, desktops, and laptops in the enterprise and collect critical security-related attributes for system entities (Table 3.1) and system events (Table 3.2).

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). For a file entity, we use volume id and data id (including inode id and file generation number\textsuperscript{1}) as its unique identifier. For a process entity, we use process id, process starting time, and an ordinal number\textsuperscript{2} as its unique identifier. For a network connection entity, as processes usually communicate with some servers using different network connections but with the same IPs and ports, treating these connections differently greatly increases the amount of data we trace and such granularity is not required in most of the cases. Thus, we use 5-tuple \((\langle srcip, srcport, dstip, dstport, protocol \rangle)\) as a network connection’s unique identifier. Failing to distinguish different entities causes problems in relating events.

\textsuperscript{1}Could be obtained via \texttt{ioctl} in Linux.

\textsuperscript{2}Processes that start at the same time are distinguished using this number.
Table 3.1: Representative attributes of system entities

<table>
<thead>
<tr>
<th>Entity</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>Name, Owner/Group, VolID, DataID, etc.</td>
</tr>
<tr>
<td>Process</td>
<td>PID, Name, User, Cmd, Binary Signature, etc.</td>
</tr>
<tr>
<td>Network Connection</td>
<td>IP, Port, Protocol</td>
</tr>
</tbody>
</table>

Table 3.2: Representative attributes of system events

<table>
<thead>
<tr>
<th>Operation</th>
<th>Read/Write, Execute, Start/End, Rename/Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time/Sequence</td>
<td>Start Time/End Time, Event Sequence</td>
</tr>
<tr>
<td>Misc.</td>
<td>Subject ID, Object ID, Failure Code</td>
</tr>
</tbody>
</table>

to entities, especially for tracking dependencies of events, which makes the system ineffective in querying precise information.

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). In particular, agent ID refers to the unique ID of the host where the entity/event is observed.

3.2.2 Data Storage

After collecting the data from the agents, we store the collected data in relational databases powered by PostgreSQL [38]. Compared to graph databases [33] and other NoSQL databases [32, 45], relational databases come with mature indexing mechanisms and thus 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 databases for efficient data I/O.

Date Deduplication and In-Memory Indexes: We observe that in the collected events, an entity often appears in multiple events. Therefore, we perform data deduplication by storing entities and events in separate tables and keeping only the unique identifiers of entities in each event. Furthermore, we maintain in-memory indexes for the already-seen entities, and keep only the unique identifiers to minimize the memory
space required by the indexes. When a new event comes, we only need to query the in-memory indexes for the entity identifiers if the entities were seen previously. Note that without such indexes, we will have to query the database to see whether the entities exist, which is prohibitively expensive due to the massive amount of entities. We also leverage RocksDB [41] as an extension for the indexes in case the memory cannot hold the them.

**Batch Commit:** Relational databases have poor write performance for large tables, which become worse when the data volume grows linearly in time. To address the problem, our server buffers the collected system events sent by the agents and writes them to the database in batches. Based on our experiments, HDD with 4 concurrent read/write can achieve 20-30K events per second while SSD can reach about 60K. We are currently looking for a multi-tier storage solution that combines both SSD (for recent days’ hot data) and HDD (for long-term cold data).

**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: we separate groups of agents into table partitions and generate 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 [28]. These databases intelligently distribute the storage and the search of events and entities based on the spatial and temporal properties of our data model.
Queries in MPP databases leverage the distributed structure and parallelism to speed up the search.

**Time Synchronization:** We correct potential time drifting of events on different agents from two aspects. At the agent (client) side, we apply synchronization protocols like Network Time Protocol (NTP) [34]. NTP can achieve better than one millisecond accuracy in local area networks [7], and the typical accuracy on the Internet ranges from about 5ms to 100ms. At the backend server side, a central clock is referenced to further check and minimize the time differences across agents' events. This two-step synchronization can construct proper happen-before relationships of events in practice, which suffices for typical security analysis. This is because, first, most security queries in attack investigations focus on events on a single host, where the total temporal order is locally enforced by our agents (using sequence numbers to represent the order of the system calls arriving at the kernel). Second, for cross-host investigations, events on different hosts (e.g., network events) are usually separated by a time interval that is larger than our synchronization granularity. Therefore, our AIQL system can construct proper happen-before relationships of events in practice.

### 3.2.3 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 [17, 45, 17, 46] that focus on the specification of relation joins or graph paths, AIQL uniquely integrates a series of 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 3.1 shows the representative rules of AIQL.
tribute relationships, which facilitate the specification of multi-step attack behaviors.

Grammar 3.1: Representative BNF grammar of AiQL

Multievent AIQL Query

AiQL provides explicit language constructs for system events (in a natural format of \{subject-operation-object\}), spatial/temporal constraints, and event temporal/attribute relationships, which facilitate the specification of multi-step attack behaviors.
Query 3.1: Data exfiltration from database server

Query 3.1 shows a multievent A1Ql query that investigates the data exfiltration from database server: the attacker leverages OSQL utility (osql.exe) to dump the database content (backup1.dmp) and then 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.dstip).

Global Constraints: The global constraint rule (⟨global_cstr⟩) specifies the constraints for all event patterns (e.g., agentid and time window in Query 3.1).

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 (i.e., file, proc, ip), 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.
Event Attribute and Temporal Relationships: The event relationship rule ($\langle \text{evt}_\text{rel} \rangle$) specifies how multiple event patterns are related. The attribute relationship rule ($\langle \text{attr}_\text{rel} \rangle$) uses attribute values of event patterns to specify their relationships. For example, in Query 3.1 the use of $f_1$ in both $\text{evt}_2$ and $\text{evt}_3$ specifies an attribute relationship between the two events: $\text{evt}_2$ and $\text{evt}_3$ are linked by the same object file entity. The temporal relationship rule ($\langle \text{temporal}_\text{rel} \rangle$) specifies temporal order (“before”, “after”, “within”) of event patterns. For example, $\text{evt}_1 \text{ before}[1-2 \text{ minutes}] \text{ evt}_2$ specifies that $\text{evt}_1$ occurs 1 to 2 minutes before $\text{evt}_2$.

Event Return and Filters: The event return rule ($\langle \text{return} \rangle$) retrieves the attributes of the matched events. Constructs such as “count”, “distinct”, and “top” are provided to manipulate the return results. “Having’ can be used to filter the returned attributes based on boolean conditions. “Sort by” can be used to sort the return rows.

Context-Aware Syntax Shortcuts: AiQL employs various language syntax shortcuts to ease the query specification process.

- Attribute inference: (1) default attribute names will be inferred if the user specifies only attribute values in an event pattern, or specifies 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 dstip for networks; (2) id will be used as the default attribute if the user specifies 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 the same entity ID in multiple event patterns implicitly means that these event patterns share the same entity.

For example, in Query 3.1 $p_1["cmd.exe"], f_1["backup1.dmp"], i_1["XXX.129"], \text{and return distinct } p_1, p_2, p_3, f_1, p_4, i_1$ will be inferred as $p_1[\text{exe_name} = \text{"cmd.exe"}], f_1[\text{name} =$
Dependency AIQL Query

AIQL provides explicit language constructs for chaining constraints among system events in the form of event path, which facilitates the specification of dependency tracking of attacks. Nodes in the path represent subject/object entities, and edges in the path represent operations (specified by the \( \langle \text{op\_edge} \rangle \) rule). The direction of the operation is indicated by the arrow (\( \rightarrow \)) on the edge, which points from the subject entity to the object entity (e.g., proc p1 \( \rightarrow \text{[write]} \) file f1). The forward and backward keywords can be used to specify the temporal order of the events on the path: forward means that the events found by the leftmost event pattern occur earliest, and backward means that the events found by the leftmost event pattern occur latest.

Query 3.2: Forward tracking for malware ramification

Query 3.2 shows a forward dependency AIQL query that investigates the ramification of a malware (\textit{info\_stealer}), which originates from Host 1 (\textit{agentid = 1}) and affects Host 2 (\textit{agentid = 2}) through an Apache web server. Lines 2-3 specify that p1 writes to f1, and then p2 reads from f1. Such syntax eliminates the need to repetitively specify the shared entity (i.e., f1) in each event pattern. An example execution result may show that p3 is the \texttt{wget} process that downloads the malicious script from Host 2. The \texttt{forward} keyword (Line 2) specifies the temporal order of the events: left event occurs earlier. The operation \texttt{connect} (Line 4) indicates that the dependency tracking is across different hosts.
Anomaly AIQL Query

AIQL provides explicit language constructs for sliding windows and a variety of aggregation functions (e.g., `count`, `avg`, and `sum`) to facilitate the specification of frequency-based system behavioral models. Furthermore, AIQL provides explicit language constructs for accesses to historical aggregate results, allowing queries to compare frequencies using historical information.

```sql
1 (at "mm/dd/yyyy") // 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 3.3:** Simple moving average for network frequency

Query 3.3 shows an anomaly AIQL query that specifies a 1-minute sliding window (Line 3) and computes a three-period simple moving average [83] (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.

AIQL supports common types of moving averages through built-in functions (SMA, CMA, WMA, EWMA [83]). For example, the computation of EWMA for network frequency with normalized deviation can be expressed as: \((freq - EWMA(freq, 0.9)) / EWMA(freq, 0.9) > 0.2\).

### 3.2.4 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 [19]. 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.

**Query Execution Pipeline**

Figure 3.3 shows the query execution pipeline. Given a AIQL query, the parser performs syntactic analysis and semantic analysis to generate a query execution context, which is an object abstraction of the input query that contains all required information for query execution. For an input multievent query, the engine executes it through the pipeline as shown in Figure 3.4. For an input dependency query, the engine compiles it to an equivalent multievent query via query rewriting.

For an input anomaly query, the engine maintains the aggregate results as historical states and performs the filtering based on the historical states. Various types of errors during the parsing and the execution will be reported.
Execution of a Multievent Query: Figure 3.4 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 (Section 3.2.2) for the matched events. The data query scheduler prioritizes the execution of data queries to optimize execution performance. Execution results of each data query are further processed by the execution engine 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 query with many constraints mixed together. Such strategy suffers from indeterministic optimizations due to the large number of constraints and often causes the query execution to last for minutes or even hours (Section 3.3.2).

Data Query Synthesis: As we store the collected data in relational databases due to the mature indexing support for efficient joins, Aiql employs a *template-based* synthesis mechanism [144] to synthesize a SQL data query for every event pattern. Query 3.4 shows the SQL query synthesis template, which consists of major clauses of SQL and a set of holes to be filled with the information from the query execution context generated by the parser.

```sql
1 SELECT |return|
2 FROM |subject_type| |subject_id|, |object_type| |object_id|, |event_type| |event_id|
3 WHERE |entity_join|
4 AND |event_op|
```
Based on this template, Aiql system synthesizes a SQL query via: (1) filling in the “SELECT” clause based on return and filters; (2) filling in the “FROM” clause based on event type; (3) filling in the “WHERE” clause based on global constraints, attribute relationships, and temporal relationships. Query 3.5 shows a synthesized SQL query of the second event pattern (evt2) in Query 3.1. With the query context abstraction and the template-based synthesis, Aiql can be easily extended to synthesize data queries in other languages, such as Cypher [17].

```
SELECT p3.exe_name
FROM process p3, file f1, fileevent evt2
WHERE (evt2.src_id = p3.id AND evt2.dst_id = f1.id)
AND evt2.optype = "write"
AND evt2.agentid = xxx /* obfuscated */
AND evt2.start_time >= xxx /*epoch timestamp*/
AND evt2.end_time <= xxx
AND p3.exe_name ilike "%sqlservr.exe"
AND f1.name ilike "%backup1.dmp"
```

**Query 3.5:** Synthesized SQL query for evt2 in Query 3.1

**Query Rewriting for Dependency Query:** The engine compiles an input dependency query to a semantically equivalent multievent query by traversing its event path and listing the event patterns successively. In this way, dependency queries can also take advantage of the domain-specific optimizations employed by the data query scheduler in the multievent query execution pipeline (Figure 3.4).

**Data Query Scheduler**

The data query scheduler in Figure 3.4 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 rela-
tionships 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** shows 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 (relationships with higher sum of scores are sorted
   in the front).

3. The main loop processes event relationships returned from the sorted list, ex-
   ecutes data queries, and generates result tuples. The engine executes the data
   query whose associated event pattern has a higher pruning score first, and lever-
   ages 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.
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: $R = \{\text{rel}(e_i, e_j)\}$

Output: Event ID tuples that satisfy all constraints

1. $\forall e_i \in E$, score($e_i$) $\leftarrow$ compute $e_i$;
2. $R_{\text{sorted}} \leftarrow \text{sort} R$;
3. Initialize empty set $\text{Exec}$, empty map $M$;
for $\text{rel}(e_i, e_j)$ in $R_{\text{sorted}}$ do
   if $e_i$ not in $\text{Exec}$ and $e_j$ not in $\text{Exec}$ then
      // Suppose score($e_i$) $\geq$ score($e_j$)
      $S_i \leftarrow \text{execute } q_i$; $\text{Exec.add}(e_i)$; // $S_i$: event ID set
      $S_j \leftarrow \text{execute } q_j$; $\text{Exec.add}(e_j)$;
      $T \leftarrow S_i \times S_j |_{\text{rel}(e_i, e_j)}$; // create tuple set from $S_i$ and $S_j$, then filter by
      $\text{rel}(e_i, e_j)$
      $M.put(e_i, T)$; $M.put(e_j, T)$;
   else if Either of $\{e_i, e_j\}$ in $\text{Exec}$ then
      // Suppose $e_i$ in $\text{Exec}$
      $S_j \leftarrow \text{execute } q_j$; $\text{Exec.add}(e_j)$;
      $T \leftarrow M.get(e_i)$; $T' \leftarrow T \times S_j |_{\text{rel}(e_i, e_j)}$; // update tuple set using $S_j$ and
      $\text{rel}(e_i, e_j)$
      $\text{replaceVals}(M, T, T')$; $M.put(e_j, T')$;
   else
      $T_i \leftarrow M.get(e_i)$; $T_j \leftarrow M.get(e_j)$;
      if $T_i = T_j$ then
         $T' \leftarrow T_i |_{\text{rel}(e_i, e_j)}$; // filter tuple set
         $\text{replaceVals}(M, T_i, T')$;
      else
         $T' \leftarrow T_i \times T_j |_{\text{rel}(e_i, e_j)}$; // merge tuple sets
         $\text{replaceVals}(M, T_i, T')$; $\text{replaceVals}(M, T_j, T')$;
   end
4. for $e_i \in E$ and $e_i$ not in $\text{Exec}$ do
   $S_i \leftarrow \text{execute } q_i$; $\text{Exec.add}(e_i)$; $M.put(e_i, S_i)$;
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
   $\text{replaceVals}(M, T, T')$; $\text{replaceVals}(M, T_j, T')$;
6. Return unique($M.values()$);

Function $\text{replaceVals} (M, T, T')$
Choose all values $T'$ stored in $M$ with $T'$;

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.
Our empirical results (Section 3.3.3) 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.

**Error Reporting and Recovery**

The error reporting module in Figure 3.2 monitors and reports different types of errors during execution.

- **Syntax error:** Errors that occur during parsing the input (i.e., syntactic analysis in Figure 3.3) that does not conform to the AIQL grammar (Grammar 3.1).

- **Semantic error:** Errors that occur during interpreting the parse tree (i.e., semantic analysis in Figure 3.3). For example, *ReservedIDError* refers to the use of reserved keywords as entity IDs.

- **Runtime error:** Errors that occur during the execution of the synthesized data queries. For example, errors reported by the underlying databases will be surfaced and reported.

AIQL supports basic error recovery mechanism for syntax errors. For example, if the input query contains an unrecognized character, such character will be automatically removed and the execution will continue.
3.2.5 Web UI

To showcase the key functionalities of the AIQL system, we built a web UI (Figure 3.5) upon Apache Tomcat [6], and made a demo video [20]. Our web UI consists of (1) an input box for entering AIQL queries, (2) an execution output area to show the synthesized SQL queries and 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 effective and efficient attack investigation: (1) syntax highlighting for query construction, (2) syntax checking for query debugging, and (3) sorting and searching for query results management.
3.3 Evaluation

We deployed the AiQL system in NEC Labs America comprising 150 hosts (10 servers, 140 employee stations). We performed a variety of attacks based on known exploits in the deployed environment and constructed 46 AiQL queries in total 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 [38], Neo4j [33], and Splunk [46]. We also evaluate the efficiency offered by our data query scheduler (Section 3.2.4) in both storage settings: PostgreSQL and Greenplum. In total, our evaluation uses 857GB of real system monitoring data (16 days; 2.5 billion events).

3.3.1 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, Neo4j Cypher, and Splunk SPL queries. We measure the execution time and the conciseness of each query. For evaluation fairness, we clean the database cache every time before executing every 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 [46] makes it inappropriate for investigating multi-step attack behaviors. Due to the limited expressiveness of SQL and Neo4j Cypher, we cannot compare the anomaly queries (e.g., Query 3.6). The 19 AiQL queries used in our performance
evaluation (Section 3.3.3) and conciseness evaluation (Section 3.3.4) are available in Appendix A.2. The 27 AiQL queries used in our case study (Section 3.3.2), as well as all SQL, Neo4j Cypher, and Splunk SPL queries used in our evaluation are available on our project website [2].

3.3.2 Case Study: APT Attack Investigation

We conduct a case study by asking a white hat hacker to perform an APT attack in the deployed environment, as shown in Figure 3.6. Below are the attack steps:

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

\textit{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 [9]) to open the backdoor to the attacker.

\textit{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.

\textit{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 to the attacker host.

Anomaly Detectors: We deployed two anomaly detectors based on existing solutions [77, 114, 145]. 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 creates 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.

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 construct 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 [113, 142]. Thus, we first construct an anomaly AiQL query that computes a moving average (SMA3) to find processes which transfer a large amount of data to this suspicious IP.

```
1 (at "mm/dd/yyyy") // date (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 3.6: AiQL anomaly query for large file transfer
Query 3.6 finishes execution within 4 seconds and identifies a suspicious process “sbblv.exe”. Note that we are unable to easily find such anomalous process using general database queries like SQL or Neo4j Cypher, since they lack support for sliding window and history state comparison. We then construct a multievent AiQL query to find the data sources for this process (Query 3.7).

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

**Query 3.7:** Starter AiQL query for $c_5$

By inspecting the execution results of Query 3.7, we identify a suspicious file “backup1.dmp” for $f_1$ 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 a “osql.exe” process is started by “cmd.exe” (OSQL utility is often involved in many SQL database attacks). Query 3.8 gives the complete query for investigating the step $c_5$.

```sql
1 (at "mm/dd/yyyy")
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
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 3.8:** Complete AiQL query for $c_5$
Step c4: The investigation for the step $c_4$ is similar to $c_5$, which includes iterative query execution and editing. Query 3.9 gives the complete AIQL query for the step $c_4$. As we can see, the attack steps involved are that the attacker first penetrates into the database server from the Windows client. Then, the attacker drops a malicious VBScript “hwvun.vbs” and executes it, which creates a malicious executable “sbblv.exe”. The executable is then executed and finally creates a reverse connection back to the attacker host.

```
1 (at "mm/dd/yyyy")
2 agentid = xxx // SQL database server (obfuscated)
3 proc p1["sqlservr.exe"] start || read || write ip i1["XXX.130"] as evt1
4 proc p1 start proc p2["cmd.exe"] as evt2
5 proc p2 write file f1["hwvun.vbs"] as evt3
6 proc p3["cscript.exe"] read file f1 as evt4
7 proc p3 write file f2["sbblv.exe"] as evt5
8 proc p3 start proc p4["sbblv.exe"] as evt6
9 proc p4 start ip i2["XXX.129"] as evt7
10 with evt1 before evt2, evt2 before evt3, evt3 before evt4, evt4 before evt5, evt5 before evt6, evt6 before evt7
11 return distinct p1, i1, p2, f1, p3, f2, p4, i2
```

Query 3.9: Complete AIQL query for $c_4$

Step c2: We investigate the alerts reported by the other anomaly detector that we deployed on the Windows client, and identify that the “excel.exe” process starts an unexpected child process “java.exe”. From this alert, we further investigate how the “java.exe” process is created and what other child processes that the “java.exe” process further creates. Query 3.10 gives the complete AIQL query for the step $c_2$. As we can see, the attacker first sends a crafted e-mail to the Windows client victim. The victim uses Outlook client to receive the e-mail and opens the malicious attachment with Excel, which creates a malware “java.exe” and executes it. The malware then leverages Notepad to create a reverse connection back to the attacker host, inviting the attacker to the intranet.

```
1 (at "mm/dd/yyyy")
```
Query 3.10: Complete AIQL query for c2

Step c3: In order to penetrate into the database server from the Windows client in the step c2, the attacker needs to obtain the database server’s IP address and the database administrator credentials, which are typically stored in the Windows domain controller. Since there are many ways for escalating privilege, without any detection alerts, we are unable to investigate further. We asked the white hat hacker who did the penetration and learned that he used a specific database cracking tool called “gsecdump.exe”. Based on this information, we finally constructed the complete AIQL for the step c3, as shown in Query 3.11.

```plaintext
agentid = xxx // Windows client
proc p1["%outlook.exe"] start proc p2["%excel.exe"] as evt1
proc p2 write || execute file f1["%java.exe"] as evt2
proc p2 start proc p3["%java.exe"] as evt3
proc p3 start proc p4["%notepad.exe"] as evt4
proc p4 start || read || write ip i1["XXX.129"] as evt5
with evt1 before evt2, evt2 before evt3, evt3 before evt4, evt4 before evt5
return distinct p1, p2, f1, p3, p4, i1
```

Query 3.11: Complete AIQL query for c3

Step c1: The behavior for the step c1 is rather normal, and is buried among other benign behaviors in sending and receiving emails. Query 3.12 gives a complete AIQL query to verify that the victim client receives an e-mail from the attacker host.

```plaintext
agentid = xxx // Windows domain controller
proc p1 read || write ip i1[srcport=445, dstip="XXX.134"] as evt1 // DC penetration using psexec protocol
proc p2["%powershell.exe"] write file f1["%gsec%"] as evt2 // transfer DB cracking tool gsecdump.exe
proc p3["%cmd.exe"] start proc p4["%gsec%"] as evt3 // dump DB administrator credentials
with evt1 before evt2, evt2 before evt3
return distinct p1, i1, p2, f1, p3, p4
```

Query 3.12: Complete AIQL query for c1
In total, we constructed 26 multievent queries and 1 anomaly query to successfully investigate the APT attack, touching 119GB of data/422 million events. Please refer to our project website [2] for the queries as well as the investigation details.

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:** Figure 3.7 shows the execution time of AIQL queries, SQL queries in PostgreSQL, and Cypher queries in Neo4j. For evaluation,
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. Table 3.3 shows the 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 \( \sim 5.9 \) hours for PostgreSQL and \( \sim 7.5 \) 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).

Note that for fair comparison, our evaluation was done by cleaning the database cache before every execution of a query. In the real-life investigation without cleaning cache, the AiqL system can achieve further performance speedup by leveraging the data locality, since adjacent investigation queries often preserve certain level of semantic similarity (e.g., the latter query is constructed from the former query by adding another event pattern).

**Conciseness:** The largest AiqL query is c4-8 (Query A.20 in Appendix A.3.1) with 7 event patterns, 25 query constraints, 109 words, and 463 characters (excluding spaces). The corresponding SQL query (Query A.21 in Appendix A.3.1) contains 77

---

3Fine-grained evaluations of the AiqL scheduling are in Section 3.3.3
Table 3.3: Aggregate statistics for case study

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

constraints (3.1x larger), 432 words (4.0x larger), and 2792 characters (6.0x larger).

The corresponding Neo4j Cypher query (Query A.22 in Appendix A.3.1) 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 Neo4j Cypher queries become verbose with many joins and constraints, posing challenges for constructing the queries for timely attack investigation.

3.3.3 Performance Evaluation

We evaluate the performance of Aiql in both storage settings (PostgreSQL and Greenplum) by constructing 19 Aiql queries (Appendix A.2) for a wide range of attack behaviors, touching 738GB/2.1 billion events. Particularly, we are interested in the efficiency speedup provided by the Aiql scheduling (Section 3.2.4) in comparison with the PostgreSQL scheduling and the Greenplum scheduling.

Attack Behaviors

Multi-Step Attack Behaviors: We asked a white hat hacker to launch another APT attack using a different set of exploits (Appendix A.1). We followed a similar investigation procedure as in Section 3.3.2 and constructed 5 complete Aiql queries to investigate the five 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).
<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>7dd95111e9e100b6243ca96b9b322120</td>
<td>Trojan.Sysbot</td>
</tr>
<tr>
<td>v2</td>
<td>425327783e88bb6492753849bc43b7a0</td>
<td>Trojan.Hooker</td>
</tr>
<tr>
<td>v3</td>
<td>ee11901739531d6963ab1ec3ecaf280</td>
<td>Virus.Autorun</td>
</tr>
<tr>
<td>v4</td>
<td>4e720458c357310da684018f4a254dd0</td>
<td>Virus.Sysbot</td>
</tr>
<tr>
<td>v5</td>
<td>7dd95111e9e100b6243ca96b9b322120</td>
<td>Trojan.Hooker</td>
</tr>
</tbody>
</table>

**Table 3.4**: Selected malware samples from Virussign

**Real-World Malware Behaviors:** We obtained a dataset of free malware samples from VirusSign [56]. We then randomly selected 5 malware samples (Table 3.4) 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 [56] (v1-v5).

**Abnormal System Behaviors:** We evaluated 6 abnormal system behaviors based on the knowledge of security experts: (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, Neo4j Cypher, or Splunk SPL queries, due to their lack of support for sliding window and history state comparison.

**Efficiency in PostgreSQL**

We evaluate the efficiency speedup provided by the AIQL scheduling (Section 3.2.4) in single-node databases. We select two baselines: (1) PostgreSQL databases that employ our data storage optimizations (Section 3.2.2). Note that this setting is different from the end-to-end efficiency evaluation in Section 3.3.2 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; Section 3.2.4). We measure the execution time of the 19 queries in Appendix A.2.

**Evaluation Results:** Figure 3.8 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 \(a_2\), \(a_4\), and \(d_2\) within 1 hour; (2) the scheduling employed by \(\text{AIQL}_\text{FF}\) and \(\text{AIQL}\) is more efficient than PostgreSQL, with 19x and 40x speedup, respectively; (3) the relationship-based scheduling employed by \(\text{AIQL}\) is more efficient than the fetch-and-filter scheduling employed by \(\text{AIQL}_\text{FF}\).

Efficiency in Parallel Databases

We evaluate the efficiency speedup provided by the \(\text{AIQL}\) scheduling \(\text{(Section 3.2.4)}\) in parallel databases. We compare the performance of \(\text{AIQL}\) scheduling in the Greenplum storage with the Greenplum scheduling (i.e., running SQLs). The Greenplum databases also employ our data storage optimizations \(\text{(Section 3.2.2)}\).
Figure 3.9: Query execution time of the scheduling employed by Greenplum and AiQL (parallel)

Evaluation Results: Figure 3.9 shows the execution time of queries in Greenplum and AiQL. We observe that: (1) in most cases, the AiQL scheduling in parallel settings achieves a comparable performance as Greenplum scheduling; (2) in certain cases (e.g., a4, d3), the AiQL 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.
Figure 3.10: Conciseness evaluation of queries written in AIQL, SQL, Neo4j Cypher, and Splunk SPL
### 3.3.4 Conciseness Evaluation

We evaluate the conciseness of queries that express the 19 attack behaviors in Section 3.3.3 (the corresponding AiQL queries are in Appendix A.2) in three metrics: the number of query constraints, the number of words, and the number of characters (excluding spaces).

**Evaluation Results:** Figure 3.10 shows the conciseness metrics of AiQL, SQL, Neo4j Cypher, and Splunk SPL queries. Table 3.5 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, Neo4j Cypher, and Splunk SPL that contain 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. For example, in Appendix A.3.2, Queries A.23 to A.25 show the AiQL, SQL, and Splunk SPL queries for $d3$. As we can see, AiQL queries are significantly more concise than SQL queries and Splunk SPL queries.

### 3.4 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 \[125, 126\] and in-memory data manipulations \[85, 98\]. To handle the increase of data size, we plan to explore more aggressive data reduction techniques in addition to existing solutions \[118, 153\] to make the system more scalable.

### 3.5 Summary

We proposed AiQL, a novel system for collecting attack provenance using system monitoring and assisting effective and efficient attack investigation. Compared with existing systems, AiQL greatly reduces the cycle time for iterative and interactive investigation of complex attacks.
Chapter 4

SAQL: A Stream-Based Query System for Real-Time Abnormal System Behavior Detection

We propose a novel system, SAQL, that enables real-time abnormal system behavior detection via querying the stream of system monitoring data. SAQL is a stream-based query system that takes as input a real-time event feed aggregated from multiple hosts in an enterprise, and provides an anomaly query engine that queries the event feed to identify abnormal system behaviors based on the specified anomaly models. We built our system (∼50K lines of Java code) based on the existing system-level monitoring tools (i.e., auditd [30], ETW [27], DTrace [22]) for data collection and the existing stream management system (i.e., Siddhi [44]) for data stream management.

Domain-Specific Query Language: To facilitate the specification of various types of anomaly models that incorporate the domain knowledge of experts, our system provides a domain-specific language, Stream-Based Anomaly Query Language (SAQL). Our SAQL language uniquely integrates a series of critical primitives for expressing major types of anomaly models. In particular, SAQL provides (1) the syntax
of event patterns to ease the task of specifying relevant system activities and their relationships, which facilitates the specification of rule-based anomaly models; (2) the constructs for sliding windows and stateful computation that allow stateful anomaly models to be computed in each sliding window over the data stream, which facilitates the specification of time-series anomaly models, invariant-based anomaly models, and outlier-based anomaly models. The specified models in SAQL are checked using continuous queries over unbounded stream of system monitoring data [95], which report the detected anomalies continuously.

Rule-based anomaly models allow system experts to specify behavior rules to detect known attack behaviors or enforce enterprise-wide security policies. Figure 4.1 shows an example rule-based anomaly, where a process (cat) accesses multiple command log files in a relatively short time period, indicating that an external user is trying to probe the useful commands issued by the legitimate users. To express such behavior, SAQL uses event patterns to express each activity in the format of \{subject-operation-object\} (e.g., proc p1 write file f1), where system entities are represented as subjects (proc p1) and objects (file f1), and interactions are represented as operations initiated by subjects and targeted on objects.

Stateful computation in sliding windows over a data stream enables the specification of stateful behavior models for detecting abnormal system behaviors such as time-series anomalies, which lack support from existing stream query systems that fo-
cus on general data streams \[156, 104, 63, 80\]. Figure 4.1 shows a time-series anomaly, where a process (\texttt{sqlservr.exe}) transfers abnormally large amount of data starting from \(e_2\). To facilitate the detection of such anomalies, SAQL provides constructs for \textit{sliding windows} that break the continuous data stream into fragments with common aggregation functions (\textit{e.g.}, \texttt{count}, \texttt{sum}, \texttt{avg}). Additionally, SAQL provides constructs to define \textit{states in sliding windows} and allow accesses to the states of past windows. These constructs facilitate the comparison with historical states and the computation of moving averages such as three-period simple moving average (SMA) \[83\].

Built upon the states of sliding windows, SAQL provides high-level constructs to facilitate the specification of invariant-based anomaly models and outlier-based anomaly models. Invariant-based anomaly models capture the invariants during training periods as models, and use the models later to detect anomalies. Figure 4.1 shows an invariant-based anomaly, where a process (\texttt{apache.exe}) starts an abnormal process (\texttt{java.exe}) that is unseen during the training period. SAQL provides constructs to define and learn the invariants of system behaviors in each state computed from a window, which allow the user to combine both states of windows and invariants learned under normal operations to detect more types of abnormal system behaviors.

Outlier-based anomaly models allow the user to identify abnormal system behaviors through peer comparison, \textit{e.g.}, finding outlier processes by comparing the abnormal processes with other peer processes. Figure 4.1 shows an outlier-based anomaly, where a process (\texttt{sqlservr.exe}) transfers abnormally larger amount of data to an IP address than other IP addresses. SAQL provides constructs to define which information of a state in a sliding window forms the points of comparison and compute clusters to identify outliers. The flexibility and extensibility introduced by SAQL allows users to use various clustering algorithms for different deployed environments.

\textbf{Query Execution Engine:} We build the query engine on top of Siddhi \[44\] to leverage its mature stream management engine. Based on the input SAQL queries, our
system synthesizes Siddhi queries to match data from the stream, and performs stateful computation and anomaly model construction to detect anomalies over the stream. One major challenge faced by this design is the scalability in handling multiple concurrent anomaly queries over the large-scale system monitoring data. Typically, different queries may access different attributes of the data using different sliding windows. To accommodate these needs, the scheme employed by the existing systems, such as Siddhi, Esper, and Flink [44, 26, 5], is to make copies of the stream data and feed the copies to each query, allowing each query to operate separately. However, such scheme is not efficient in handling the big data collected from system monitoring.

To address this challenge, we devise a master-dependent-query scheme that identifies compatible queries and groups them to use a single copy of the stream data to minimize the data copies. Our system first analyzes the submitted queries with respect to the temporal dimension in terms of their sliding windows and the spatial dimension in terms of host machines and event attributes. Based on the analysis results, our system puts the compatible queries into groups, where in each group, a master query will directly access the stream data and the other dependent queries will leverage the intermediate execution results of the master query. Note that such optimization leverages both the characteristics of the spatio-temporal properties of system monitoring data and the semantics of SAQL queries, which would not be possible for the queries in general stream-based query systems [44, 26, 95, 5].

**Deployment and Evaluation:** We deployed the system in NEC Labs America comprising 150 hosts. We performed a wide range of attack behaviors in the deployed environment, and evaluated the system using 1.1TB of real system monitoring data (containing 3.3 billion events): (1) our case study on four major types of attack behaviors (17 SAQL queries) shows that our SAQL system has a low alert detection latency (<2s); (2) our pressure test shows that our SAQL system has a high system throughput (110000 events/s) for a single representative rule-based query that monitors file
accesses, and can scale to \(\sim 4000\) hosts on the deployed server; (3) our performance evaluation using 64 micro-benchmark queries shows that our SAQL system is able to efficiently handle concurrent query execution and achieves more efficient memory utilization compared to Siddhi, achieving 30\% average saving.

The SAQL system proposed in this thesis was published and presented at The 27th USENIX Security Symposium [91]. Furthermore, we made a demo video [21] for showcasing the key functionalities of our SAQL system, and wrote a demonstration proposal [92]. All 17 SAQL queries used in the case study (Section 4.3.2) are available in Appendix B.1. All 64 micro-benchmark queries used in the performance evaluation (Section 4.3.4) are available on our project website [42]. To the best of our knowledge, SAQL is the first work that facilitates real-time abnormal system behavior detection via querying the stream of system monitoring data.

This chapter is organized as follows. In Section 4.1, we present an overview of the SAQL system. In Section 4.2, we present the design details of the data collection, domain-specific query language, and query execution engine. We also present a few representative SAQL queries for showcasing the expressiveness of SAQL. In Section 4.3, we present an extensive evaluation of the effectiveness of SAQL in a variety of metrics. We discuss the aspects of future improvement in Section 4.4, and summarize this chapter in Section 4.5.

4.1 Overview

Figure 4.2 shows the architecture of the SAQL system. We deploy monitoring agents across servers, desktops, and laptops in the enterprise to monitor system-level activities by collecting information about system calls from kernels. System monitoring data for Windows, Linux, and Mac OS are collected via ETW event tracing [27],
Linux Audit Framework [30], and DTrace [22]. The collected data is sent to the central server, forming an event stream.

The SAQL system takes SAQL queries from users, and reports the detection alerts over the event stream. The system consists of two components: (1) the language parser, implemented using ANTLR 4 [4], performs syntactic and semantic analysis of the input queries and generates an anomaly model context for each query. An anomaly model context is an object abstraction of the input query that contains all the required information for query execution and anomaly detection; (2) the execution engine, built upon Siddhi [44], monitors the data stream and reports the detection alerts based on the execution of the anomaly model contexts.

The execution engine has four sub-modules: (1) the multievent matcher matches the events in the stream against the event patterns specified in the query; (2) the state maintainer maintains the states of each sliding window computed from the matched events; (3) the concurrent query scheduler divides the concurrent queries into groups based on the master-dependent-query scheme [Section 4.2.4] to minimize the need for data copies; (4) the error reporter reports errors during the query execution.
4.2 Design of The SAQL System

4.2.1 Data Collection

System monitoring data represents various system activities in the form of events along with time [111, 112, 96, 107]. Each event can naturally be described as a system entity (subject) performing some operation on another system entity (object). For example, a process reads a file or a process accesses a network connection. An APT attack needs multiple steps to succeed, such as target discovery and data exfiltration, as illustrated in the cyber kill chain [15]. Therefore, multiple attack footprints might be left as “dots”, which can be captured precisely by system monitoring.

System monitoring data records system audit events about the system calls that are crucial in security analysis [111, 112, 96, 107]. The monitored system calls are mapped to three major types of system events: (1) process creation and destruction, (2) file access, and (3) network access. Existing work has shown that on main-stream operating systems (Windows, Linux and OS X), system entities in most cases are files, processes, and network connections [111, 112, 96, 107]. Thus, in this work, we consider system entities as files, processes, and network connections in our data model. We define an interaction among entities as an event, which is represented using the triple \( \langle \text{subject}, \text{operation}, \text{object} \rangle \), which consists of the initiator of the interaction, the type 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 events into three types according to the type of their object entities, namely file events, process events, and network connection events.

We develop data collection agents based on mature system monitoring frameworks: auditd [30] for Linux, ETW [27] for Windows, and DTrace [22] for MacOS. Our agents are deployed across servers, desktops, and laptops in the enterprise and collect
critical security-related attributes for system entities (Table 4.1) and system events (Table 4.2).

The attributes of an entity include the properties to describe the entities (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 an event 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). In particular, agent ID refers to the unique ID of the host where the entity/event is collected.

### 4.2.2 Query Language Design

**SAQL** is designed to facilitate the task of expressing a variety of anomaly models by incorporating the domain knowledge of security experts. **SAQL** provides explicit constructs to specify system entities/events, as well as event relationships. This facilitates the specification of rule-based anomaly models to detect known attack behaviors or enforce enterprise-wide security policies. **SAQL** also provides constructs for sliding windows and stateful computation that allow stateful anomaly models to be computed in each sliding window over the data stream. This facilitates the specification of time-series anomaly models, invariant-based anomaly models, and outlier-based anomaly models, which lack support from existing stream query systems.
and stream-based anomaly detection systems. Grammar 4.1 shows the representative rules of SAQL. We omit the terminal symbols.

**Multievent Pattern Matching**

SAQL provides the event pattern syntax (in the format of \{subject-operation-object\}) to describe system activities, where system entities are represented as subjects and objects, and interactions are represented as operations initiated by subjects and targeted on objects. Besides, the syntax directly supports the specification of event temporal relationships and event attribute relationships, which facilitates the specification of complex system behavioral rules.

**Global Constraint:** The \(\text{\langle global\_cstr\rangle}\) rule specifies the constraints for all event patterns (e.g., \text{agentid = 1} in Query 4.5 specifies that all event patterns occur on the same host).

**Event Pattern:** The \(\text{\langle evt\_patt\rangle}\) rule specifies an event pattern, including the subject/object entity (\(\text{\langle entity\rangle}\)), the event operation (\(\text{\langle op\_exp\rangle}\)), the event ID (\(\text{\langle evt\rangle}\)), and the optional sliding window (\(\text{\langle wind\rangle}\)). The \(\text{\langle entity\rangle}\) rule consists of the entity type (file, process, network connection), the optional entity ID, and the optional attribute constraints expression (\(\text{\langle attr\_exp\rangle}\)). Logical operators (\&\&, ||, !) can be used in \(\text{\langle op\_exp\rangle}\) to form complex operation expressions (e.g., \text{proc p read || write file f}).

The \(\text{\langle attr\_exp\rangle}\) rule specifies an attribute expression which supports the use of the logical operators, the comparison operators (=, ! =, >, >=, <, <=), the arithmetic operators (+, -, *, /), the aggregation functions, and the stateful computation-related operators (e.g., \text{proc p[pid = 1 && name = "chrome.exe"]}).

**Sliding Window:** The \(\text{\langle wind\rangle}\) rule specifies the sliding windows for stateful computation. For example, \text{%time(10 min)} in Query 4.3 specifies a sliding window whose width is 10 minutes. An optional step size can be provided (e.g., \text{%time(10 min)(1 min)} indicates a step size of 1 minute).
Grammar 4.1: Representative BNF grammar of SAQL
**Event Temporal Relationship:** The \( \langle \text{temp\_rel} \rangle \) rule specifies the temporal dependencies among event patterns. For example, \( \text{evt1} \rightarrow \text{evt2} \rightarrow \text{evt3} \) in Query 4.2 specifies that \( \text{evt1} \) occurs first, then \( \text{evt2} \), and finally \( \text{evt3} \). Finer-grained control of temporal distance can also be provided. For example, \( \text{evt1} \rightarrow [1-2 \text{ min}] \text{evt2} \rightarrow [1-2 \text{ min}] \text{evt3} \) indicates that the time span between the two events is 1 to 2 minutes.

**Event Attribute Relationship:** Event attribute relationships can be included in the alert rule (\( \langle \text{alert} \rangle \)) to specify the attribute dependency of event patterns (e.g., \( \text{alert} \text{evt1}.\text{agentid} = \text{evt2}.\text{agentid} \& \& \text{evt1}.\text{dst\_id} = \text{evt2}.\text{src\_id} \) for two event patterns \( \text{evt1} \) and \( \text{evt2} \) indicates that the two events occur at the same host and are “physically connected”: the object entity of \( \text{evt1} \) is exactly the subject entity of \( \text{evt2} \)).

**Context-Aware Syntax Shortcuts:** SAQL employs a variety of language syntax shortcuts to ease the query specification process.

- **Attribute inferences:** (1) default attribute names will be inferred if only attribute values are specified in an event pattern, or only entity IDs are specified in event return. We select the most commonly used attributes in security analysis as default attributes: \( \text{name} \) for files, \( \text{exe\_name} \) for processes, and \( \text{dstip} \) for network connections. For example, in Query 4.2, \( \text{file f1["%gsecdump.exe"]} \) is equivalent to \( \text{file f1[name="%gsecdump.exe"]} \), and \( \text{return p1} \) is equivalent to \( \text{return p1.exe\_name} \); (2) \( \text{id} \) will be used as default attribute if only entity IDs are specified in the alert condition. For example, given two processes \( p1 \) and \( p2 \), \( \text{alert p1 = p2} \) is equivalent to \( \text{alert p1.id = p2.id} \).

- **Optional ID:** the ID of entity/event can be omitted if it is not referenced in event relationships or event return. For example, in \( \text{proc p open file} \), we can omit the file entity ID if we will not reference its attributes later.

- **Entity ID Reuse:** Reused entity IDs in multiple event patterns implicitly indicate the same entity.
Stateful Computation

Based on the constructs of sliding windows, SAQL provides constructs for stateful computation, which consists of two major parts: defining states based on sliding windows and accessing states of current and past windows to specify time-series anomaly models, invariant-based anomaly models, and outlier-based anomaly models.

**State Block:** The \( \langle \text{state-def} \rangle \) rule specifies a state block by specifying the state count, block ID, and multiple state fields. The state count indicates the number of states for the previous sliding windows to be stored (\( e.g., \) Line 2 in [Query 4.3]). If not specified, only the state of the current window is stored by default (\( e.g., \) Line 2 in [Query 4.4]). The \( \langle \text{state-field} \rangle \) rule specifies the computation that needs to be performed over the data in the sliding window, and associates the computed value with a variable ID. SAQL supports a broad set of numerical aggregation functions (\( e.g., \) sum, avg, count, median, percentile, stddev, etc.) and set aggregation functions (\( e.g., \) set, multiset).

**State Invariant:** The \( \langle \text{state-inv} \rangle \) rule specifies invariants of system behaviors and updates these invariants using states computed from sliding windows (\( i.e., \) invariant training), so that users can combine both states of windows and invariants learned to detect more types of abnormal system behaviors. For example, Lines 5-8 in [Query 4.4] specifies an invariant \( a \) and trains it using the first 10 window results.

**State Cluster:** The \( \langle \text{state-cluster} \rangle \) rule specifies clusters of system behaviors, so that users can identify abnormal behaviors through peer comparison. The cluster specification requires the specification of the points of comparison using peer reference keywords \( \langle \text{peer-ref} \rangle \) (\( e.g., \) all), distance metric, and clustering method. SAQL supports common distance metrics (\( e.g., \) Manhattan distance, Euclidean distance) and major clustering algorithms (\( e.g., \) K-means \[100\], DBSCAN \[87\], and hierarchical clustering \[100\]). For example, Line 6 in [Query 4.5] specifies a cluster of the one-dimensional points \( ss.amt \) using Euclidean distance and DBSCAN algorithm. SAQL
also provides language extensibility that allows other clustering algorithms and metrics to be used through mechanisms such as Java Native Interface (JNI) and Java Naming and Directory Interface (JNDI).

### Alert Condition Checking

The \textit{(alert)} rule specifies the condition (a boolean expression) for triggering the alert. This enables SAQL to specify a broad set of detection logics for time-series anomaly models (\textit{e.g.}, Line 5 in \texttt{Query 4.3}), invariant-based anomaly models (\textit{e.g.}, Line 9 in \texttt{Query 4.4}), and outlier-based anomaly models (\textit{e.g.}, Line 7 in \texttt{Query 4.5}). Note that in addition to the moving average detection logic specified in \texttt{Query 4.3}, the flexibility of SAQL also enables the specification of other well-known logics, such as 3-sigma rule \cite{76} (\textit{e.g.}, alert \texttt{ss.amt}>3*stddev(all(ss.amt))+avg(all(ss.amt))) and IQR rule \cite{76} (\textit{e.g.}, alert \texttt{ss.amt}>1.5*iqr(all(ss.amt))+q3(all(ss.amt))).

### Return and Filters

The \textit{(report)} rule specifies the desired attributes of the qualified events to return as results. Constructs such as \texttt{group by}, \texttt{sort by}, and \texttt{top} can be used for further results manipulation and filtering. These constructs are useful for querying the most active processes and IP addresses, as well as specifying threshold-based anomaly models without explicitly defining states. For example, \texttt{Query 4.1} computes the IP frequency of each process in a 1-minute sliding window and returns the active processes with a frequency greater than 100.

1. proc p start ip i as evt #time(1 min)
2. group by p
3. alert freq > 100
4. return p, count(i) as freq

\textbf{Query 4.1:} Threshold-based IP Frequency Anomaly
4.2.3 Example SAQL Queries

We present several representative SAQL queries that specify major types of anomaly models. The point is not to assess the quality of these models, but to provide examples of language constructs that are essential in specifying anomaly models, which lack good support from existing query tools.

**Rule-Based Anomaly Model:** Advanced cyber attacks typically include a series of steps that exploit vulnerabilities across multiple systems for stealing sensitive information [3, 1]. Query 4.2 shows a SAQL query for describing an attack step that reads external network (\(\text{evt1}\)), downloads a database cracking tool \(\text{gsecdump.exe}\) (\(\text{evt2}\)), and executes (\(\text{evt3}\)) it to obtain database credentials. It also specifies that these events should occur in an ascending temporal order (Line 4).

```
1 proc p1 read || write ip i1[src_ip != "internal_address"] as evt1
2 proc p2["%powershell.exe"] write file f1["%gsecdump.exe"] as evt2
3 proc p3["%cmd.exe"] start proc p4["%gsecdump.exe"] as evt3
4 with evt1 -> evt2 -> evt3
5 return p1, i1, p2, f1, p3, p4 // p1 -> p1.exe_name, i1 -> i1.dstip, f1 -> f1.name
```

**Query 4.2:** A rule-based SAQL query

**Time-Series Anomaly Model:** SAQL query provides the constructs of sliding windows to enable the specification of time-series anomaly models. For example, a SAQL query may monitor the amount of data sent out by certain processes and detect unexpectedly large amount of data transferred within a short period. This type of query can detect network spikes [51, 55], which often indicates a data exfiltration. Query 4.3 shows a SAQL query that monitors the network usage of each application and raises an alert when the network usage is abnormally high. It specifies a 10-minute sliding window (Line 1), collects the amount of data sent through network within each window (Lines 2-4), and computes a moving average to detect the spikes of network data transfers (Line 5). In the query, \(\text{ss}[0]\) represents the state of the current window while \(\text{ss}[1]\) and \(\text{ss}[2]\) represent the states of the two past windows respectively (\(\text{ss}[2]\) occurs
earlier than \(ss[1]\)). Existing stream query systems and anomaly systems \([95, 104, 63]\) lack the expressiveness of stateful computation in sliding windows to support such anomaly models.

```plaintext
1 proc p write ip i as evt #time(10 min)
2 state[3] ss {
3   avg_amount := avg(evt.amount)
4 } group by p
5 alert (ss[0].avg_amount > (ss[0].avg_amount + ss[1].avg_amount + ss[2].avg_amount) / 3) &&
   (ss[0].avg_amount > 10000)
6 return p, ss[0].avg_amount, ss[1].avg_amount, ss[2].avg_amount
```

Query 4.3: A time-series SAQL query

**Invariant-Based Anomaly Model:** Invariant-based anomaly models capture the invariants during training periods as models, and use the models later to detect anomalies. To achieve invariant-based anomaly detection, SAQL provides constructs of invariant models and learning specifics to define and learn invariants of system behaviors, which allows users to combine both stateful computation and invariants learned under normal operations to detect more types of abnormal system behaviors \([71]\). Query 4.4 shows a SAQL query that specifies a 10-second sliding window (Line 1), maintains a set of child processes spawned by the Apache process (Lines 2-4), uses the first ten time windows for training the model (Lines 5-8), and starts to detect abnormal child processes spawned by the Apache process (Line 10). The model specified in the Lines 5-8 represents the set of names of the processes forked by the Apache process in the training stage. During the online detection phase, this query generates alerts when a process with a new name is forked by the Apache process. General stream query systems without the support of stateful computation and invariant models cannot express such types of anomaly models. Note that the invariant definition allows multiple aggregates to be defined.

```plaintext
1 proc p1["%apache.exe"] start proc p2 as evt #time(10 s)
2 state ss {
3   set_proc := set(p2.exe_name)
```
Outlier-Based Anomaly Model: Outlier-based anomaly models allow users to identify abnormal system behavior through peer comparison, e.g., finding outlier processes by comparing the abnormal processes with other peer processes. To detect outlier-based anomalies, SAQL provides constructs of outlier models to define which information in a time window forms a multidimensional point and compute clusters to identify outliers. Query 4.4 shows a SAQL query that (1) specifies a 10-minute sliding window (Line 2), (2) computes the amount of data sent through network by the `sqlservr.exe` process for each outgoing IP address (Lines 3-5), and (3) identifies the outliers using DBSCAN clustering (Lines 6-8) to detect the suspicious IP that triggers the database dump. Note that Line 6 specifies which information of the state forms the points of comparison and how the “distance” among these points should be computed (e.g., “ed” representing Euclidean Distance). These language constructs enable SAQL to express models for peer comparison, which has limited support from the existing querying systems where only simple aggregations such as max/min are supported [95, 44, 26].

```sql
agentid = 1 // sqlserver host
proc p["sqlservr.exe"] read || write ip as evt #time(10 min)
state ss {
    amt := sum(evt.amount)
} group by i.dstip
cluster(points=all(ss.amt), distance="ed", method="DBSCAN(100000, 5)")
alert cluster.outlier && ss.amt > 100000
return i.dstip, ss.amt
```
Query 4.5: An outlier-based SAQL query using clustering

In addition to querying outliers through clustering, SAQL also supports querying outliers through aggregation comparison. For example, in Query 4.5 replacing the `alert` statement with `alert ss.amt>1.5*iqr(all(ss.amt))+q3(all(ss.amt))` gives interquartile range (IQR)-based outlier detection [76], and replacing the `alert` statement with `alert ss.amt>3*stddev(all(ss.amt))+avg(all(ss.amt))` gives 3-sigma-based outlier detection [76]. SAQL also supports querying outliers through sorting, and reports the top sorted results as alerts, which is useful in querying the most active processes or IP addresses.

4.2.4 Query Execution Engine

The SAQL execution engine in Figure 4.2 takes the event stream as input, executes the anomaly model contexts generated by the parser, and reports the detection alerts. To make the system more scalable in supporting the execution of multiple concurrent queries, the SAQL execution engine employs a master-dependent-query scheme that groups semantically compatible queries to share a single copy of the stream data for query execution. In this way, the SAQL system significantly reduces the unnecessary data copies of the stream.

Query Execution Pipeline

We built the SAQL query engine upon Siddhi [44] to leverage its mature stream management engine in terms of event model, stream processing, and stream query. Given a SAQL query, the parser performs syntactic analysis and semantic analysis to generate an anomaly model context. The concurrent query scheduler inside the query optimizer analyzes the newly arrived anomaly model context against the existing anomaly model contexts of the queries that are currently running, and computes
an optimized execution schedule by using the master-dependent-query scheme. The multievent solver analyzes event patterns and their dependencies in the SAQL query, and retrieves the matched events by issuing a Siddhi query to access the data from the stream. If the query involves stateful computation, the state maintainer leverages the intermediate execution results to compute and maintain query states. Alerts will be generated if the alert conditions are met for the submitted query.

**Concurrent Query Scheduler**

The concurrent query scheduler in Figure 4.2 schedules the execution of concurrent queries. A straightforward scheduling strategy is to make copies of the stream data and feed the copies to each query, allowing each query to operate separately. However, system monitoring produces a huge amount of daily logs [118, 153], and such copy scheme incurs high memory usage, which greatly limits the scalability of the system.

**Master-Dependent-Query Scheme:** To efficiently support concurrent query execution, the concurrent query scheduler adopts a *master-dependent-query scheme*. In the scheme, only master queries have direct access to the data stream, and the execution of the dependent queries depends on the execution of their master queries. Given that the execution pipeline of a query typically involves four phases (*i.e.*, event pattern matching, stateful computation, alert condition checking, and attributes return), the key idea is to maintain a map $M$ from a master query to its dependent queries, and *let the execution of dependent queries share the intermediate execution results of their master query in certain phases*, so that unnecessary data copies of the stream can be significantly reduced.

[Algorithm 2](#) shows the scheduling algorithm:

1. The scheme first checks if $M$ is empty (*i.e.*, no concurrent running queries). If so, the scheme sets $newQ$ as a master query, stores it in $M$, and executes it.
Algorithm 2: Master-dependent-query scheme

Input: User submitted new SAQL query: \( \text{newQ} \)
Map of concurrent master-dependent queries: \( M = \{ \text{masQ}_i \rightarrow \{ \text{depQ}_{ij} \} \} \)

Output: Execution results of \( \text{newQ} \)

if \( M.\text{isEmpty} \) then
    return \( \text{execAsMas(} \text{newQ}, M \text{)} \);
else
    for \( \text{masQ}_i \) in \( M.\text{keys} \) do
        \( \text{covQ} = \text{constructSemanticCover}(\text{masQ}_i, \text{newQ}); \)
        if \( \text{covQ} \neq \text{null} \) then
            if \( \text{covQ} \neq \text{masQ}_i \) then
                \( \text{replMas(} \text{masQ}_i, \text{covQ}, M \text{)} ; \)
                \( \text{addDep(} \text{covQ}, \text{newQ}) ; \)
        return \( \text{execDep(} \text{newQ}, \text{covQ}) ; \)
        return \( \text{execAsMas(} \text{newQ}, M \text{)} ; \)

Function \( \text{constructSemanticCover}(\text{masQ}, \text{newQ}) \)
if Both \( \text{masQ} \) and \( \text{newQ} \) define a single event pattern then
    if \( \text{masQ} \) and \( \text{newQ} \) share the same event type, operation type, and sliding window type then
        Construct the event pattern cover \( \text{evtPattCovQ} \) by taking the union of their attributes and agent IDs and the GCD of their window lengths;
    if Both \( \text{masQ} \) and \( \text{depQ} \) define states then
        if \( \text{masQ} \) and \( \text{depQ} \) have the same sliding window length and \( \text{masQ} \)
        defines a super set of state fields of \( \text{depQ} \) then
            Construct the state cover \( \text{stateCovQ} \) by taking the union of their state fields;
        return \( \text{covQ} \) by concatenating \( \text{evtPattCovQ} \), \( \text{stateCovQ} \), and the rest parts of \( \text{masQ} \);
    return \( \text{null} \);

Function \( \text{execAsMas(} \text{newQ}, M \text{)} \)
Make \( \text{newQ} \) as a new master and execute it;

Function \( \text{addDep(} \text{masQ}, \text{depQ}, M \text{)} \)
Add \( \text{depQ} \) to the dependencies of \( \text{masQ} \);

Function \( \text{replMas(} \text{oldMasQ}, \text{newMasQ}, M \text{)} \)
Replace the old master \( \text{oldMasQ} \) with the new master \( \text{newMasQ} \) and update dependencies;

Function \( \text{execDep(} \text{depQ}, \text{masQ}) \)
if \( \text{depQ} == \text{masQ} \) then
    return execution results of \( \text{masQ} \);
else if Both \( \text{masQ} \) and \( \text{depQ} \) define states then
    if \( \text{masQ} \) and \( \text{depQ} \) have the same sliding window length and \( \text{masQ} \)
    defines a super set of state fields of \( \text{depQ} \) then
        Fetch the state aggregation results of \( \text{masQ} \), enforce additional filters, and feed into the execution pipeline of \( \text{depQ} \);
    else
        Fetch the matched events of \( \text{masQ} \), enforce additional filters, and feed into the execution pipeline of \( \text{depQ} \);
2. If $M$ is not empty, the scheme checks $newQ$ against every master query $masQ_i$ for compatibility and tries to construct a semantic cover $covQ$. If the construction is successful, the scheme then checks whether $covQ$ equals $masQ_i$.

3. If $covQ$ is different from $masQ_i$, the scheme updates the master query by replacing $masQ_i$ with $covQ$ and updates all the dependent queries of $masQ_i$ to $covQ$.

4. The scheme then adds $newQ$ as a new dependent query of $covQ$, and executes $newQ$ based on $covQ$.

5. Finally, if there are no master queries found to be compatible with $newQ$, the scheme sets $newQ$ as a new master query, stores it in $M$, and executes it.

Two key steps in Algorithm 2 are `constructSemanticCover()` and `execDep()`. The construction of a semantic cover requires that (1) the $masQ$ and the $depQ$ both define a single event pattern, and (2) their event types, operation types, and sliding window types must be the same\(^1\). The scheme then explores the following four optimization dimensions: event attributes, agent ID, sliding window, and state aggregation. Specifically, the scheme first constructs an event pattern cover by taking the union of the two queries’ event attributes and agent IDs, and taking the greatest common divisor (GCD) of the window lengths. It then constructs a state block cover by taking the union of the two queries’ state fields (if applicable), and returns the semantic cover by concatenating the event pattern cover, the state block cover, and the rest parts of $masQ$.

The execution of $depQ$ depends on the execution of $masQ$. If two queries are the same, the engine directly uses the execution results of $masQ$ as the execution results of $depQ$. Otherwise, the engine fetches the intermediate results from the execution pipeline of $masQ$ based on the level of compatibility. The scheme currently enforces

\(^1\)We leave the support for multiple event patterns for future work.
the results sharing in two execution phases: event pattern matching and stateful computation: (1) if both \textit{dep} and \textit{masQ} define states and their sliding window lengths are the same, the engine fetches the state aggregate results of \textit{masQ}; (2) otherwise, the engine fetches the matched events of \textit{masQ} without its further state aggregate results. The engine then enforces additional filters and feeds the filtered results into the rest of the execution pipeline of \textit{depQ} for further execution.

### 4.3 Evaluation

We deployed the \textit{Saqql} system in NEC Labs America comprising 150 hosts (10 servers, 140 employee stations; generating around 3750 events/s). To evaluate the expressiveness of \textit{Saqql} and the \textit{Saqql}’s overall effectiveness and efficiency, we first perform a series of attacks based on known exploits in the deployed environment and construct 17 \textit{Saqql} queries (available in Appendix B.1) to detect them. Next, we conduct a pressure test to measure the maximum performance that our system can achieve. Finally, we conduct a performance evaluation on 64 micro-benchmark queries (available on our project website [42]) to evaluate the effectiveness of our query engine in handling concurrent queries. In total, our evaluations use 1.1TB of real system monitoring data (containing 3.3 billion system events).

#### 4.3.1 Evaluation Setup

The evaluations are conducted on a server with an Intel(R) Xeon(R) CPU E1650 (2.20GHz, 12 cores) and 128GB of RAM. The server continuously receives a stream of system monitoring data collected from the hosts deployed with the data collection agents. We developed a web-based client for query submission and deployed the \textit{Saqql} system on the server for query execution. To easily reproduce the attack scenarios for the performance evaluation in Section 4.3.4, we additionally stored the collected
data in databases and developed a stream replayer to replay the system monitoring data from the databases. Our stream replayer has a web-based UI (Figure 4.3) that lets the user choose the hosts and the start/end time to replay the specific part of system monitoring data.

4.3.2 Case Study of Four Major Types of Attacks

We performed four major types of attack behaviors in the deployed environment based on known exploits: (1) APT attack [3, 1], (2) SQL injection attack [81], (3) Bash shellshock command injection attack [11], and (4) suspicious system behaviors.
Attack Behaviors

**APT Attack:** We ask a white hat hacker to perform an APT attack in the deployed environment, as shown in [Figure 4.4](#). 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 client and runs the macro, which downloads and executes a malicious script (CVE-2008-0081 [9]) to open a backdoor for 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 steal the database credentials.

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

- **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 to his host.

For each attack step, we construct a rule-based anomaly query (i.e., Queries B.1 to B.5 in Appendix B.1). Besides, we construct 3 advanced anomaly queries:

- We construct an invariant-based anomaly query (Query B.6 in Appendix B.1) to detect the scenario where Excel executes a malicious script that it has never executed before: The invariant contains all unique processes started by Excel in the first 100 sliding windows. During the detection phase, new processes that deviate from the invariant will be reported as alerts. This query can be used to detect the unseen suspicious Java process started by Excel (i.e., step c2).

- We construct a time-series anomaly query (Query B.7 in Appendix B.1) based on simple moving average (SMA) to detect the scenario where abnormally high
volumes of data are exchanged via network on the database server (i.e., step c5): For every process on the database server, this query detects the processes that transfer abnormally high volumes of data to the network. This query can be used to detect the large amount of data transferred from the database server.

- We also construct an outlier-based anomaly query (Query B.8 in Appendix B.1) to detect processes that transfer high volumes of data through the network (i.e., step c5): The query detects such processes through peer comparison based on DBSCAN clustering algorithm. The detection logic here is different from Query B.7 which detects anomalies through comparison with the historical states based on SMA.

Note that the construction of these 3 queries assumes no knowledge of the detailed attack steps.

**SQL Injection Attack:** We conduct a SQL injection attack [?] for a typical web application server configuration. The setup has multiple web application servers that accept incoming web traffics to load balance. Each of these web servers connects to a single database server to authenticate users and serves dynamic contents. However, these web applications provide limited input sanitization and thus are susceptible to SQL injection attack.

We use SQLMap [48] to automate the attack against one of the web application servers. In the process of detecting and exploiting SQL injection flaws and taking over the database server, the attack generates an excessive amount of network traffic between the web application server and the database server. We construct an outlier-based anomaly query (Query B.9 in Appendix B.1) to detect abnormally large data transfers to external IP addresses.

**Bash Shellshock Command Injection Attack:** We conduct a command injection attack against a system that installs an outdated Bash package susceptible to the
Shellshock vulnerability \[11\]. With a crafted payload, the attacker initiates a HTTP request to the web server and opens a Shell session over the remote host. The behavior that the web server creates a long-running Shell process is an outlier pattern. We construct an invariant-based anomaly query (Query B.10 in Appendix B.1) to learn the invariant of child processes of Apache, and use it to detect any unseen child process (i.e., /bin/bash in this attack).

**Suspicious System Behaviors:** Besides known threats, security analysts often have their own definitions of suspicious system behaviors, such as accessing credential files using unauthorized software and running forbidden software. We construct 7 rule-based queries to detect a representative set of suspicious behaviors:

- Forbidden Dropbox usage (Query B.11 in Appendix B.1): finding the activities of Dropbox processes.
- Command history probing (Query B.12 in Appendix B.1): finding the processes that access multiple command history files in a relatively short period.
- Unauthorized password files accesses (Query B.13 in Appendix B.1): finding the unauthorized processes that access the protected password files.
- Unauthorized login logs accesses (Query B.14 in Appendix B.1): finding the unauthorized processes that access the log files of login activities.
- Unauthorized SSH key files accesses (Query B.15 in Appendix B.1): finding the unauthorized processes that access the SSH key files.
- Forbidden USB drives usage (Query B.16 in Appendix B.1): finding the processes that access the files in the USB drive.
- IP frequency analysis (Query B.17 in Appendix B.1): finding the processes with high frequency network accesses.
Table 4.3: Execution statistics of 17 SAQL queries for four major types of attacks

<table>
<thead>
<tr>
<th>Saql Query</th>
<th>Alert Detection Latency</th>
<th>Num. of States</th>
<th>Tot. State Size</th>
<th>Avg. State Size</th>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>apt-c1</td>
<td>≤1ms</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>10%</td>
<td>1.7GB</td>
</tr>
<tr>
<td>apt-c2</td>
<td>≤1ms</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>10%</td>
<td>1.8GB</td>
</tr>
<tr>
<td>apt-c3</td>
<td>6ms</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>8%</td>
<td>1.6GB</td>
</tr>
<tr>
<td>apt-c4</td>
<td>10ms</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>10%</td>
<td>1.5GB</td>
</tr>
<tr>
<td>apt-c5</td>
<td>3ms</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>10%</td>
<td>1.6GB</td>
</tr>
<tr>
<td>apt-c5-invariant</td>
<td>≤1ms</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>8%</td>
<td>1.8GB</td>
</tr>
<tr>
<td>apt-c5-timeseries</td>
<td>≤1ms</td>
<td>812</td>
<td>3321</td>
<td>4.09</td>
<td>6%</td>
<td>2.2GB</td>
</tr>
<tr>
<td>apt-c5-outlier</td>
<td>2ms</td>
<td>812</td>
<td>3321</td>
<td>4.09</td>
<td>8%</td>
<td>2.2GB</td>
</tr>
<tr>
<td>shellshock</td>
<td>5ms</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>8%</td>
<td>2.7GB</td>
</tr>
<tr>
<td>sql-injection</td>
<td>1776ms</td>
<td>14</td>
<td>13841</td>
<td>988.6</td>
<td>8%</td>
<td>1.9GB</td>
</tr>
<tr>
<td>dropbox</td>
<td>2ms</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>8%</td>
<td>1.2GB</td>
</tr>
<tr>
<td>command-history</td>
<td>≤1ms</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>10%</td>
<td>2.2GB</td>
</tr>
<tr>
<td>password</td>
<td>≤1ms</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>10%</td>
<td>2.7GB</td>
</tr>
<tr>
<td>login-log</td>
<td>≤1ms</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>10%</td>
<td>2.2GB</td>
</tr>
<tr>
<td>sshkey</td>
<td>≤1ms</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>10%</td>
<td>2.1GB</td>
</tr>
<tr>
<td>usb</td>
<td>≤1ms</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>9%</td>
<td>2.1GB</td>
</tr>
<tr>
<td>ipfreq</td>
<td>≤1ms</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>10%</td>
<td>2.1GB</td>
</tr>
</tbody>
</table>

Query Execution Statistics

To demonstrate the effectiveness of the SAQL system in supporting timely anomaly detection, we measure the following performance statistics of the query execution:

- **Alert detection latency**: the difference between the time that the anomaly event gets detected and the time that the anomaly event enters the SAQL engine.
- **Number of states**: the number of sliding windows encountered from the time that the query gets launched to the time that the anomaly event gets detected.
- **Average state size**: the average number of aggregation results per state.

The results are shown in Table 4.3. We observe that: (1) the alert detection latency is low (≤10ms for most queries and <2s for all queries). For *sql-injection*, the latency is a bit larger due to the additional complexity of the specified DBSCAN clustering algorithm in the query; (2) the system is able to efficiently support 150 enterprise hosts, with <10% CPU utilization and <2.7GB memory utilization. Note that this is far from the full processing power of our system on the deployed server, and our system is able to support a lot more hosts (as experimented in Section 4.3.3); (3) the number of states and the average state size vary with a number of factors, such as query running time, data volume, and query attributes (e.g., number of
agents, number of attributes, attribute filtering power). Even though the amount of system monitoring data is huge, a SAQL query often restricts one or several data dimensions by specifying attributes. Thus, the state computation is often maintained in a manageable level.

4.3.3 Pressure Test

We conduct a pressure test of our system by replicating the data stream, while restricting the CPU utilization to certain levels. When we conduct the experiments, we set the maximum Java heap size to be 100GB so that memory will not be a bottleneck. We deploy a query that retrieves all file events as the representative rule-based query, and measure the system throughput to demonstrate the query processing capabilities of our system.

Evaluation Results: Figure 4.5 shows the throughput of the SAQL system under different CPU utilizations. We observe that using a deployed server with 12 cores, the SAQL system achieves a maximum throughput of 110000 events/s. Given that our deployed enterprise environment comprises 150 hosts with 3750 events generated...
### Figure 4.6: Event attributes

<table>
<thead>
<tr>
<th>Memory (GB)</th>
<th>SAQL</th>
<th>Siddhi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of concurrent queries</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) Sensitive file accesses  
(b) Browsers access files  
(c) Processes access networks  
(d) Processes spawn

### Figure 4.7: Sliding window

per second, we can estimate that the SAQL system deployed on this server can support \( \sim 4000 \) hosts. While such promising results demonstrate that our SAQL system deployed on only one server can easily support far more than hundreds of hosts for many organizations, there are other factors that can affect the performance of the system. First, queries that involve temporal dependencies may cause more computation on the query engine, and thus could limit the maximum number of hosts that our SAQL system can support. Second, if multiple queries are running concurrently, multiple copies of the data stream are created to support the query computation, which would significantly compromise the system performance. Our next evaluation demonstrates the impact of concurrent queries and how our master-dependent-query scheme mitigates the problem.
4.3.4 Performance Evaluation of Concurrent Query Execution

To evaluate the effectiveness of our query engine (i.e., master-dependent-query scheme) (Section 4.2.4) in handling concurrent queries, we construct a micro-benchmark that consists of 64 SAQL queries and measure the memory usage during the query execution. We select Siddhi [44], one of the most popular stream processing and complex event processing engines, for baseline comparison.

**Micro-Benchmark Construction:** We construct our micro-benchmark queries by extracting critical attributes from the attacks in Section 4.3.2. In particular, we specify the following four attack categories:

- **Sensitive file accesses:** finding processes that access the files `/etc/passwd`, `.ssh/id_rsa`, `.bash_history`, and `/var/log/wtmp`.

**Figure 4.8:** Agent ID

**Figure 4.9:** State aggregation

---

2Note that there are no existing standards or benchmark cases for evaluating the performance of concurrent queries in the context of abnormal system behavior detection.
• **Browsers access files:** finding files accessed by the processes `chrome`, `firefox`, `iexplore`, and `microsoftedge`.

• **Processes access networks:** finding network accesses of the processes `dropbox`, `sqlservr`, `apache`, and `outlook`.

• **Processes spawn:** finding processes spawn by the processes `/bin/bash`, `/usr/bin/ssh`, `cmd.exe`, and `java`.

We also specify the following four **evaluation categories** for query variations, which correspond to the four **optimization dimensions** in Section 4.2.4:

• **Event attributes:** we vary from 1 attribute to 4 attributes. The attributes are chosen from one of the attack categories. The default is 4 attributes.

• **Sliding window:** we vary from 1 minute to 4 minutes. The default is 1 minute.

• **Agent ID:** we vary from 1 agent to 4 agents. The default is to avoid the agent ID specification (i.e., the query matches all agents).

• **State aggregation:** we vary from 1 aggregation type to 4 aggregation types, which are chosen from the pool `{count, sum, avg, max}`. The default is to avoid the state specification (i.e., no states defined).

We construct 4 queries for each evaluation category and each attack category. In total, we construct 64 queries for the micro-benchmark. For each SAQL query, we construct an equivalent Siddhi query. Note that unlike SAQL which provides explicit constructs for stateful computation, Siddhi as well as other stream-based query systems [44, 26, 95, 5], do not provide the native support for these concepts, making these tools unable to specify advanced anomaly models (i.e., time-series anomaly models, invariant-based anomaly models, outlier-based anomaly models). Thus, for the “state evaluation category”, we only construct Siddhi queries that monitor the same event
pattern without stateful computation. **Query 4.6** shows an example micro-benchmark query for the joint category “sensitive file accesses & state aggregation”.

```plaintext
1 proc p read || write file f["/etc/passwd" || ".ssh/id_rsa" || ".bash_history" || "/var/log/wtmp"] as evt #time(1 min)
2 state ss {
3   e1 := count(evt.id)
4   e2 := sum(evt.amount)
5   e3 := avg(evt.amount)
6   e4 := max(evt.amount)
7 } group by p
8 return p, ss.e1, ss.e2, ss.e3, ss.e4
```

**Query 4.6**: Example micro-benchmark query

**Evaluation Results**: For each evaluation category and each attack category, we vary the number of concurrent queries from 1 to 4 and measure the corresponding memory usage. **Figures 4.6** to **4.9** show the results. We observe that: (1) as the number of concurrent queries increases, the memory usage increases of Siddhi are much higher than the memory usage increases of SAQL in all evaluation settings; (2) when there are multiple concurrent queries in execution, SAQL requires a smaller memory usage than Siddhi in all evaluation settings (30% average saving when there are 4 concurrent queries). Such results indicate that the master-dependent-query scheme employed in our query engine is able to save memory usage by sharing the intermediate execution results among dependent queries. On the contrary, the Siddhi query engine performs data copies, resulting in significantly more memory usage than our query engine. Note that for evaluation fairness, we use the replayer (Section 4.3.1) to replay a large volume of data in a short period of time. Thus, the memory measured in **Figures 4.6** to **4.9** is larger than the memory measured in the case study (Table 4.3), where we use the real-time data streams. Nevertheless, this does not affect the relative improvement of SAQL over Siddhi in terms of memory utilization.
4.4 Discussion

Scalability: The collection of system monitoring data and the execution of SAQL queries can be potentially parallelized with distributed computing. Parallelizing the data collection involves allocating computing resources (i.e., computational nodes) to disjoint sets of enterprise hosts to form sub-streams. Parallelizing the SAQL query execution can be achieved through a query-based manner (i.e., allocating one computing resource for executing a set of queries over the entire stream), a sub-stream-based manner (i.e., allocating one computing resource for executing all compatible queries over a set of sub-streams), or a mixed manner. Nonetheless, the increasing scale of the deployed environment, the increasing number of submitted queries, and the diversity and semantic dependencies among these queries bring significant challenges to parallel processing. Thus, the adaptation of our master-dependent-query scheme to such complicated scenarios is an interesting research direction that requires non-trivial efforts. In this work, however, we do not enable distributed computation in our query execution. Instead, we collect system monitoring data from multiple hosts, model the data as a single holistic event stream, and execute the queries over the stream in a centralized manner. Nevertheless, we build our system on top of Siddhi, which can be easily adapted to a distributed mode by leveraging Apache Storm [59]. Again, we would like to point out that the major focus of our work is to provide a useful interface for investigators to query a wide range of abnormal behaviors from system audit logs, which is orthogonal to the computing paradigms of the underlying stream processing systems.

System Entities and Data Reduction: Our current data model focuses on files, processes, and network connections. In future work, we plan to expand the monitoring scope by including inter-process communications such as pipes in Linux. We also plan to incorporate finer granularity system monitoring, such as execution partition
to record more precise activities of processes \[123, 126\] and in-memory data manipulations \[85, 98\]. Such additional monitoring data certainly adds a lot more pressure to the SAQL system, and thus more research on data reduction, besides the existing works \[118, 153\], should be explored.

**Master-Dependent Query:** Our optimization focuses on the queries that share the pattern matching results and the stateful computation results. More aggressive sharing could include alerts and even results reported by the alerts, which we leave for future work.

**Anomaly Models:** We acknowledge that while SAQL supports major anomaly models used in commonly observed attacks, there are many more anomaly models that are valuable for specialized attacks. Our SAQL now allows easy plugins for different clustering algorithms, and we plan to make the system extensible to support more anomaly models by providing interfaces to interact with the anomaly models written in other languages.

**Alert Fusion:** Recent security research \[132, 84, 150\] has shown promising results in improving detect accuracy using alert fusion that considers multiple alerts. While this is beyond the scope of this work, our SAQL can be extended with the syntax that supports the specifications of the temporal relationships among alerts. More sophisticated relationships would require further design on turning each SAQL query into a module and chaining the modules using various computations.

### 4.5 Summary

We proposed SAQL, a novel stream-based system for real-time abnormal system behavior detection via querying the stream of system monitoring data. Compared with existing stream processing systems, SAQL has a domain-specific query language that
integrates a series of critical primitives for expressing a wide range of anomaly models, and is more efficient in memory utilization.
Chapter 5

SysRep: Towards Automatic Attack Investigation via Weight-Aware Reputation Propagation from System Monitoring

We proposed a novel system, SysRep, that enables automatic attack investigation via a reputation propagation paradigm built upon system monitoring. SysRep first assigns discriminative weights to edges in a dependency graph\(^1\) to construct a *weighted dependency graph* and then propagates reputation scores in a weight-aware manner from seed sources to system entities in the POI events (referred to as *POI entities*).

The key insight of SysRep is leveraging discriminative edge weights to differentiate *critical edges* (e.g., downloading malicious scripts) from the non-critical edges.

\(^1\)Nodes on the dependency graph represent system entities such as files, processes, and network connections, and edges represent system auditing events between the two entities such as file read or write, process creation, and network access. Details are described in [Section 5.1.2](#).
(e.g., irrelevant file copies). Critical edges represent the events that are more important with respect to the POI events for the attack, including the events that execute malicious payloads and the events that are required to correlate the malicious payloads with the POI events. These discriminative edge weights enable SysRep to automatically reconstruct the attack sequence. The reputation propagation paradigm enables SysRep to automatically determine the suspiciousness/trustworthiness of POI entities based on whether the system entities originate from suspicious sources (e.g., USB sticks and suspicious IPs) or trusted sources (e.g., verified binaries and trusted websites). The synergy of attack sequence reconstruction and reputation propagation in SysRep facilitate automatic attack investigation.

**Challenges:** Due to the characteristics of system dependency graph, achieving discriminative weight assignment and reputation propagation for automatic attack investigation faces three major challenges. (1) **Unequal Impacts of Edges:** in a dependency graph produced by causality analysis, critical edges that reveal the attack provenance are often buried in a huge number of non-critical edges [111, 112, 117, 121]. If the reputation propagation treats all edges equally, then it cannot ensure that the reputation scores propagated through the critical edges will have the more impact on the POI entities than those propagated via non-critical edges. As a result, POI entities that are supposed to be correlated with suspicious sources may be incorrectly correlated with irrelevant system entities as their sources. (2) **Diversified Attack Cases:** critical edges in different attack scenarios often present different properties, and thus relying on a single feature, such as outgoing degree, for computing edge weights is often insufficient to oversee a diversified set of attack cases. (3) **Long Dependency Path:** due to the multi-stage nature of APT attacks [3, 1], the dependency path from suspicious sources to POI entities often has many hops. If the reputation propagates in a distributed manner (i.e., a node distributes its reputation along its outgoing
edges in certain propagation) \[133, 75\], the node’s reputation will degrade rapidly in the middle of the path, resulting in little impact on the POI entities.

**Novel Techniques of SysRep:** To address the aforementioned challenges, SysRep employs three novel techniques:

1. **Weighted Reputation Propagation:** to ensure that the reputation propagated through critical edges has more impact on the POI entities than non-critical paths, SysRep first assigns discriminative weights to edges in the dependency graph to represent the importance of the edges with respect to the POI events, and then propagates reputation, proportional to the edge weights, along the paths in the graph.

2. **Discriminative Local Feature Projection:** to model the importance of edges with respect to the POI event, SysRep extracts three discriminative features, instead of relying on a single “golden” feature, from a system auditing event, including (i) **relative data size difference**, which measures the distance between the size of data processed by the system call and the size of POI entity; (ii) **relative time difference**, which measures the distance between the timestamps of the dependency event and the POI event; (iii) **concentration degree**, which measures the ratio of the indegree to the outdegree of the involved system entity, which is particularly useful for smoothing out the impacts of system libraries with no incoming edges and long-running processes with many outgoing edges [Section 5.3.3].

To compute a weight score, instead of adopting a standard classification-based approach (i.e., training a classifier using the three features and outputting a score as the weight), which has limited generalization capability in our problem context (e.g., due to the very limited training data, highly imbalanced classes, etc.), SysRep leverages ideas from dimensionality reduction and discriminant analysis. Rather than computing a projection vector globally, which may result in serious bias due to the large number of edges, SysRep employs a novel *discriminative local feature projection* mechanism based on an extended version of Linear Discriminant Analy-
sis (LDA) \cite{128}. For each node, the mechanism computes a discriminative projection vector locally for all its incoming edges and computes a weight for each incoming edge by projecting its three features. This mechanism helps avoid the undesired impacts of unlinked edges to the node, while ensuring that the weights of critical edges are maximally separated from the weights of non-critical edges (Section 5.3.4);

(3) Reputation Inheritance: to avoid the fast degradation of reputation during propagation, SysRep employs an inheritance fashion opposed to the distribution fashion: when a node propagates its reputation to a receiving node, it considers only the impact it may have, measured by the edge weight, on the receiving node, without distributing its reputation equally among all downstream nodes. This scheme preserves source reputations on a group of highly-related nodes (reflected by weights) even when propagating along long paths (Section 5.3.5).

**Evaluation:** We implemented and deployed SysRep in a server to collect real system monitoring data, and evaluated SysRep in both benign and malicious scenarios. We performed 8 tasks that inject benign and malicious payloads through key system interfaces (e.g., web downloads and shell executions) and 5 APT attacks in the deployed environment, and applied SysRep to perform automatic attack investigation. During our evaluation, the server continues to resume its routine tasks to emulate the real-world deployment where irrelevant system activities and attack activities co-exist. In total, we collected \(~2\) billion auditing events for the attacks. For evaluation purpose, we set the range of the reputation score to be \([0.0, 1.0]\), with 0.0 for suspicious sources and 1.0 for trusted sources.

**Reputation Propagation.** To evaluate the effectiveness of SysRep in reputation propagation, we compare the reputation scores of POI entities against the expected values (0.0 for malicious scenarios and 1.0 for benign scenarios). Our results show that SysRep is able to accurately propagate the reputation scores from trusted and sus-
picious sources to the POI entities (averagely 0.99 for benign scenarios and averagely 0.03 in malicious scenarios).

To demonstrate the effectiveness of SysRep’s discriminative local feature projection, we compare SysRep with three other weight computation approaches: (1) LP-Fixed, which leverages a fixed set of parameters for projection; (2) LP-Global, which groups all edges on the dependency graph into two clusters and derives a projection to separate the edges in different clusters; (3) LP-Global+, which is the same as LP-Global but removes outlier edges. The results show that SysRep on average improves LP-Fixed, LP-Global, and LP-Global+ by 64.60%, 79.22%, 70.36%, respectively, in benign scenarios and malicious scenarios.

We also compare SysRep with the edge priority computation used in the state-of-the-art causality analysis work, PrioTracker [121]. Note that the goal of PrioTracker is to improve the precision of causality analysis, which is orthogonal to our goal of enabling automatic investigation via reputation propagation on the weighted dependency graph. Nevertheless, for fair comparison, we adapt their edge priority computation algorithm to assign weights to edges and apply the same propagation algorithm to propagate reputation scores. The results show that SysRep achieves 57.22% improvement over PrioTracker in benign scenarios and average 87.22% improvement over PrioTracker in malicious scenarios.

*Attack Sequence Reconstruction.* To evaluate the effectiveness of SysRep in attack sequence reconstruction, we choose a range of threshold values to prune edges whose weights are below the threshold, and inspect the remaining edges to measure the possible loss of critical edges. The results show that for all attacks, the reputation scores of critical edges (mostly > 0.9) are well separated from the non-critical edges (mostly < 0.1), demonstrating the effectiveness of our discriminative weights. Additionally, setting threshold values within [0.1, 0.25] can prune more than 90% of the irrelevant edges while preserving the critical edges. To the best of our knowledge,
**Figure 5.1:** Partial dependency graph of a motivating data leakage attack case. Rectangles denote processes, ovals denote files, and parallelograms denote network sockets. Yellow ovals are the files leaked in this attack. Nodes in the attack path have red frames while normal activities and system libraries are in dashed green rectangles. Critical edges are represented with solid red arrows. Non-critical edges are represented with dashed arrows. The complete graph contains 5,817 nodes and 215,380 edges. Note that the reputation of the suspicious source on the top left propagates mainly through the critical edges.

AIQL is the first work that facilitates automatic attack investigation via reputation on weighted system monitoring.

This chapter is organized as follows. In **Section 5.1** we present the background of system monitoring and causality analysis, as well as a motivating example for showcasing the usage of SysRep in automatic investigation. In **Section 5.2** we present an overview of the SysRep system. In **Section 5.3** we present the design details of the three phases of SysRep, including: dependency graph generation, discriminative weight computation, and attack investigation. In **Section 5.4**, we present an extensive evaluation of the effectiveness of SysRep over a wide range of attacks. We discuss the aspects of future improvement in **Section 5.5** and summarize this chapter in **Section 5.6**.
Table 5.1: Representative attributes of system entities

<table>
<thead>
<tr>
<th>Entity</th>
<th>Attributes</th>
<th>Shape in Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>Name, Path</td>
<td>Ellipse</td>
</tr>
<tr>
<td>Process</td>
<td>PID, Name, User, Cmd</td>
<td>Square</td>
</tr>
<tr>
<td>Network Connection</td>
<td>IP, Port, Protocol</td>
<td>Parallelogram</td>
</tr>
</tbody>
</table>

Table 5.2: Representative attributes of system events.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Read/Write, Execute, Start/End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Start Time/End Time, Duration</td>
</tr>
<tr>
<td>Misc.</td>
<td>Subject ID, Object ID, Data Amount, Failure Code</td>
</tr>
</tbody>
</table>

5.1 Background and Motivating Example

5.1.1 System Monitoring

System monitoring collects auditing events about system calls that are crucial in security analysis, describing the interactions among system entities. As shown in previous studies [111, 112, 96, 107, 91, 94, 115, 124, 121, 102], on mainstream operating systems (Windows, Linux, and Mac OS), system entities in most cases are files, processes, and network connections, and the collected system calls are mapped to three major types of system events: (i) file access, (ii) processes creation and destruction, and (iii) network access. As such, we consider system entities as files, processes, and network connections. We consider a system event as the interaction between two system entities represented as \( \langle \text{subject}, \text{operation}, \text{object} \rangle \). Subjects are processes originating from software applications (e.g., Chrome), and objects can be files, processes, and network connections. We categorize system events into three types according to the types of their object entities, namely file events, process events, and network events.

Both entities and events have critical security-related attributes [Tables 5.1 and 5.2]. The attributes of entities include the properties to support various security analyses (e.g., file name, process name, and IP address), and the unique identifiers to distinguish entities (e.g., file path, process name and PID, IP and port). The attributes of events include event origins (e.g., start time/end time), operations (e.g., file read/write), and other security-related properties (e.g., failure code).
5.1.2 Causality Analysis

Causality analysis [111, 112, 96, 110, 121] analyzes the auditing events to infer the dependencies among system entities and present the dependencies as a directed graph. In the dependency graph $G(E, V)$, a node $v \in V$ represents a process, a file, or a network network. An edge $e(u, v) \in E$ indicates a system auditing event involving two entities $u$ and $v$ (e.g., process creation, file read or write, and network access), and its direction (from the source node $u$ to the sink node $v$) indicates the data flow. Each edge is associated with a time window, $tw(e)$. We use $ts(e)$ and $te(e)$ to represent the starting time and the ending time of $e$. Formally, in the dependency graph, for two event edges $e_1(u_1, v_1)$ and $e_2(u_2, v_2)$, there exists causal dependency between $e_1$ and $e_2$ if $v_1 = u_2$ and $ts(e_1) < te(e_2)$.

Causality analysis enables two important security applications: (i) backward causality analysis that helps identify the entry points of attacks, and (ii) forward causality analysis that helps investigate the ramifications of attacks. Given a POI event $e_s(u, v)$, a backward causality analysis traces back from the source node $u$ to find all events that have causal dependencies on $u$, and a forward causality analysis traces forward from the sink node $v$ to find all events on which $v$ has causal dependencies.

5.1.3 Motivating Example

[Figure 5.1] shows an example dependency graph of a data leakage attack: the attacker exploited the shellshock bug to execute [curl in the target system and downloaded a backdoor program bdoor. The attacker then executed the bdoor program to open a backdoor on port 4444. Through the backdoor, the attacker collected sensitive data using [tar, bzip2, and gpg, and uploaded the collected data to a remote host. The POI entity is a network connection to the suspicious remote host. The complete graph
contains 5,817 nodes and 215,380 edges, which is impractical for a human analyst to manually inspect.

**SYSREP** enables automatic attack investigation by assigning weights to edges and propagating reputation scores from seed sources to POI entities. In this example, we set the reputation score for the suspicious network source (i.e., the top left parallelogram) to 0.0 and the reputation scores for system libraries to 0.5.

**Challenges:** From Figure 5.1, we can see that the critical edges (i.e., the red bold edges) representing the attack sequence are buried in many irrelevant system activities, such as remote login via ssh and normal program execution like Java and Python. Thus, **SYSREP** needs to assign high weights to the critical edges to distinguish them from edges representing irrelevant system activities, and propagates reputation scores based on the weights to ensure that the critical edges have the most impact on the POI entity. Furthermore, the dependency path from the suspicious source to the POI entity has more than 10 hops, where each node in the middle of the path also receives reputation from other sources. As such, **SYSREP** needs to make sure that the reputation propagation does not suffer from fast degradation in a long path with many irrelevant nodes that can potentially distort the final reputation score.

**Techniques of SYSREP:** **SYSREP** first computes the weights using three novel discriminative features (Section 5.3.3). In this example, compared with other (i.e., non-critical) edges, the critical edges have more similar data amount as the data amount transmitted in the POI event, the start timestamps of these edges are closer to the POI event, and the incoming and outgoing edges are relatively small for the nodes involved in the critical edges. Thus, by combining these features into the discriminative edge weights (Section 5.3.4), **SYSREP** maximizes the differences between critical edges and other edges, and reveals the critical edges. Based on the weights, **SYSREP** adopts an inheritance fashion that considers the impact of the edges to propagate the reputation scores from the seed sources in the graph (Section 5.3.5). From Figure 5.1
we observe that the reputation scores of nodes on the critical edges are close to 0, as compared to the irrelevant system libraries with reputation scores close to 0.5. By connecting the nodes with low reputation scores and the critical edges, we are able to reconstruct the attack sequence. We also observe that the POI entity receives a very low reputation score that is almost 0, which indicates that it comes from a suspicious source as expected.

5.2 Overview

Figure 5.2 illustrates the architecture of the SysRep system. SysRep consists of three phases: (1) dependency graph generation, (2) graph preprocessing & discriminative weight computation, and (3) attack investigation. In Phase I, SysRep leverages mature system auditing tools (e.g., auditd [30], ETW [27], DTrace [22], and Sysdig [50]) to collect system-level audit logs about system calls. Given a POI event, SysRep parses the collected logs and performs causality analysis [111, 112] to generate the dependency graph for the event (Section 5.3.1). In Phase II, SysRep preprocesses the graph by merging the same type of edges between two nodes that occur within a time window threshold to reduce the graph size and splitting the nodes to remove parallel edges. This process transforms the large dependency graph into a simple directed graph (Section 5.3.2), which is easier for weight computation and reputation propagation. To produce a weighted dependency graph such that critical edges are easily distinguishable from non-critical edges by their weights, for each edge, SysRep extracts three novel features that model the importance of the event.
edge with respect to the POI event from three dimensions: time dimension, data amount dimension, and structure dimension (Section 5.3.3). SysRep then employs a novel local feature projection mechanism to project the three features to a combined, discriminative weight (Section 5.3.4). In Phase III, SysRep enables automatic attack investigation by (1) propagating reputation from seeds sources on the weighted dependency graph to automatically determine the suspiciousness/trustworthiness of POI entities (Section 5.3.5) and (2) providing a suggested range of threshold values based on the discriminative edge weights to automatically reconstruct the attack sequence by distinguishing critical edges from other edges.

## 5.3 Design of The SysRep System

In this section, we present the three phases and the components of SysRep in detail.

### 5.3.1 Phase I: Dependency Graph Generation

**System Auditing:** SysRep leverages system auditing tools \([30, 27, 22, 50]\) to collect information about system calls from the kernel, and then parses the collected events to build a global dependency graph. SysRep focuses on three types of system events: (i) file access, (ii) processes creation and destruction, and (iii) network access. Table 5.3 shows the representative system calls (in Linux) processed by SysRep. Particularly, for a process entity, we use the process name and PID as its unique identifier. For a file entity, we use the absolute path as its unique identifier. For a network connection entity, as processes usually communicate with some servers using different network
Algorithm 3: Backward Causality Analysis

**Input:** POI Event: $e_s$, Global Dependency Graph: $G$  
**Output:** Dependency Graph for the POI Event: $G_d$

$G_d \leftarrow \text{new Graph}()$

Queue.add($e_s$)

while Queue is not Empty do

    $u \leftarrow \text{Queue.pop}().\text{source}$

    set $\leftarrow G\text{.incomingEdgeOf}(u)$

    for $e \in \text{set}$ do

        if $ts(e) < te(e_s)$ then

            $G_d\text{.add}(e)$

            Queue.add($e$)

connections but with the same IPs and ports, treating these connections differently greatly increases the amount of data we trace and such granularity is not required in most of the cases \[121, 94, 91\]. Thus, we use 5-tuple ($\langle \text{srcip, srcport, dstip, dstport, protocol} \rangle$) as a network connection’s unique identifier. Failing to distinguish different entities causes problems in relating events to entities and tracking the dependencies of events. For each system call, SysRep extracts the security-related attributes of entities (Table 5.1) and events (Table 5.2). SysRep filters out failed system calls, which could cause false dependencies among events.

**Causality Analysis:** Given a POI event $e_s$, SysRep applies causality analysis (Section 5.1.2) to produce a dependency graph $G_d$, as shown in Algorithm 3. Algorithm 3 adds $e_s$ to a queue (Line 2), and repeats the process of finding eligible incoming edges of the edges (i.e., incoming edges of the source nodes of edges) in the queue (Lines 3-9) until the queue is empty. The output is a dependency graph that only contains relevant system entities and events with respect to the given POI event.

### 5.3.2 Phase II: Graph Preprocessing

SysRep performs graph preprocessing to transform the dependency graph from a directed multigraph to a simple directed graph with no parallel edges.

\[\text{Forward causality analysis can be implemented in a similar way.}\]
**Edge Merge:** The dependency graph produced by causality analysis often has many edges between the same pair of nodes [153]. The reason for generating these excessive edges is that the OS typically finishes a read/write task (e.g., file read/write) by distributing the data to multiple system calls and each system call processes only a portion of the data. Inspired by the recent work that proposes Causality Preserved Reduction (CPR) [153] for dependency graph reduction, SysRep merges the edges between two nodes. As shown in the study [153], CPR does not work well for processes that have many interleaved read and write system calls, which introduces excessive causality. As such, SysRep adopts a more aggressive approach: for edges between two nodes that represent the same system call, SysRep will merge them into one edge if the time differences of these edges are smaller than a given time threshold. Empirically, we set the time threshold as 10 seconds, which is large enough for most processes to finish file transfers and network communications in modern computers. Since such merge is performed after the dependency graph generation, all the dependencies are still preserved but only the time windows of certain edges are merged.

**Node Split:** After the edge merge, the dependency graph may still have parallel edges (i.e., edges indicating read or write from different system calls). For example, a process may receive data from a network socket via both `read` and `recvfrom`. These parallel edges create complications for weight computation: for a node, some of its incoming edges’ time windows may violate the causal dependencies on the outgoing edges.

To address the problem, SysRep first enumerates all the pairs of nodes that have parallel edges. For a pair of nodes \((u, v)\) that have parallel edges from \(u\) to \(v\), SysRep splits \(u\) into multiple copies and assigns each copy to one parallel edge, so that each copy of \(u\) only has one outgoing edge to \(v\). The copies of \(u\) also inherit all \(u\)’s incoming edges that have causal dependencies on the \(u\)’s outgoing edges. The output after the node split is a simple directed dependency graph without parallel edges.
5.3.3 Phase II: Feature Extraction

For each edge, SYSREP extracts three novel discriminative features that model its importance with respect to the POI event.

- **Relative Data Amount Difference**: For an edge $e(u, v)$, we measure the distance between the size of data processed by the system call and the size of POI entity. The intuition is that the smaller the distance is, the more important this edge is to the data flow in the POI event.

  \[
  f_D(e) = \frac{1}{|s_e - s_{e_s}| + \alpha},
  \]

  Eq. (5.1) gives the relative data amount difference feature, where $s_e$ and $s_{e_s}$ represent the data amount associated with the event edge $e$ and the POI event $e_s$. Note that we use a small positive number $\alpha$ to handle the special case when $e$ is the POI event. Thus, the POI event will have the highest data amount difference feature value $f_D(e_s) = 1/\alpha$. Empirically, we set $\alpha = 1e^{-4}$.

- **Relative Time difference**: For an edge $e(u, v)$, we measure the distance between its end time $t_e(e)$ and the end time of the POI event $t_e(e_s)$. The intuition is that the event that occurred closer to the POI event is more temporally related to the POI event.

  \[
  f_T(e) = \ln(1 + 1/|t_e - t_{e_s}|),
  \]

  Eq. (5.2) gives the relative time difference feature, where $t_e$ and $t_{e_s}$ represent timestamp values (we use the event end time) of the event edge $e$ and the POI event $e_s$. To handle the special case when $e$ is the POI event (i.e., $|t_e - t_{e_s}| = 0$), we use one tenth of the minimal time unit (nanosecond) in the audit logging.
framework (i.e., $1e^{-10}$) to compute its feature: $f_{T(e)} = \ln(1 + 1e10)$. This ensures that the POI event has the highest feature value.

- **Concentration Degree**: One important category of non-critical edges that often appear in a causal graph are events that access system libraries [153, 148]. These edges are often associated with considerable data amount and occur at various timestamps, and hence using only $f_{T(e)}$ and $f_{D(e)}$ is less effective in distinguishing critical edges from them. To address this challenge, we observe that most system library nodes are source nodes in the corresponding edges and do not have any incoming edges. Another category of non-critical edges are events that involve long-running processes as source nodes, which often have few incoming edges but many outgoing edges. Based on this observation, we define the concentration degree for the edge $e(u, v)$ as:

$$f_{C(e)} = \frac{\text{InDegree}(u)}{\text{OutDegree}(u)},$$

(5.3)

where $\text{InDegree}(u)$ and $\text{OutDegree}(u)$ represent the in-degree and out-degree of the source node $u$ in $e(u, v)$. For seed nodes, since they are very important in initiating the reputation propagation, we set their concentration degree to be 1. This feature helps smooth out the impacts of system libraries with no incoming edges and long-running processes with many outgoing edges.

**Local Feature Normalization**: Before using the three features to compute edge weights, it is often a good practice to scale them in the same range [66, 89]. Global feature scaling using standard methods such as range normalization and standardization does not make much sense in our context, since a node is only affected by its parents but not by its children or siblings. Recognizing this, SysRep adopts a local feature scaling scheme: for an edge $e(u, v)$, SysRep locally normalizes its each feature by the sum of corresponding features of all incoming edges of the sink node.
This scheme enables the three features of all \( v \)'s incoming edges to be compared on the same scale.

### 5.3.4 Phase II: Weight Computation

For each edge, \textit{SysRep} computes a discriminative weight using its three normalized features. The weight models the aggregated importance of the event edge with respect to the POI event in the attack sequence reconstruction, so that critical edges can be easily distinguished from non-critical edges.

One approach to compute a weight score is to adopt a classification-based approach to train a binary classifier using the three features and output a probability score. Though this approach has demonstrated its applicability for several domains \cite{90,91}, it faces significant limitations in our problem context. To achieve a high classification accuracy, supervised learning-based approaches often require large amount of training data and the training data and the test data come from the same distribution \cite{66,89}. However, in our context, features are extracted with respect to the specific POI event, and thus the model learned for one type of attack with a specific POI can hardly generalize to other types of attacks with different POIs. The highly imbalanced classes (\textit{i.e.}, the very few number of critical edges as compared to non-critical edges) further impede the model generalization.

Recognizing such limitations, \textit{SysRep} leverages ideas from dimensionality reduction and discriminant analysis \cite{89}. Specifically, \textit{SysRep} employs a novel \textit{discriminative local feature projection mechanism} based on an extended version of Linear Discriminant Analysis (LDA) \cite{128}, and computes a projection vector to project the three-dimensional feature vector to a one-dimensional weight, while ensuring that the projected weights of critical edges and non-critical edges are maximally separated.

We next present the weight computation mechanism (Algorithm 4) in detail.
Algorithm 4: Weight Computation

Input: Dependency Graph for the POI event: $G_d$
Output: Weighted Dependency Graph, $G_{wd}$

for $v \in G_d$ do
  $Set \leftarrow G_d$\_incomingEdgeOf($v$)
  $group_1, group_2 \leftarrow$ Multi-KMeans++($Set$)
  $(\omega_D^*, \omega_T^*, \omega_C^*) \leftarrow$ extendedLDA($group_1, group_2$)
  for $e \in Set$ do
    $W_e = \omega_D^* f_D(e) + \omega_T^* f_T(e) + \omega_C^* f_C(e)$
  $W' = \sum_{e\in Set} W_e$
  for $e \in Set$ do
    $W_e \leftarrow W_e/W'$

Step 1: Local Edge Clustering: Due to the localized nature of our problem context and features (Section 5.3.3), rather than computing a projection vector globally for all edges, SysRep computes a projection vector locally for the incoming edges of each node. This helps avoid the bias introduced by the large number of irrelevant edges.

In Step 1, for each node, SysRep locally clusters all its incoming edges in two groups, which is a prerequisite for discriminant analysis. Specifically, SysRep adopts Multi-KMeans++ clustering algorithm [67]. Based on KMeans, KMeans++ uses a different method for choosing the initial seeds to avoid poor clustering. Multi-KMeans++ is a meta algorithm that performs $n$ runs of KMeans++ and then chooses the best clustering that has the lowest distance variance over all clusters. Empirically, we set $k = 2$ (since we want two groups) and $n = 20$ (to ensure that the best clustering is chosen).

Step 2: Local Feature Projection via Extended LDA: For each node, after locally clustering its incoming edges in two groups, SysRep employs an extended version of Linear Discriminant Analysis (LDA) [128] to compute a projection vector, and applies the projection vector to the feature vector of each incoming edge to compute an edge weight.

Formally, for a node $v$, we denote the feature vectors of its $N$ incoming edges as $x^1, x^2, \ldots, x^N$, which are clustered in two groups: group $g_1$ contains $N_1$ edges,
and group $g_2$ contains $N_2$ edges (i.e., $N_1 + N_2 = N$). The group mean vectors are $\mu_1 = \frac{1}{N_1} \sum_{x \in g_1} x$ and $\mu_2 = \frac{1}{N_2} \sum_{x \in g_2} x$. The between-group scatter matrix is defined as $S_b = (\mu_1 - \mu_2)(\mu_1 - \mu_2)^T$. The within-group scatter matrix is defined as $S_w = \sum_{x \in g_i} (x - \mu_i)(x - \mu_i)^T$. LDA is a technique that seeks to reduce dimensionality while preserving as much of the group discriminatory information as possible. Specifically, LDA finds a projection vector $\omega$ that maximizes the following Fisher criterion:

$$J(\omega) = \frac{\omega^T S_b \omega}{\omega^T S_w \omega}$$

(5.4)

In order words, LDA looks for the best projection direction such that the projected samples from the same group are close to each other (as enforced by the denominator $\omega^T S_w \omega$), and the projected samples from different groups are far away from each other as possible (as enforced by the numerator $\omega^T S_b \omega$). Solving the optimization problem yields:

$$\omega^* = \arg\max J(\omega) = S_w^{-1}(\mu_1 - \mu_2)$$

(5.5)

Denoting the projection vector as $\omega^* = (\omega_D^*, \omega_T^*, \omega_C^*)$, for an incoming edge $e(u, v)$ of node $v$, its (unnormalized) weight $W_{e_{UN}}$ is computed as:

$$W_{e_{UN}} = \omega_D^* f_D(e) + \omega_T^* f_T(e) + \omega_C^* f_C(e)$$

(5.6)

The applicability of Eq. (5.5) requires that $S_w$ is nonsingular (i.e., $S_w^{-1}$ exists). However, this criterion may be violated quite often in our problem context, due to the large imbalance between the number of critical edges and the number of non-critical edges. Furthermore, standard LDA only ensures that the projected values of different groups are largely separated, rather than guaranteeing which group has higher projected values, while our goal is to make critical edges have higher weights than non-critical edges. Another limitation of standard LDA is that sometimes the
projected values are negative, while we require the edge weights to be a non-negative number to model the importance with respect to the POI event.

Recognizing such limitations, we extend the standard LDA in three aspects.

(1) Handling singular $S_w$: When $S_w$ is singular, we select the projection vector (we normalize it first) from the following two candidates that results in a larger Fisher criterion numerator (i.e., $\omega^T S_b \omega$):

- $S_w^+ (\mu_1 - \mu_2)$. $S_w^+$ is the Moore-Penrose inverse of $S_w$.
- $\mu_1 - \mu_2$ (i.e., the direction of group mean difference)

(2) Negating the projection vector by condition: To make critical edges have higher projected values, we negate the projection vector by condition. Note that this problem is fundamentally challenging, since we do not have labels for critical edges and thus we do not know which group contains critical edges. We approach this problem using a set of heuristics:

I If all three dimensions of the projection vector are non-positive, negate.

II If all three dimensions of the projection vector are non-negative, do not negate.

III Else, negate by condition:

i If only one group has seed edges and this group has a lower projected mean, negate. This is based on the insight that seed edges should have high weights.

ii Else, if the group with a smaller size has a lower projected mean, negate. This is based on the insight that the number of critical edges is smaller than the number of non-critical edges in most cases.

(3) Scaling the projected weights: We scale the projected weights to the range $[0, 1]$. We further add a small offset (we use 1/100 of the difference between the
smallest value and the second smallest value) to each scaled weight, so that all scaled weights are positive.

**Step 3: Local Edge Weight Normalization:** Same as local feature normalization (Section 5.3.1), for an edge $e(u, v)$, we *locally normalize* its scaled weight by the sum of corresponding weights of all incoming edges of the sink node $v$:

$$W_e = \frac{W_{e_{UN}}}{\sum_{e' \in \text{IncomingEdge}(v)} W_{e'_{UN}}}$$

(5.7)

After normalization, the weights of all edges are in the range $(0, 1]$, and the sum of weights of all incoming edges of any node (except for nodes that do not have incoming edges) is equal to 1. Note that for nodes that have only one incoming edge, we skip the local clustering and projection process and directly set its final weight to 1. This completes the Phase II by producing a weighted dependency graph with discriminative weights to distinguish critical edges from non-critical edges.

### 5.3.5 Phase III: Attack Investigation

**Reputation Propagation:** To perform reputation propagation, SysRep assigns initial reputations to seed sources and propagates the reputations on the weighted dependency graph using an inheritance fashion.

*Reputation Scheme.* The design goal of the reputation scheme is to capture the fact that if a file (or program) is downloaded or installed from a reputable source, it should be more reliable than the file (program) downloaded from an suspicious website or from an suspicious equipment, and the reputation of files (programs) from the reliable sources should be higher than the reputation of those from suspicious sources. As such, we define the reputation of a system entity to be $[0.0, 1.0]$, where a file (program) from trusted sources should have a reputation closer to 1.0, and a file (program) from suspicious sources should have a reputation closer to 0.0.
Seed Sources and System Libraries. Seed sources are the nodes without incoming edges in the dependency graph, which require initial reputations. They are usually trusted sources such as trusted websites/programs/services like official Microsoft updater or Chrome updater (assigned high reputations), system libraries like libc (assigned neutral reputations), or untrusted sources like malicious domains and unknown programs (assigned with low reputations). We propose that trusted sources are assigned reputation 1.0, system libraries that can be used by both legitimate users and the attacker are assigned reputation 0.5, and untrusted sources are assigned 0.0. Note that SysRep allows the security analyst to adjust the seed assignment according to the domain knowledge of the enterprise systems or special needs. Furthermore, the process of seed assignment can be largely automated using external reputation sources (e.g., libraries can be easily labeled using a database and sources can be labeled via IP/binary reputations [57]).

Reputation Inheritance. Based on the edge weights and the chosen seeds, we iteratively update other nodes’ reputation values. To prevent fast degradation of reputation values, SysRep updates a node’s reputation using an inheritance fashion:

\[ R_v = \sum_{e \in \text{IncomingEdge}(v)} R_{\text{sourceNode}(e)} \times W_e, \]  

(5.8)

where \( \text{IncomingEdge}(v) \) represents the set of incoming edges of \( v \), \( \text{sourceNode}(e) \) represents the source node of \( e \), \( R_{\text{sourceNode}(e)} \) represents the reputation of \( \text{sourceNode}(e) \), and \( W_e \) represents the weight of \( e \).

Algorithm 5 gives the details of reputation propagation. A node \( v \) in the dependency graph receives its reputation by inheriting the weighted reputations from all of its parent nodes (Lines 7-12). Note that if we adopt a distribution fashion that ensures the sum of reputations for \( v \)'s children nodes to be equal to \( v \)'s reputation, then the reputation will degrade quickly in a few hops, which does not work well.
for dependency graphs in the context of APT attack investigation that often have many paths with many hops. The process of reputation propagation is an iterative process. For each iteration, the algorithm computes the sum of reputation differences for all the nodes (Line 11, Line 13). The propagation terminates when the aggregate difference between the current iteration and the previous iteration is smaller than a threshold $\delta$ (Line 1), indicating that the reputations for all nodes become stable. Empirically, we set $\delta = 1e - 13$. Note that the reputations of the seed nodes remain unchanged (Lines 4-5).

**Attack Sequence Reconstruction:** SysRep leverages the weights in the dependency graph for reconstructing the attack sequence, which consists of critical edges and their nodes. Since the weights computed by SysRep aim to maximize the difference between critical and non-critical edges, we can specify a minimum weight as a threshold and hide the edges with weights below that threshold. For example, the weights of edges from the irrelevant system activities (e.g., daily tasks or file indexing) usually have lower weights than those representing the attack provenance. By carefully choosing this threshold, SysRep can filter out these irrelevant system activities while retaining the attack provenance (Section 5.4.2).
In practice, security analysts can first hide the non-critical edges using a higher threshold to focus on the investigation of the attack, and gradually show part of the non-critical edges by lowering the threshold to get more context of the attack-related activities. For example, certain system libraries may reveal the functionality that the malicious payloads possess.

5.4 Evaluation

We built SYSREP upon Sysdig [50], and deployed our tool in a server to collect system auditing events and perform attack investigation. The server is used by other users to perform daily tasks, so that enough noise of irrelevant system activities can be collected. We performed a series of attacks based on known exploits in the deployed environment, and applied SYSREP to perform attack investigation on the attacks, demonstrating the effectiveness of SYSREP. In total, our evaluations use real system monitoring data that consists of 2 billion events. Each attack is done with the time gap being at least 1 hour.

Specifically, we conduct three sets of evaluations. First, to evaluate the effectiveness of SYSREP in propagating reputations, we compare the reputation scores of the POI entities against the expected reputation scores in benign scenarios (POI entities coming from trusted sources) and attack scenarios (POI entities coming from suspicious sources). We also compare the results with three other weight computation approaches. Second, we compare the weight computation in SYSREP with the edge prioritization of the state-of-the-art causality analysis, PrioTracker [121], which prioritizes edges during dependency search based on the fanout of nodes. Finally, we evaluate the effectiveness of SYSREP in revealing critical edges for attack sequence reconstruction.
5.4.1 Evaluation Setup

The evaluations are conducted on a server with an Intel(R) Xeon(R) CPU E5-2637 v4 (3.50GHz), 256GB RAM running 64bit Ubuntu 18.04.1. We performed 8 tasks to inject benign and malicious payloads into the system through key system interfaces that are vulnerable for attacks. We also performed 5 APT attacks in the deployed environment. We then collected the system auditing events and applied SysRep to analyze the events.

Benign and Malicious Payloads Through Key System Interfaces

We performed 8 tasks that employ common system interfaces to inject benign or malicious payloads. These representative system interfaces are commonly exploited in attacks [127].

- File merge: 2File, 3File

- Shell execution: shell-script (list all files in the Home folder and write the results to a file)

- File download: curl, wget, shell-wget (wget called by a shell script), python-wget (wget called by a Python script)

- File transfer: scp

APT Attacks

We performed five APT attacks that capture the important traits of APT attacks depicted from the Cyber Kill Chain framework [15]. Note that an APT attack consists of a series of steps, and some steps may not be captured by system monitoring (e.g., user inputs and inter-process communications). Such limitations can be addressed by employing more powerful auditing tools, but it is out of the scope of this paper.
Thus, we identified 10 key steps that are related to POI entities for our evaluations in the five APT attacks.

**APT Attack 1: Zero-Day Penetration to Target Host:** The scenario emulates the attacker’s behavior who penetrates the victim’s host leveraging previously unknown zero-day attack. Zero-day vulnerabilities are attack vectors that are previously unknown to the community, therefore allow the attacker to put their first step into their targets. In our case, we assume that the `bash` binary in victim’s host is outdated and vulnerable to shellshock [11]. The victim computer hosts web service that has CGI written as BASH script. The attacker can run an arbitrary command when she passes the specially crafted attack string as one of environment variable. Leveraging the vulnerability, the attacker runs a series of remote commands to plant and run initial attack by: (1) transferring the payload (`penetration-c1`), (2) changing its permission, and (3) running the payload to bootstrap its campaign (`penetration-c2`).

**APT Attack 2: Password Cracking After Shellshock Penetration:** After initial shellshock penetration, the attacker first connects to Cloud services (e.g., Dropbox, Twitter) and downloads an image where C2 (Command and Control) host’s IP address is encoded in EXIF metadata (`password-crack-c1`). The behavior is a common practice shared by APT attacks [105, 58] to evade the network-based detection system based on DNS blacklisting.

Using the IP, The malware connects to C2 host. C2 host directs the malware to take some lateral movements, including a series of stealthy reconnaissance maneuvers. In this stage, the attacker generally takes a number of actions. Among those, we emulate the password cracking attack. The attacker downloads password cracker payload (`password-crack-c2`) and runs it against password shadow files (`password-crack-c3`).

**APT Attack 3: Data Leakage After Shellshock Penetration:** After lateral movement stage, 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 activity, copying and compressing of important files, and transferring to its C2 host. The attacker scans the file system, scrap files into a single compress file and transfer it back to C2 host (data-leakage).

**APT Attack 4: Command-Line Injection with Input Sanitization Failures:**
Different from the previous shellshock case, a program may contain vulnerabilities introduced by developer errors and this can also be a initial attack vector that invites the attacker into their target systems. To represent such cases, we wrote a web application prototype that fails to sanitize inputs for a certain web request, hence allows Command line Injection attack. Our prototype service mimics the Jeep-Cherokee attack case [130] which implements a remote access using the conventional web service API that internally uses DBUS service to run the designated commands. Due to the developer mistake, the web service fails to sanitize the remote inputs, the attacker can append arbitrary commands followed by semi-colon (;). Leveraging this vulnerability, we can download backdoor program (command-injection-c1) and collect sensitive data (command-injection-c2).

**APT Attack 5: VPNFilter:** We prototyped a famous IoT attack campaign, VPNFilter malware [43], which infected millions of dozens of different IoT devices exploiting a number of known or zero-day vulnerabilities [13, 13]. The attack’s significance lies in how the malware operates during its lateral movement stage following its initial penetration. The campaign employs up-to-date hacker practices to bypass conventional security solutions based on static blacklisting approaches and has an architecture to download plug-in payload on-demand, at run-time. We prototyped the malware referring to one of its sample for x86 architecture [138].

The VPNFilter stage 1 malware accesses a public image repository to get an image. In the EXIF metadata of the image, it contains the IP address for the stage 2 host.
Table 5.4: POI reputations of benign payloads through key system interfaces (expected 1.0)

<table>
<thead>
<tr>
<th>Attack/Task</th>
<th>LP-Fixed</th>
<th>PrioTracker</th>
<th>LP-Global</th>
<th>LP-Global+</th>
<th>SysRep</th>
</tr>
</thead>
<tbody>
<tr>
<td>3File</td>
<td>0.85</td>
<td>0.50</td>
<td>0.50</td>
<td>0.98</td>
<td>1.00</td>
</tr>
<tr>
<td>2File</td>
<td>0.80</td>
<td>0.50</td>
<td>0.50</td>
<td>0.50</td>
<td>1.00</td>
</tr>
<tr>
<td>curl</td>
<td>0.70</td>
<td>0.60</td>
<td>0.51</td>
<td>0.51</td>
<td>0.99</td>
</tr>
<tr>
<td>shell_script</td>
<td>0.62</td>
<td>0.50</td>
<td>0.50</td>
<td>0.57</td>
<td>0.96</td>
</tr>
<tr>
<td>python_wget</td>
<td>0.71</td>
<td>0.65</td>
<td>0.61</td>
<td>0.63</td>
<td>0.99</td>
</tr>
<tr>
<td>scp</td>
<td>0.74</td>
<td>1.00</td>
<td>0.93</td>
<td>0.92</td>
<td>1.00</td>
</tr>
<tr>
<td>shell_wget</td>
<td>0.80</td>
<td>0.75</td>
<td>0.82</td>
<td>0.89</td>
<td>0.98</td>
</tr>
<tr>
<td>wget</td>
<td>0.71</td>
<td>0.54</td>
<td>0.50</td>
<td>0.50</td>
<td>1.00</td>
</tr>
<tr>
<td>avg</td>
<td>0.74</td>
<td>0.63</td>
<td>0.61</td>
<td>0.69</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Table 5.5: POI reputations of malicious payloads through key system interfaces (expected: 0.0)

<table>
<thead>
<tr>
<th>Attack/Task</th>
<th>LP-Fixed</th>
<th>PrioTracker</th>
<th>LP-Global</th>
<th>LP-Global+</th>
<th>SysRep</th>
</tr>
</thead>
<tbody>
<tr>
<td>3File</td>
<td>0.15</td>
<td>0.50</td>
<td>0.49</td>
<td>0.02</td>
<td>~0.00</td>
</tr>
<tr>
<td>2File</td>
<td>0.20</td>
<td>0.50</td>
<td>0.50</td>
<td>0.50</td>
<td>~0.00</td>
</tr>
<tr>
<td>curl</td>
<td>0.30</td>
<td>0.40</td>
<td>0.49</td>
<td>0.49</td>
<td>0.01</td>
</tr>
<tr>
<td>shell_script</td>
<td>0.38</td>
<td>0.50</td>
<td>0.50</td>
<td>0.50</td>
<td>0.04</td>
</tr>
<tr>
<td>python_wget</td>
<td>0.29</td>
<td>0.35</td>
<td>0.39</td>
<td>0.37</td>
<td>0.01</td>
</tr>
<tr>
<td>scp</td>
<td>0.26</td>
<td>~0.00</td>
<td>0.07</td>
<td>0.08</td>
<td>~0.00</td>
</tr>
<tr>
<td>shell_wget</td>
<td>0.20</td>
<td>0.25</td>
<td>0.18</td>
<td>0.11</td>
<td>0.02</td>
</tr>
<tr>
<td>wget</td>
<td>0.21</td>
<td>0.46</td>
<td>0.50</td>
<td>0.50</td>
<td>~0.00</td>
</tr>
<tr>
<td>avg</td>
<td>0.25</td>
<td>0.37</td>
<td>0.39</td>
<td>0.32</td>
<td>0.01</td>
</tr>
</tbody>
</table>

(vpnfilter-c1). It downloads the VPNFilter stage2 from the stage2 server, and runs it (vpnfilter-c2).

5.4.2 Evaluation Results

Reputation Propagation

We evaluate the effectiveness of SysRep in identifying whether POI entities come from trusted sources or suspicious sources via reputation propagation in both normal and malicious scenarios, respectively.

Evaluation Setup: All the nodes that have no incoming edges are considered as seed nodes. We set the reputation of seed sources representing trusted sources to 1.0, and seed sources representing system libraries to 0.5. In malicious scenarios and APT attacks, we set the reputation of seed sources representing suspicious sources to
<table>
<thead>
<tr>
<th>Attack</th>
<th>LP-Fixed</th>
<th>PrioTracker</th>
<th>LP-Global</th>
<th>LP-Global+</th>
<th>SysRep</th>
</tr>
</thead>
<tbody>
<tr>
<td>data-leakage</td>
<td>0.16</td>
<td>~0.00</td>
<td>0.16</td>
<td>0.15</td>
<td>~0.00</td>
</tr>
<tr>
<td>password-crack-c1</td>
<td>0.04</td>
<td>0.25</td>
<td>0.03</td>
<td>~0.00</td>
<td>~0.00</td>
</tr>
<tr>
<td>password-crack-c2</td>
<td>0.40</td>
<td>~0.00</td>
<td>0.50</td>
<td>~0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>password-crack-c3</td>
<td>0.04</td>
<td>~0.00</td>
<td>~0.00</td>
<td>0.02</td>
<td>~0.00</td>
</tr>
<tr>
<td>penetration-c1</td>
<td>0.13</td>
<td>0.30</td>
<td>0.43</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>penetration-c2</td>
<td>0.13</td>
<td>0.43</td>
<td>0.13</td>
<td>0.13</td>
<td>0.04</td>
</tr>
<tr>
<td>command-injection-c1</td>
<td>0.27</td>
<td>~0.00</td>
<td>0.49</td>
<td>~0.00</td>
<td>~0.00</td>
</tr>
<tr>
<td>command-injection-c2</td>
<td>0.30</td>
<td>~0.00</td>
<td>0.24</td>
<td>0.17</td>
<td>0.04</td>
</tr>
<tr>
<td>vpnfilter-c1</td>
<td>0.05</td>
<td>~0.00</td>
<td>0.02</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>vpnfilter-c2</td>
<td>0.04</td>
<td>~0.00</td>
<td>~0.00</td>
<td>~0.00</td>
<td>~0.00</td>
</tr>
<tr>
<td><strong>avg</strong></td>
<td>0.15</td>
<td>0.10</td>
<td>0.20</td>
<td>0.05</td>
<td>0.01</td>
</tr>
</tbody>
</table>

We propagate the reputation from seed sources according to Algorithm 5 and record the reputation scores of the POI entities (referred to as POI reputation). An effective approach will lead to a higher POI reputation in the benign scenarios and a lower POI reputation in the malicious scenarios.

To demonstrate the effectiveness of SysRep’s weight computation, we compare the POI reputations computed by the following four weight computation approaches:

- **LP-Fixed**: We select a fixed parameter vector \( (0.1, 0.5, 0.4) \) empirically and normalize it to be the projection vector.

- **LP-Global**: We globally cluster all edges in the graph using Multi-KMeans++ and compute the projection vector using extended LDA.

- **LP-Global+**: Same as previous one, but for nodes that have only one incoming edge \( (i.e., \) outlier edges), we do not consider these edges in the global clustering and global projection vector computation, and directly assign their final weights to 1.

- **SysRep**: This is the one described in Section 5.3.4. We locally cluster the incoming edges of every node using Multi-KMeans++ and locally compute the projection vector using extended LDA.
Furthermore, we compare our weight computation approach with the event priority computation used in the state-of-the-art causality analysis approach, PrioTracker \[121\]. PrioTracker mainly uses the fanout of nodes to prioritize the dependencies in the causality analysis for a given POI event. While they also use a rareness score based on the reference models built upon normal activities in their proprietary environment, their reference models are not publicly available and difficult to generalize from their organizations to our deployed environment. Given that the rareness score accounts for only a small portion of their priority (i.e., 27\%), we use only the fanout of nodes to compute edge priorities.\footnote{Note that the reference models are complementary to SysRep, and SysRep can easily integrate the rareness score with our weights to compute final edge weights.} We then adapt the computed priories as the edge weights, and apply our algorithm for reputation propagation. In this way, we can do a fair comparison between SysRep and PrioTracker in reputation propagation.

Tables 5.4 and 5.5 show the results for benign and malicious payloads through key system interfaces. Table 5.6 shows the results for the APT attacks.

**POI Reputations of SysRep:** The results demonstrate that SysRep effectively propagates the reputation scores from trusted and suspicious sources to the POI entities (averagely 0.99 for benign scenarios and averagely 0.03 in malicious scenarios). This indicates the effectiveness of our weight computation (Section 5.3.4) and reputation inheritance (Section 5.3.5).

**Comparisons of Four Weight Computation Approaches:** We have the following observations: (1) The performance of LP-\textsc{Global+} significantly improves over LP-\textsc{Global}. This shows the effectiveness and necessity of treating outlier edges differently when doing weight computation. (2) LP-\textsc{Fixed} performs better than LP-\textsc{Global} and LP-\textsc{Global+} in system tasks through key system interfaces in both benign and malicious scenarios (Tables 5.4 and 5.5). This shows that the dependency graph is quite diverse and it is difficult to separate all edges into two discriminative
groups. However, treating the outlier edges differently in LP-GLOBAL+ improves over LP-FIXED for APT attacks. (3) SysRep achieves the best performance in most of the cases. Specifically, SysRep achieves 34.67%, 62.82%, and 43.76% improvement over LP-FIXED, LP-GLOBAL, and LP-GLOBAL+ in benign scenarios (Table 5.4) and average 94.52%, 95.61%, and 86.21% improvement in malicious scenarios (Tables 5.5 and 5.6). The results clearly demonstrate the necessity and superiority of clustering and projecting edges locally for each sink node. Note that this approach also treats outliers locally by directly setting their weights to 1, and thus SysRep embraces the merits of LP-GLOBAL+ and achieves the best performance.

**Comparison with PrioTracker:** The results demonstrate that SysRep achieves an average of 57.22% improvement over PrioTracker in benign scenarios (Table 5.4) and an average of 87.22% improvement over PrioTracker in malicious scenarios (Tables 5.5 and 5.6). As we can see from Table 5.6 while the average POI reputation achieved by PrioTracker is 0.10, it achieves bad performance for penetration-c1 (0.30), penetration-c2 (0.43), and password-crack-c1 (0.25), which incorrectly labels the malicious payloads as neutral, while SysRep correctly assigns low POI reputations (≤ 0.04) for all these attack steps. These results demonstrate the superiority of SysRep’s discriminative weight computation over PrioTracker’s priority computation.

**Attack Sequence Reconstruction**

The important goal of attack sequence reconstruction is to filter out as many irrelevant edges as possible while preserving critical edges in the dependency graph. Based on the edge weights (Section 5.3.4), SysRep hides non-critical edges whose weights are below a threshold. To provide a guidance on choosing this threshold, we test the filtering performance on all the attacks studied in Section 5.4.2 by using a range of values from 0.05 to 0.95 with a pace of 0.05 as the thresholds.
Figure 5.3: Effectiveness of filtering. The percentage of edges remaining after filtering drops significantly at $T = 0.10$ and remains stable below 10%.

Figure 5.4: Critical edge loss from filtering. All missing points are distributed between $T = 0.25$ and $T = 1.0$.

Figure 5.3 shows the average percentage of remaining edges after edge filtering. We observe that when the threshold reaches 0.10, the average percentage of remaining edges is 9.8%, and further increasing the threshold from 0.10 to 0.90 only results in a slightly increased amount of pruned edges (2.2%). Such results indicate that most of the edges having reputation scores below 0.10 or above 0.90.
While a higher threshold can hide more irrelevant edges, it may cause the loss of critical edges as well. Thus, we define the missing point as the highest threshold that preserves all the critical edges for a dependency graph, and measure the missing points for all the attacks. Figure 5.4 shows the cumulative distribution of the missing points for all the attacks. We observe that: (1) all missing points are greater than 0.25, and (2) about 80% of the missing points are greater than 0.90.

Combining the results in Figure 5.3 and Figure 5.4, we conclude that the reputation scores of almost all the critical edges are above 0.90, while the reputation scores of most of the non-critical edges are below 0.10. Such results clearly demonstrate the effectiveness of SysRep in leveraging the discriminative weights (Section 5.3.4) to distinguish critical edges from non-critical edges. Furthermore, based on Figure 5.4, we can suggest an optimal range of the threshold: [0.1, 0.25].

5.5 Discussion

Alternative Approaches for Weight Computation: As shown in Section 5.4.2, the SysRep achieves the best performance in all the compared weight computation approaches, especially much better than empirically setting a fixed projection vector. As mentioned in Section 5.3.4, classification-based approaches have very limited generalization capability in our problem context due to the lack of enough training samples and the highly specialized context introduced by the POI event. Among unsupervised learning approaches, approaches based on anomaly detection might be a substitution for KMeans. In order to compute the best projection vector, we extend the standard LDA from several aspects including handling the singular within-group scatter matrix $S_w$, negating the projection vector by condition to ensure that critical edges have higher projected weights than non-critical edges, and scaling the projected weights to a positive range. We acknowledge that there might be other ways to ex-
tend the standard LDA and other methods to achieve discriminative dimensionality reduction \cite{128,146}, which we leave for future exploration.

**Parallelization:** Part of the construction of weighted dependency graph can be potentially parallelized with distributed computing. The dependency graph produced by causality analysis \cite{111,112} can be parallelized by searching the dependency separately. The feature extraction for each edge is independent and can be parallelized. In the scenarios where multiple hosts are involved, dependencies on each host can be precomputed in parallel and thus cross-host causality analysis becomes the concatenation of multiple generated dependency graphs. The weight computation can be parallelized by processing the set of incoming edges of each node separately. However, the reputation propagation cannot be easily separated into independent components for parallel processing (though, we can convert Eq. (5.8) into a matrix-vector product form to save CPU cycles), due to the dependencies on the computation results, and the massive dependencies among system events also pose significant challenges for parallelization. Weighted causality analysis with parallelization is an interesting research direction that requires non-trivial efforts.

**Attacks Against the Reputation System:** In practice, the attacker, with some knowledge about the proposed system, may optimize its attack strategy to stay under cover. For instance, (i) to have a lower time weight, the attacker may inject the malicious files earlier but start the attack later, or (ii) to have a lower data weight, the attacker may perform the attack using multiple processes and each process is only associated with small data amount (i.e., the attacker distributes the malicious behavior to multiple processes). An attacker may also choose to gain reputation first before attacking the system or stay under cover and attack probabilistically. In this paper, we do not consider the potential attacks against the reputation system, which we leave for future work.
**Industrial View:** Because APT attacks consist of many small steps over a long period of time, even though security experts can obtain the system-wide log, it is time-consuming to manually inspect the daunting number of edges in the system dependency graph, and thus it is hard to discover the complete attack steps and reconstruct the attack sequence. Moreover, depending on the individual’s capability, the quality of the analysis may vary a lot. By enabling automatic investigation, SysRep not only reduces the time consumption of the analysis, but also mitigates the dependency on the capability of the security analysts. This makes SysRep highly applicable to small-scale businesses that are not affordable to hire a large team of security analysts to conduct labor-intensive investigation.

### 5.6 Summary

We proposed SysRep, a novel system that enables automatic attack investigation via reputation propagation and attack sequence reconstruction on weighted dependency graph. The evaluations on a wide range of operations that exploit key system interfaces as well as attacks demonstrate the practical efficacy of SysRep.
Chapter 6

Conclusion

In this thesis, we investigated the design of novel systems for effective and efficient forensic analysis of advanced cyber attacks (e.g., APT attacks). First, we proposed 

AiQL, a novel system for collecting attack provenance using system monitoring and assisting timely attack investigation. The AiQL 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 domain-specific characteristics of the data and the semantics of the query to efficiently schedule the query execution. Compared with existing systems, the AiQL system greatly reduces the cycle time for iterative and interactive attack investigation.

Next, we proposed SaQL, a novel stream-based system for real-time abnormal system behavior detection via querying the stream of system monitoring data. The SaQL system provides a domain-specific language, Stream-Based Anomaly Query Language (SaQL), which is specially designed to facilitate the task of expressing a wide range of anomaly models while incorporating the domain knowledge of experts. The SaQL language provides the constructs of event patterns to easily specify relevant system
activities and their relationships, and the constructs to perform stateful computation by defining states in sliding windows and accessing historical states to compute anomaly models. With these constructs, SAQL allows security analysts to express models for (1) rule-based anomalies, (2) time-series anomalies, (3) invariant-based anomalies, and (4) outlier-based anomalies. Compared with existing stream processing systems, SAQL provides a unified interface for expressing a wide range of anomaly models and is more efficient in memory utilization.

Finally, we proposed SysRep, a novel system that enables automatic attack investigation. SysRep assigns discriminative weights to edges in a dependency graph built from system auditing events and propagates reputation scores in the weighted graph from seed sources to POI events. The discriminative edge weights enable SysRep to reveal critical edges for automatically reconstructing the attack sequence. The reputation propagation paradigm enables SysRep to automatically determine the suspiciousness/trustworthiness of POI entities based on whether the system entities originate from suspicious sources or trusted sources. The evaluations on a wide range of operations that exploit key system interfaces as well as attacks demonstrate the practical efficacy of SysRep in enabling automatic attack investigation.
Appendix A

AIQL

A.1 APT Attack in Performance Evaluation and Conciseness Evaluation

Figure A.1 shows the environmental setup for the APT attack used in Sections 3.3.3 and 3.3.4. Table A.1 shows the exploits used in each step. The attacker aims to enumerate all user and host information stored in domain controller including the database server IP address and its administrator credential. With this, the attacker can transfer the database data dump back to her host, emulating a typical APT attack that leads to data leakage. We use $a1$-$a5$ to denote the attack behaviors that correspond to the 5 attack steps:

$a1$ Initial Compromise: The attacker first exploits the UnReal IRC server remote code execution vulnerability [10] to create a telnet connection to the attacker.

$a2$ Malware Infection: The attacker then uploads 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 its intranet, the attack could easily leverage other vulnerabilities, such as [12], to escalate its privilege and exe-
Figure A.1: Environmental setup for the APT attack in Sections 3.3.3 and 3.3.4

Table A.1: List of vulnerabilities and tools for the attack

<table>
<thead>
<tr>
<th>Step</th>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>CVE-2010-2075 [10]</td>
<td>UnReal IRC server remote code execution vulnerability</td>
</tr>
<tr>
<td>a2</td>
<td>Trojan malware</td>
<td>Create a reverse channel to the designated attacker host</td>
</tr>
<tr>
<td></td>
<td>Windows memory dump tools</td>
<td>Mimikatz, Kiwi</td>
</tr>
<tr>
<td>a4</td>
<td>Samba PsExec</td>
<td>Windows remote execution to create reverse connection.</td>
</tr>
<tr>
<td></td>
<td>Windows credential editors</td>
<td>Pwdump7.exe, WCE.exe</td>
</tr>
<tr>
<td>a5</td>
<td>SQL server dump</td>
<td>OSQL utility</td>
</tr>
</tbody>
</table>

cute memory dumping tools (e.g., Mimikatz, Kiwi) to obtain administrator credentials.

**a4 Obtain User Credentials:** Next, the attacker uses the credentials to penetrate into the domain controller and executes password dumping tools (e.g., pwdump7.exe, wce.exe) to obtain all users’ credentials.

**a5 Data Exfiltration:** Finally, the attacker penetrates into the database server and dumps the database data back to her host.
A.2 AIQL Queries in Performance Evaluation and Conciseness Evaluation

We present the 19 AIQL queries that we constructed for the performance evaluation (Section 3.3.3) and the conciseness evaluation (Section 3.3.4). For privacy purposes, we anonymize the IP addresses and the agent IDs in the presented queries.

A.2.1 Multi-Step Attack Behaviors

Query A.1: AIQL query for $a_1$

```
agentid = XXX // Linux web server
(at "4/25/2016")
proc p2 start proc p1["%telnet%"] as evt1
proc p1 start ip ipp[srcport=4444 || dstport = 4444] as evt2
proc p3 read file f2["%trojan.exe"] as evt3
with evt1 before evt2, evt2 before evt3
return distinct p2, p1, ipp.srcip, ipp.srcport, ipp.dstip, ipp.dstport, p3, f2
```

Query A.2: AIQL query for $a_2$

```
agentid = XXX // Windows client
(at "4/25/2016")
proc p1 write file f1["%trojan%"] as evt1
proc p2["%trojan%" start ip ipp [dstport = 4444]] as evt2
proc p2["%trojan%" read||write ip ipp2 [dstport = 4444]] as evt3
with evt1 before evt2, evt2 before evt3
return distinct p1, f1, p2, ipp, ipp2
```

Query A.3: AIQL query for $a_3$

```
agentid = XXX // Windows client
(at "4/25/2016")
proc p1["%trojan%"] start proc p2["%notepad%"] as evt1
proc p2 start ip ipp [dstport = 4444] as evt2
proc p2 read||write ip ipp2 [dstport = 4444] as evt3
with evt1 before evt2, evt2 before evt3
return distinct p1, p2, ipp.dstip, ipp.dstport, ipp2
```
agentid = XXX // Windows domain controller
(at "4/25/2016")
proc p1 read ip ipp as evt1
proc p1 write file f1["%pwdump7.exe%"||"%libeay32.dll%"] as evt2
proc p2 execute file f1 as evt3
with evt1 before evt2, evt2 before evt3
return distinct p1, ipp.dstip, ipp.dstport, f1, p2

Query A.4: A1QL query for a4

agentid = XXX // Database server
(at "5/20/2016")
proc p1 write file f1["%db.bak%"] as evt1
proc p2["%powershell%"] start ip ipp as evt2
proc p2["%powershell%"] read file f2["%db.bak%"] as evt3
proc p2 write ip ipp as evt4
with evt2 after evt1, evt3 after evt2, evt4 after evt3
return distinct p1, f2, p2, ipp

Query A.5: A1QL query for a5

A.2.2 Dependency Tracking Behaviors

agentid = XXX
(at "4/8/2016")
return distinct f1, p1, f2, p2, f3

Query A.6: A1QL query for d1

agentid = XXX
(at "4/7/2016")
return distinct f1, p1, f2, p2, f3

Query A.7: A1QL query for d2
Query A.8: AiQL query for d3

A.2.3 Real-World Malware Behaviors

Query A.9: AiQL query for v1

Query A.10: AiQL query for v2

Query A.11: AiQL query for v3
proc p2 execute file f1 as evt2
proc p2 start proc p3["%netsh%"] as evt3
with evt1 before evt2, evt2 before evt3
return distinct p1, f1, p2, p3

Query A.12: AiQL query for v4

agentid = XXX
(at "4/30/2016")
proc p1["%virus%7dd%"] start proc p2 as evt1
proc p1 execute file f1["%iexplore%"] as evt2
proc p2 read||write ip ipp as evt3
with evt1 before evt3, evt2 before evt3
return distinct p1, p1.pid, p2, p2.pid, f1, ipp

Query A.13: AiQL query for v5

A.2.4 Abnormal System Behaviors

agentid = XXX
(from "12/08/2015 00:00:00" to "12/12/2015 23:59:59")
proc p read file f["%/.lesshst" || "%/.viminfo" || "%/.bash_history" || "%/.pgadmin_histoqueries"] as evt
return distinct p, f, evt.starttime, evt.endtime

Query A.14: AiQL query for s1

agentid in (XXX1, XXX2, XXX3)
(from "12/08/2015 00:00:00" to "12/12/2015 23:59:59")
proc p[!"%apache%"] start ip ipp[srcport = 80] as evt
return distinct p, ipp, evt.agentid

Query A.15: AiQL query for s2

agentid in (XXX1, XXX2, XXX3)
(from "12/08/2015 00:00:00" to "12/12/2015 23:59:59")
proc p read ip ipp
return p, ipp, count

Query A.16: AiQL query for s3
A.3 Selected SQL, Neo4j Cypher, and Splunk SPL Queries

We present a few selected SQL, Neo4j Cypher, and Splunk SPL queries to demonstrate that these query languages are cumbersome to use, and that our AiQL language significantly improves the query conciseness and is much more convenient in specifying complex attack behaviors. For a complete list of queries used in our case study
(Section 3.3.2), performance evaluation (Section 3.3.3), and conciseness evaluation (Section 3.3.4), please refer to our project website [2].

A.3.1 AIQL, SQL, and Neo4j Cypher Queries for c4-8

Query A.20: AIQL query for c4-8
and evt2.agentid in (XXX)
and (evt2.starttime >= 1492574400000 and evt2.starttime <= 1492660799999)
/* evt3 */
and (pp2_f1.path ilike '%hwvun.vbs')
and (evt3.srclid = __sub1_dup.id and evt3.dstid = f1.id and evt3.path = pp2_f1.id)
and {((evt3.optype = 'Access_File' or evt3.optype = 'Open_File_Descriptor') and evt3.
  access = 'Write_Content'))
and evt3.agentid in (XXX)
and (evt3.starttime >= 1492574400000 and evt3.endtime <= 1492660799999)
/* evt4 */
and (p3.exe_name ilike '%cscript.exe')
and (evt4.srclid = p3.id and evt4.dstid = __obj2_dup.id and evt4.path = pp3___obj2_dup.id)
  access = 'Read_Content'))
and evt4.agentid in (XXX)
and (evt4.starttime >= 1492574400000 and evt4.endtime <= 1492660799999)
/* evt5 */
and (pp4_f2.path ilike '%sbblv.exe')
and (evt5.srclid = __sub3_dup.id and evt5.dstid = f2.id and evt5.path = pp4_f2.id)
and {((evt5.optype = 'Access_File' or evt5.optype = 'Open_File_Descriptor') and evt5.
  access = 'Write_Content'))
and evt5.agentid in (XXX)
and (evt5.starttime >= 1492574400000 and evt5.endtime <= 1492660799999)
/* evt6 */
and (p4.exe_name ilike '%sbblv.exe')
and (evt6.srclid = __sub4_dup.id and evt6.dstid = p4.id)
and (evt6.optype = 'Start_Processlet')
and evt6.agentid in (XXX)
and (evt6.starttime >= 1492574400000 and evt6.starttime <= 1492660799999)
/* evt7 */
and (i2.dstip = 'XXX.129')
and (evt7.srclid = __sub5_dup.id and evt7.dstid = i2.id)
and (evt7.optype = 'Start_IP_Connection')
and evt7.agentid in (XXX)
and (evt7.starttime >= 1492574400000 and evt7.endtime <= 1492660799999)
/* attribute relationships */
and p1.id = __sub0_dup.id
and p2.id = __sub1_dup.id
and f1.id = __obj2_dup.id
and p3.id = __sub3_dup.id
and p3.id = __sub4_dup.id
Query A.21: SQL query for c4-8

```sql
match (p1:Processlet) -[evt1:IpchannelEvent]-> (i1:Ipchannel),
(__sub0_dup:Processlet) -[evt2:ProcessletEvent]-> (p2:Processlet),
(__sub1_dup:Processlet) -[evt3:FileEvent]-> (f1:File),
(pp2_f1:Path),
(p3:Processlet) -[evt4:FileEvent]-> (__obj2_dup:File),
(pp3_obj2_dup:Path),
(__sub3_dup:Processlet) -[evt5:FileEvent]-> (f2:File),
(pp4_f2:Path),
(__sub4_dup:Processlet) -[evt6:ProcessletEvent]-> (p4:Processlet),
(__sub5_dup:Processlet) -[evt7:IpchannelEvent]-> (i2:Ipchannel)
where (p1.exe_name =~ '(?i).*sqlservr.exe') /* evt1 */
and (i1.dstip = 'XXX.130')
and (evt1.optype = 'Start_IP_Connection' or ((evt1.optype = 'Transfer_IP_Data' or evt1.
          optype = 'Open_IP_Connection_Descriptor') and evt1.access = 'Read_Content') or ((evt1.
          optype = 'Transfer_IP_Data' or evt1.optype = 'Open_IP_Connection_Descriptor') and evt1
          .access = 'Write_Content'))
and evt1.agentid in [XXX]
and (evt1.starttime >= 1492574400000 and evt1.endtime <= 1492660799999)
/* evt2 */
and (p2.exe_name =~ '(?i).*cmd.exe')
and (evt2.optype = 'Start_Processlet')
and evt2.agentid in [XXX]
and (evt2.starttime >= 1492574400000 and evt2.starttime <= 1492660799999)
/* evt3 */
and (pp2_f1.path =~ '(?i).*hwvun.vbs')
and (evt3.path = pp2_f1.id)
and (((evt3.optype = 'Access_File' or evt3.optype = 'Open_File_Descriptor') and evt3.
        access = 'Write_Content'))
and evt3.agentid in [XXX]
and (evt3.starttime >= 1492574400000 and evt3.endtime <= 1492660799999)
/* evt4 */
```
and (p3.exe_name =~ '(?i).*cscript.exe')
and (evt4.path = pp3_obj2_dup.id)
access = 'Read_Content'))
and evt4.agentid in [XXX]
and (evt4.starttime >= 1492574400000 and evt4.endtime <= 1492660799999)
/* evt5 */
and (pp4_f2.path =~ '(?i).*sbblv.exe')
and (evt5.path = pp4_f2.id)
and (((evt5.optype = 'Access_File' or evt5.optype = 'Open_File_Descriptor') and evt5.
access = 'Write_Content'))
and evt5.agentid in [XXX]
and (evt5.starttime >= 1492574400000 and evt5.endtime <= 1492660799999)
/* evt6 */
and (p4.exe_name =~ '(?i).*sbblv.exe')
and (evt6.optype = 'Start_Processlet')
and evt6.agentid in [XXX]
and (evt6.starttime >= 1492574400000 and evt6.starttime <= 1492660799999)
/* evt7 */
and (i2.dstip = 'XXX.129')
and (evt7.optype = 'Start_IP_Connection')
and evt7.agentid in [XXX]
and (evt7.starttime >= 1492574400000 and evt7.endtime <= 1492660799999)
/* attribute relationships */
and p1.id = __sub0_dup.id
and p2.id = __sub1_dup.id
and f1.id = __obj2_dup.id
and p3.id = __sub3_dup.id
and p3.id = __sub4_dup.id
and p4.id = __sub5_dup.id
/* temporal relationships */
and evt1.starttime < evt2.starttime
and evt2.starttime < evt3.starttime
and evt3.starttime < evt4.starttime
and evt4.starttime < evt5.starttime
and evt5.starttime < evt6.starttime
and evt6.starttime < evt7.starttime
return distinct p1.exe_name, i1.dstip, p2.exe_name, pp2_f1.path, p3.exe_name, pp4_f2.path,
p4.exe_name, i2.dstip;

Query A.22: Neo4j Cypher query for c4-8
A.3.2 AIQL, SQL, and Splunk SPL Queries for d3

(at "08/27/2016")

forward: proc p1["%/bin/cp%",agentid = XXX1] ->[write] file f1["%malicious%"] <-[read]

return distinct p1, f1, p2, p3, f2

---

Query A.23: AIQL query for d3

```sql
select distinct p1.exe_name, pp0_f1.path, p2.exe_name, p3.exe_name, pp4_f2.path
from processlet p1, file f1, path pp0_f1, fileevent evt0, processlet p2, file __obj0_dup,
    path pp1___obj0_dup, fileevent evt1, processlet __sub1_dup, ipchannel conn_ip0,
    ipchannlevent cevt1, processlet p3, ipchannel conn_ip1, ipchannlevent cevt2,
    processlet __sub2_dup, file f2, path pp4_f2, fileevent evt3
where (p1.exe_name ilike '%/bin/cp%' and p1.agentid = XXX1) /* evt0 */
    and (pp0_f1.path ilike '%malicious%')
    and (evt0.srcid = p1.id and evt0.dstid = f1.id and evt0.path = pp0_f1.id)
    and (evt0.agentid = XXX1)
    and (((evt0.optype = 'Access_File' or evt0.optype = 'Open_File_Descriptor') and evt0.
        access = 'Write_Content'))
    and (evt0.starttime >= 1472270400000 and evt0.endtime <= 1472356799999)
    and (p2.exe_name ilike '%apache%') /* evt1 */
    and (pp1___obj0_dup.path ilike '%malicious%')
    and (evt1.srcid = p2.id and evt1.dstid = __obj0_dup.id and evt1.path = pp1___obj0_dup.id)
    and (evt1.agentid = XXX1)
    and (((evt1.optype = 'Access_File' or evt1.optype = 'Open_File_Descriptor') and evt1.
        access = 'Read_Content'))
    and (evt1.starttime >= 1472270400000 and evt1.endtime <= 1472356799999)
    and (__sub1_dup.exe_name ilike '%apache%') /* cevt1 */
    and (cevt1.srcid = __sub1_dup.id and cevt1.dstid = conn_ip0.id)
    and (cevt1.agentid = XXX1)
    and (((cevt1.optype = 'Open_IP_Connection_Descriptor' or cevt1.optype = 'Close_IP_Connection_Descriptor'))
    and (cevt1.starttime >= 1472270400000 and cevt1.endtime <= 1472356799999)
    and (p3.agentid = XXX2) /* cevt2 */
    and (cevt2.srcid = p3.id and cevt2.dstid = conn_ip1.id)
    and (cevt2.agentid = XXX2)
```
and ((cevt2.optype = 'Open_IP_Connection_Descriptor' or cevt2.optype = 'Close_IP_Connection_Descriptor'))
and (cevt2.starttime >= 1472270400000 and cevt2.endtime <= 1472356799999)
and (__sub2_dup.agentid = XXX2) /* evt3 */
and (pp4_f2.path ilike '%malicious%')
and (evt3.srcid = __sub2_dup.id and evt3.dstid = f2.id and evt3.path = pp4_f2.id)
and (evt3.agentid = XXX2)
and (((evt3.optype = 'Access_File' or evt3.optype = 'Open_File_Descriptor') and evt3.
  access = 'Write_Content'))
and (evt3.starttime >= 1472270400000 and evt3.endtime <= 1472356799999)
and __obj0_dup.id = f1.id
and __sub1_dup.id = p2.id
and __sub2_dup.id = p3.id
and conn_ip0.srcip = conn_ip1.dstip
and conn_ip0.dstip = conn_ip1.srcip
and conn_ip0.srcport = conn_ip1.dstport
and conn_ip0.dstport = conn_ip1.srcport
and conn_ip0.protocol = conn_ip1.protocol
and evt0.starttime_seq < evt1.starttime_seq
and abs(cevt1.starttime - cevt2.starttime) < 1000

Query A.24: SQL query for d3

source=fileEvent AND operation="write" AND agentid=XXX1 AND day="8/27/2016" /* evt0 */
| join file_id max=0 [search source=fileInfo AND file_name="*malicious*"]
| join process_id max=0 [search source=processInfo AND process_name="*/bin/cp*"]
| rename file_id as f1_id
| rename process_name as p1_name
| rename file_name as f1_name
| rename starttime as t1
| join max=0 [search source=fileEvent AND operation="read" AND agentid=XXX1 AND day="
  8/27/2016" /* evt1 */
| join file_id max=0 [search source=fileInfo AND file_name="*malicious*"]
| join process_id max=0 [search source=processInfo AND process_name="*/apache*"]
| rename process_id as p2_id
| rename file_id as f1_id_2
| rename process_name as p2_name
| rename starttime as t2
]
| join max=0 [search source=networkEvent AND (operation="open" OR operation="close") AND agentid=XXX1 AND day="8/27/2016" /* cevt1 */ |
| join ip_id max=0 [search source=processInfo AND process_name="*apache*"] |
| rename process_id as p2_id_2 |
| rename src_ip as conn_ip0_srcip |
| rename dst_ip as conn_ip0_dstip |
| rename src_port as conn_ip0_srcport |
| rename dst_port as conn_ip0_dstport |
| rename protocol as conn_ip0_protocol |
| rename starttime as t3 |
|
| join max=0 [search source=networkEvent AND (operation="open" OR operation="close") AND agentid=XXX2 AND day="8/27/2016" /* cevt2 */ |
| join process_id max=0 [search source=processInfo] |
| join ip_id max=0 [search source=networkInfo ] |
| rename process_id as p3_id |
| rename process_name as p3_name |
| rename process_id as p2_id_2 |
| rename src_ip as conn_ip1_srcip |
| rename dst_ip as conn_ip1_dstip |
| rename src_port as conn_ip1_srcport |
| rename dst_port as conn_ip1_dstport |
| rename protocol as conn_ip1_protocol |
| rename starttime as t4 |
|
| join max=0 [search source=fileEvent AND operation="write" AND agentid=XXX2 AND day="8/27/2016" /* evt3 */ |
| join file_id max=0 [search source=fileInfo AND file_name="*malicious*"] |
| join process_id max=0 [search source=processInfo] |
| rename process_id as p3_id_2 |
| rename file_name as f2_name |
| rename starttime as t5 |
|
| where t1 < t2 AND t2 < t3 AND t3 < t4 AND t4 < t5 AND abs(t3-t4) < 1000 |
| where f1_id = f1_id_2, p2_id = p2_id_2, p3_id = p3_id_2, conn_ip0_srcip = conn_ip1_dstip, conn_ip0_dstip = conn_ip1_srcip, conn_ip0_srcport = conn_ip1_dstport, conn_ip0_dstport = conn_ip1_srcport, conn_ip0_protocol = conn_ip1_protocol |
| dedup p1_name, f1_name, p2_name, p3_name, f2_name |
| table p1_name, f1_name, p2_name, p3_name, f2_name |
Query A.25: Splunk SPL query for $d^3$
Appendix B

SAQL

B.1 SAQL Queries in Case Study

We present the 17 SAQL queries that we constructed for the case study for detecting four major types of attack behaviors (Section 4.3.2). For privacy purposes, we anonymize the IP addresses and the agent IDs in the presented queries.

B.1.1 APT Attack

1 proc p1["%smtp%"] read||write ip i1[srcip="XXX" && srcport=25 && protocol=6] as evt1[agentid = XXX] // mail server, SMTP connection from the router to the mail server
2 proc p2["%imap%"] read||write ip i2[srcip="XXX" && srcport=143 && dstip="XXX" && dstport =51962 && protocol=6] as evt2[agentid = XXX] // mail server, IMAP connection from the mail server to the client
3 proc p3["%outlook%"] read||write ip i3[srcip="XXX" && srcport=51960 && dstip="XXX" && dstport=143 && protocol=6] as evt3[agentid = XXX] // windows client, client’s outlook reads email data
4 with evt1 -> evt2 -> evt3
5 return p1, i1, p2, i2, p3, i3, evt1.starttime, evt2.starttime, evt3.starttime

Query B.1: apt-c1

1 agentid = XXX // Windows client
2 proc p1["%outlook.exe"] start proc p2["%excel.exe"] as evt1 // outlook starts excel
3 proc p2 start proc p3["%java.exe"] as evt2 // excel starts malware (java) process
proc p3 start proc p4["%notepad.exe"] as evt3 // malware (java) starts notepad
proc p4 read||write ip i1["XXX"] as evt4 // notepad connects to the attacker host
with evt1 -> evt2 -> evt3 -> evt4
return p1, p2, p3, p4, i1, evt1.starttime, evt2.starttime, evt3.starttime, evt4.starttime

Query B.2: apt-c2

agentid = XXX // Windows domain controller
proc p1 read || write ip i1[srcport=445 && dstip="XXX"] as evt1 // attacker penetrates to the DC host using psexec protocol
proc p2["%powershell.exe"] write file f1["%gsecdump%"] as evt2 // attacker transfers the DB cracking tool gsecdump.exe
proc p3["%cmd.exe"] start proc p4["%gsecdump%"] as evt3 // attacker executes gsecdump.exe to dump DB administrator credentials
with evt1 -> evt2 -> evt3
return p1, i1, p2, f1, p3, p4, evt1.starttime, evt2.starttime, evt3.starttime

Query B.3: apt-c3

agentid = XXX // Database server
proc p1["%sqlservr.exe"] read||write ip i1[srcip="XXX" && srcport=1433 && dstip="XXX" && dstport=52038 && protocol=6] as evt1 // attacker connects to the SQL server using DB administrator credentials
proc p2 start proc p3["%cmd.exe"] as evt2 // SQL server starts cmd
proc p3 read || write file f1["%hwvun.vbs"] as evt3 // cmd writes malware sbblv.exe
proc p4["%cscript.exe"] write file f2["%sbblv.exe"] as evt4
proc p4["%sbblv.exe"] start ip i2[srcip="XXX" && srcport=61060 && dstip="XXX" && dstport=445 && protocol=6] as evt5 // malware connects back to the attacker host
with evt1 -> evt2 -> evt3 -> evt4 -> evt5
return p1, i1, p2, f1, p3, f2, p4, i2, evt1.starttime, evt2.starttime, evt3.starttime,
    evt4.starttime, evt5.starttime

Query B.4: apt-c4

agentid = XXX // Database server
proc p1["%cmd.exe"] start proc p2["%osql.exe"] as evt1 // attacker executes osql.exe on the sql server
proc p3["%sqlservr.exe"] write file f1["%backup1.dmp"] as evt2 // attacker dumps the DB content
proc p4["%sbblv.exe"] read file f1 as evt3 // malware reads the dump
proc p4 read || write ip i1[dstip="XXX"] as evt4 // malware transfers the dump to the attacker

return p1, i1, p2, f1, p3, f2, p4, i2, evt1.starttime, evt2.starttime, evt3.starttime,
    evt4.starttime, evt5.starttime
with evt1 -> evt2 -> evt3 -> evt4
return p1, p2, p3, f1, p4, i1, evt1.starttime, evt2.starttime, evt3.starttime, evt4.starttime, evt4.amount

Query B.5: apt-c5

proc p1["%excel.exe"] start proc p2 as evt #time(5 second)
state ss {
    set_proc := set(p2.exe_name)
} group by p1, evt.agentid
invariant[100][offline] {
    a := empty_set
    a = a union ss.set_proc
}
alert |ss.set_proc diff a| > 0
return p1, evt.agentid, ss.set_proc

Query B.6: apt-c2-invariant

agentid = XXX // Database server
proc p write ip i as evt #time(10 min)
state[3] ss {
    avg_amount := avg(evt.amount)
} group by p
alert (ss[0].avg_amount > (ss[0].avg_amount + ss[1].avg_amount + ss[2].avg_amount) / 3) &&
    (ss[0].avg_amount > 10000)
return p, ss[0].avg_amount, ss[1].avg_amount, ss[2].avg_amount

Query B.7: apt-c5-timeseries

agentid = XXX// Database server
proc p write ip i as evt #time(1 min)
state ss {
    avg_amount := avg(evt.amount)
} group by p
cluster(points=all(ss.avg_amount), distance="ed", method="DBSCAN(1000, 5)")
alert cluster.outlier && ss.avg_amount > 1000000
return p, ss.avg_amount

Query B.8: apt-c5-outlier
B.1.2 SQL Injection Attack

agentid = XXX // sqlserver host

proc p["%sqlservr.exe"] read || write ip i as evt #time(10 min)

state ss {
  amt := sum(evt.amount)
} group by i.dstip

cluster(points=all(ss.amt), distance="ed", method="DBSCAN(100000, 5)")

alert cluster.outlier && ss.amt > 1000000

return i.dstip, ss.amt

Query B.9: sql-injection

B.1.3 Bash Shellshock Command Injection Attack

proc p1["%apache2%"] start proc p2 as evt #time(10 s)

state ss {
  set_proc := set(p2.exe_name)
} group by p1

invariant[10][offline] {
  a := empty_set // invariant init
  a = a union ss.set_proc //invariant update
}

alert |ss.set_proc diff a| > 0

return p1, ss.set_proc

Query B.10: shellshock

B.1.4 Suspicious System Behaviors

proc p["%dropbox%"] start ip i as evt

return p, i, evt.agentid, evt.starttime, evt.endtime

Query B.11: dropbox

proc p read || write file f["%.viminfo" || ".bash_history" || ".zsh_history" || ".lesshst" || ".pgadmin_histoqueries" || ".mysql_history"] as evt

return p, f, evt.agentid, evt.starttime, evt.endtime
Query B.12: command-history

1. `proc p read || write file f[/etc/passwd] as evt`
2. `return p, f, evt.agentid, evt.starttime, evt.endtime`

Query B.13: password

1. `proc p write file f[/var/log/wtmp || /var/log/lastlog] as evt`
2. `return p, f, evt.agentid, evt.starttime, evt.endtime`

Query B.14: login-log

1. `proc p read || write file f[%.ssh/id_rsa || %.ssh/id_dsa] as evt`
2. `return p, f, evt.agentid, evt.starttime, evt.endtime`

Query B.15: sshkey

1. `proc p read || write file f[bustype = "USB"] as evt`
2. `return p, f, evt.agentid`

Query B.16: usb

1. `proc p start ip ipp #time(1 min)`
2. `group by p`
3. `alert freq > 100`
4. `return p, count(ipp) as freq`

Query B.17: ipfreq
Bibliography


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


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


[21] Demo video of the SAQL system. [https://youtu.be/3S7D5jVoR2c](https://youtu.be/3S7D5jVoR2c).


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


[59] WSO2 clustering and deployment guide. https://docs.wso2.com/display/CLUSTER44x/


[100] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.


[129] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, R Sekar, and VN Venkatakrishnan. HOLMES: real-time APT detection through correlation


