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

Peng Gao, *Princeton University;* Xusheng Xiao, *Case Western Reserve University;*
Ding Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, and Chung Hwan Kim, *NEC Laboratories America, Inc.;* Sanjeev R. Kulkarni and Prateek Mittal, *Princeton University*

https://www.usenix.org/conference/usenixsecurity18/presentation/gao-peng

This paper is included in the Proceedings of the 27th USENIX Security Symposium.

August 15–17, 2018 • Baltimore, MD, USA

ISBN 978-1-939133-04-5

Open access to the Proceedings of the 27th USENIX Security Symposium is sponsored by USENIX.

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

Peng Gao[1] Xusheng Xiao[2] Ding Li[3] Zhichun Li[3] Kangkook Jee[3]
Zhenyu Wu[3] Chung Hwan Kim[3] Sanjeev R. Kulkarni[1] Prateek Mittal[1]
[1]*Princeton University* [2]*Case Western Reserve University* [3]*NEC Laboratories America, Inc.*
[1]{*pgao,kulkarni,pmittal*}*@princeton.edu* [2]*xusheng.xiao@case.edu* [3]{*dingli,zhichun,kjee,adamwu,chungkim*}*@nec-labs.com*

Abstract

Recently, advanced cyber attacks, which consist of a sequence of steps that involve many vulnerabilities and hosts, compromise the security of many well-protected businesses. This has led to the solutions that ubiquitously monitor system activities in each host (big data) as a series of events, and search for anomalies (abnormal behaviors) for triaging risky events. Since fighting against these attacks is a time-critical mission to prevent further damage, these solutions face challenges in incorporating *expert knowledge* to perform *timely anomaly detection* over the large-scale provenance data.

To address these challenges, we propose a novel 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 behaviors based on the specified anomalies. To facilitate the task of expressing anomalies based on expert knowledge, our system provides a domain-specific query language, SAQL, which allows analysts to express models for (1) *rule-based anomalies*, (2) *time-series anomalies*, (3) *invariant-based anomalies*, and (4) *outlier-based anomalies*. We deployed our system in NEC Labs America comprising 150 hosts and evaluated it using 1.1TB of real system monitoring data (containing 3.3 billion events). Our evaluations on a broad set of attack behaviors and micro-benchmarks show that our system has a low detection latency (<2s) and a high system throughput (110,000 events/s; supporting ∼4000 hosts), and is more efficient in memory utilization than the existing stream-based complex event processing systems.

1 Introduction

Advanced cyber attacks and data breaches plague even the most protected companies [9, 16, 14, 23, 11]. The recent massive Equifax data breach [11] has exposed the sensitive personal information of 143 million US customers. Similar attacks, especially in the form of advanced persistent threats (APT), are being commonly observed. These attacks consist of *a sequence of steps* across *many hosts* that exploit different types of vulnerabilities to compromise security [25, 2, 1].

To counter these attacks, approaches based on *ubiquitous system monitoring* have emerged as an important solution for actively searching for possible anomalies, then to quickly triage the possible significant risky events [63, 64, 52, 40, 62, 74, 73, 68]. System monitoring observes *system calls* at the kernel level to collect information about system activities. The collected data from system monitoring facilitates the detection of abnormal system behaviors [39, 66].

However, these approaches face challenges in detecting multiple types of anomalies using system monitoring data. First, fighting against attacks such as APTs is a time-critical mission. As such, we need a *real-time* anomaly detection tool to search for a "needle in a haystack" for preventing additional damage and for system recovery. Second, models derived from data have been increasingly used in detecting various types of risky events [66]. For example, system administrators, security analysts and data scientists have extensive domain knowledge about the enterprise, including expected system behaviors. A key problem is how we can provide a real-time tool to detect anomalies while *incorporating the knowledge* from system administrators, security analysts and data scientists? Third, system monitoring produces huge amount of daily logs (∼50GB for 100 hosts per day) [69, 88]. This requires *efficient* real-time data analytics on the large-scale provenance data.

Unfortunately, none of the existing stream-based query systems and anomaly detection systems [91, 51, 59, 68] provide a comprehensive solution that addresses all these three challenges. These systems focus on specific anomalies and are optimized for general purpose data streams, providing limited support for users to spec-

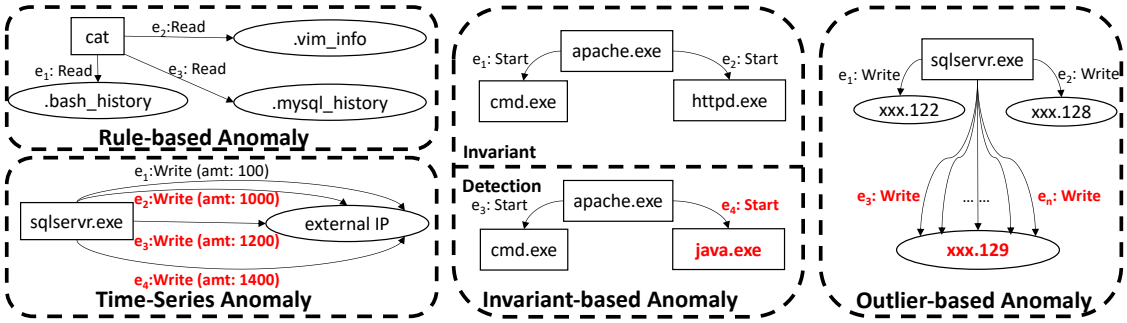


Figure 1: Major types of abnormal system behaviors ($e_1, \ldots, e_n$ are shown in ascending temporal order.)

ify anomaly models by incorporating domain knowledge from experts.

Contributions: We design and build a novel stream-based real-time query system. Our system takes as input a real-time event feed aggregated from multiple hosts in an enterprise, and provides an anomaly query engine. The query engine provides a novel interface for users to submit anomaly queries using our *domain-specific language*, and checks the events against the queries to detect anomalies in real-time.

Language: To facilitate the task of expressing anomalies based on domain knowledge of experts, our system provides a domain-specific query language, *Stream-based Anomaly Query Language (*SAQL*)*. 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 anomalies*; (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 anomalies*, *invariant-based anomalies*, and *outlier-based anomalies* (more details in Section 2.2). The specified models in SAQL are checked using *continuous queries* over unbounded streams of system monitoring data [51], which report the detected anomalies continuously.

Rule-based anomalies allow system experts to specify rules to detect known attack behaviors or enforce enterprise-wide security policies. Figure 1 shows an example rule-based anomaly, where a process (`cat`) accesses multiple command log files in a relatively short time period, indicating an external user 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 focus on general data streams [91, 59, 30, 42]. Figure 1 shows a time-series anomaly, where a process (`sqlservr.exe`) transfers abnormally large amount of data starting from e_2. To facilitate the detection of such anomalies, SAQL provides constructs for *sliding windows* that break the continuous data stream into fragments with common aggregation functions (*e.g.,* `count`, `sum`, `avg`). Additionally, SAQL provides constructs to define *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) [55].

Built upon the states of sliding windows, SAQL provides high-level constructs to facilitate the specification of invariant-based and outlier-based anomalies. Invariant-based anomalies capture the invariants during training periods as models, and use the models later to detect anomalies. Figure 1 shows an invariant-based anomaly, where a process (`apache.exe`) starts an abnormal process (`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 users to combine both states of windows and invariants learned under normal operations to detect more types of abnormal system behaviors.

Outlier-based anomalies allow users to identify abnormal system behavior through peer comparison, *e.g.,* finding outlier processes by comparing the abnormal processes with other peer processes. Figure 1 shows an outlier-based anomaly, where a process (`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 a point and compute clusters to identify outliers. The flexibility and extensibility introduced by SAQL allows users to use various clustering algorithms for different deployed environments.

Execution Engine: We build the query engine on top of Siddhi [20] 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 [20, 12, 4], 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 [20, 12, 51, 4].

Deployment and Evaluation: We built the whole SAQL system (around 50,000 lines of Java code) based on the existing system-level monitoring tools (i.e., auditd [15] and ETW [13]) and the existing stream management system (i.e., Siddhi [20]). We deployed the system in NEC Labs America comprising 150 hosts. We performed a broad set 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 ∼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. All the evaluation queries are available on our *project website [19]*.

Table 1: Representative attributes of system entities

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

2 Background and Examples

In this section, we first present the background on system monitoring and then show SAQL queries to demonstrate the major types of anomaly models supported by our system. 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.

2.1 System Monitoring

System monitoring data represents various system activities in the form of events along with time [63, 64, 52, 60]. 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 [28]. 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 [63, 64, 52, 60]. 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 mainstream operating systems (Windows, Linux and OS X), system entities in most cases are files, network connections and processes [63, 64, 52, 60]. 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 ⟨*subject, operation, object*⟩. We categorize events into three types according to the type of their object entities, namely *file events*, *process events*, and *network connection events*.

Entities and events have various attributes (Tables 1 and 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.

Table 2: Representative attributes of system events

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

2.2 SAQL Queries for Anomalies

We next present how to use SAQL as a unified interface to specify various types of abnormal system behaviors.

Rule-based Anomaly: Advanced cyber attacks typically include a series of steps that exploit vulnerabilities across multiple systems for stealing sensitive information [2, 1]. Query 1 shows a SAQL query for describing an attack step that reads external network (`evt1`), downloads a database cracking tool `gsecdump.exe` (`evt2`), and executes (`evt3`) it to obtain database credentials. It also specifies these events should occur in 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.dst_ip, f1 -> f1.name
```

Query 1: A rule-based SAQL query

Time-Series Anomaly: 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 [24, 26], which often indicates a data exfiltration. Query 2 shows a SAQL query that monitors 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 the moving average to detect spikes of network data transfers (Line 5). In the query, `ss[0]` means the state of the current window while `ss[1]` and `ss[2]` represent the states of the two past windows respectively (`ss[2]` occurs earlier than `ss[1]`). Existing stream query systems and anomaly systems [51, 59, 30] lack the expressiveness of stateful computation in sliding windows to support such anomaly models.

```
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 2: A time-series SAQL query

Invariant-based Anomaly: Invariant-based anomalies 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 [35]. Query 3 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 is 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.

```
1 proc p1["%apache.exe"] start proc p2 as evt #time(10
      s)
2 state ss {
3   set_proc := set(p2.exe_name)
4 } group by p1
5 invariant[10][offline] {
6   a := empty_set // invariant init
7   a = a union ss.set_proc //invariant update
8 }
9 alert |ss.set_proc diff a| > 0
10 return p1, ss.set_proc
```

Query 3: An invariant-based SAQL query

Outlier-based Anomaly: Outlier-based anomalies 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 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 a point and how the "distance" among these points should be computed ("`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 aggregation such as max/min are supported [51, 20, 12].

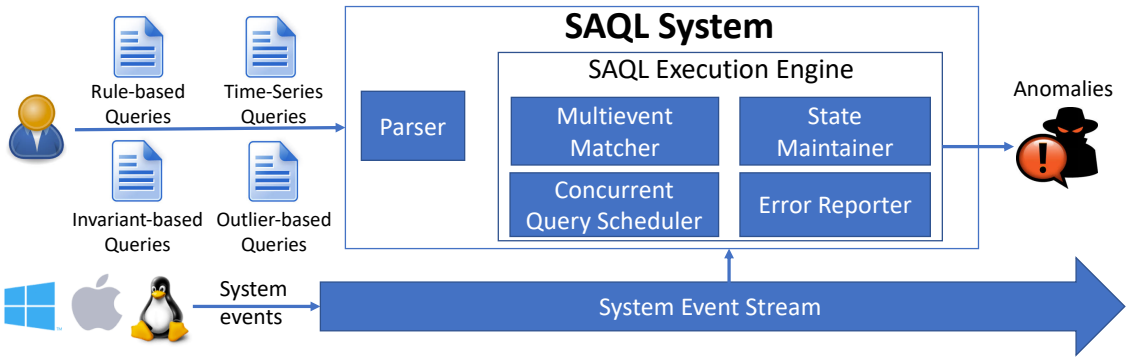


Figure 2: The architecture of SAQL system

```
agentid = 1 // 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 4: An outlier-based SAQL query using clustering

In addition to querying outliers through clustering, SAQL also supports querying through aggregation comparison. For example, in Query 4, 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 [38], and replacing the `alert` statement with `alert ss.amt>3*stddev(all(ss.amt))+avg(all(ss.amt))` gives 3-sigma-based outlier detection [38]. SAQL also supports querying outliers through sorting, and reports top sorted results as alerts, which is useful in querying most active processes or IP addresses.

3 System Overview and Threat Model

Figure 2 shows the SAQL system architecture. 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 [13], Linux Audit Framework [15], and DTrace [8]. The collected data is sent to the central server, forming an event stream.

The SAQL system takes SAQL queries from users, and reports the detected alerts over the event stream. The system consists of two components: (1) the language parser, implemented using ANTLR 4 [3], 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 the query execution and anomaly detection; (2) the execution engine, built upon Siddhi [20], monitors the data stream and reports the detected 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 5.2) to minimize the need for data copies; (4) the error reporter reports errors during the execution.

Threat Model: SAQL is a stream-based query system over system monitoring data, and thus we follow the threat model of previous works on system monitoring data [63, 64, 69, 68, 32, 50]. We assume that the system monitoring data collected from kernel space [15, 13] are not tampered, and that the kernel is trusted. Any kernel-level attack that deliberately compromises security auditing systems is beyond the scope of this work.

We do consider that insiders or external attackers have full knowledge of the deployed SAQL queries and the anomaly models. They can launch attacks with seemingly "normal" activities to evade SAQL's anomaly detection, and may hide their attacks by mimicking peer hosts' behaviors to avoid SAQL's outlier detection.

4 SAQL Language Design

SAQL is designed to facilitate the task of expressing anomalies based on the domain knowledge of experts. SAQL provides explicit constructs to specify system entities/events, as well as event relationships. This facilitates the specification of rule-based anomalies 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 anomalies, invariant-based anomalies, and outlier-based anomalies, which lack support from existing stream query systems and stream-based anomaly de-

tection systems. Grammar 1 shows the representative rules of SAQL. We omit the terminal symbols.

4.1 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 attribute relationships, which facilitates the specification of complex system behavioral rules.

Global Constraint: The ⟨*global_cstr*⟩ rule specifies the constraints for all event patterns (*e.g.,* `agentid = 1` in Query 4 specifies that all event patterns occur on the same host).

Event Pattern: The ⟨*evt_patt*⟩ rule specifies an event pattern, including the subject/object entity (⟨*entity*⟩), the event operation (⟨*op_exp*⟩), the event ID (⟨*evt*⟩), and the optional sliding window (⟨*wind*⟩). The ⟨*entity*⟩ rule consists of the entity type (file, process, network connection), the optional entity ID, and the optional attribute constraints expression (⟨*attr_exp*⟩). Logical operators (&&, ||, !) can be used in ⟨*op_exp*⟩ to form complex operation expressions (*e.g.,* `proc p read || write file f`). The ⟨*attr_exp*⟩ 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.,* `proc p[pid = 1 && name = "%chrome.exe"]`).

Sliding Window: The ⟨*wind*⟩ rule specifies the sliding windows for stateful computation. For example, `#time(10 min)` in Query 2 specifies a sliding window whose width is 10 minutes. An optional step size can be provided (*e.g.,* `#time(10 min)(1 min)` indicates a step size of 1 minute).

Event Temporal Relationship: The ⟨*temp_rel*⟩ rule specifies the temporal dependencies among event patterns. For example, `evt1->evt2->evt3` in Query 1 specifies that `evt1` occurs first, then `evt2`, and finally `evt3`. Finer-grained control of temporal distance can also be provided. For example, `evt1 ->[1-2 min] evt2 ->[1-2 min] 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 (⟨*alert*⟩) to specify the attribute dependency of event patterns (*e.g.,* `alert evt1.agentid = evt2.agentid && evt1.dst_id = evt2.src_id` for two event patterns `evt1` and `evt2` indicates that the two events occur at the same host and are "physically connected": the object entity of `evt1` is exactly the subject entity of `evt2`).

⟨*saql*⟩ ::= (⟨*global_cstr*⟩)* (⟨*evt_patt*⟩)+ ⟨*temp_rel*⟩? ⟨*state*⟩? ⟨*groupby*⟩? ⟨*alert*⟩? ⟨*return*⟩ ⟨*sortby*⟩? ⟨*top*⟩?

Data types:

⟨*num*⟩ ::= ⟨*int*⟩ | ⟨*float*⟩
⟨*val*⟩ ::= ⟨*int*⟩ | ⟨*float*⟩ | ⟨*string*⟩
⟨*val_set*⟩ ::= '(' ⟨*val*⟩ (',' ⟨*val*⟩)* ')'
⟨*id*⟩ ::= ⟨*letter*⟩(⟨*letter*⟩ | ⟨*digit*⟩)*
⟨*attr*⟩ ::= ⟨*id*⟩ ('[' ⟨*int*⟩ ']')? ('.' ⟨*id*⟩)?

Multievent pattern matching:

⟨*global_cstr*⟩ ::= ⟨*attr_exp*⟩
⟨*evt_patt*⟩ ::= ⟨*entity*⟩ ⟨*op_exp*⟩ ⟨*entity*⟩ ⟨*evt*⟩? ⟨*wind*⟩?
⟨*entity*⟩ ::= ⟨*entity_type*⟩ ⟨*id*⟩ ('[' ⟨*attr_exp*⟩']')?
⟨*op_exp*⟩ ::= ⟨*op*⟩
| '!'⟨*op_exp*⟩
| ⟨*op_exp*⟩ ('&&' | '||') ⟨*op_exp*⟩
| '(' ⟨*op_exp*⟩ ')'
⟨*evt*⟩ ::= 'as' ⟨*id*⟩ ('[' ⟨*attr_exp*⟩']')?
⟨*wind*⟩ ::= '#' ⟨*time_wind*⟩ | ⟨*length_wind*⟩
⟨*time_wind*⟩ ::= 'time' '(' ⟨*num*⟩ ⟨*time_unit*⟩')' ('['⟨*num*⟩ ⟨*time_unit*⟩']')?
⟨*length_wind*⟩ ::= 'length' '(' ⟨*int*⟩')'
⟨*attr_exp*⟩ ::= ⟨*attr*⟩ | ⟨*val*⟩
| ⟨*attr_exp*⟩ ⟨*bop*⟩ ⟨*attr_exp*⟩
| ⟨*attr_exp*⟩ ('&&' | '||') ⟨*attr_exp*⟩
| '!'⟨*attr_exp*⟩
| '(' ⟨*attr_exp*⟩ ')'
| ⟨*attr*⟩ 'not'? 'in' ⟨*val_set*⟩
| ⟨*agg_func*⟩ '(' ⟨*attr_exp*⟩ (',' ⟨*attr_exp*⟩)*')'
| ⟨*attr_exp*⟩ ⟨*set_op*⟩ ⟨*attr_exp*⟩
| '|' ⟨*attr_exp*⟩ '|'
| ⟨*peer_ref*⟩ '(' ⟨*attr_exp*⟩')'
⟨*temp_rel*⟩ ::= 'with' ⟨*id*⟩ (('->'|'<-') ('[' ⟨*num*⟩ '-' ⟨*num*⟩ ⟨*time_unit*⟩']')? ⟨*id*⟩)+

Stateful computation:

⟨*state*⟩ ::= ⟨*state_def*⟩ ⟨*state_inv*⟩? ⟨*state_cluster*⟩?
⟨*state_def*⟩ ::= 'state' ('[' ⟨*int*⟩ ']')? ⟨*id*⟩ '{' ⟨*state_field*⟩ ⟨*state_field*⟩*'}' ⟨*groupby*⟩
⟨*state_field*⟩ ::= ⟨*id*⟩ ':=' (⟨*agg_func*⟩ | ⟨*set_func*⟩) '(' ⟨*attr*⟩ ')' ⟨*groupby*⟩?
⟨*state_inv*⟩ ::= 'invariant' '[' ⟨*int*⟩ ']' '[' ⟨*train_type*⟩ ']'? '{' ⟨*inv_init*⟩+ ⟨*inv_update*⟩+ '}'
⟨*inv_init*⟩ ::= ⟨*id*⟩ ':=' (⟨*num*⟩|⟨*empty_set*⟩)
⟨*inv_update*⟩ ::= ⟨*id*⟩ '=' ⟨*attr_exp*⟩
⟨*state_cluster*⟩ ::= 'cluster' '(' ⟨*point_def*⟩ ',' ⟨*distance_def*⟩ ',' ⟨*method_def*⟩ ')'
⟨*point_def*⟩ ::= 'points' '=' ⟨*peer_ref*⟩ '(' ⟨*attr*⟩ (',' ⟨*attr*⟩)* ')'
⟨*distance_def*⟩ ::= 'distance' '=' ⟨*dist_metric*⟩
⟨*method_def*⟩ ::= 'method' '=' ⟨*cluster_method*⟩ '(' ⟨*num*⟩ (',' ⟨*num*⟩)* ')'

Alert condition checking:

⟨*alert*⟩ ::= 'alert' ⟨*attr_exp*⟩

Return and filters:

⟨*return*⟩ ::= 'return' ⟨*res_pair*⟩ (', ' ⟨*res_pair*⟩)*
⟨*res_pair*⟩ ::= ⟨*attr_exp*⟩ ('as' ⟨*id*⟩)?
⟨*groupby*⟩ ::= 'group by' ⟨*attr*⟩ (',' ⟨*attr*⟩)*
⟨*sortby*⟩ ::= 'sort by' ⟨*attr*⟩ (',' ⟨*attr*⟩)* ('asc' | 'desc')?
⟨*top*⟩ ::= 'top' ⟨*int*⟩

Grammar 1: Representative BNF grammar of SAQL

Context-Aware Syntax Shortcuts:

- *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: `name` for files, `exe_name` for processes, and `dst_ip` for network connections. For example, in Query 1, `file f1["%gsecdump .exe"]` is equivalent to `file f1[name="%gsecdump.exe"]`, and `return p1` is equivalent to `return p1.exe_name`; (2) `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`, `alert p1 = p2` is equivalent to `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 `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.

4.2 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 anomalies, invariant-based anomalies, and outlier-based anomalies.

State Block: The ⟨*state_def*⟩ 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 2). If not specified, only the state of the current window is stored by default (*e.g.,* Line 2 in Query 3). The ⟨*state_field*⟩ 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`). After specifying the state block, security analysts can then reference the state fields via the state ID to construct time-series anomaly models (*e.g.,* Line 5 in Query 2 specifies a three-period simple moving average (SMA) [55] time-series model to detect network spikes).

State Invariant: The ⟨*state_inv*⟩ 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 3 specifies an invariant `a` and trains it using the first 10 window results.

State Cluster: The ⟨*state_cluster*⟩ 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 using peer reference keywords ⟨peer_ref⟩ (*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 [56], DBSCAN [48], and hierarchical clustering [56]). For example, Line 6 in Query 4 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).

4.3 Alert Condition Checking

The ⟨*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 anomalies (*e.g.,* Line 5 in Query 2), invariant-based anomalies (*e.g.,* Line 9 in Query 3), and outlier-based anomalies (*e.g.,* Line 7 in Query 4). Note that in addition to the moving average detection logic specified in Query 2, the flexibility of SAQL also enables the specification of other well-known logics, such as 3-sigma rule [38] (*e.g.,* `alert ss.amt>3*stddev(all(ss.amt))+avg(all(ss.amt))`) and IQR rule [38] (*e.g.,* `alert ss.amt>1.5*iqr(all(ss.amt))+q3(all(ss.amt))`).

4.4 Return and Filters

The ⟨*report*⟩ rule specifies the desired attributes of the qualified events to return as results. Constructs such as `group by`, `sort by`, and `top` can be used for further result 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, Query 5 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
```

Query 5: Threshold-based IP Frequency Anomaly

5 SAQL Execution Engine

The SAQL execution engine in Figure 2 takes the event stream as input, executes the anomaly model contexts

generated by the parser, and reports the detected alerts. To make the system more scalable in supporting multiple concurrent queries, the 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 data copies of the stream.

5.1 Query Execution Pipeline

The query engine is built upon Siddhi [20], so that our SAQL can 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 leveraging 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 queries.

5.2 Concurrent Query Scheduler

The concurrent query scheduler in Figure 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 huge amount of daily logs [69, 88], 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 1 shows the scheduling algorithm:

Algorithm 1: Master-dependent-query scheme

```
Input: User submitted new SAQL query: newQ
       Map of concurrent master-dependent queries:
       M = {masQ_i → {depQ_ij}}
Output: Execution results of newQ
if M.isEmpty then
    return execAsMas(newQ, M);
else
    for masQ_i in M.keys do
        covQ = constructSemanticCover(masQ_i, newQ);
        if covQ ≠ null then
            if covQ ≠ masQ_i then
                replMas(masQ_i, covQ, M);
            addDep(covQ, newQ);
            return execDep(newQ, covQ);
    return execAsMas(newQ, M);

Function constructSemanticCover(masQ, newQ)
    if Both masQ and newQ define a single event pattern then
        if masQ and newQ share the same event type, operation
          type, and sliding window type then
            Construct the event pattern cover evtPattCovQ by
              taking the union of their attributes and agent IDs
              and the GCD of their window lengths;
            if Both masQ and depQ define states then
                if masQ and depQ have the same sliding
                  window length and masQ defines a super set
                  of state fields of depQ then
                    Construct the state cover stateCovQ by
                      taking the union of their state fields;
            return covQ by concatenating evtPattCovQ,
              stateCovQ, and the rest parts of masQ;
    return null;
Function execAsMas(newQ, M)
    Make newQ as a new master and execute it;
Function addDep(masQ, depQ, M)
    Add depQ to the dependencies of masQ;
Function replMas(oldMasQ, newMasQ, M)
    Replace the old master oldMasQ with the new master
      newMasQ and update dependencies;
Function execDep(depQ, masQ)
    if depQ == masQ then
        return execution results of masQ;
    else if Both masQ and depQ define states then
        if masQ and depQ have the same sliding window length
          and masQ defines a super set of state fields of depQ
          then
            Fetch the state aggregation results of masQ,
              enforce additional filters, and feed into the
              execution pipeline of depQ;
    else
        Fetch the matched events of masQ, enforce additional
          filters, and feed into the execution pipeline of depQ;
```

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.
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 1 are *constructSemanticCover*() and *execDep*(). The construction of a semantic cover requires that (1) the *masQ* and *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 the results sharing in two execution phases: event pattern matching and stateful computation: (1) if both *dep* and *masQ* define states and their sliding window lengths are the same, the engine fetches the state aggregate results of *masQ*; (2) otherwise, the engine fetches the matched events of *masQ* without its further state aggregate results. The engine then enforces additional filters and feed the filtered results into the rest of the execution pipeline of *depQ* for further execution.

6 Deployment and Evaluation

We deployed the SAQL system in NEC Labs America comprising 150 hosts (10 servers, 140 employee stations; generating around 3750 events/s). To evaluate the expressiveness of SAQL and the SAQL's overall effectiveness and efficiency, we first perform a series of attacks based on known exploits in the deployed environment and construct 17 SAQL queries to detect them. We further conduct a pressure test to measure the maximum performance that our system can achieve. Finally, we conduct a performance evaluation on a micro-benchmark (64 queries) 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). All the attack queries are available in Appendix, and all the micro-benchmark queries are available on our *project website [19]*.

[1] We leave the support for multiple event patterns for future work

6.1 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 SAQL system on the server for query execution. To reproduce the attack scenarios for the performance evaluation in Section 6.4, we stored the collected data in databases and developed a stream replayer to replay the system monitoring data from the databases.

6.2 Attack Cases Study

We performed four major types of attack behaviors in the deployed environment based on known exploits: (1) APT attack [2, 1], (2) SQL injection attack [43, 78], (3) Bash shellshock command injection attack [7], and (4) suspicious system behaviors.

6.2.1 Attack Behaviors

APT Attack: We ask white hat hackers to perform an APT attack in the deployed environment, as shown in Figure 3. 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 [6]) 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 credentials of the database.

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 7 to 11). Besides, we construct 3 advanced anomaly queries:

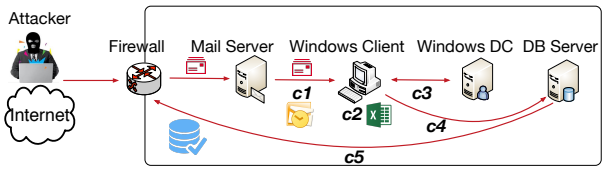


Figure 3: Environmental setup for the APT attack

- We construct an invariant-based anomaly query (Query 12) 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 13) based on 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 14) to detect processes that transfer high volumes of data to the network (*i.e.,* step *c5*): The query detects such processes through peer comparison based on DBSCAN. The detection logic here is different from Query 13, which detects anomalies through comparison with 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 [54] 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 [22] 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 15) 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 [7]. 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 of the web server in creating a long-running Shell process is an outlier pattern. We construct an invariant-based anomaly query (Query 16) 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 17): finding the activities of Dropbox processes.
- Command history probing (Query 18): finding the processes that access multiple command history files in a relatively short period.
- Unauthorized password files accesses (Query 19): finding the unauthorized processes that access the protected password files.
- Unauthorized login logs accesses (Query 20): finding the unauthorized processes that access the log files of login activities.
- Unauthorized SSH key files accesses (Query 21): finding the unauthorized processes that access the SSH key files.
- Forbidden USB drives usage (Query 22): finding the processes that access the files in the USB drive.
- IP frequency analysis (Query 23): finding the processes with high frequency network accesses.

6.2.2 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 3. We observe that: (1) the alert detection latency is low ($\leq$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

Table 3: Execution statistics of 17 SAQL queries for four major types of attacks

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

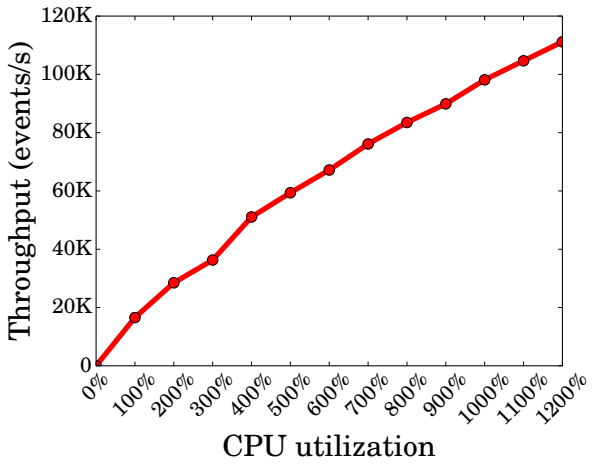


Figure 4: Throughput of the SAQL system under different CPU utilizations.

in Section 6.3); (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.

6.3 Pressure Test

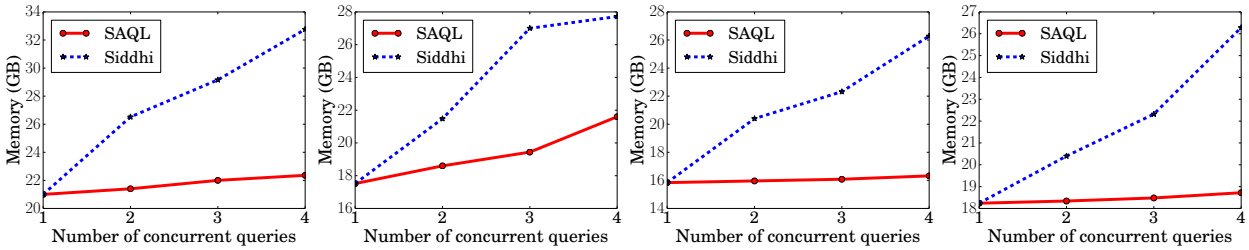
We conduct a pressure test of our system by replicating the data stream, while restricting the CPU utilization to certain levels [5]. When we conduct the experiments, we set the maximum Java heap size to be 100GB so that memory will not be a bottleneck. We deploy a query that retrieves all file events as the representative rule-based query, and measure the system throughput to demonstrate the query processing capabilities of our system.

Evaluation Results: Figure 4 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 per second, we can estimate that the SAQL system on this server can support ∼4000 hosts. While such promising results demonstrate that our SAQL system deployed in 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 demonstrate the impact of concurrent queries and how our master-dependent-query scheme mitigates the problem.

6.4 Performance Evaluation of Concurrent Query Execution

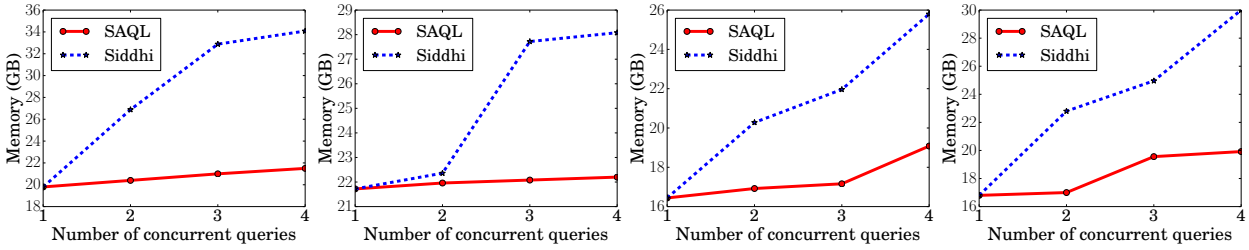
To evaluate the effectiveness of our query engine (*i.e.,* master-dependent-query scheme) in handling concurrent queries, we construct a micro-benchmark that consists of 64 queries and measure the memory usage during the execution. We select Siddhi [20], one of the most popular stream processing and complex event processing engines, for baseline comparison.

Micro-Benchmark Construction: We construct our micro-benchmark queries by extracting critical attributes



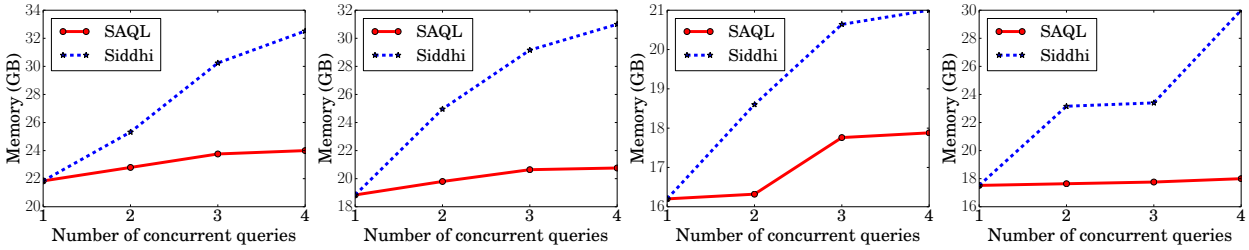
(a) Sensitive file accesses **(b)** Browsers access files **(c)** Processes access networks **(d)** Processes spawn

Figure 5: Event attributes



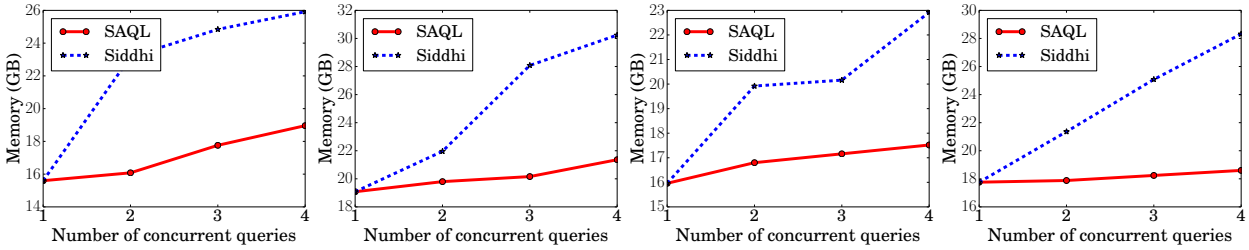
(a) Sensitive file accesses **(b)** Browsers access files **(c)** Processes access networks **(d)** Processes spawn

Figure 6: Sliding window



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

Figure 7: Agent ID



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

Figure 8: State aggregation

from the attacks in Section 6.2.1. 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`.
- *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 5.2:

- *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 [20, 12, 51, 4], do not provide the native support for these concepts, making these tools unable to specify advanced anomaly models (*i.e.,* time-series anomalies, invariant-based anomalies, outlier-based anomalies). Thus, for the "state evaluation category", we only construct Siddhi queries that monitor the same event pattern without stateful computation. Query 6 shows an example micro-benchmark query for the joint category "sensitive file accesses & state aggregation".

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

Query 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 5 to 8 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 require 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 6.1) to replay a large volume of data in a short period of time. Thus, the memory measured in Figures 5 to 7 is larger than the memory measured in the case study (Table 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.

7 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 substream-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 [27]. 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 broad set 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 [74, 75] and in-memory data manipulations [46, 53]. 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 [69, 88], should be explored.

Master-Dependent Query: Our optimization focuses on the queries that share the pattern matching results and 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 admit 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 al-

lows 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 [77, 45, 85] shows 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.

8 Related Work

Audit Logging and Forensics: Significant progress has been made to leverage system-level provenance for forensic analysis, with the focus on generating provenance graphs for attack causality analysis [74, 75, 63, 64, 32, 69, 88]. Recent work also investigates how to filter irrelevant activities in provenance graphs [71] and how to reduce the storage overheads of provenance graphs generated in distributed systems such as data centers [57]. These systems consider historical logs and their contributions are orthogonal to the contribution of SAQL, which provides a useful and novel interface for investigators to query abnormal behaviors from the stream of system logs. Nevertheless, SAQL can be interoperated with these systems to perform causality analysis on the detected anomalies over the concise provenance graphs.

Gao et al. [50] proposed AIQL which enables efficient attack investigation by querying historical system audit logs stored in databases. AIQL can be used to investigate the real-time anomalies detected by our SAQL system over the stream of system monitoring data. Together, these two systems can provide a better defense against advanced cyber attacks.

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 cryptographic systems [33, 34, 70], secure overlay networks [61, 72], and network intrusions [36, 44, 82, 86] and obfuscations [47]. These languages are explicitly designed to solve domain specific problems, providing specialized constructs for their particular problem domain and eschewing irrelevant features. In contrast to these languages, the novelty of SAQL focuses on how to specify anomaly models as queries and how to execute the queries over system monitoring data.

Security Anomaly Detection: Anomaly detection techniques have been widely used in detecting malware [58, 83, 65, 67], preventing network intrusion [89, 90, 80], internal threat detection [81], and attack prediction [87]. Rule-based detection techniques characterize normal behaviors of programs through analysis and detect unknown behaviors that have not been observed during the characterization [49, 58]. Outlier-based detection techniques [89, 90, 80] 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, SAQL provides a unified interface to express anomalies based on domain knowledge of experts.

Complex Event Processing Platforms & Data Stream Management Systems: Complex Event Processing (CEP) platforms, such as Esper [12], Siddhi [20], Apache Flink [4], and Aurora [29] match continuously incoming events against a pattern. Unlike traditional database management systems where a query is executed on the stored data, 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. Wukong+S [91] builds a stream querying platform that can query both the stream data and stored data. Data stream management systems [79], such as CQL [51], manage multiple data streams and provide a query language to process the data over the stream. These CEP platforms are useful in managing large streams of data. Thus, they can be used as a management infrastructure for our approach. However, these CEP systems alone do not provide language constructs to support stateful computation in sliding windows, and thus lack the capability to express stateful anomaly models as our system does.

Stream Computation Systems: Stream computation systems allow users to compute various metrics based on the stream data. These systems include Microsoft StreamInsight [31], MillWheel [30], Naiad [76], and Puma [41]. 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.

Other System Analysis Languages: Splunk [21] and Elasticsearch [10] are platforms that automatically parse general application logs, and provide a keyword-based search language to filter entries of logs. OSQuery [17, 18] allows analysts to use SQL queries to probe the real-time system status. However, these systems and the languages themselves cannot support anomaly detection and do not support stateful computation in sliding windows. Other languages, such as Weir [37] and StreamIt [84],

focus on monitoring the system performance, and lack support for expressing anomaly models.

9 Conclusion

We have presented a novel stream-based query system that takes a real-time event feed aggregated from different hosts under monitoring, and provides an anomaly query engine that checks the event stream against the queries submitted by security analysts to detect anomalies in real-time. Our system provides a *domain-specific language*, SAQL, which is specially designed to facilitate the task of expressing anomalies based on domain knowledge. SAQL 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*. Our evaluation results on 17 attack queries and 64 micro-benchmark queries show that the SAQL system has a low alert detection latency and a high system throughput, and is more efficient in memory utilization than the existing stream processing systems.

10 Acknowledgements

We would like to thank the anonymous reviewers and our shepherd, Prof. Adam Bates, for their insightful feedback in finalizing this paper. This work was partially supported by the National Science Foundation under grants CNS-1553437 and CNS-1409415. Any opinions, findings, and conclusions made in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Advanced persistent threats: How they work. https://www.symantec.com/theme.jsp?themeid=apt-infographic-1.

[2] Anatomy of advanced persistent threats. https://www.fireeye.com/current-threats/anatomy-of-a-cyber-attack.html.

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

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

[5] Cpulimit. https://github.com/opsengine/cpulimit.

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

[7] CVE-2014-6271. https://nvd.nist.gov/vuln/detail/CVE-2014-6271.

[8] DTrace. http://dtrace.org/.

[9] Ebay Inc. to ask Ebay users to change passwords. http://blog.ebay.com/ebay-inc-ask-ebay-users-change-passwords/.

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

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

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

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

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

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

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

[17] osquery. https://osquery.io/.

[18] osquery for security. https://medium.com/@clong/osquery-for-security-b66fffdf2daf.

[19] SAQL: A stream-based query system for real-time abnormal system behavior detection. https://sites.google.com/site/saqlsystem/.

[20] Siddhi complex event processing engine. https://github.com/wso2/siddhi.

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

[22] SQLMap. http://sqlmap.org.

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

[24] Top 5 causes of sudden network spikes. https://www.paessler.com/press/pressreleases/top_5_causes_-of_sudden_spikes_in_traffic.

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

[26] Using Splunk to detect DNS tunneling. https://www.sans.org/reading-room/whitepapers/dns/splunk-detect-dns-tunneling-37022.

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

[28] Cyber kill chain, 2017. http://www.lockheedmartin.com/us/what-we-do/aerospace-defense/cyber/cyber-kill-chain.html.

[29] ABADI, D. J., CARNEY, D., ÇETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. Aurora: A new model and architecture for data stream management. *The VLDB Journal 12*, 2 (2003), 120–139.

[30] AKIDAU, T., BALIKOV, A., BEKIROĞLU, K., CHERNYAK, S., HABERMAN, J., LAX, R., MCVEETY, S., MILLS, D., NORDSTROM, P., AND WHITTLE, S. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow. 6*, 11 (2013), 1033–1044.

[31] ALI, M. An introduction to Microsoft SQL server streaminsight. In *COM.Geo* (2010).

[32] BATES, A. M., TIAN, D., BUTLER, K. R. B., AND MOYER, T. Trustworthy whole-system provenance for the Linux kernel. In *USENIX Security* (2015).

[33] BHARGAVAN, K., CORIN, R., DENILOU, P. M., FOURNET, C., AND LEIFER, J. J. Cryptographic protocol synthesis and verification for multiparty sessions. In *CSF* (2009).

[34] BHARGAVAN, K., FOURNET, C., CORIN, R., AND ZALINESCU, E. Cryptographically verified implementations for TLS. In *CCS* (2008).

[35] BLUM, A. On-line algorithms in machine learning. In *Developments from a June 1996 Seminar on Online Algorithms: The State of the Art* (1998).

[36] BORDERS, K., SPRINGER, J., AND BURNSIDE, M. Chimera: A declarative language for streaming network traffic analysis. In *USENIX Security* (2012).

[37] BURTSEV, A., MISHRIKOTI, N., EIDE, E., AND RICCI, R. Weir: A streaming language for performance analysis. In *PLOS* (2013).

[38] CASELLA, G., AND BERGER, R. L. *Statistical inference*, vol. 2. Duxbury Pacific Grove, 2002.

[39] CHANDOLA, V., BANERJEE, A., AND KUMAR, V. Anomaly detection: A survey. *ACM Comput. Surv. 41*, 3 (2009), 15:1–15:58.

[40] CHANDRA, R., KIM, T., SHAH, M., NARULA, N., AND ZELDOVICH, N. Intrusion recovery for database-backed web applications. In *SOSP* (2011).

[41] CHEN, G. J., WIENER, J. L., IYER, S., JAISWAL, A., LEI, R., SIMHA, N., WANG, W., WILFONG, K., WILLIAMSON, T., AND YILMAZ, S. Realtime data processing at Facebook. In *SIGMOD* (2016).

[42] CHEN, Q., HSU, M., AND ZELLER, H. Experience in continuous analytics as a service (CaaaS). In *EDBT/ICDT* (2011).

[43] CLARKE, J. *SQL injection attacks and defense*, 1st ed. Syngress Publishing, 2009.

[44] CUPPENS, F., AND ORTALO, R. LAMBDA: A language to model a database for detection of attacks. In *RAID* (2000).

[45] DEBAR, H., AND WESPI, A. Aggregation and correlation of intrusion-detection alerts. In *RAID* (2001).

[46] DOLAN-GAVITT, B., HODOSH, J., HULIN, P., LEEK, T., AND WHELAN, R. Repeatable reverse engineering with PANDA. In *PPREW-5* (2015).

[47] DYER, K. P., COULL, S. E., AND SHRIMPTON, T. Marionette: A programmable network traffic obfuscation system. In *USENIX Security* (2015).

[48] ESTER, M., KRIEGEL, H.-P., SANDER, J., AND XU, X. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD* (1996).

[49] FORREST, S., HOFMEYR, S. A., SOMAYAJI, A., AND LONGSTAFF, T. A. A sense of self for unix processes. In *IEEE S&P* (1996).

[50] GAO, P., XIAO, X., LI, Z., JEE, K., XU, F., KULKARNI, S. R., AND MITTAL, P. AIQL: Enabling efficient attack investigation from system monitoring data. In *USENIX ATC* (2018).

[51] GAROFALAKIS, M. N., GEHRKE, J., AND RASTOGI, R., Eds. *Data stream management - processing high-speed data streams*. Springer, 2016.

[52] GOEL, A., PO, K., FARHADI, K., LI, Z., AND DE LARA, E. The taser intrusion recovery system. In *SOSP* (2005).

[53] GUO, Z., WANG, X., TANG, J., LIU, X., XU, Z., WU, M., KAASHOEK, M. F., AND ZHANG, Z. R2: An application-level kernel for record and replay. In *OSDI* (2008).

[54] HALFOND, W. G., VIEGAS, J., ORSO, A., ET AL. A classification of SQL-injection attacks and countermeasures. In *ISSSE* (2006).

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

[56] HAN, J., PEI, J., AND KAMBER, M. *Data mining: concepts and techniques*. Elsevier, 2011.

[57] HASSAN, W. U., LEMAY, M., AGUSE, N., BATES, A., AND MOYER, T. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *NDSS* (2018).

[58] HOFMEYR, S. A., FORREST, S., AND SOMAYAJI, A. Intrusion detection using sequences of system calls. *J. Comput. Secur. 6*, 3 (1998), 151–180.

[59] HOSSBACH, B., AND SEEGER, B. Anomaly management using complex event processing: Extending data base technology paper. In *EDBT* (2013).

[60] JIANG, X., WALTERS, A., XU, D., SPAFFORD, E. H., BUCHHOLZ, F., AND WANG, Y.-M. Provenance-aware tracing of worm break-in and contaminations: A process coloring approach. In *ICDCS* (2006).

[61] KILLIAN, C. E., ANDERSON, J. W., BRAUD, R., JHALA, R., AND VAHDAT, A. M. Mace: Language support for building distributed systems. In *PLDI* (2007).

[62] KIM, T., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Intrusion recovery using selective re-execution. In *OSDI* (2010).

[63] KING, S. T., AND CHEN, P. M. Backtracking intrusions. In *SOSP* (2003).

[64] KING, S. T., MAO, Z. M., LUCCHETTI, D. G., AND CHEN, P. M. Enriching intrusion alerts through multi-host causality. In *NDSS* (2005).

[65] KOLBITSCH, C., COMPARETTI, P. M., KRUEGEL, C., KIRDA, E., ZHOU, X., AND WANG, X. Effective and efficient malware detection at the end host. In *USENIX Security* (2009).

[66] KRUEGEL, C., VALEUR, F., AND VIGNA, G. *Intrusion detection and correlation - challenges and solutions*, vol. 14. Springer, 2005.

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

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

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

[70] LIU, C., WANG, X. S., NAYAK, K., HUANG, Y., AND SHI, E. ObliVM: A programming framework for secure computation. In *IEEE S&P* (2015).

[71] LIU, Y., ZHANG, M., LI, D., JEE, K., LI, Z., WU, Z., RHEE, J., AND MITTAL, P. Towards a timely causality analysis for enterprise security. In *NDSS* (2018).

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

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

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

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

[76] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: A timely dataflow system. In *SOSP* (2013).

[77] NING, P., CUI, Y., AND REEVES, D. S. Constructing attack scenarios through correlation of intrusion alerts. In *CCS* (2002).

[78] NYSTROM, M. *SQL injection defenses*, 1st ed. O'Reilly, 2007.

[79] PANIGATI, E., SCHREIBER, F. A., AND ZANIOLO, C. Data streams and data stream management systems and languages. In *Data Management in Pervasive Systems*. 2015, pp. 93–111.

[80] PORTNOY, L., ESKIN, E., AND STOLFO, S. Intrusion detection with unlabeled data using clustering. In *DMSA* (2001).

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

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

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

[84] THIES, W., KARCZMAREK, M., AND AMARASINGHE, S. P. StreamIt: A language for streaming applications. In *CC* (2002).

[85] VALEUR, F., VIGNA, G., KRUEGEL, C., AND KEMMERER, R. A. A comprehensive approach to intrusion detection alert correlation. *TDSC 1*, 3 (2004), 146–169.

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

[87] VEERAMACHANENI, K., ARNALDO, I., KORRAPATI, V., BASSIAS, C., AND LI, K. AI^2: Training a big data machine to defend. In *BigDataSecurity* (2016).

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

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

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

[91] ZHANG, Y., CHEN, R., AND CHEN, H. Sub-millisecond stateful stream querying over fast-evolving linked data. In *SOSP* (2017).

Appendix

A SAQL Queries in Attack Cases Study

We present the 17 SAQL queries that we construct in the case study, which are used detect the four major types of attack behaviors (Section 6.2.1). For privacy purposes, we anonymize the IP addresses and the agent IDs in the presented queries.

A.1 APT Attack

```
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
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
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
with evt1 -> evt2 -> evt3
return p1, i1, p2, i2, p3, i3, evt1.starttime, evt2.
    starttime, evt3.starttime
```

Query 7: apt-c1

```
agentid = XXX // windows client
proc p1["%outlook.exe"] start proc p2["%excel.exe"]
    as evt1 // outlook starts excel
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 8: 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 9: apt-c3

```
agentid = XXX // db 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 p1 start proc p2["%cmd.exe"] as evt2 // SQL
    server starts cmd
proc p2 read || write file f1["%hwvun.vbs"] as evt3
    // cmd writes malware sbblv.exe
proc p3["%cscript.exe"] write file f2["%sbblv.exe"]
    as evt4
proc p4["%sbblv.exe"] start ip i2[srcip="XXX" &&
    srcport=61060 && dstip="XXX" && dstport=443 &&
    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 10: apt-c4

```
agentid = XXX // db 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
with evt1 -> evt2 -> evt3 -> evt4
return p1, p2, p3, f1, p4, i1, evt1.starttime, evt2.
     starttime, evt3.starttime, evt4.starttime, evt4.
     amount
```

Query 11: 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 12: apt-c2-invariant

```
agentid = XXX // db 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 13: apt-c5-timeseries

```
agentid = XXX// db 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 14: apt-c5-outlier

A.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 15: sql-injection

A.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 16: shellshock

A.4 Suspicious System Behaviors

```
proc p["%dropbox%"] start ip i as evt
return p, i, evt.agentid, evt.starttime, evt.endtime
```

Query 17: 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 18: command-history

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

Query 19: password

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

Query 20: login-log

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

Query 21: sshkey

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

Query 22: usb

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

Query 23: ipfreq