This is Why We Can't Cache Nice Things: Lightning-Fast Threat Hunting using Suspicion-Based Hierarchical Storage

Wajih Ul Hassan[◇], Ding Li[‡], Kangkook Jee[∘], Xiao Yu[*], Kexuan Zou[◇], Dawei Wang[◇], Zhengzhang Chen[*], Zhichun Li[*], Junghwan "John" Rhee[†], Jiaping Gui[*], Adam Bates[◇]

[◇]University of Illinois at Urbana-Champaign [‡]Peking University [∘]University of Texas at Dallas
[†]University of Central Oklahoma [*]NEC Laboratories America, Inc.

ABSTRACT

Recent advances in the causal analysis can accelerate incident response time, but only after a causal graph of the attack has been constructed. Unfortunately, existing causal graph generation techniques are mainly offline and may take hours or days to respond to investigator queries, creating greater opportunity for attackers to hide their attack footprint, gain persistency, and propagate to other machines. To address that limitation, we present Swift, a threat investigation system that provides high-throughput causality tracking and real-time causal graph generation capabilities. We design an in-memory graph database that enables space-efficient graph storage and online causality tracking with minimal disk operations. We propose a hierarchical storage system that keeps forensically-relevant part of the causal graph in main memory while evicting rest to disk. To identify the causal graph that is likely to be relevant during the investigation, we design an asynchronous cache eviction policy that calculates the most suspicious part of the causal graph and caches only that part in the main memory. We evaluated Swift on a real-world enterprise to demonstrate how our system scales to process typical event loads and how it responds to forensic queries when security alerts occur. Results show that Swift is scalable, modular, and answers forensic queries in real-time even when analyzing audit logs containing tens of millions of events.

KEYWORDS

Auditing, Data Provenance, Digital Forensics

ACM Reference Format:

Wajih Ul Hassan[◇], Ding Li[‡], Kangkook Jee[∘], Xiao Yu[*], Kexuan Zou[◇], Dawei Wang[◇], Zhengzhang Chen[*], Zhichun Li[*], Junghwan "John" Rhee[†], Jiaping Gui[*], Adam Bates[◇]. 2020. This is Why We Can't Cache Nice Things: Lightning-Fast Threat Hunting using Suspicion-Based Hierarchical Storage. In *Annual Computer Security Applications Conference (ACSAC 2020), December 7–11, 2020, Austin, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3427228.3427255

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACSAC 2020, December 7–11, 2020, Austin, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8858-0/20/12…$15.00
https://doi.org/10.1145/3427228.3427255

1 INTRODUCTION

Modern organizational networks are sprawling and diverse, hosting data of tremendous value to malicious actors. Unfortunately, due to the complexity of organizations and time-consuming nature of threat investigations, attackers are able to dwell on the target system for longer periods. In slow-moving targeted attacks (e.g., Equifax [29]), the amount of damage wrought by the attacker grows exponentially as their dwell time in the system increases [18], with a recent study reporting that it costs organizations $32,000 for each day an attacker persists in the network [36]. This situation is made even worse when considering fast-spreading attacks; the infamous Slammer worm [65] that infected more than 75,000 hosts within the first ten minutes of its release, and recent ransomware attacks [9, 19, 24] exhibit a similar replication factor. Regardless of the specific attack, delayed response times imply significantly larger negative consequences. Thus, to minimize repercussions of intrusions, cyber analysts require tools that facilitate fast and interactive threat hunting.

Given its vital importance, what are the key factors that determine the success of the threat hunting process? The various steps involved in post-breach threat hunting [14] are summarized in Figure 1. Effectiveness is usually measured using two metrics in industry [5]: 1) *Mean-time-to-detect* (MTTD), which measures the time required for the organization's Threat Detection Software (TDS) to detect suspicious activity and raise a security alert; and 2) *Mean-time-to-know* (MTTK), which measures the time required for cyber analysts to make sense of alert and unearth evidence that the alert is indicative of a true attack. Depending upon the volume of threat alerts and the analysis tools available to the analyst, this process can typically range from hours to days for an individual threat alert [15, 18].

Recently, threat hunting has become a subject of renewed interest in the literature, primarily due to advancements in *causal analysis* [30, 31, 38, 39, 43–45, 47–49, 53, 54, 58, 60, 61] that can can reduce MTTK during the post-breach threat hunting process. Causality analysis incrementally parses audit log events generated by system-level logging tools (e.g., Linux Audit [4]) into causal graphs (i.e., *provenance graphs*) that encode the dependency relationships between subjects (e.g., processes) and objects (e.g., files) in the system. Such graphs not only provide the historical context needed by analysts to quickly understand alerts, but have also been shown to be useful for alert management (e.g., triage [41], correlation [40, 63], cyber threat intelligence [62, 69]).

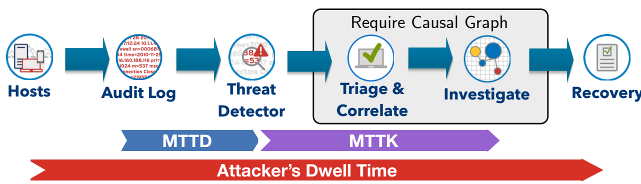

Figure 1: Typical post-breach threat hunting in an enterprise. Both alert management (e.g., triage) and investigation steps require causal graphs of generated alerts.

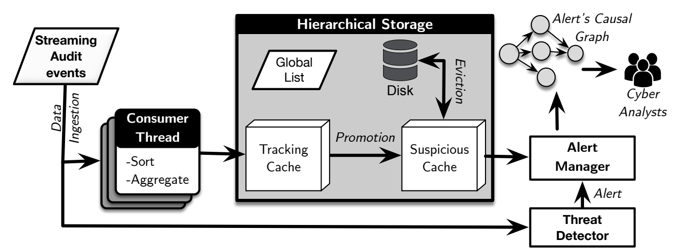

Figure 2: Overview of SWIFT architecture.

Unfortunately, at present the performance of causal analysis is a limiting factor to their widespread adoption – early attempts to deploy these techniques in practice reported graph construction times ranging from hours to days and unwieldy audit logs that reached terabytes in size over just a week (e.g., [57]). These existing tools fall under two categories: 1) disk-based offline approaches (e.g., [41, 57]) that incur significant I/O bottleneck and takes hours to respond to each query, thereby increasing MTTK; and 2) memory-based online approaches (e.g., [31, 44]) that require the *whole* causal graph to be stored in main-memory for analysis, which cannot scale to even modestly-sized organizations. As neither approach is a practical candidate for deployment, prior work has sought to improve the performance of causal analysis through various forms of graph reduction and compression (e.g., [32, 40, 42, 46, 55, 71, 73, 74]). By reducing the number of log events to process, those techniques have indeed improved query latency and alleviated the burdens of long-term storage. However, these approaches potentially affect the fidelity of logs for answering key forensic queries.[1] Further, over longer periods those techniques do not provide a scalable solution to log analysis and management.

In this work, we propose a causal analysis and alert management framework that can process logs and forensic queries as quickly as the system event stream. Unfortunately, building a highly scalable real-time causality tracker is a daunting task. The challenge comes from the *volume* and *velocity* of system events that are in large enterprises. Three key challenges need to be answered before we can build this scalable mechanism:

- **C1** *Scalable Ingest:* How can we continuously ingest and process upwards of terabytes of system events per day?
- **C2** *Fast Graph Retrieval:* How can we quickly recover causal graphs of recent alerts, especially when alerts' dependencies may extend back weeks into the past?
- **C3** *Efficient Alert Management:* How can we incorporate causality analysis into real-time alert management to help cyber analysts cope with the deluge of alerts?

1.1 Approach Overview & Contributions

To address these challenges, we designed SWIFT[2], a causality tracker for which scalability and performance are first-class citizens. Figure 2 presents an overview of the SWIFT architecture. Enterprise-wide audit logs are first collected into a Kafka broker [11] and then fetched by the consumer threads of SWIFT. Each consumer thread buffers the events for a certain configurable window, organizes the out-of-order events based on their timestamps, and merges continuous events that have the same source and destination.[3] Then, these audit log events are fed into a novel hierarchical storage management (HSM) system.

The challenge of *scalable ingestion* (C1) is met by the first contribution of this paper, a novel vertex-centric graph schema and database that is tailored for online causality analysis. This in-memory causal graph database allows SWIFT to quickly identify the causal relationships of streaming events with all causally-related events that occurred previously. We show that our graph database is space-efficient and is an enabling factor in providing real-time query results without significant disk I/O during our experiments.

The challenge of *fast graph retrieval* (C2) is resolved through the introduction of a causal graph HSM that consists of a two-layered memory cache (the tracking cache and suspicious cache, respectively), and a disk. This HSM automatically moves causal graph segments between main-memory and disk to achieve high-throughput data ingestion and low-latency query results. However, incorporating an HSM into an existing causal analysis framework is non-trivial – a generic cache eviction strategy would regularly evict forensically-relevant events, leading to increased disk access and high query latency.

Our solution to eviction is based on two distinct insights that motivate our two-layered memory cache design. The first insight is that of temporal locality; recent events have a high probability of dependence with upcoming events in the near future, Based on this observation we formulated an *Epochal Causality Hypothesis*, described in Section 6.1, and store recent events in the tracking cache. As events age out of the tracking cache, a decision must be made as to which events are likely to be used in forensic queries and should thus be retained in memory.

To identify forensically-relevant events, we formulated a *Most Suspicious Causal Paths Hypothesis* which states that, given a suspicious influence score algorithm (e.g., [33, 41, 50, 51, 57]) that satisfies three key properties described in Section 6.2, we can calculate the most suspicious causal paths in an online fashion (on time-evolving graphs); as these paths are more likely to be associated with a true attack, they are also the most likely to be queried and should thus be retained in the suspicious cache. Note that a causal graph consists of one or more causal paths (further described in Section 4). Finally, to quickly identify top-k most suspicious causal paths seen so far

[1] For example, LogGC removes subgraphs associated with closed sockets and thus could obscure data exfiltration attempts [55], while Winnower may prevent attack attribution by abstracting remote IP addresses [42].

[2] SWIFT is a recursive acronym for Swift investigator for threat alerts.

[3] Most of the operating systems introduce several system-level events for single file operation. Aggregating these events together does not affect the correctness of causality analysis but saves substantial space.

in the enterprise, SWIFT also maintains a Global List that stores pointers to such paths.

The final contribution of this paper considers the matter of efficient alert management (C3), which is a vital consideration to mitigating threat alert fatigue [18]. SWIFT includes an alert management layer on top of its HSM. When alerts are fired by a connected TDS (e.g., Splunk [70]), SWIFT automatically leverages its suspicious influence scores to perform alert triage based on historical context,[4] allowing the analyst to investigate the most likely threats first. Further, during online causality tracking, SWIFT keeps track of all previously-fired alerts. When an alert has a causal relation with a previously fired alert, SWIFT fuses these events into a single causal graph to display to the analyst.

1.2 Summary of Results

We deployed and evaluated our system at NEC Labs America, comprised of 191 hosts. Our case studies on this testbed confirm that SWIFT can retrieve the most critical parts of an APT attack from a database of over 300 million events in just 20 ms. SWIFT successfully classified 140 security alerts and responded to forensic queries in less than 2 minutes, reducing the latency of the state-of-the-art alert triage tools by 5 hours. With this result, we estimate that SWIFT can scale to monitor upwards of 4,000 hosts on a single server. Further, SWIFT can scale to support thousands of monitored hosts on a single machine using just 300 MB memory, thus addressing a central limitation of existing causal analysis techniques. *We clarify at the outset that SWIFT does not improve or detract from the efficacy of its two modular components, the underlying TDS (e.g., [70]) and suspicious influence scoring algorithm (e.g., [41, 57]); instead, SWIFT seeks to improve security by dramatically improving the speed and scalability of causality-based threat hunting solutions.*

2 RELATED WORK

Performance of Causal Analysis. Several threat investigation systems, such as PrioTracker [57], SAQL [37], and NoDoze [41] have been proposed to improve the performance of causal analysis in enterprises. Those systems use disk-based approach and may take hours to respond to each query. In a large enterprise with high-speed alerts, such response times are ineffective, increasing MTTK and attacker's dwell time.

CamQuery system [68] supports scalable online analysis of causal graphs. However, CamQuery only supports iterative computation of queries as pre-written programs. It does not support full forensic querying which cannot be known ahead of time and thus cannot be used for active threat hunting. SLEUTH [44], HOLMES [63], and POIROT [62] use in-memory graph database to provide real-time forensic analysis; however, they require whole causality data to be stored in main memory for forensic analysis. Thus, those systems cannot scale to enterprises that usually produce terabytes of data per week [57].

KCAL [59] proposed a kernel-level cache to remove redundant causal events and reduce the overhead of log transfer from kernel to user-space. However, the log is eventually stored on disk which incurs slow response times during forensic analysis. Moreover, KCAL does not provide any scalable solution for causal analysis on enterprise-wide data.

Graph Databases.. Given the high-throughput and low-latency requirements of large-scale streaming systems, key-value storage systems are shifting to in-memory designs [25, 56, 66]. Existing graph databases (e.g., Redis [26], Neo4J [21], and Stinger [35]) cannot be used directly in forensic analysis domain because of two main reasons. First, these databases need to *load* and *keep* the whole causal graph in the main memory to enable forensic analysis queries (e.g., backward tracing). In large enterprises where terabytes of data needs to be loaded from disk for long-running attack campaigns, this approach incurs a significant I/O bandwidth. Even assigning large main memory is prohibitively impractical for large enterprises because they need to store six months of log data, which is the average attacker dwell time, in the main-memory to provide real-time causal analysis. For example, NEC Labs America, with 191 hosts, generated about 20 TB of audit log in six months. So 20 TB of audit log needs to be stored in the main memory to provide real-time causal analysis at that enterprise.

Second, all-purpose graph databases incur a lot of space overhead to maintain every edge/vertex because they need to support most of the graph algorithms (e.g., clustering coefficient) not just forensic analysis algorithms. On the other hand, SWIFT provides causal graph database which is optimized for forensic analysis and keeps only forensically-relevant graph in the main memory to enable fast alert's causal graph generation.

Alert Correlation.. Alert correlation techniques assist security analysts by correlating similar threat alerts. Existing systems use statistical-, heuristic-, and probabilistic-based methods [34, 67, 72] to derive correlations between generated threat alerts. Moreover, security information and event management (SIEM) [16, 28] systems use similar approaches for alert correlation. We argue that these techniques are based on mere event correlations, while through causal analysis we can establish actual system-layer dependencies between events as provided by SWIFT.

3 THREAT MODEL AND ASSUMPTIONS

This work considers a large enterprise environment, comprised of upwards of thousands of machines, that is the target of a sophisticated and well-funded remote attacker. The attacker follows the pattern of rapid cyber attacks which are both *fast* – taking minutes to spread through the whole enterprise and *disruptive* – creating significant business disruption by establishing persistence, privilege escalation and lateral movement.

We make the following assumptions about the environment. We assume that there is a kernel-level causality tracker running on each host in the enterprise (e.g., Linux Audit, LPM [31]). We also assume the presence of one or many threat detection systems that generate threat alerts in real-time (e.g., [1, 2, 6, 16, 70]); recall that our solution will use causal analysis to triage, correlate, and provide historical context to these alerts. Like other work in this space (e.g., [40, 42, 43, 45, 53]), we assume that the causality tracker is not compromised and that the audit logs are correct at the time of forensic analysis. Hardware-layer or physical attacks, as well as side-channels, are designated as out of the scope of this paper.

[4] Prior work [41] has shown that incorporating historical context into alert triage may reduce the false positives of a commercial TDS by up to 84%.

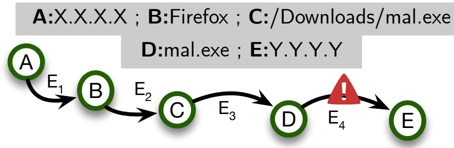

Figure 3: An example causal graph. The event (edge) E_4 has triggered an alert. The full graph describes the causality of the alert.

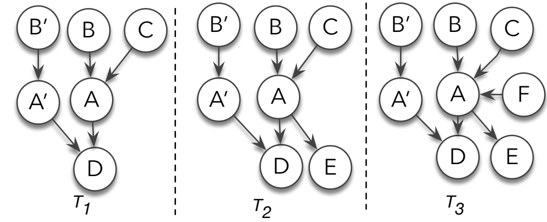

Figure 4: Example of causal graph database updates over time.

4 PRELIMINARIES

4.1 Causality Analysis

Audit logs are a set of records that provide a detailed history of the activities that have affected an operating system. Audit support is included in all major operating system families, such as the Linux Audit [4] and Event Tracing for Windows (ETW) [3]. Causality trackers incrementally parse events in these audit logs into causal graphs of the form $G =< V, E >$. V is a set of vertices representing different system entities (e.g., processes, file) that are identified by various metadata such PID and file path. E is a set of edges defined by the 4-tuple (src, dst, t, rel) where rel is a causal relationship type between vertex src and vertex dst that occurred at time t. Because each *threat alert* in the system is an event associated with an edge $e \in E$, a cyber analyst can issue a *backward tracing* query on the causal graph to identify the root cause of e, then issue a *forward tracing* query to identify other ramifications of the same attack.

Figure 3 shows the simplified causal graph generated for a threat alert triggered by event E_4, which documents a process `mal.exe` initiating a network connection to IP `Y.Y.Y.Y`. The full graph shows the backward trace, or *provenance*, of E_4, revealing that `mal.exe` was downloaded from IP `X.X.X.X`. This contextual information can help cyber analysts to validate and investigate the generated alert. A causal graph consists of one or more causal paths, which are defined as follows:

Def. 1. ***Causal Path.*** *A causal path P of a event e_a represents a chain of events that led to e_a and chain of events induced by e_a in the future. It is a temporally ordered sequence of events and represented as $P := \{e_1, \ldots, e_a, \ldots, e_n\}$ of length n. Each event can have multiple causal paths where each path represents one possible flow of information through e_a.*

4.2 Suspicious Influence Score

When analyzing causal paths, it is desirable to understand how the suspiciousness of each event relates to the whole. Here, our *suspicion* may relate purely to an event's rarity, but may also incorporate other knowledge sources besides frequency, such as IP blacklists or antivirus signatures. To evaluate the suspiciousness of an entire path, we introduce the notion of a suspicious influence score. We say that a path exerts "suspicious influence" because it influences the level of suspicion that we have for future events, including alert events.

Def. 2. ***Suspicious Influence Score.*** *For a causal path $P := \{e_1, \ldots, e_i, \ldots, e_n\}$ where the suspiciousness score for event e_i is given by $AS(e_i)$, the suspicious influence score $AS(P)$ is a function that combines the suspiciousness score of each event in the path P.*

Many prior works satisfy this definition for a suspicious influence scoring algorithm, e.g., [33, 41, 50, 51, 57]. In our approach, we require the scoring algorithm to satisfy three specific properties: Cumulativity, Temporality, and Monotonicity. Combined, these properties will allow Swift to track causality in an online fashion with a low time complexity and minimal disk operations. To better explain these properties, we use Figure 4 as an example.

The first property, **Cumulativity**, means that the suspicious influence score of a path can be calculated from the suspicious influence score of its prefix and the suspiciousness score of its last event. For example, in Figure 4, to calculate the suspicious influence score of the causal path $P_1 = \{B \rightarrow A \rightarrow D\}$, we only need to know the suspicious influence score of $P_1' = \{B \rightarrow A\}$ and the suspiciousness score of event $A \rightarrow D$. This property guarantees that while adding new events to an existing path, Swift does not need to backtrack the existing path to generate the suspicious influence score for the newly extended path.

The second property, **Temporality**, means that an event can only affect the suspicious influence score of events that happen after it. For two events $e_1 = \{V_1 \rightarrow V_2\}$ and $e_2 = \{V_2 \rightarrow V_3\}$, event $AS(e_2)$ depends on $AS(e_1)$ only if e_1 happens before e_2. This is intuitive from an information flow perspective, as V_2 will not have been inform by V_1 until after e_1 occurs. For example, at time T_2 in Figure 4, events $A \rightarrow E$ and $A \rightarrow D$ do not depend on event $F \rightarrow A$ because this occurred at time T_3. Therefore, we do not calculate the suspicious influence scores $AS(F \rightarrow A \rightarrow E)$ or $AS(F \rightarrow A \rightarrow D)$.

The third property **Monotonicity**, means that when a new event is appended to two existing paths it does not change the suspicious influence score of the existing paths. Let $P_1 = \{P_1' \rightarrow S \rightarrow D\}$ and $P_2 = \{P_2' \rightarrow S \rightarrow D\}$, where P_1' and P_2' are distinct causal paths prefixes and $\{S \rightarrow D\}$ is a new event shared by P_1 and P_2. The monotonicity property states that if $AS(P_1' \rightarrow S) > AS(P_2' \rightarrow S)$ then it must also be true that $AS(P_1) > AS(P_2)$. For example in Figure 4, if $AS(B \rightarrow A) > AS(C \rightarrow A)$ at time T_1 then it must also be true that $AS(B \rightarrow A \rightarrow E) > AS(C \rightarrow A \rightarrow E)$ at time T_2. This property helps ensure the correctness of our online causality tracking.

5 VERTEX-CENTRIC CAUSAL GRAPH

In this section, we first explain different graph formats and describe their merits and limitations for fast causal analysis. Then, we present the graph format used by Swift.

5.1 Graph Representation

There are two major data formats for graphs [52]. First, the *Edge List* format is a collection of edges, each a pair of vertices, that captures

the incoming data in their arrival order. Second, the *Adjacency List* format manages the neighbors of each vertex in separate per-vertex edge arrays. In Edge Lists, the neighbors for each vertex are scattered across the data structure, making it difficult to traverse the graph quickly. On the other hand, in Adjacency Lists vertex neighbors are easy to reference, making them better suited for causal graph traversal.

In our causal graph schema, each system subject and object is represented as a vertex in the causal graph and stored as an entry in a key-value storage. In each key-value pair $\langle Key, Val \rangle$, Key is the unique identifier representing the vertex and Val is a list of three entries. For a vertex K this list is as follows:

(1) A list of K's parent vertices' unique identifiers, $L_{parents}$. Each parent identifier is associated with a timestamp for the event's creation and the edge relationship type.
(2) A list of K's child vertices' unique identifiers $L_{children}$. Each child identifier is associated with a timestamp for the event's creation and the edge relationship type.
(3) An ordered list $PATH_{abnormal}$ of the m most suspicious causal paths that end with vertex K, sorted in order of each path's suspicious influence score.

This graph representation is specifically tailored towards forensic analysis queries, i.e., backward and forward tracing queries. We use the same graph representation for both main-memory and on-disk storage. Recall that a major goal of Swift is to provide hierarchical storage that can quickly query the most suspicious causal graphs. Our graph schema supports this through the $PATH_{abnormal}$ objects, which are sorted in a descending order of their suspicious influence scores. Note that each vertex has a set of causal paths that end at it, even though these may be sub-paths of other paths. For example, in Figure 4, vertex A has two paths in its $PATH_{abnormal}$, $P_1 = \{B \rightarrow A\}$ and $P_2 = \{C \rightarrow A\}$. These two paths are the sub-paths of $P_3 = \{B \rightarrow A \rightarrow D\}$ and $P_4 = \{C \rightarrow A \rightarrow D\}$, respectively.

5.2 Suspicious Causal Paths

For a vertex K, each path in $PATH_{abnormal}$ is a tuple in the form of $(P, S, t, Rel, Rank)$: P is the unique identifier of the parent vertex of K in a given causal path; S is the suspicious influence score of the path; t is the timestamp of the edge event $P \rightarrow K$; Rel is edge relationship between K and P; and $Rank$ is the relative score ranking of all the paths that end at $P \rightarrow K$. In the case when multiple edges with the same edge relationship Rel exists between two vertices, we keep only the latest timestamp. This is because ignoring the previous edges does not affect the correctness of forensic analysis, as shown by previous works (e.g., [44, 55]).

We use Figure 4 as an example to explain our design of $PATH_{abnormal}$. Note that we do not show edge relationships in this figure and rest of the paper for simplicity although we do store edge relationships in our schema. In Figure 4 there are three paths ending at the vertex D, which are $P_1 = \{B \rightarrow A \rightarrow D\}$, $P_2 = \{C \rightarrow A \rightarrow D\}$, and $P_3 = \{B' \rightarrow A' \rightarrow D\}$. Assume the suspicious influence score and the timestamps of P_1, P_2, and P_3 are S_1, S_2, and S_3 and t_1, t_2, t_3, respectively. If $S_1 > S_2 > S_3$, then $PATH_{abnormal}$ of D is $[(A, S_1, t_1, Rel_1 0), (A, S_2, t_2, Rel_2, 1), (A', S_3, t_3, Rel_3, 0)]$. For the tuple $(A, S_1, t_1, Rel_1, 0)$, it means that the parent of the given causal path is A, its suspicious influence score is S_1, the event $A \rightarrow D$ happens at time t_1 with edge relationship Rel_1 and its suspicious score ranks the first among all paths which have the last edge as $A \rightarrow D$.

Algorithm 1: PathDiscover

```
   Inputs : V, R, Seen
   Output: PATH
1  Parent = GetParent(V,R)
2  if Parent = Null then
3  |  return Null
4  if Parent ∈ Seen then
5  |  return Null
6  Seen ← Parent
7  ParentRank = GetRank(V,R)
8  PATH = PathDiscover(Parent,ParentRank, Seen)
9  Append(PATH,Parent)
10 return PATH
```

Limiting the Size of $PATH_{abnormal}$ The number of paths that end at each vertex is exponential to the number of vertices. Maintaining a $PATH_{abnormal}$ that contains all paths is not realistic. To address this limitation, in our design of Swift, the length of $PATH_{abnormal}$ is limited to m. Limiting the size of $PATH_{abnormal}$ means that for each vertex in the causal graph, Swift only keeps the top m most suspicious paths that end at that vertex in memory. Note that this does not affect the completeness of the whole causal graph since the complete parent and child list for each vertex is maintained on disk. It only affects the paths that can be retrieved quickly from the main memory. Based on the Hypothesis H2, these suspicious paths are more likely to represent attacks. Thus, it is reasonable for us to limit the size of $PATH_{abnormal}$ for each vertex.

5.3 Graph Query

Our design of the causal graph schema and database allows fast recovery of a causal path with the unique identifier of its last vertex and its index in $PATH_{abnormal}$. The time complexity of the recovering process is $O(n)$, where n is the length of the causal path. The algorithm is outlined in Algorithm 1. The inputs are the vertex V, the relative ranking R of the causal path in V, and $Seen$, which is a hashmap of the previously-visited vertices during path discovery. This hashmap is used to halt recursion in the case of a cycle. The output of Algorithm 1 is the discovered path.

We use Figure 4 as an example to explain the recovering process. Assume that Swift wants to recover the highest scoring path $P_1 = \{B \rightarrow A \rightarrow D\}$. To do so, Swift only needs to have the last vertex D and the relative ranking (index), which is 0. To recover the full path, Swift refers to the first element in its $PATH_{abnormal}$ and recovers the parent in the given path, which is A, and gets the relative ranking of the path in A, which is also 0. Then this process is recursively repeated on A and its ranking until the whole path is recovered.

6 HIERARCHICAL STORAGE MANAGEMENT

6.1 Tracking Cache

Swift takes the stream of audit log events and identifies causal relationships between each new event and past events in order to build a causal graph. The role of the tracking cache is to ensure

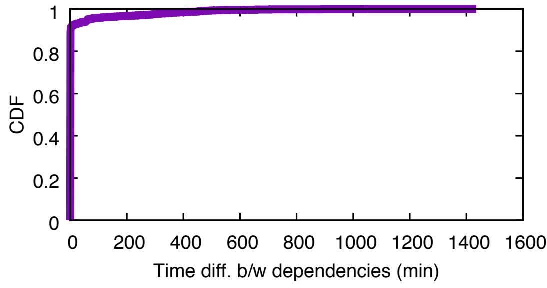

Figure 5: CDF of the time difference between a newly generated event and the event's immediate dependencies (i.e., parents). 98% of events' immediate dependencies occurred less than 15 minutes ago, providing empirical evidence for the Epochal Causality Hypothesis.

that the events most relevant to the graph building process are consistently available in the main memory. Our approach to assuring fast access to causally-related past events is based on the following hypothesis:

H1 Epochal Causality Hypothesis. *Events which are recently accessed during causal graph generation are accessed again in a short epoch of time ($\Delta T_{promote}$), and thus should not be evicted from the main memory in that epoch.*

An empirical validation of hypothesis H1 is given in Figure 5 based on the audit stream of a 191 host enterprise. This CDF shows that the immediate dependencies (i.e., parents) of 98% of newly created events were created within a short epoch prior (< 15 mins). In other words, if we can design a cache that can store the most recent 15 minutes of events in the main memory, we will eliminate 98% of disk accesses.

6.1.1 Tracking Algorithm & Eviction Policy. Algorithm 2 outlines the high level steps of our online tracking algorithm. At the high level, it takes the causal graph database (*GDB*) and an incoming event (*E*) as the input, and adds *E*'s subject and object to the causal graph database as two vertices. At the same time, the algorithm calculates and updates $PATH_{abnormal}$, which represents the most suspicious paths that end with the object of *E*. The time complexity of updating the $PATH_{abnormal}$ is O(1).

The first step of the tracking algorithm is to check if the subject and the object of the event exist in the *GDB* (lines 1-2). RetreiveOrCreate does this work. Given the system entities (the subject or the object), RetreiveOrCreate tries to first fetch it from the main memory. If the system entity does not exist in the main memory, RetreiveOrCreate tries to fetch it from the disk. If the system entity still does not exist in the hard disk, RetreiveOrCreate will create a new entry in the causal graph database.

Once the subject and object have been retrieved, Swift updates the parent and child list for the subject and object based on edge relationship *Rel* (lines 4 - 5 and lines 12 - 13). Then it updates the $PATH_{abnormal}$ list of the children (lines 6 - 9 and lines 16 - 19). To do so, Swift enumerates each element in the $PATH_{abnormal}$ of the subject (line 6), calculates a score from each element in the subject's $PATH_{abnormal}$ (line 7) and updates the $PATH_{abnormal}$ of the object with the new score, the relative ranking the in subject (*Index*), and the new time stamp of the event (line 8). Finally, Swift updates the *GDB* of the subject and the object in the main memory.

Algorithm 2: TrackObject

```
   Inputs :GDB, E
 1 Sub = RetreiveOrCreate(E.sub, GDB)
 2 Obj = RetreiveOrCreate(E.obj, GDB)
 3 if IsParent(Sub, E.Rel) then
 4 |   Sub.AddChild(Obj)
 5 |   Obj.AddParent(Sub)
 6 |   for Index, (P, S, t, R) ∈ Sub.PATH_abnormal do
 7 |   |   ChildScore = CalculateScore(S,Sub, Obj, E)
 8 |   |   Obj.AddToPath(Sub, ChildScore, E.t, Index)
 9 else
10 |   Obj.AddChild(Sub)
11 |   Sub.AddParent(Obj)
12 |   for Index, (P, S, t, R) ∈ Obj.PATH_abnormal do
13 |   |   ChildScore = CalculateScore(S,Sub, Obj, E)
14 |   |   Sub.AddToPath(Obj, ChildScore, E.t, Index)
15 GDB.Update(Sub)
16 GDB.Update(Obj)
17 return
```

The time complexity of Algorithm 2 is $O(1)$. Since we have limited the size of the $PATH_{abnormal}$ as a constant, the time complexity of the loop between line 5 and line 8 is constant. Due to the same reason, the time complexity of AddToPath is also $O(1)$. After each epoch $\Delta T_{promote}$, Swift evicts system objects (vertices) from tracking cache to the suspicious cache if they have not been accessed in the last epoch. Vertices that have been accessed during the past epoch are retained in the tracking cache for the next epoch.

6.2 Suspicious Cache

After being evicted from the tracking cache, vertex entries are moved to the suspicious cache. The goal of the second cache is to retain vertex entries for all vertices that fall on the Top *K* most suspicious causal paths throughout the history of system execution. The intuition behind the suspicious cache is based on the Hypothesis H2.

H2 Most Suspicious Causal Paths Hypothesis. *If a path in the causal graph contains multiple suspicious (anomalous) events, it is much more likely to be associated with a true attack.*

Recent studies provide evidence for this hypothesis, and in fact are the inspiration for the present study – Hassan et al. [41] present an alert triage system that ranks alerts based on the aggregate anomalousness of their causal paths, observing that this approach can be used to eliminate 84% of false alerts from a commercial Threat Detection Softwares (TDS). Liu et al. [57] present an optimization for forward trace queries that prioritizes the search of anomalous paths in order to construct attack graphs more quickly. While these results are encouraging, both of these systems rely on disk-based graph storage and are thus subject to extremely high latencies when traversing causal graphs; our observation is that this hypothesis can also inform the design of a forensic cache. Because true attacks are likely to fall on the most suspicious (anomalous) causal paths, our system should prioritize the retention of events associated with such paths. This will increase the likelihood that all forensically-relevant information will exist in main memory at the time of the investigation.

The specific goal of the suspicious cache is to retain vertices that appear in the Top K most suspicious causal paths; we call this set of K paths the Global List (GL). Each element in Global List is a pair (V, R), where V is a vertex in the causal graph database and R is the index of the causal path in V's $PATH_{abnormal}$ list. As discussed in Section 5, we can recover the full causal path efficiently with this pair.

For a causal path P to be in GL, P must meet three conditions: (1) the suspicious influence score of P is among the top K most suspicious paths in history; (2) P is not a sub-path of another causal path (e.g., the path $\{B \longrightarrow A\}$ in Figure 4 could not be in GL because it is a sub-path of $\{B \longrightarrow A \longrightarrow D\}$; and (3) P is in the $PATH_{abnormal}$ of at least one $vertex$. The third condition alleviates a possible "spoofing attack" that spoils the cache of Swift, which we discuss in Section 9.

6.2.1 Suspicious Cache Eviction Policy. Based on the Hypothesis H2, Swift maintains the top-K most suspicious causal paths in the memory to support low-latency attack investigation. To achieve this goal with our two-layer cache design, we introduce a time-window ΔT_{evict} to evict objects from suspicious cache and GL to the disk. At a pre-defined time interval, ΔT_{evict}, Swift asynchronously runs an eviction algorithm to move the vertices that are not contained in a GL path to the disk. Algorithm 3 outlines the high-level steps of the eviction process from suspicious cache to the disk. Its inputs are GL and suspicious cache. The algorithm first enumerates every tuple in GL, recovering the causal path from the tuple using Algorithm 1. Then, for each vertex in the recovered path, it taints the vertex as "TO_KEEP" (lines 1 - 5). After the tainting process, Swift evicts the key-value pairs in the *SuspiciousCache* that do not have the "TO_KEEP" taint (lines 6 - 11). Note that Algorithm 1 accounts for possible cycles in the graph.

6.2.2 Correctness. The equation we use for suspicious influence scoring in Swift's implementation is given by Equation 1 in Section 8. Based on the Cumulativity this equation, the time complexity of CalculateScore is also $O(1)$. The correctness of Algorithm 2 is guaranteed by the Monotonicity and the Temporality of Equation 1. Due to the Temporality of Equation 1, Algorithm 2 only needs to update the $PATH_{abnormal}$ for the object. It does not need to further propagate the suspicious influence score to the successors. Due to the Monotonicity, the new top m most suspicious paths of the object can only be from the old $PATH_{abnormal}$ of the object or the new causal paths generated from the top m most suspicious paths of the subject. Thus, to calculate the new $PATH_{abnormal}$, it is safe to only enumerate the items in $PATH_{abnormal}$ of the subject.

Complexity. Our eviction algorithm runs in $O(N)$ time complexity, where N is the total number of vertices present in the main-memory, since it has to taint all the vertices which belong to the Global List path and evict the vertices that do not belong to the Global List path.

Algorithm 3: Eviction

```
   Inputs : GL, SuspicousCache
1  for (V, R) ∈ GL do
2  |   PATH= PathDiscover(V, R)
3  |   for N ∈ PATH do
4  |   |   Taint(N)
5  for < K, V > ∈ SuspiciousCache do
6  |   if CheckNotTaint(< K, V >) then
7  |   |   EVICT(< K, V >)
8  return
```

7 ALERT MANAGEMENT

The alert management layer provides three fundamental capabilities: context-based alert triage, alert correlation, and suspicious causal graph generation. These capabilities are based on Swift's HSM which allows them to be real-time. Context-based alert triage is achieved by the propagating and storing of suspicious influence scores along with each causal path in the database. Note that the suspicious influence scores are calculated during online tracking (Algorithm 2). As discussed previously, the greater the suspicious influence score of an alert, the more suspicious that alert will be and should therefore be investigated first. As soon as alerts are fired during threat hunting process (shown in Figure 1), Swift iteratively sorts alerts based on suspicious influence scores. In the alert management stage, Swift only needs to retrieve the previously-calculated suspicious influence scores from the HSM, assuring that alert triage can occur in real-time.

Alert correlation and concise causal graph generation are realized automatically by our HSM design. Swift uses the suspicious cache to retain the causal paths of those previously triggered alerts that have higher suspicious influence scores. To correlate two alerts, Swift only needs to query the suspicious cache to figure out if the most recently triggered alert's causal path is associated with any alerts that were triggered in the past. To support causal graph generation, Swift provides two types of queries to retrieve the causal graph of alerts: concise queries and complete queries. The concise query returns the most suspicious causal subgraph related to an alert, which is stored entirely in the suspicious cache. The complete query returns the whole causal graph by fetching paths from both the suspicious cache and, if needed, the disk.

8 EVALUATION

In this section, we focus on evaluating the efficacy, usefulness, and scalability of Swift as a real-time forensic analysis in an enterprise. In particular, we investigated the following research questions (RQs):

- ***RQ1*** How effective is Swift in threat alert investigation?
- ***RQ2*** What are the insights into the events that are cached vs spilled to disk by Swift?
- ***RQ3*** How scalable is Swift?
- ***RQ4*** Can the time saved using Swift help an enterprise to thwart an attack?
- ***RQ5*** How efficient is Swift at alert management?

8.1 Implementation

We implement Swift for an enterprise environment and collected system event logs generated by Windows ETW [3] and Linux Auditd [4] using Kafka producers. We wrote our own consumer threads to fetch audit logs from Kafka producers. Swift uses the Guava Cache by Google [17] to maintain the causal graph database in

the main-memory. This cache supports timed eviction and asynchronous batch writes. Swift uses RocksDB [27] as the persistent key-value storage. The batch mode in RocksDB provides high rate for read and write.

In our implementation, we use the method proposed by Hassan et al. [41] to calculate the suspicious influence score because it satisfies all the three properties mentioned in Section 4.2. Particularly, for a causal path P, we calculate its Suspicious Influence Score $SIS(P)$ with Equation 1.

$$SIS(P) = 1 - \prod_{i=1}^{l} IN(SRC_i) \times M(\varepsilon_i) \times OUT(DST_i) \times \alpha \qquad (1)$$

The details about the above-mentioned equation can be found in [41]. At a high-level, IN and OUT are two vectors that quantify the likelihood that the vertex is a source or destination of information flow, respectively. M is the transition probability from SRC_i vertex to DST_i vertex. α is a normalization factor. IN, OUT, M, and α are parameterized based on observations of historic benign data from the enterprise deployment. This equation satisfies all three properties mentioned in Section 4.2. *Cumulativity* is satisfied because this equation calculates score of each event by taking the product of all previous events' aggregate score and the new event's score. *Temporality* is preserved because the product is taken over a causal path, which is sorted temporally by definition. If a new event is added to two paths, the subtraction of their $SIS(P)$ will be multiplied by the same factors, which will not change their orders. Therefore, *monotonicity* is satisfied.

8.2 Experiment Setup

We collected system events and threat alerts at NEC Labs America. In total, we monitored 191 hosts (51 Linux and 140 Windows OS) for 10 days. We deployed Swift on a server with Intel® Xeon(R) CPU E5-2660 @ 2.20GHz and 64 GB memory running Ubuntu 16.04 OS. We connected Swift to ASI [13], a commercial anomaly-based TDS, to generate alerts. During the engagement, we injected 10 APT attacks over a period of 10 days. These APT attacks were designed by expert analysts employed at NEC Labs America. A short description of these attacks is shown in Table 1. On each day we injected one attack, except for 3 attacks (Datatheft, ShellShock, and Netcat backdoor) which were ran on the same day.

We collected more than 1 TB worth of audit logs with around 1 billion system events from 191 hosts over period of 10 days. The APT attack traces constitute less than 0.0005% of the total audit logs collected from the enterprise. Meanwhile, we also monitored these logs with a commercial TDS [13]. This underlying TDS generated 140 threat alerts over a period of 10 days. Out of these 140 alerts, 12 were true alerts generated by our simulated APT attacks, while the rest were false alerts.

To evaluate Swift against a baseline approach, we re-implement NoDoze based on its description in [41]. We chose this as a baseline because it is one of the most recent offline approach that can perform: 1) suspicious score assignment, 2) automated alert triage and 3) causality graph generation. Further, our decision to implement Swift using NoDoze's suspicious influence scoring algorithm permits an apples-to-apples comparison when evaluating Swift's HSM. We used 20 consumer threads to consume audit logs from Kafka producers and then we performed forensic analysis in real-time. Note that 20 threads is also the maximum number of threads supported by the machine we use in our evaluation.

Parameters. We set $\Delta T_{promote}$ = 800 *seconds*, GL size K = 3000, $PATH_{abnormal}$ size $m = K/3$, and ΔT_{evict} = 1600 *seconds* in all experiments unless we explicitly note otherwise. We chose these values because they generate the optimal throughput and can hold all the suspicious data in our enterprise. However, we also discovered that it is flexible to choose the parameters for Swift since the throughput is not heavily affected by the value of the parameters. A detailed discussion of how we derived these values for this enterprise environment is included in Appendix A.1.

RQ1: Effectiveness in Alert Investigation

To answer this question, we used Swift to generate the *most suspicious causal graph* for all 140 threat alerts, measuring the response time for answering each causal graph query. We issued each query at the end of the day, *not* immediately following the attack, which ensured: 1) all attack related events had been evicted from the tracking cache, and were thus either in the suspicious cache or on disk; 2) a steady state for the HSM where all promotion and eviction cycles were completed for that day. We manually verified the fidelity of Swift's causal graph for each alert against the graphs generated by the baseline approach, checking that Swift returned all of the critical events necessary to explain the attack.

The results for Swift are shown in Figure 6; Swift was able to respond in less than one second for 80% of the alerts because of our novel suspicion-based HSM. In total, Swift took less than two minutes to generate the concise causal graphs for all alerts. We compare these results to the baseline approach in Figure 7, noting that the scale on the x-axis has changed from seconds to minutes. It took more than 1 hour for the baseline approach to process the same set of alerts. Moreover, the baseline approach took more than three minutes for 40% of the alerts and more than 20 minutes for 25% of the alerts, *in the worst case taking more than an hour to finish*. Such a slow response time is problematic, especially considering realistic scenarios in which the processing latency for one alert adds to the queuing latency of hundreds of other alerts in the stack (discussed more in **RQ3**).

A breakdown of performance results for each attack are shown in Table 2. The rightmost columns show the response time for the baseline method, Swift, and observed speedup. In all cases, Swift generated the causal graph for the attack in less than 3 milliseconds, whereas the baseline required nearly 5 minutes in the worst case. *Comparing the two techniques, we observe a speed up of up to 1.3 million times (Shellshock)*. In spite of the performance increase, it may at first glance seem that the performance of the baseline approach is acceptable. One reason for this is that the underlying TDS used in our experiments itself maintained a 15GB event cache that was able to store part of the attack provenance for the baseline (compare this to the 300MB cache required by Swift, which we will show in Section 8.2). More importantly, a limitation of our evaluation is that it does not capture longitudinal attack patterns that are commonly observed in-the-wild, e.g., the 4.5 month attack window of the Equifax breach [29]. In such circumstances, the TDS

Table 1: APT attack scenarios used in our evaluation with short their descriptions.

Attacks	Short Description
VPNFilter [10]	An attacker used known vulnerabilities [7] to penetrate into an IoT device and overwrite system files for persistence. It then connected to outside to connect to C2 host and download attack modules.
Redis-Server	Example case study in Section 8.2
wget-gcc [74]	Malicious source files were downloaded and then compiled.
WannaCry [9]	An attacker exploits EternalBlue [20] vulnerability in enterprise to gain access to machines and then attacker encrypts data on those machines.
Data Theft [57]	An attacker downloaded a malicious bash script on the data server and used it to exfiltrate all the confidential documents on the server.
ShellShock [7]	An attacker utilized an Apache server to trigger the Shellshock vulnerability in Bash multiple times.
Netcat Backdoor [8]	An attack downloaded the netcat utility and used it to open a Backdoor, from which a Persistent Netcat port scanner was then downloaded and executed using PowerShell
Cheating Student [61]	A student downloaded midterm scores from Apache and uploaded a modified version.
passwd-gzip-scp [74]	An attack stole user account information from passwd file, compressed it using gzip and transferred the data to a remote machine
Jeep-Cherokee [64]	An attack remotely exploits in-car information system and gains control over physical components (*e.g.*, wheels, breaks, engines) by sending out commands via CANBUS.

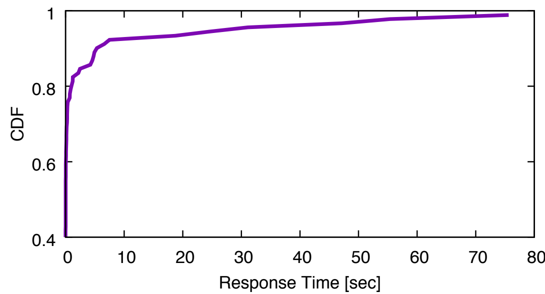

Figure 6: Response times *in seconds* **to return concise causal graphs of threat alerts using Swift.**

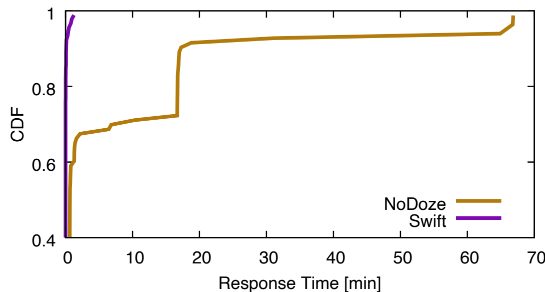

Figure 7: Response times *in minutes* **to return concise causal graphs of threat alerts using Swift as compared to NoDoze (baseline). Note that the Swift CDF is the same as in Figure 6 on a different scale.**

cache would be useless and the baseline may take hours or days to process individual alerts.

Reasons for Milli-second Level Response Time. To further investigate the reason for the time reduction, we also studied what was maintained in the memory for these attacks. In Table 2, the column "All Events" represents all enterprise-wide system events collected while "Critical Events" represents only attack-related events. "%Cached" shows the percentage of events cached in the memory *end of the day.* Our experiment shows that Swift had a much lower response time because it effectively cached most of the events that were related to attacks in the main memory even if the size of the cache was small compared to the size of total events. Particular, on average, by maintaining about 0.04% of total events of a day, Swift can maintain on average 90% of attack-related events in its cache. In other words, Swift was able to significantly reduce disk IOs while generating causal graphs for attacks. This result also validates our Hypothesis H2. Reasons for why Swift cannot maintain 100% of the attack-related events in its cache will be discussed in **RQ2**.

RQ2: Insights into Cached vs Spilled Events

To further study how causal events are handled in the Swift HSM (i.e. which events are cached, as opposed to being spilled to disk), we select a ransomware attack as a case study from the 10 attacks in Table 2. In this attack, a misconfigured Redis server [25] allows an attacker to log into the server via the `ssh` service as root [23]. The attacker first connects directly to a misconfigured Redis server over its default port, executes the `Flushall` command to erase the whole database, uploads their `ssh` key to the database, then obtains root access to the server by using `CONFIG` to copy the database to the root's `.ssh` directory and renaming it to `authorized_keys`. Once in the enterprise network, the attacker moves laterally in their search for valuable data while simultaneously encrypting data by running an `encryptor` that was downloaded from their remote server. *Time is crucial in this scenario – the earlier we investigate and respond to the attack, the more valuable data we can save.*

This attack generated two alerts which are marked in red dashed arrows in Figure 8. However, these true alerts are among a deluge of unrelated false alerts being generated by TDS, making it critical to quickly identify the true alerts and take actions to prevent damages. Fortunately, Swift assigns suspicious influence scores in real-time; when **Alert 1** arrives, Swift automatically remembers its suspiciousness score and propagates this score to its successors. When **Alert 2** fires, Swift combines the suspiciousness influence scores from **Alert 1** in $O(1)$ time. This means that as soon as **Alert 2** is fired by TDS, Swift can instantaneously generates the most suspicious causal graph and correlate the alerts.

Figure 8 shows the simplified causal graph of this attack. In this graph, we use diamonds to represent sockets, oval nodes to

Table 2: Comparison of Swift's effectiveness against baseline. "#Alerts" shows how many alerts are associated with the particular attack.

Attacks	Reference	#Alerts	All Events		Critical Events		Response Time		
			Total	%Cached	Total	%Cached	Baseline	Swift	Speedup
VPNFilter	NoDoze [41]	1	150M	0.06%	15	100%	0.5 min	0.65 ms	46,000×
Redis-Server	Case Study Sec. 8.2	2	100M	0.07%	29	86%	1.1 min	0.1 ms	660,000×
wget	Xu et al. [74]	1	160M	0.03%	15	100%	0.7 min	1.1 ms	28,000×
WannaCry	NoDoze [41]	2	139M	0.05%	21	90%	1.5 min	0.2 ms	450,000×
Data Theft	PrioTracker [57]	1	366M	0.04%	13	88%	3 min	0.3 ms	600,000×
ShellShock	CVE-2014-6271	1	366M	0.04%	25	82%	2 min	0.09 ms	1,333,000×
Netcat Backdoor	Backdoor [8]	1	366M	0.04%	14	85%	1.8 min	0.8 ms	135,000×
Cheating Student	ProTracer [61]	1	336M	0.03%	37	94%	2.1 min	1.9 ms	66,000×
passwd-gzip-scp	Xu et al. [74]	1	335M	0.02%	25	90%	4.6 min	2.8 ms	98,000×
Jeep-Cherokee	Exploit Vehicle [64]	1	129M	0.06%	16	94%	1.2 min	1.2 ms	60,000×

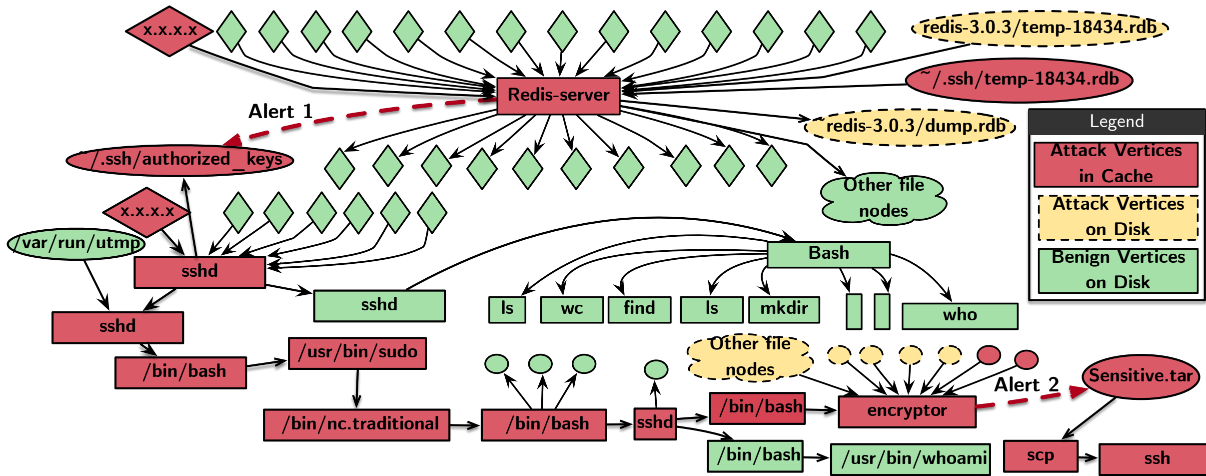

Figure 8: Simplified causal graph of the simulated ransomware attack. Swift keeps part of the causal graph related to the ransomware attack in the main-memory (red vertices), and part of that graph (yellow vertices) is spilled to the disk. Causal graph not related to the attack (green vertices) is spilled to the disk.

represent files, and boxes to represent processes. In Figure 8, the red vertices represent the most suspicious causal graph which is cached in the main-memory. Yellow vertices are related to attack but spilled to disk while green vertices are not related to attack (benign) which are also spilled to disk. Due to dependency explosion problem (false dependency) [61] benign vertices become part of attack's causal graph. Swift shows the most suspicious graph (red vertices) to cyber analyst accelerate investigation and assist cyber analyst to *quickly* identify the root cause (`X.X.X.X` connection to process `Redis-server`) and ramification (`Sensitive.tar` read by process `scp`) of this attack using this subgraph.

As can be seen in Table 2, 14% of attack-related vertices (yellow vertices) were spilled to the disk. The main reason for this was our conservative Global List size ($k = 3000$); these attack-related vertices fell outside of the top-k most suspicious paths, leading to their eviction from the suspicious cache. *We found in our experiments that increasing the Global list size from $k = 3000$ to $k = 5000$ was sufficient to store 100% of attack-related vertices in the cache.* In considering the $k = 3000$ configuration, some temporary files created by the `Redis-server` process, such as `/redis-3.0.3/temp-18434.rdb`, are assigned low suspicious scores because redis regularly creates many such files. However, the temporary file `~/.ssh/temp18434.rdb` was highly unusual because `Redis-server` never writes to the `~/.ssh` folder. As a result, it had a high suspiciousness score and was retained in cache. Note that missing some temporary files from the causal graph does not break causal analysis since we can still identify the root cause and ramifications using red vertices alone. Further, cyber analysts can still retrieve these yellow vertices from disk later for further investigation.

RQ3: Scalability

Throughput. We define the throughput of Swift as the maximum number of events that Swift can process under different configuration values of the global list size k, the eviction time window ΔT_{evict}, the promotion epoch $\Delta T_{promote}$, and the number of threads. To stress test Swift, we replayed the audit logs from our enterprise engagement at the maximal speed. The results of our throughput experiment are shown in Figure 9. Since our eviction algorithm is *asynchronous*, the throughput does not change under different configurations except when we change the number of consumer threads. We can see that Swift can process up to 100,000

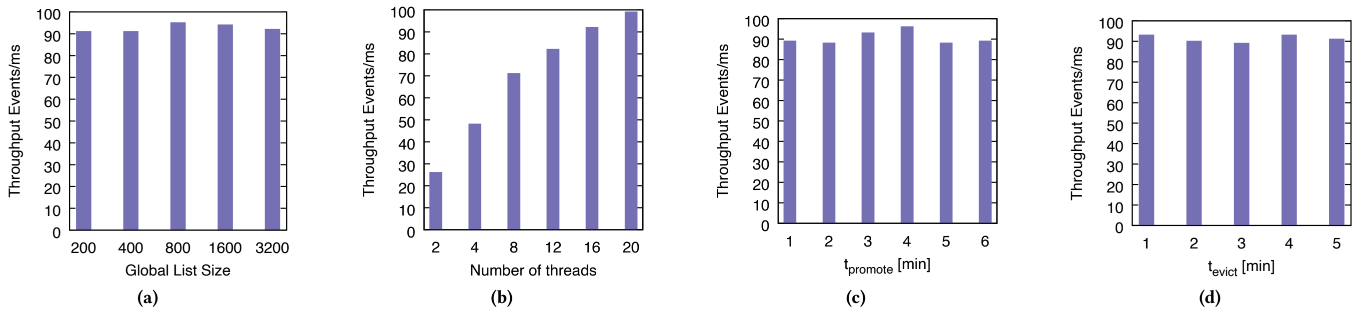

Figure 9: Throughput of Swift under different configuration values.

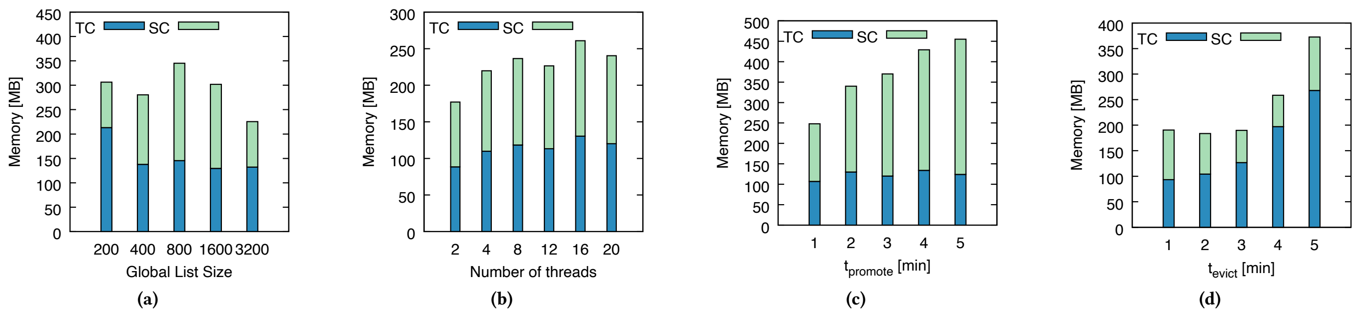

Figure 10: Max. memory usage of Swift under different configuration values when ran for one day. TC stands for tracking cache and SC stands for suspicious cache.

events/sec when the number of threads is 20, which was the maximum number of threads allowed by our machine. Note that, in our experiment, each of the 191 hosts generated less than 5,000 events/sec on average, which is far less than the maximal throughput of Swift. Assuming that this event generation rate holds, *our prototype implementation of Swift can scale to support up to 4,000 hosts with a single server.*

Memory Usage. Another aspect of scalability is memory usage. In our implementation of Swift, memory is consumed by two components: the Kafka framework and the cache for events. Since Kafka is only used as a black-box infrastructure in our implementation and could have very different configurations in practice, we focus on the memory usage of the cache. In our experiment, we first measured the maximum memory used by both tracking cache and suspicious cache under different configuration values while monitoring all the 191 hosts for one day. The results are shown in Figure 10. Changing global list size and threads does not affect the maximum usage of tracking cache and suspicious cache. Increasing the $\Delta T_{promote}$ increases the size of tracking cache because events stay longer there. On the other hand, increasing ΔT_{evict} window increases the suspicious cache usage since eviction algorithm runs after long time. Our experiment shows that in general, Swift could process the workload for 191 hosts in an enterprise with 300 MB memory. For a server with 64 GB memory, as we have used in our experiment, it is possible to handle thousands of hosts at the same time.

RQ4: Benefits of Time Saved

Using causal analysis in state-of-the-art alert triage systems [41], it takes on average 1 min to respond to forensic queries, with a worst case performance of 2.5 hours; because response time grows linearly with graph size, we can expect alerts related to sophisticated intruders to fall closer to this worst-case because they employ a "low and slow" attack approach. On the other hand, Swift responds to queries in just 0.1 sec on average, with worst case performance of 1 minute. This effectively provides investigators with alert context (i.e., causal graphs) as soon as the alert is triggered.

Still, it could be argued that an average response time of 1 minute (as opposed to Swift's 100 milliseconds) is suitably fast for cyber analysts. However, *it is important to consider the fact investigation latency compounds as the number of alerts increases.* Recent studies [12, 18, 22] have shown that organizations receive around 10,000 alerts per week. For simplicity, let us assume that all 10,000 alerts need to be investigated,[5] and that a true attack falls at each quartile (i.e., alerts 2500, 5000, etc.) of the stack. For the first quartile, NoDoze [41] imposes at least 41 hours of latency due to causal analysis, while Swift will impose just 4 minutes of latency. By the last quartile, NoDoze will have imposed 166 hours of latency, while Swift introduces just 16 minutes. Further, we can assign a financial cost to this difference – studies have shown that it costs

[5] In practice, alert triage systems may be used to condense or procedurally exclude some alerts so that they need not be investigated; however, this exercise demonstrates the value of eliminating causal analysis latency from the threat investigation process.

an organization $32,000 for every day an attacker stays in the network [36]. Thus, for just the attack in the last quartile, SWIFT could save the organization up to $221,244 as compared to the previous state-of-the-art.

RQ5: Effectiveness in Alert Management

To answer this research question, we measured the performance and accuracy of SWIFT as an alert management system. We used HSM's suspicious influence scores for all the alerts and triaged alerts based on scores. After that we compared our accuracy and performance with the baseline approach [41]. Since our suspicious influence scores were similar to the baseline approach, SWIFT has the same accuracy (false and true positive rates) as the baseline approach. However, the performance of SWIFT is magnitudes of times better than baseline.

The performance of SWIFT over the baseline approach is measured in terms of response time. As we have already shown in the CDF in Figure 7 that it took total of 5 hours to rank all the 140 alerts using baseline. The reason for this is that baseline as an offline approach first generate the causal graph using disk storage for each alert. After that, it assigns suspiciousness influence scores to each alert's graph and then triage them based on these scores. On the other hand it took SWIFT around 1 minute to rank all the 140 threat alerts because it generates causal graph in an online fashion and assigns suspiciousness influence scores as events arrive and keeps most suspicious causal graphs in the main-memory. Thus, as soon as alerts are fired by underlying TDS, SWIFT already has its graph with suspiciousness score and just need to lookup this score from the cache to triage which $O(1)$ time. Since SWIFT also keeps track of all the alerts fired on causal graph, it instantaneously correlates new alert with all the previous alerts that are causally related.

9 DISCUSSION & LIMITATIONS

Design of suspicious influence scoring system. SWIFT is effective with an arbitrary suspicious influence scoring system that satisfies all three properties described in Section 4.2. In Section 8.1, we implemented an anomaly-based scoring system as the reference in our evaluation. Other scoring systems, such as rule-based or label-propagation-based systems, can also be applied as long as they meet the three requirements.

Possible Attacks. One possible attack to spoil the cache of SWIFT is by exploiting Hypothesis H1 – the adversary may conduct an attack in a longer time window so that the causal paths of the attack in the cache are eventually replaced by causal paths of other attacks and suspicious activities. This attack can be alleviated by allocating large memory (Global List size). As long as there is enough space, SWIFT will maintain the suspicious causal paths in the cache. If there is not enough memory space, choosing which suspicious paths to keep in the cache would be a trade-off. We leave this discussion for our future work. Adversaries may also try to spoil the cache of SWIFT by generating anomalous events and causal paths in provenance data. This can be solved by having more accurate underlying anomaly detection techniques. In this paper, we apply one commercial tool [13] to detect anomalies. Although it is important to improve the accuracy of anomaly detection, it is orthogonal to our study. Nevertheless, even if the cache is spoiled, an investigator can still generate a complete causal graph but with some delay due to disk IO.

Adversaries may try to spoil the cache system of SWIFT to degenerate its responsiveness by having a "spoofing attack". The adversary may conduct an attack that contains a vertex that is involved in more than K most suspicious paths to occupy the whole global list (e.g. unzipping more than K files from a .ZIP package). Under the "spoofing attack," causal paths of other vertices are evicted to the disk so the performance of SWIFT to investigate other vertices is degraded to existing offline solutions. To address this attack, SWIFT only selects candidates from the $PATH_{abnormal}$ of each vertex. Since the size of $PATH_{abnormal}$ of each vertex is limited to m, each vertex will only occupy at most m slots in the global list.

Another type of spoofing attack is that the adversary may generate a lot of different suspicious events to occupy the cache. Since we keep the longest path in the cache, the adversary needs to generate a huge number of **independent** suspicious events, which do not have causal dependencies, to spoil the cache. However, if an attacker tries to produce a lot independent suspicious events then it defeats the "low and slow" strategy used by attackers and generates a strong indication of an attack which a threat hunter can immediately spot. Moreover, this problem is equivalent to the problem of having too many suspicious paths that the cache cannot hold, which we leave for future work.

Applicability. The SWIFT approach is generic to provide broad support for fast and interactive threat hunting in enterprises provided that system-level audit logs are being collected and there is an underlying threat detector which monitors enterprise-wide activities. The two key hypotheses presented in Section 6, upon which SWIFT is built, are enterprise agnostic. These hypotheses are derived from fundamental characteristics of system-level audit logs [3, 4]. This ensures that our techniques can be applied in different enterprises without sacrificing performance and accuracy.

10 CONCLUSION

Threat hunting using causality analysis has an insatiable demand for high throughput and low latency investigation queries. In this paper, we present SWIFT, an online causality tracker that directly works on audit logs provided from commodity systems in an enterprise. SWIFT is capable of identifying in-progress threats and provides quick investigation capabilities to a cyber analyst before serious damage is inflicted. We implemented SWIFT in 7k LoC of Java and deployed at NEC Labs America. Our evaluation results show that SWIFT can precisely capture all the causality related to the true attacks and store them in the main-memory which subsequently speeds up after-the-fact investigation.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments and suggestions. Wajih Ul Hassan was partially supported by the Sohaib and Sara Abbasi Computer Science Fellowship. This work was funded in part by the NSF under contracts CNS-16-57534 and CNS-17-50024. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of their employers or the sponsors.

REFERENCES

[1] [n.d.]. Cortex XDR. https://www.paloaltonetworks.com/cortex/cortex-xdr.
[2] [n.d.]. CrowdStrike. https://www.crowdstrike.com/.
[3] [n.d.]. Event tracing. https://docs.microsoft.com/en-us/windows/desktop/ETW/event-tracing-portal.
[4] [n.d.]. The Linux audit daemon. https://linux.die.net/man/8/auditd.
[5] [n.d.]. MTTD vs MTTK. https://www.threatstack.com/blog/how-to-use-automation-to-decrease-mean-time-to-know.
[6] [n.d.]. Netwrix Auditor. https://www.netwrix.com/network_auditing_software_features.html.
[7] 2014. CVE-2014-6271. https://nvd.nist.gov/vuln/detail/CVE-2014-6271.
[8] 2018. Persistent netcat backdoor. https://www.offensive-security.com/metasploit-unleashed/persistent-netcat-backdoor/.
[9] 2018. Ransom.Wannacry. https://symc.ly/2NSK5Rg.
[10] 2018. VPNFilter: New Router Malware with Destructive Capabilities. https://symc.ly/2IPGGVE.
[11] 2019. Apache Kafka. https://kafka.apache.org/.
[12] 2019. Automated Incident Response: Respond to Every Alert. https://swimlane.com/blog/automated-incident-response-respond-every-alert/.
[13] 2019. Automated Security Intelligence (ASI). https://www.nec.com/en/global/techrep/journal/g16/n01/160110.html.
[14] 2019. Breach Detection. https://link.medium.com/6HpgbLgZuW.
[15] 2019. Cyber Threat Hunting Review. https://blog.usejournal.com/cyber-threat-hunting-basics-52fca11a4e1d.
[16] 2019. Endpoint Monitoring & Security. https://logrhythm.com/solutions/security/endpoint-threat-detection/.
[17] 2019. Google core libraries for Java. https://github.com/google/guava.
[18] 2019. How Many Alerts is Too Many to Handle? https://www2.fireeye.com/StopTheNoise-IDC-Numbers-Game-Special-Report.html.
[19] 2019. How WannaCrypt attacks. https://www.zdnet.com/article/how-wannacrypt-attacks/.
[20] 2019. MS17-010 EternalBlue SMB Remote Windows Kernel Pool Corruption. https://www.rapid7.com/db/modules/exploit/windows/smb/ms17_010_eternalblue.
[21] 2019. Neo4j. https://neo4j.com/.
[22] 2019. New Research from Advanced Threat Analytics. https://prn.to/2uTiaK6.
[23] 2019. Over 18,000 Redis Instances Targetted. https://duo.com/decipher/over-18000-redis-instances-targeted-by-fake-ransomware.
[24] 2019. Petya ransomware outbreak. https://www.symantec.com/blogs/threat-intelligence/petya-ransomware-wiper.
[25] 2019. Redis in-memory data structure store. https://redis.io/.
[26] 2019. RedisGraph - a graph database module for Redis. https://oss.redislabs.com/redisgraph/.
[27] 2019. RocksDB | A persistent key-value store. https://rocksdb.org/.
[28] 2019. What is SIEM? https://logz.io/blog/what-is-siem/.
[29] 2 [n.d.]. Equifax says cyberattack may have affected 143 million in the U.S. https://www.nytimes.com/2017/09/07/business/equifax-cyberattack.html.
[30] Adam Bates, Wajih Ul Hassan, Kevin Butler, Alin Dobra, Bradley Reaves, Patrick Cable, Thomas Moyer, and Nabil Schear. 2017. Transparent web service auditing via network provenance functions. In *WWW*.
[31] Adam Bates, Dave Tian, Kevin R. B. Butler, and Thomas Moyer. 2015. Trustworthy whole-system provenance for the Linux kernel. In *USENIX Security*.
[32] Chen Chen, Harshal Tushar Lehri, Lay Kuan Loh, Anupam Alur, Limin Jia, Boon Thau Loo, and Wenchao Zhou. 2017. Distributed Provenance Compression. In *SIGMOD*.
[33] Marco Cova, Davide Balzarotti, Viktoria Felmetsger, and Giovanni Vigna. 2007. Swaddler: An approach for the anomaly-based detection of state violations in web applications. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 63–86.
[34] Hervé Debar and Andreas Wespi. 2001. Aggregation and correlation of intrusion-detection alerts. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 85–103.
[35] David Ediger, Rob McColl, Jason Riedy, and David A Bader. 2012. Stinger: High performance data structure for streaming graphs. In *Conference on High Performance Extreme Computing*. IEEE.
[36] FireEye. 2019. Incident Investigation. https://www.fireeye.com/solutions/incident-investigation.html.
[37] Peng Gao, Xusheng Xiao, Ding Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, Chung Hwan Kim, Sanjeev R. Kulkarni, and Prateek Mittal. 2018. SAQL: A Stream-based Query System for Real-Time Abnormal System Behavior Detection. In *USENIX Security Symposium*.
[38] Ashish Gehani and Dawood Tariq. 2012. SPADE: Support for provenance auditing in distributed environments. In *Middleware* (Montreal, Quebec, Canada).
[39] Xueyan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. 2020. Unicorn: Runtime Provenance-Based Detector for Advanced Persistent Threats. In *NDSS*.
[40] Wajih Ul Hassan, Adam Bates, and Daniel Marino. 2020. Tactical Provenance Analysis for Endpoint Detection and Response Systems. In *IEEE S&P*.
[41] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. 2019. NoDoze: Combatting threat alert fatigue with automated provenance triage. In *NDSS* (San Diego, CA).
[42] Wajih Ul Hassan, Mark Lemay, Nuraini Aguse, Adam Bates, and Thomas Moyer. 2018. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *NDSS* (San Diego, CA).
[43] Wajih Ul Hassan, Mohammad A Noureddine, Pubali Datta, and Adam Bates. 2020. OmegaLog: High-Fidelity Attack Investigation via Transparent Multi-layer Log Analysis. In *NDSS*.
[44] Md Nahid Hossain, Sadegh M Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R Sekar, Scott D Stoller, and VN Venkatakrishnan. 2017. SLEUTH: Real-time attack scenario reconstruction from COTS audit data. In *USENIX Security*.
[45] Md Nahid Hossain, Sanaz Sheikhi, and R Sekar. 2020. Combating Dependence Explosion in Forensic Analysis Using Alternative Tag Propagation Semantics. In *IEEE S&P*.
[46] Md Nahid Hossain, Junao Wang, R. Sekar, and Scott D. Stoller. 2018. Dependence-Preserving data compaction for scalable forensic analysis. In *USENIX Security Symposium*.
[47] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. 2017. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *CCS*. ACM.
[48] Samuel T. King and Peter M. Chen. 2003. Backtracking Intrusions. In *SOSP*. ACM.
[49] Samuel T King, Zhuoqing Morley Mao, Dominic G Lucchetti, and Peter M Chen. 2005. Enriching Intrusion Alerts Through Multi-Host Causality.. In *NDSS*.
[50] Christopher Kruegel, Darren Mutz, William Robertson, and Fredrik Valeur. 2003. Bayesian event classification for intrusion detection. In *19th Annual Computer Security Applications Conference, 2003. Proceedings*. IEEE, 14–23.
[51] Christopher Kruegel and Giovanni Vigna. 2003. Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 251–261.
[52] Pradeep Kumar and H. Howie Huang. 2019. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *USENIX FAST*.
[53] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela Ciocarlie, et al. 2018. MCI: Modeling-based Causality Inference in Audit Logging for Attack Investigation. In *NDSS*.
[54] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High accuracy attack provenance via binary-based execution partition. In *NDSS* (San Diego, CA).
[55] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. LogGC: Garbage collecting audit log. In *CCS*.
[56] Hyeontaek Lim, Donsu Han, David G Andersen, and Michael Kaminsky. 2014. MICA: A holistic approach to fast in-memory key-value storage. USENIX.
[57] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. 2018. Towards a Timely Causality Analysis for Enterprise Security. In *NDSS*.
[58] Shiqing Ma, Kyu Hyung Lee, Chung Hwan Kim, Junghwan Rhee, Xiangyu Zhang, and Dongyan Xu. 2015. Accurate, low cost and instrumentation-free security audit logging for Windows. In *ACSAC*. ACM.
[59] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. 2018. Kernel-supported cost-effective audit logging for causality tracking. In *USENIX ATC*.
[60] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2017. MPI: Multiple Perspective Attack Investigation with Semantic Aware Execution Partitioning. In *USENIX Security*.
[61] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. ProTracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS* (San Diego, CA).
[62] Sadegh M. Milajerdi, Birhanu Eshete, Rigel Gjomemo, and V.N. Venkatakrishnan. 2019. POIROT: Aligning Attack Behavior with Kernel Audit Records for Cyber Threat Hunting. In *CCS*. ACM.
[63] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. N. Venkatakrishnan. 2019. HOLMES: Real-Time APT Detection through Correlation of Suspicious Information Flows. In *IEEE S&P*.
[64] Charlie Miller and Chris Valasek. 2015. Remote exploitation of an unaltered passenger vehicle. (2015).
[65] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. 2003. Inside the Slammer Worm. *IEEE Security and Privacy* 1, 4 (July 2003), 33–39. https://doi.org/10.1109/MSECP.2003.1219056
[66] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. [n.d.]. Scaling Memcache at Facebook.
[67] Steven Noel, Eric Robertson, and Sushil Jajodia. 2004. Correlating intrusion events and building attack scenarios through attack graph distances. In *Computer Security Applications Conference, 2004. 20th Annual*. IEEE, 350–359.
[68] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, and Margo Seltzer. 2018. Runtime analysis of whole-system provenance. In *CCS*. ACM.

[69] Xiaokui Shu, Frederico Araujo, Douglas L Schales, Marc Ph Stoecklin, Jiyong Jang, Heqing Huang, and Josyula R Rao. 2018. Threat intelligence computing. In *ACM CCS*.
[70] Splunk Inc. [n.d.]. splunk. https://www.splunk.com.
[71] Yutao Tang, Ding Li, Zhichun Li, Mu Zhang, Kangkook Jee, Xusheng Xiao, Zhenyu Wu, Junghwan Rhee, Fengyuan Xu, and Qun Li. 2018. Nodemerge: Template based efficient data reduction for big-data causality analysis. In *CCS*. ACM.
[72] Alfonso Valdes and Keith Skinner. 2001. Probabilistic alert correlation. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 54–68.
[73] Yulai Xie, Kiran-Kumar Muniswamy-Reddy, Darrell D. E. Long, Ahmed Amer, Dan Feng, and Zhipeng Tan. 2011. Compressing Provenance Graphs.
[74] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. 2016. High Fidelity Data Reduction for Big Data Security Dependency Analyses. In *CCS*.

A APPENDIX

A.1 Optimal Parameters

Swift has three configurable parameters:$\Delta T_{promote}$, Global list size k, and ΔT_{evict}. In this section, we evaluate our reasons of choosing the optimal configuration in our experiments.

A.1.1 Promotion Epoch. The Epochal Causality Hypothesis H1 for causal graph pattern access states that we need to keep events for a certain epoch ($\Delta T_{promote}$) in the tracking cache to minimize the miss ratio of the main-memory for causal events. So, in order to find the promotion epoch $\Delta T_{promote}$ for our enterprise dataset, we set out to experiment with different $\Delta T_{promote}$. Intuitively, larger the $\Delta T_{promote}$ means that our tracking cache miss ratio will be less; however, this also means that we will be keeping more events in the tracking cache. So essentially we need to find a sweet spot between miss ratio and maximum size of tracking cache. Therefore, we define the tracking cache (TC) utilization as follows:

$$TC_{utilization} = Miss_{ratio} \times Max_{events}$$

In the above equation, we multiple the miss ratio by max events that are present in tracking cache during each run to get utilization. Our results are shown in the Figure 11. We can see that with 800 sec we get maximum utilization where we have both low miss ratio and low maximum TC size.

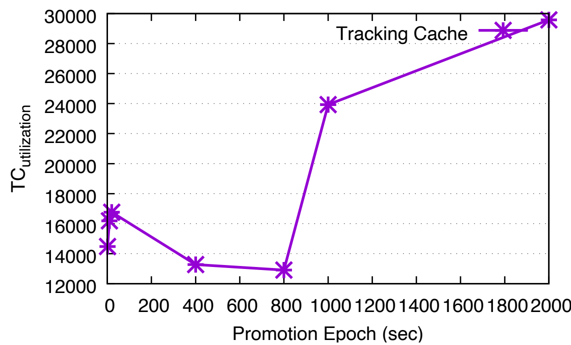

Figure 11: Finding optimal Promotion Epoch $\Delta T_{promote}$ for our experiment dataset.

A.1.2 Global List Size. Global list size k is directly correlated with number of most anomalous paths any enterprise wants to store in the main-memory. Given an enterprise with larger resources can potentially store more anomalous paths in the main-memory which can accelerate the investigation of threats alerts later. One thing that is affected by the large global list size is the time to complete each eviction cycle after every ΔT_{evict} time windows as shown in the Figure 12a. However, since our eviction algorithm is asynchronous it does not affect the throughput of our system.

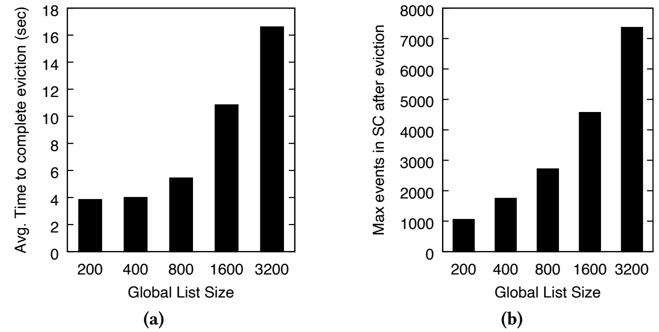

Figure 12: Effect of changing global list size on average time to complete on eviction cycle and max number of events in the suspicious cache (SC) after eviction.

Another factor that can be used to find out required global list size for an enterprise is the number of alerts that are generated by enterprise's underlying TDS. Since these threat alerts are related to anomalous behaviour, by storing large global list size we can store more information regarding these alerts in the main-memory all the time. NEC Labs America generated 300 threat alerts per day. Since our attack dataset spans over 10 days we set the global list size to be 3000 so that we can store at least 10 days of threat alerts' provenance data in the main-memory. Figure 12b shows how increasing the global list size increased the overall size of main-memory.

A.1.3 Eviction Window. We run eviction algorithm after every ΔT_{evict} time window. Greater the eviction window time, the greater Swift will take to complete one eviction cycle as shown in the Figure 13. However, since our eviction algorithm is asynchronous it does not matter how much time it takes to complete one eviction cycle. For experiments we picked ΔT_{evict} = 1600 since it worked well during our deployment; however, any value of ΔT_{evict} can be picked by the user as long as it is after $\Delta T_{promote}$ time window.

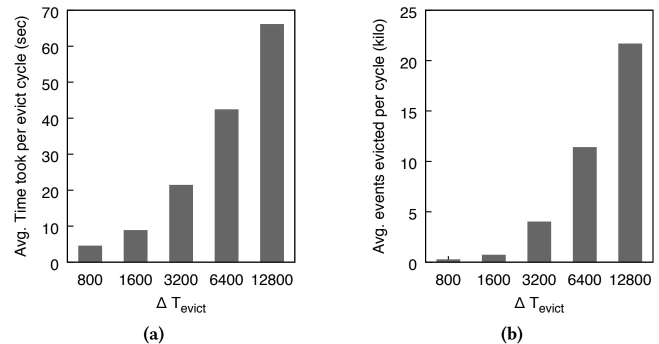

Figure 13: Effect of changing ΔT_{evict} on average time to complete on eviction cycle.