UNICORN: Runtime Provenance-Based Detector for Advanced Persistent Threats

Xueyuan Han*, Thomas Pasquier†, Adam Bates‡, James Mickens* and Margo Seltzer§
*Harvard University
{hanx,mickens}@g.harvard.edu
†University of Bristol
thomas.pasquier@bristol.ac.uk
‡University of Illinois at Urbana-Champaign
batesa@illinois.edu
§University of British Columbia
mseltzer@cs.ubc.ca

***Abstract*—Advanced Persistent Threats (APTs) are difficult to detect due to their "low-and-slow" attack patterns and frequent use of zero-day exploits. We present UNICORN, an anomaly-based APT detector that effectively leverages data provenance analysis. From modeling to detection, UNICORN tailors its design specifically for the unique characteristics of APTs. Through extensive yet time-efficient graph analysis, UNICORN explores provenance graphs that provide rich contextual and historical information to identify stealthy anomalous activities without pre-defined attack signatures. Using a graph sketching technique, it summarizes long-running system execution with space efficiency to combat slow-acting attacks that take place over a long time span. UNICORN further improves its detection capability using a novel modeling approach to understand long-term behavior as the system evolves. Our evaluation shows that UNICORN outperforms an existing state-of-the-art APT detection system and detects real-life APT scenarios with high accuracy.**

I. INTRODUCTION

Advanced Persistent Threats (APT) are becoming increasingly common [9]. The long timescale over which such attacks take place makes them fundamentally different from more conventional attacks. In an APT, the adversary's goal is to gain control of a specific (network of) system(s) while remaining undetected for an extended period of time [116]. The adversary often relies on zero-day exploits [96, 116] to gain a foothold in the victim system.

Traditional detection systems are not well-suited to APTs. Detectors dependent on malware signatures are blind to attacks that exploit new vulnerabilities [20]. Anomaly-based systems typically analyze series of system calls [114] and log-adjacent system events [130], but most of them [36, 81, 92, 109] have difficulty modeling long-term behavior patterns. Further, they are susceptible to evasion techniques, because they typically inspect only short sequences of system calls and events. As a result, they have exhibited little success in detecting APTs. Systems that attempt to capture long-term program behavior [112] limit their analysis to event co-occurrence to avoid high computational and memory overheads.

More recent work [15, 19, 83, 87] suggests that data provenance might be a better data source for APT detection. Data provenance represents system execution as a directed acyclic graph (DAG) that describes information flow between system subjects (e.g., processes) and objects (e.g., files and sockets). It connects causally-related events in the graph, even when those events are separated by a long time period. Thus, even though systems under APT attack usually behave similarly to unattacked systems, the richer contextual information in provenance allows for better separation of benign and malicious events [55].

However, leveraging data provenance to perform runtime APT analysis is difficult. Provenance graph analysis is computationally demanding, because the graph size grows continuously as APTs slowly penetrate a system. The necessary contextual analysis, which requires large graph components, makes the task even more challenging. One approach to provenance-based APT detection [87] uses simple edge-matching rules based on prior attack knowledge, but this makes it difficult to detect new classes of APTs [15]. Provenance-based anomaly detection systems rely on graph neighborhood exploration to understand normal behavior through either static [53] or dynamic [83] models. However, practical computational constraints limit the scope of the contextual analysis that is feasible. Thus, current systems suffer from some combination of the following three problems: 1) static models cannot capture long term system behavior, 2) the low-and-slow APT approach can gradually poison dynamic models, and 3) approaches that require in-memory computation [83, 87] scale poorly in the presence of long-running attacks [87].

We introduce UNICORN, a provenance-based anomaly detector capable of detecting APTs. UNICORN uses graph sketching to build an incrementally updatable, fixed size, longitudinal graph data structure [1] that enables efficient computation of graph statistics [33]. This longitudinal nature of the structure permits extensive graph exploration, which allows UNICORN

[1]An algorithm that builds a graph data structure is called a *graph kernel*, which is an overloaded term that also refers to functions that compare the similarity between two graphs. We adopt the second definition here.

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24046
www.ndss-symposium.org

to track stealthy intrusions. The fixed size and incrementally updatable graph data structure obviates the need for an in-memory representation of the provenance graph, so UNICORN is scalable with low computational and storage overheads. It can succinctly track a machine's entire provenance history from boot to shutdown. UNICORN directly models a system's evolving behavior during training, but it does not update models afterward, preventing an attacker from poisoning the model.

We make the following contributions:

- We present a provenance-based anomaly detection system tailored to APT attacks.
- We introduce a novel sketch-based, time-weighted provenance encoding that is compact and able to usefully summarize provenance graphs over long time periods.
- We evaluate UNICORN against both simulated and real-life APT attack scenarios. We show that UNICORN can detect APT campaigns with high accuracy. Compared to previous work, UNICORN significantly improves precision and accuracy by 24% and 30%, respectively.
- We provide an open source implementation.

II. BACKGROUND

Traditional anomaly-based intrusion detection systems (IDS) analyze system call traces from audit systems [37, 81, 130, 134]. However, for APT detection, whole-system provenance [100] is a superior data source.

A. Challenges of Syscall Traces

The system call abstraction provides a simple interface by which user-level applications request services of the operating system. As the mechanism through which system services are invoked, the system call interface is also generally the entry point for attackers trying to subvert a system [61]. Therefore, system call traces have long been regarded as the *de facto* information source for intrusion detection [37].

However, existing systems capture unstructured collections of syscall audit logs, requiring analysis to make sense of such information. Given the low-and-slow nature of APTs, analyzing individual syscall logs to detect point-wise outliers is often fruitless [23], as is inspection of short system call sequences. Such analysis does not reflect the historical context of each syscall event, resulting in high false positive rates and mimicry attacks that evade detection [98, 123]. In contrast, data provenance encodes historical context into causality graphs [21].

Data provenance can be used to model a variety of event sequences in computing, including syscall audit logs. Indeed, there exist frameworks that reconstruct provenance graph structures from audit data streams to allow for better reasoning about system execution [44]. However, such post-hoc approaches make it difficult to ensure graph correctness [104]; it is difficult to prove completeness, trustworthiness, or reliability of the syscall traces from which the graph is built, because many syscall interposition techniques suffer from concurrency issues [43, 125]. It is easy to bypass library-wrapper-based syscall capture mechanisms [63], while user-space mechanisms (e.g., `ptrace`) incur unacceptable runtime performance overheads [43] and are susceptible to race conditions [63]. The same race condition issues also plague in-kernel mechanisms (e.g., `Systrace` [105], Janus [46]), resulting in time-of-check-to-time-of-use (TOCTTOU), time-of-audit-to-time-of-use (TOATTOU), and time-of-replacement-to-time-of-use (TORTTOU) bugs [125]. For example, many IDS [81, 130] analyze syscalls and their arguments to defend against mimicry attacks, but TOATTOU bugs cause the captured syscall arguments to be different from the true values accessed in the kernel [43]. Perhaps more importantly, instead of a single graph of system execution, syscall-based provenance frameworks produce many disconnected graphs, because they cannot trace the interrelationships of kernel threads that do not make use of the syscall interface. Consequently, such frameworks can rarely detect the stealthy malicious events found in APTs.

B. Whole-System Provenance

Whole-system provenance collection runs at the operating system level, capturing all system activities and the interactions between them [104]. OS-level provenance systems such as Hi-Fi [104], LPM [16], and CamFlow [100] provide strong security and completeness guarantees with respect to information flow capture. This completeness is particularly desirable in APT scenarios as it captures long-distance causal relationships enabling contextualized analysis, even if a malicious agent manipulates security-sensitive kernel objects to hide its presence.

We use CamFlow [100] as the reference implementation throughout the paper, although there exist other whole-system provenance implementations; in § VI, we show that UNICORN works seamlessly with other capture mechanisms as well. CamFlow adopts the Linux Security Modules (LSM) framework [89] to ensure high-quality, reliable recording of information flows among data objects [45, 101]. LSM eliminates race conditions (e.g., TOCTTOU attacks) by placing mediation points inside the kernel instead of at the system call interface [61].

C. Summary and Problem Statement

Prior research [15, 83, 87] explored the use of data provenance for APT detection. However, these approaches all suffer from some combinations of the following limitations:

Ⓛ1: Pre-defined edge-matching rules are overly sensitive and make it difficult to detect zero-day exploits common in APTs [87].
Ⓛ2: Constrained provenance graph exploration provides only limited understanding of information context critical to detect low-profile anomalies. For example, graph exploration is limited to small graph neighborhoods, single node/edge attributes, and truncated subgraphs [15, 53, 83].
Ⓛ3: System behavior models fail to cater to the unique characteristics of APT attacks. Static models cannot capture dynamic behavior of long-running systems [53], while dynamic modeling during runtime risks poisoning from the attackers [83].
Ⓛ4: Provenance graphs are stored and analyzed only in memory, sacrificing long-term scalability [83, 87].

UNICORN addresses those issues. We formalize the system-wide intrusion detection problem in APT campaigns as a real-time graph-based anomaly detection problem on large, attributed, streaming whole-system provenance graphs. At any point in time, the entirety of a provenance graph, captured from system boot to its current state is compared against a behavior model consisting of known good provenance graphs. The system is considered under attack if its provenance graph deviates significantly from the model. For APT detection, an ideal provenance-based IDS must:

- Continuously analyze provenance graphs with space and time efficiency while taking full advantage of rich information content provided by whole-system provenance graphs;
- Take into consideration the entire duration of system execution without making assumptions of attack behavior;
- Learn only normal system behavior changes but not those directed by the attackers.

III. Threat Model

We assume APT scenarios for host intrusion detection: an attacker illegitimately gains access to a system and plans to remain there for an extended period of time without being detected. The attacker may conduct the attack in several phases and use a variety of techniques during each phase [131]. The goal of UNICORN is to detect such attacks at any stage by interpreting the provenance generated by the host. We assume that, prior to the attack, UNICORN thoroughly observes the host system during normal operation and that no attacks arise during this initial modeling period.

The integrity of the data collection framework is central to UNICORN's correctness. We assume that Linux Security Modules [89], which is the Linux reference monitor implementation, correctly provides reference monitor guarantees [10] for CamFlow [100]. Specifically, we assume that LSM integrity is provided via an attested boot sequence [16]. We make similar integrity assumptions for other data collection frameworks. UNICORN can further safeguard its data source by streaming it across the network. While we primarily envision UNICORN as an endpoint security monitor, UNICORN's ability to stream provenance data enables off-host intrusion detectors that are not co-located with a potentially compromised machine.

For the remainder of the paper, we assume the correctness of the kernel, the provenance data, and the analysis engine. We instead focus on UNICORN's analytic capabilities.

IV. Design

UNICORN is a host-based intrusion detection system capable of simultaneously detecting intrusions on a collection of networked hosts. We begin with a brief overview of UNICORN and then follow with a detailed discussion of each system component in the following sections. Fig. 1 illustrates UNICORN's general pipeline.

① Takes as input a labeled, streaming provenance graph. UNICORN accepts a stream of attributed edges produced by a provenance capture system running on one or more networked hosts. Provenance systems construct a single, whole-system provenance DAG with a partial-order guarantee, which allows for efficient streaming computation (§ IV-B) and fully contextualized analysis (Ⓛ2). We present UNICORN using CamFlow [100], although it can obtain provenance from other

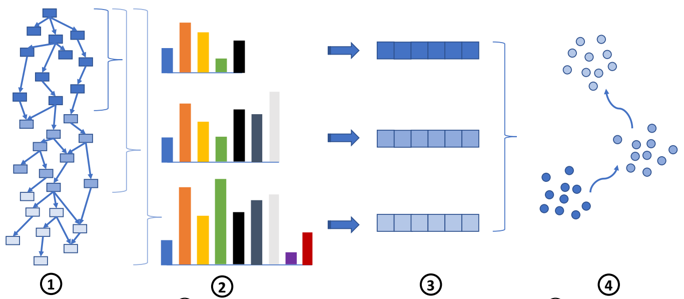


Fig. 1: UNICORN ① takes a streaming provenance graph, ② periodically summarizes graph features into histograms, and then ③ creates fixed-size graph sketches. The resulting clustering-based model ④ captures the dynamics of system execution. During deployment, graph sketches are created through the same steps (①, ② and ③) and then compared against the model in ④.

systems, such as LPM [16] and Spade [44], the latter of which interoperates with commodity audit systems such as Linux Audit and Windows ETW.

② Builds at runtime an in-memory histogram. UNICORN efficiently constructs a streaming graph histogram that represents the entire history of system execution, updating the counts of histogram elements as new edges arrive in the graph data stream. By iteratively exploring larger graph neighborhoods, it discovers causal relationships between system entities providing execution context. This is UNICORN's first step in building an efficient data structure that facilitates contextualized graph analysis (Ⓛ2). Specifically, each element in the histogram describes a unique substructure of the graph, taking into consideration the heterogeneous label(s) attached to the vertices and edges within the substructure, as well as the temporal order of those edges.

To adapt to expected behavioral changes during the course of normal system execution, UNICORN periodically discounts the influence of histogram elements that have no causal relationships with recent events (Ⓛ3). Slowly "forgetting" irrelevant past events allows us to effectively model metastates (§ IV-D) throughout system uptime (e.g., system boot, initialization, serving requests, failure modes, etc.). However, it does not mean that UNICORN forgets informative execution history; rather, UNICORN uses information flow dependencies in the graph to keep up-to-date important, relevant context information. Attackers can slowly penetrate the victim system in an APT, hoping that a time-based IDS eventually forgets this initial attack, but they cannot break the information flow dependencies that are essential to the success of the attack [87].

③ Periodically, computes a fixed-size graph sketch. In a pure streaming environment, the number of unique histogram elements can grow arbitrarily large as UNICORN summarizes the entire provenance graph. This variation in size makes it challenging to efficiently compute similarity between two histograms and impractical to design algorithms for later modeling and detection. UNICORN employs a similarity-preserving hashing technique [132] to transform the histogram to a *graph sketch* [7]. The graph sketch is incrementally maintainable, meaning that UNICORN does not need to keep the entire provenance graph in memory; its size is constant (Ⓛ4). Additionally, graph sketches preserve normalized Jaccard similarity [64] between two graph histograms. This distance-preserving property is particularly important to the clustering algorithm in our later analysis, which is based on the same graph similarity metric.

④ Clusters sketches into a model. UNICORN builds a normal system execution model and identifies abnormal activities without attack knowledge (Ⓛ1). However, unlike traditional clustering approaches, UNICORN takes advantage of its streaming capability to generate models that are *evolutionary*. The model captures behavioral changes within a single execution by clustering system activities at various stages of its execution, but UNICORN does not modify models dynamically during runtime when the attacker may be subverting the system (Ⓛ3). It is therefore more suitable for long-running systems under potential APT attacks.

A. Provenance Graph

Provenance graphs are increasingly popular for attack analysis [15, 40, 79, 102] and are attractive for APT detection. In particular, provenance graphs capture causality relationships between events. Causal connectivity facilitates reasoning over events that are temporally distant, thus useful in navigating through APT's low-and-slow attack pattern. Audit log analysis frequently relies on temporal relationships, while provenance analysis leverages causality relationships, producing a more meaningful model of system behavior.

UNICORN compares two system executions based on the similarity between their corresponding provenance graphs. UNICORN always considers the entire provenance graph to detect long-running attacks. There exist many graph similarity measures, but many approaches (e.g., graph isomorphism) are too restrictive (i.e., require two graphs to be exactly or largely identical) [110], because even normal executions often produce slightly different provenance graphs. Whole-system provenance graphs can grow large quickly [100], so NP-complete [42, 95] and even polynomial algorithms [71, 122] are too computationally expensive for streaming settings. As we show in the following sections, UNICORN's graph similarity algorithm does not exhibit these problems.

B. Constructing Graph Histograms

Our goal is to efficiently compare provenance graphs while tolerating minor variations in normal executions. The two criteria we have for an algorithm are: 1) the representation should take into account long-term causal relationships, and 2) we must be able to implement the algorithm on the real-time streaming graph data so that we can thwart intrusions when they happen (not merely detect them).

We adapt a linear-time, fast Weisfeiler-Lehman (WL) subtree graph kernel algorithm based on one dimensional WL test of isomorphism [126]. The WL test of isomorphism and its subtree kernel variation [110] are known for their discriminative power for a broad class of graphs, beyond many state-of-the-art graph learning algorithms (e.g., graph neural networks [52, 129]).

Our use of the WL subtree graph kernel hinges on our ability to construct a histogram of vertices that captures the graph structure surrounding each vertex. We bin vertices according to augmented vertex labels that describe fully the R-hop neighborhood of the vertex. We construct these augmented vertex labels by iterative label propagation; we provide an intuitive description here and a more formal one below. For simplicity of exposition, assume we have an entire static graph. A single relabeling step takes as input a vertex label, the labels of all its incoming edges, and the labels of the source vertices of all those edges. It then outputs a new label for the vertex representing the aggregation of all the input labels. We repeat this process for every vertex, and then repeat the entire procedure R times to construct labels describing an R-hop neighborhood. Once we have constructed augmented vertex labels for every vertex in the graph, we create a histogram whose buckets correspond to these labels. The WL test of isomorphism compares two graphs based on these augmented vertex labels; two graphs are similar if they have similar distributions across similar sets of labels.

Algorithm 1: Graph Histogram Generation

```
      Input  : $G = (V, E, \mathcal{F}^v, \mathcal{F}^e, \mathcal{C}), R$
      Output: Histogram $H$
1     for $i \leftarrow 1$ to $R$ do
2         foreach $v \in V$ do
3             $M \leftarrow \{\}$
4             if $i == 1$ then
5                 $l_0(v) \leftarrow \mathcal{F}^v(v)$
6             else if $i == 2$ then
7                 $TS \leftarrow \{\}$
8                 foreach $e \in \texttt{In}(v)$ do
9                     $w \leftarrow \texttt{Source}(e)$
10                    $M \leftarrow M + \{\mathcal{F}^e(e) :: l_1(w)\}$
11                    $T(w) \leftarrow \mathcal{C}(e)$
12                    $TS \leftarrow TS + \{T(w)\}$
13                $T(v) \leftarrow \texttt{Min}(TS)$
14            else
15                $TS \leftarrow \{\}$
16                foreach $w \in \mathcal{N}(v)$ do
17                    $M \leftarrow M + \{l_{i-1}(w)\}$
18                    $TS \leftarrow TS + \{T(w)\}$
19                $T(w) \leftarrow \texttt{Min}(TS)$
20            $\texttt{Sort}(M)$ based on timestamps $T(w)$, $\forall w$ whose label is
                included in $M$
21            $\texttt{s}_v \leftarrow l_{i-1}(v) + \texttt{Concat}(M)$
22        foreach $v \in V$ do
23            $l_i(v) \leftarrow \texttt{Hash}(\texttt{s}_v)$
24            if $l_i(v) \notin H$ then $H[l_i(v)] \leftarrow 1$
25            else $H[l_i(v)] \leftarrow H[l_i(v)] + 1$
```

Alg. 1 presents the algorithm more formally. We define a static graph G as a 5-tuple $(V, E, \mathcal{F}^v, \mathcal{F}^e, \mathcal{C})$, where V is the set of vertices, E is the set of directed edges, i.e., $e = (u, v) \in E$ and $(u, v) \neq (v, u)\ \forall\, u, v \in V$. $\mathcal{F}^v : V \rightarrow \Sigma$ is a function that assigns labels from an alphabet Σ to each vertex in the graph, and similarly, $\mathcal{F}^e : E \rightarrow \Psi$ assigns labels from an alphabet Ψ to each edge. $\mathcal{C}$ is a function that records the timestamp of each edge. In line 1, R is the number of neighborhood hops that each vertex explores to generate histogram elements. UNICORN creates a histogram representation of an entire graph by examining rooted subtrees around every vertex in the graph. By considering such non-linear substructures, in addition to the attributes of each vertex/edge, UNICORN preserves structural equivalence of provenance graphs, which has been demonstrated to outperform linear approaches such as random walk [94]. M is a list of neighboring vertex and/or edge labels of a vertex v and $l_i(v)$ is the label (i.e., histogram element) of the vertex v in the i^{th} iteration (or the $(i-1)$-hop neighborhood, where the 0-hop neighborhood is the vertex itself). TS is a list of timestamps of the neighboring vertices. In line 8, the function $\texttt{In}(v)$ returns all the incoming edges of the vertex v and the function $\texttt{Source}(e)$ in line 9 returns the source vertex of the edge e. T records the timestamp of each

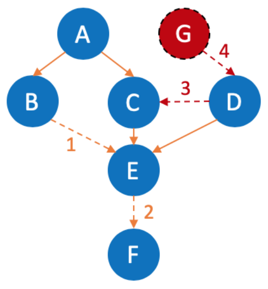


Fig. 2: A simple, abstracted provenance graph where dotted edges arrive after all solid edges have been processed and dotted vertices are newly-arrived. If the provenance graph observes partial ordering, edge 1 must arrive before edge 2 and only the local structures of vertex E and F need to be computed. However, in a provenance graph where partial ordering is *not* observed, we can encounter edge 3 and 4 and the newly-arrived vertex G (in dotted red) after the solid edges. In such a case, with R = 2, we need to update both vertex C and E (descendants of D) for edge 3 and vertex C, D, and E (descendants of G) for edge 4.

vertex v. In line 16, $\mathcal{N}(v) = \{w \mid (w, v) \in E\}$ is the set of vertices to which v is connected by an in-coming edge.

Our goal is to construct histograms where each element of the histogram corresponds to a unique vertex label, which captures the vertex's R-step in-coming neighborhood. Labels capture information about the edges in the neighborhood and the identities of the vertices in that neighborhood, including complex contextual information that reveals causal relationships among objects, subjects, and activities [94]. We bootstrap the labels as follows: each vertex begins with its own initial label. We then incorporate 1-hop neighbors by adding the labels of the incoming edges and the initial vertex labels of the sources of those edges. After this bootstrapping process, every vertex label now represents both vertex and edge labels so that as we expand to increasingly large neighborhoods, we need only add labels for the sources of all the incoming edges (since those labels already incorporate edge labels from their incoming edges). We sort all the labels by the timestamp of their corresponding edge, respecting the sequence of events in the system, and then hash the label list to facilitate fast lookup and bookkeeping. Some vertices/edges may have multiple labels, but the same hashing trick permits multi-label computation at negligible cost.

Streaming Variant and Complexity. In a streaming environment, we run Alg. 1 only on newly-arriving vertices and on vertices whose in-coming neighborhood is affected by new edges. In provenance graphs that use multiple vertices per provenance entity or activity to represent different versions or states of the corresponding object [90], we need to compute/update *only* the neighborhood of the destination vertex for each new edge, because all incoming edges to a vertex arrive before any outgoing ones [101]. UNICORN takes advantage of this partial ordering to minimize computation (Fig. 2), which is particularly important during exploration of large neighborhoods. Our implementation (§ VI) further reduces computation using batch processing. Consequently, the practical runtime complexity of the streaming variant of Alg. 1 is approximately equivalent to that of the original 1-dimensional Weisfeiler-Lehman algorithm [110]: using R-hop neighborhoods, the complexity is $O(R|E|)$.

Discounting Histogram Elements for Concept Drift. In APT scenarios, the long duration of a potential attack suggests that a good model must include the system's long-term behavior. However, system behavior often changes over time, which results in changes in the underlying statistical properties of the streaming provenance graph. This phenomenon is referred to as concept drift [120].

UNICORN accounts for such changes in system behavior through the use of exponential weight decay [70] on histogram element counts to gradually forget outdated data. It assigns weights that are inversely proportional to the age of the data [73]. For each element h in the histogram H, as a new data item x_t (i.e., a hashed label $l_i(v)$ from Alg. 1 line 23) streams in at time t, instead of counting H_h as:

$$H_h = \sum_t \mathbb{1}_{x_t = h}$$

UNICORN discounts the influence of existing data according to their age:

$$L_h = \sum_t w_t \mathbb{1}_{x_t = h}$$

where $\mathbb{1}_{cond}$ is an indicator function that returns 1 when $cond$ is true and 0 otherwise, and $w_t = e^{-\lambda \Delta t}$. UNICORN increments t monotonically with the number of edges added to the graph. We use L instead of H to denote weighted histograms (Table X).

Applicability in Intrusion Detection Scenarios. The gradually forgetting approach helps UNICORN focus on the current dynamics of system execution (i.e., the most recent part of the provenance graph), as well as any parts of the graph that are causally related to the current execution (based on the path length), while maintaining fading "memory" of the past, the rate of which is controlled by the weight decay factor λ. Notably, regardless of how temporally distant an event has occurred from the current state of system execution, if it is causally related to a current object/activity, that event and its surrounding neighborhood will contribute to the histogram without discount. Therefore, provenance graphs ensure that relevant contextual information remains in the analysis regardless of time. In § IV-D, we discuss how UNICORN models a system's evolutionary behavior.

C. Generating Graph Sketches

The graph histogram is a simple vector space graph statistic that describes system execution. However, unlike traditional histogram-based similarity analysis [13, 24, 133], UNICORN constantly updates the histogram as new edges arrive. Measuring the similarity between *streaming* histograms is challenging, because the number of histogram elements is not known *a priori* and changes continuously. Moreover, we should measure the similarity based on the underlying *distribution* of graph features, instead of absolute counts. However, most existing machine learning [31, 57] and data mining [34, 54] techniques applicable to modeling system behavior from graph histograms require fixed-length numerical vectors.

One naive solution is to enumerate all possible histogram elements beforehand, e.g., through combinatorial enumeration and manual feature engineering. However, given a potentially large number of vertex and edge labels, exacerbated by the multi-label nature of the graph and a possibly large number of neighborhood iterations, the resulting histogram will be sparse and thus problematic in terms of space and time complexity. Alternatively, manual feature engineering might reduce the size of the histogram, but it is time-consuming and inevitably requires arbitrary handling of unseen histogram elements.

We instead use locality sensitive hashing [59], also called similarity-preserving data sketching [124], which is commonly used in classifying [5] and filtering [12] high-cardinality data streams. UNICORN employs HistoSketch [132], an approach based on consistent weighted sampling [82], that efficiently maintains compact and fixed-size sketches for streaming histograms. HistoSketch is a constant time algorithm, so it is fast enough to support real-time streaming analysis on rapidly growing provenance graphs. It also unbiasedly approximates histograms based on their *normalized, generalized* min-max (Jaccard) similarity [128]. Jaccard similarity [27] has been successfully applied in a variety of machine-learning-based real-world problems, such as document analysis [111] and malware classification [107] and detection [32]. We review HistoSketch in Appendix § B. For proof of correctness of the algorithm, we refer interested readers to the original work by Yang et al. [132].

D. Learning Evolutionary Models

Given graph sketches and a similarity metric, clustering is a common data mining approach to identify outliers. However, conventional clustering approaches fail to capture a system's evolutionary behavior [8]. APT scenarios are sufficiently long term that failing to capture this behavior leads to too many false positives [83]. UNICORN leverages its streaming capability to create evolutionary models that capture normal changes in system behavior. Crucially, it builds these evolutionary models during training, not during deployment. Systems that dynamically evolve models during deployment risk poisoning their models during an APT's drawn out attack phase [83].

UNICORN creates a series of temporally-ordered sketches during training. It then clusters this sketch sequence from a single server using the well-known K-medoids algorithm [65], using the silhouette coefficient to determine the optimal value of K [108]. The clusters represent "meta-states" of system execution, e.g., startup, initialization, steady state behavior. UNICORN then uses both the temporal ordering of the sketches in all clusters and the statistics of each cluster (e.g., diameter, medoid) to produce a model of the system's evolution. Alg. 2 describes the construction of the evolutionary model. Each sketch $S(t)$ belongs to a single cluster indexed k. The evolution E is an ordered list of cluster indices, whose order is determined by the temporally-ordered sketches $S(t)$.

For each training instance, UNICORN creates a model that captures the changes of system execution states during its runtime. Intuitively, this is similar to an automaton [62, 109] that tracks the state of system execution. The final model consists of many sub-models from all the provenance graphs in the training data. With evolutionary modeling, UNICORN learns system behavior at many points in time; with the gradually forgetting scheme (§ IV-B), at *any* point in time, UNICORN is able to focus on the most relevant activities.

Algorithm 2: Generating an Evolution Trace

```
Input  : Sketches $S(t), t = 0, \cdots, T$ of a streaming provenance graph,
         ordered by time $t$
Output: Evolution List $E$
1 $E = \{\}$
2 for $t \leftarrow 0$ to $T$ do
3     $k$ = BelongsTo(S(t))
4                                          /* $k \in \{1, \cdots, K\}$ */
5     if Empty($E$) || Tail($E$) != $k$ then $E = E :: k$
```

E. Detecting Anomalies

During deployment, anomaly detection follows the same streaming paradigm described in previous sections. UNICORN periodically creates graph sketches as the histogram evolves from the streaming provenance graph. Given a graph sketch, UNICORN compares the sketch to all the sub-models learned during modeling, fitting it to a cluster in each sub-model. UNICORN assumes that monitoring starts from system boot and tracks system state transitions within each sub-model. To be considered valid in any sub-model, a sketch must either fit into the current state or (one of) the next state(s) (i.e., in the cases where the sketch captures state transition in system execution); otherwise, it is considered anomalous. Thus, we detect two forms of anomalous behavior: sketches that do not fit into existing clusters and invalid transitions between clusters.

V. IMPLEMENTATION

We use the vertex-centric graph processing framework, GraphChi [75] to implement UNICORN's graph processing algorithms in C++; we implement its data parsing and modeling components in Python.

GraphChi [75] is a disk-based system that efficiently computes on large graphs with billions of edges on a single computer. Using GraphChi, UNICORN achieves efficient analysis performance without needing to store the entire provenance graph in memory. UNICORN relies on two important features of GraphChi:

1) GraphChi uses a Parallel Sliding Windows (PSW) algorithm to split the graph into shards, with approximately the same number of edges in each shard; it computes on each shard in parallel. The algorithm allows fast vertex and edge updates to disk with only a small number of non-sequential disk accesses. This allows UNICORN to analyze the whole provenance graph independent of memory constraints.

2) UNICORN leverages GraphChi's efficient computation on streaming graphs. Per edge updates are efficient in their use of I/O, and selective scheduling reduces computation. UNICORN's guaranteed partial ordering (§ IV-B) minimizes the number of vertices it visits even when the neighborhood hop parameter, R, is large. Batching edge additions, rather than processing one edge at a time, makes processing even faster.

Appendix § A provides details on obtaining and testing our open-source implementation.

VI. Evaluation

We analyzed approximately 1.5 TB of system monitoring data containing approximately 2 billion OS-level provenance records from various tracing systems, demonstrating the applicability of our approach. Our evaluation addresses the following research questions:

Q1. Can UNICORN accurately detect anomalies in long-running systems under APT attacks? (§ VI-A, § VI-B, § VI-C)

Q2. How important are the design decisions we made that are tailored to the characteristics of APTs? (§ VI-C)

Q3. Does UNICORN's gradually forgetting scheme improve understanding of system behavior? (§ VI-C)

Q4. How effective are UNICORN's evolutionary models compared to existing clustering-based approaches that use static snapshots? (§ VI-A)

Q5. Is UNICORN fast enough to perform realtime monitoring and detection without falling behind? (§ VI-E)

Q6. What are UNICORN's memory and CPU utilization during system execution? (§ VI-F)

We compare UNICORN to StreamSpot, a state-of-the-art anomaly detector that has shown promising results for APT attacks. We show that multiple factors lead to UNICORN's higher detection accuracy; both its multi-hop graph exploration (Q2) and the evolutionary modeling scheme (Q4) are better suited for provenance-based APT detection.

We then explore the efficacy of UNICORN with three real-life APT attack datasets, all of which were captured during a red-team vs. blue-team adversarial engagement organized by U.S. DARPA and are publicly available [66]. We separate DARPA's datasets based on the underlying provenance capture systems (i.e., Cadets, ClearScope, and THEIA). We show that UNICORN can detect system anomalies during real APT campaigns (Q1).

Lastly, we create our own simulated APT supply-chain attack datasets (SC-1 and SC-2) using CamFlow in a controlled lab environment to demonstrate that UNICORN is performant in terms of processing speed (Q5) and CPU and memory efficiency (Q6). We show that it is capable of detecting simulated APT attacks using evolutionary modeling (Q4) with diverse normal and background activities. We further demonstrate that detection capability can be improved with contextualized graph exploration (Q2), and that the gradually forgetting scheme improves detection accuracy, because it helps better understand system behavior (Q3).

Properly evaluating and benchmarking security systems is difficult. We adhere to the guidelines proposed by Kouwe et al. [121] to the best of our ability. For example, the testbed that we designed to create the SC datasets ensures proper documentation of experiment specifications and easy reproduction of evaluation data and results. We also use 5-fold cross-validation to provide a more accurate evaluation [78].

A. UNICORN *vs. State-of-the-Art*

StreamSpot [83] is a clustering-based anomaly detection system that processes streaming heterogeneous graphs. It extracts local graph features from single node/edge labels through breadth-first traversal on each graph node and vectorizes them for classification. StreamSpot models only a single snapshot of every training graph and dynamically maintains its clusters during test time by updating the parameters of the clusters.

Experiment	Dataset	# of Graphs	Avg. |V|	Avg. |E|	Preprocessed Data Size (GiB)
StreamSpot	YouTube	100	8,292	113,229	0.3
	Gmail	100	6,827	37,382	0.1
	Download	100	8,831	310,814	1
	VGame	100	8,637	112,958	0.4
	CNN	100	8,990	294,903	0.9
	Attack	100	8,891	28,423	0.1

TABLE I: Characteristics of the StreamSpot dataset. The dataset is publicly available only in a preprocessed format.

Experimental Dataset. The StreamSpot dataset contains information flow graphs derived from six scenarios, five of which are benign [84]. Each scenario runs 100 times, producing 100 graphs for each. Using the Linux `SystemTap` logging system [60], the benign scenarios record system calls from normal browsing activities, such as watching YouTube videos and checking Gmail, while the attack scenarios involve a drive-by download from a malicious URL that exploits a Flash vulnerability and gains root access to the visiting host. The original scripts and the precise attack are unknown to us, however, the datasets are publicly available [84], and we confirmed with the authors that the information flow graphs were constructed from all system calls on a machine from the start of a task until its termination. Table I summarizes the dataset.

We use this dataset to fairly compare UNICORN with StreamSpot. These nicely isolated scenarios may not represent today's typical workloads. However, they provide insight into how the different systems perform relative to each other. Furthermore, we might also interpret them as a proxy for the design pattern of today's microservice architecture [93]. As we will see next, UNICORN performs particularly well in such scenarios, but it is also capable of accurately detecting anomalies in more diverse, heterogeneous computing environments (§ VI-B and § VI-C). We discuss this point further in § VII.

Experiment	Precision	Recall	Accuracy	F-Score
StreamSpot (baseline)	0.74	N/A	0.66	N/A
$R = 1$	0.51	1.0	0.60	0.68
$R = 3$	0.98	0.93	0.96	0.94

TABLE II: Comparison to StreamSpot on the StreamSpot dataset. We estimate StreamSpot's average accuracy and precision from the figure included in the paper [83], which does not report exact values. They did not report recall or F-score.

Experimental Results. We compare UNICORN to StreamSpot, using StreamSpot's own dataset. We configure UNICORN to use a sketch size $|S| = 2000$ and examine with different neighborhood sizes, $R = 1$ (equivalent to StreamSpot) and $R = 3$. As shown in Table II, UNICORN's ability to trivially consider larger neighborhoods ($R = 3$) produces significant

Experiment	# of Test Graphs	# of FPs (R = 1)	# of FPs (R = 3)
YouTube	25	14	0
Gmail	25	19	0
Download	25	25	2
VGame	25	20	0
CNN	25	18	0

TABLE III: Decomposition of UNICORN's false positive results of the StreamSpot dataset.

precision/accuracy improvement. The detailed precision results in Table III further show that analyzing larger neighborhoods greatly reduces the false positive rate. This supports our hypothesis that contextual analysis is crucial. UNICORN raises false positive alarms only on the Download dataset, the most diverse dataset of StreamSpot's benign datasets (also manifested in its large average number of edges in Table I). We discuss the importance of graph exploration in more depth in § VI-C.

The following sections evaluate UNICORN on various real-life and simulated APT attacks. Unfortunately, we were unable to evaluate StreamSpot on these datasets, because it could not handle the large number of edge types present nor could it scale to the size of the graphs.

B. DARPA TC Datasets

Next, we demonstrate that UNICORN can effectively detect APTs utilizing data from a variety of different provenance capture systems.

DARPA's Transparent Computing program focuses on developing technologies and prototype systems to provide both forensic and detection of APTs.

Experiment	Dataset	# of Graphs	Avg. $\|V\|$	Avg. $\|E\|$	Raw Data Size (GiB)
DARPA CADETS	Benign	66	59,983	4,811,836	271
	Attack	8	386,548	5,160,963	38
DARPA ClearScope	Benign	43	2,309	4,199,309	441
	Attack	51	11,769	4,273,003	432
DARPA THEIA	Benign	2	19,461	1,913,202	4
	Attack	25	275,822	4,073,621	85

TABLE IV: Characteristics of graph datasets used in the DARPA experiments.

Experimental Datasets. The DARPA datasets (Table IV) were collected from a network of hosts during the 2-week long third adversarial engagement of the DARPA Transparent Computing program. The engagement involved various teams responsible for collecting audit data from different platforms (e.g., Linux, Windows, BSD), launching attacks during the engagement period, and analyzing the data to detect attacks and perform forensic analysis. The red team that carried out attacks also generated benign background activity (e.g., web browsing), thus allowing us to model normal system behavior.

The CADETS dataset was captured via the **C**ausal, **A**daptive, **D**istributed, and **E**fficient **T**racing **S**ystem (CADETS) on FreeBSD [1]. ClearScope instruments the entire Android mobile software stack to capture provenance of the operations of mobile devices [3]. THEIA is a system for tagging and tracking multi-level host events [35]; it instruments Ubuntu Linux machines during the engagement.

The experiment simulated an enterprise setup [87] including security-critical services such as a web server, an SSH server, an Email server, and an SMB server (for shared file access). The red team carried out various nation-state and common threats through the use of, e.g., a Firefox backdoor, a Nginx backdoor, and phishing emails. Detailed descriptions of the attacks are available online [66].

Experimental Results. We partition each benign dataset into a training set (90% of the graphs) and a test dataset (10% of the graphs). We use the same sketch size ($|S| = 2000$) and neighborhood hop ($R = 3$) as in the StreamSpot experiment.

Experiment	Precision	Recall	Accuracy	F-Score
DARPA CADETS	0.98	1.0	0.99	0.99
DARPA ClearScope	0.98	1.0	0.98	0.99
DARPA THEIA	1.0	1.0	1.0	1.0

TABLE V: Experimental results of the DARPA datasets.

Table V shows that UNICORN's analytics framework generalizes to different provenance capture systems and various provenance graph structures. UNICORN's high performance suggests that it can accurately detect anomalies in long-running systems of various platforms. During the engagement, the red team launched APT attacks using different attack vectors and the attacks account for less than 0.001% of the audit data volume [87]. UNICORN's anomaly-based detection mechanism identifies those attacks without prior attack knowledge, even though they are embedded in an abundance of benign activity.

We note that some existing systems (Holmes [87] and Poirot [86]) also use the DARPA dataset for evaluation. Comparison between UNICORN and these systems is difficult, because they use a rule-based approach that requires *a priori* expert knowledge to construct a model. UNICORN is fundamentally different, using an unsupervised learning model, requiring no expert input. However, UNICORN's performance is comparable based on the number of detected attacks: UNICORN detects all attacks on FreeBSD and Linux as do Holmes and Poirot. We discuss the differences between these systems and UNICORN in greater detail in § VIII.

C. Supply Chain Attack Scenarios

We designed two APT attack scenarios to run in a controlled lab environment. These experiments evaluate the importance of graph analysis and evolutionary modeling (this section and § VI-D) to show that UNICORN is able to perform efficient, realtime monitoring (§ VI-E). Additionally, we use these experiments to understand how UNICORN performs when faced with attacks that behave similarly to normal system workloads. We carefully design benign and attack scenarios to achieve this goal. Previous experiments, conducted by others, do not guarantee similarity between benign and attack scenarios.

Experiment	Dataset	# of Graphs	Avg. $\|V\|$	Avg. $\|E\|$	Raw Data Size (GiB)
SC-1	Benign	125	265,424	975,226	64
	Attack	25	257,156	957,968	12
SC-2	Benign	125	238,338	911,153	59
	Attack	25	243,658	949,887	12

TABLE VI: Characteristics of the datasets used in the supply-chain APT attack experiments.

Experimental Datasets. We simulated two APT supply-chain attacks (SC-1 and SC-2) on a Continuous Integration (CI) platform and used CamFlow (v0.5.0) to capture whole-system provenance, including background activity, during both benign and attack scenarios. For each scenario, the experiment ran for three days. To facilitate reproduction, we leverage virtualization technology as our test harness and provide automated scripts for push-button replication of the experiments that generated the data (see Appendix § A for details).

To simulate APT attacks, we follow the typical cyber kill chain model that consists of roughly 7 nonexclusive phases, i.e., *reconnaissance* (identify a target and explore its vulnerabilities), *weaponize* (design a backdoor and a penetration plan),

	Batch Size	Sketch Size	Hop Count	Decay Factor	Sketch Interval
Baseline	6,000	2,000	3	0.02	3,000

TABLE VII: UNICORN configurations for supply-chain APT attack scenarios.

delivery (deliver the weapon), *exploitation* (victim triggers the vulnerability), *installation* (install the backdoor or malware), *command and control (C&C)* (give remote instructions to the victim), and *actions on objectives* [131].

In the SC-1 experiment, the attacker identified an enterprise CI server that routinely `wgets` Debian packages from various repositories. She discovered that the server runs GNU `wget` version 1.17, which is vulnerable to arbitrary remote file upload when the victim requests a malicious URL to a compromised server (CVE-2016-4971) [47] (*reconnaissance*). The attacker embedded a common remote access trojan (RAT) into a Debian package and compromised one of the repositories so that any request to download the legitimate package is redirected to the attacker's FTP server that hosts the RAT-embedded package (*delivery*). As the CI server downloaded (*exploitation*) and installed (*installation*) the package, it also unknowingly installed the trojan software. The RAT established a C&C channel with the attacker, creating a reverse TCP shell session on the CI server (*C&C*). The attacker then modified the CI server configuration (*actions on objectives*) to gain control of the CI deployment output. The SC-2 experiment had a similar setup but the attacker exploited a different vulnerability from GNU Bash version 4.3, which allows remote attackers to execute arbitrary code via crafted trailing strings after function definitions in Bash scripts (CVE-2014-6271). Both scenarios represent the *supply-chain compromise* as the initial access mechanism to subvert a company's software distribution channel to spread malware [48, 67].

In both experiments, we model normal behavior of the victim system (i.e., the CI server). Table VI summaries the datasets.

Experiment	Precision	Recall	Accuracy	F-Score
SC-1	0.85	0.96	0.90	0.90
SC-2	0.75	0.80	0.77	0.78

TABLE VIII: Experimental results of the supply-chain APT attack scenarios.

Experimental Results. We split 125 benign graphs randomly into 5 groups to enable 5-fold cross validation. After we use UNICORN to model normal behavior on the training set, which consists of 100 benign graphs (i.e., 4 groups), we evaluate it on the remaining 25 benign graphs (i.e., the 5^{th} group) for false positive validation. We also evaluate the model on the 25 attack graphs for false negatives (Table VI). We repeat this procedure for each group and report the mean evaluation results. Table VII summarizes the configuration for the experiments and Table VIII shows the experimental results.

We see in Table VIII that UNICORN is able to detect attacks with high accuracy with only a small number of false alarms (as reflected in precision and recall). We observe that UNICORN creates many clusters in the sub-models during modeling, and the majority of true alarms originate from the first several clusters as UNICORN tracks system state transition. This suggests that UNICORN's evolutionary model captures system behavior changes and that it is able to detect attacks in their early stages, i.e., the initial supply-chain access point. This has important implications. First, traditional clustering approaches that use static snapshots to build the initial model generate a large number of false positives (§ VI-A). These can easily overwhelm system administrators and cause "threat fatigue", leading to alert dismissal. In supply-chain attacks, the attackers can take advantage of this initial stage to break into an enterprise network. UNICORN reduces false positives as its evolutionary models precisely but flexibly define normal system behavior. We further note that dynamically adapting the model during runtime is also suboptimal in APT scenarios, because once the attackers break into the network from the initial supply chain, they can guide the model to slowly and gradually penetrate the network without the model raising an alarm. Second, while many APTs abandon stealth in later attack stages, making detection easier, UNICORN raises alarms in earlier stages, thus preventing damage that may have already occurred when APTs unmask their behavior. For example, we observe that 50% of the attacks in SC-1 were detected when malicious packages were just delivered to the victim machine, and all of them were flagged right after installation.

It is more difficult to detect attacks in the SC scenarios than in the DARPA ones. The attackers in the DARPA datasets spend time finding vulnerabilities in the system, and that behavior appears in the traces. In contrast, in the SC scenarios, the attacker has *a priori* knowledge of the target system (i.e., we act as both the attacker and the victim), so we can launch an attack without any prior unusual behavior. This partially explains UNICORN's lower performance on the SC datasets.

In § VI-D, we conduct additional experiments on SC-1 to demonstrate the importance of graph exploration in detecting anomalies in provenance graphs.

D. Influence of Graph Analysis on Detection Performance

We now analyze the importance of UNICORN's key parameters using the SC-1 dataset. We use the same setup from § VI-C as our baseline configurations (Table VII). We then vary parameters independently to examine the impact of each. Fig. 3 shows the experimental results.

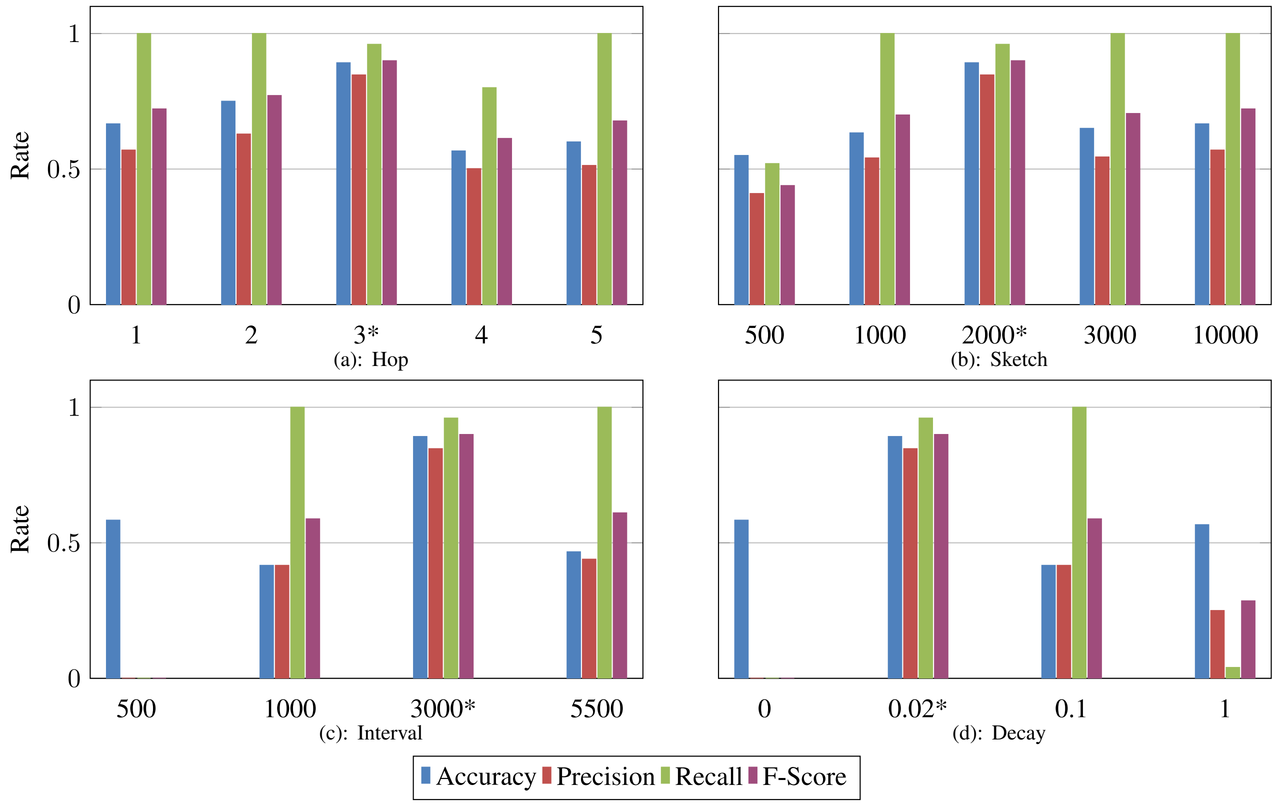
Batch Size (BS). This indicates the number of edges submitted to GraphChi at once; it does not affect detection performance.

Hop Count (HC). This defines the size of the neighborhood used to characterize each vertex; it is a measure of the expressiveness of the features in our sketches. Larger hop counts capture more contextual information, some of which might be irrelevant, which can mask potential attacks [83, 87], as shown in Fig. 3(a).

Sketch Size (SS). This is the size of our fixed-size histogram representation. Larger SS allows UNICORN to include more information about the evolving graph, thus reducing the error of approximating normalized min-max similarity (§ IV-C). However, a large SS ultimately leads to the curse of dimensionality [38] in clustering (§ IV-D). Fig. 3(b) confirms that, in general, detection precision, recall, and accuracy improve as we increase SS up to a point, after which it degrades.

Interval of Sketch Generation (SG). This is the number of edges added to the graph between the construction of new sketches. Smaller SG makes adjacent sketches look similar to each other, which can produce higher false negative rates

Fig. 3: Detection performance (precision, recall, accuracy, and F-score) with varying hop counts (Fig. 3(a)), sketch sizes (Fig. 3(b)), intervals of sketch generation (Fig. 3(c)), and decay factor (Fig. 3(d)). Baseline values (*) are used by the controlled parameters (that remain constant) in each figure.



and lower recall and accuracy. Meanwhile, given fixed-size sketches, which are approximations, a larger SG leads to coarser-grained changes, which also makes graphs look too similar to each other. In Fig. 3(c), we observe that SG = 500 edges per new sketch cannot detect any attack graphs, resulting in 0 recall and undefined precision and F-score. We obtain optimal results when setting the interval to be around $3,000$ in the SC-1 experiment.

Weighted Decay Factor (DF). This determines the rate at which we forget the past. We observe that both never-forget ($\lambda = 0.0$) and always-forget ($\lambda = 1.0$) yield unsatisfactory results, while a slow decay rate (around 0.02) achieves a good balance between factoring past and current graph components into the analysis.

E. Processing Speed

In the previous section, we show how UNICORN's parameters influence detection performance; this section and the one that follows examine UNICORN's runtime overhead. These sections consider only the CamFlow implementation, which has been shown to produce negligible overhead [100], allowing us to isolate UNICORN's performance characteristics. We use Amazon EC2 i3.2xlarge Linux machines with 8 vCPUs and 61GiB of memory.

Runtime performance is important in APT scenarios where an IDS is constantly monitoring the system in real time. We analyze the SC-1 experiment using the same baseline settings and parameters as in the previous section. (We also evaluate batch size here, which has no impact on accuracy.) Together, these two sections illustrate the tradeoff between accuracy and runtime performance.

Fig. 4 shows the total number of edges processed over time as a metric to quantify UNICORN's processing speed. The CamFlow lines (in blue) represent the total number of edges generated by the capture system; the closer other lines are to this line, the better the runtime performance, meaning that UNICORN "keeps up" with the capture system.

Batch Size (BS). Fig. 4(a) shows that runtime performance improves as we increase BS. We use BS of 6,000 as it approximates CamFlow. There is marginal improvement as we increase BS above 6,000.

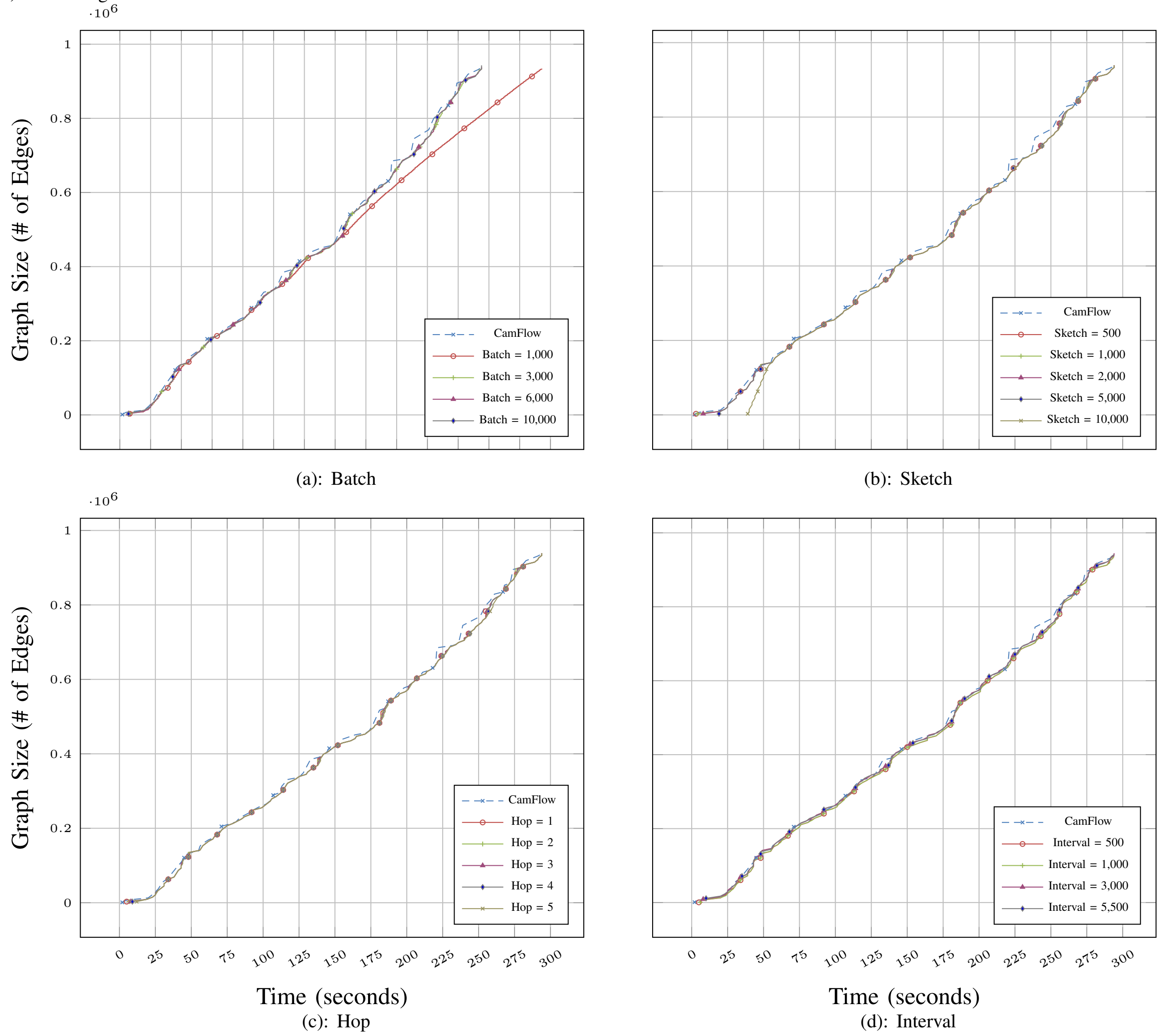
Sketch Size (SS). As shown in Fig. 4(b), SS has minimal impact on runtime performance, except during the beginning of the experiment when UNICORN initializes the sketch, which requires more computation with larger sketch sizes. Afterwards, runtime performance is similar among different SS, thanks to UNICORN's fast, incremental sketch update.

Hop Count (HC). Fig. 4(c) shows that HC has minimal impact on provenance graphs because of the graph structure and its adaption of fast WL algorithm.

Interval of Sketch Generation (SG) & Weighted Decay Factor (DF). Neither SG nor DF affect runtime performance (we omit DF from the figure).

Overall, we show that UNICORN runtime is relatively insensitive to these parameters. This means that UNICORN can perform realtime intrusion detection with parameters optimized for detection accuracy.

Fig. 4: Total number of processed edges over time (in seconds) in the SC-1 experimental workload with varying batch sizes (Fig. 4(a)), sketch sizes (Fig. 4(b)), hop counts (Fig. 4(c)), and intervals of sketch generation (Fig. 4(d)). Dashed blue line represents the speed of graph edges streamed into UNICORN for analysis. Triangle maroon baseline has the same configurations as those used in our experiments and indicates the values of the controlled parameters (that remain constant) in each figure.



(a): Batch

(b): Sketch

(c): Hop

(d): Interval

Configuration Parameter	Parameter Value	Max Memory Usage (MB)
Hop Count	R = 1	562
	R = 2	624
	R = 3	687
	R = 4	749
	R = 5	812
Sketch Size	$\|S\|$ = 500	312
	$\|S\|$ = 1,000	437
	$\|S\|$ = 2,000	687
	$\|S\|$ = 5,000	1,374
	$\|S\|$ = 10,000	2,498

TABLE IX: Memory overheads with varying hop counts and sketch sizes. The highlighted configurations gave the best detection performance.

F. *CPU & Memory Utilization*

We evaluate UNICORN's CPU utilization and memory overheads for a system under relatively heavy workload, i.e., CI performing kernel compilation. We show that UNICORN exhibits low CPU utilization and memory overheads.

Fig. 5(a) shows the average CPU utilization due to UNICORN over a long-running experiment using the baseline configuration. The average CPU utilization stabilizes around 12.3% on a single CPU. Fig. 5(b) shows the per-vCPU and the average CPU utilization of the same experiment (but with a shorter timeline to avoid cluttering). Note that the parameters discussed in the previous sections do not significantly impact average CPU utilization.

PassMark Software's Enterprise Endpoint Security performance benchmark [14] shows an average CPU utilization of 26%-90% for commercial products such as Symantec Endpoint Protection, Kaspersky Endpoint Security, and McAfee Endpoint Security. However, direct comparison is difficult. Research IDS we surveyed (§ VIII) do not report metrics that allow meaningful comparison. We leave the design of meaningful performance benchmarks for IDS to future work.

Table IX shows memory overheads for the same workload under two different parameters. Other parameters in the basic configurations do not significantly influence memory consumption.

The graph histogram and the random variables sampled for sketch generation represent the majority of UNICORN's memory usage. The size of the histogram is proportional to the number of unique labels, which in turn is determined by the size of the neighborhood that each vertex explores. Table IX shows that as we increase the number of neighborhood hops, UNICORN requires more memory for the histogram. In theory, the total number of unique labels is bounded by the number of possible combinations of node/edge types within $|R|$ hops. In practice, however, many parts of a provenance graph exhibit similar structures and therefore the value is significantly lower than the theoretical upper bound. For example, the maximum memory usage of the experiment corresponding to Fig. 5(a) remains around 680MB, which is acceptable for modern systems. As we increase the sketch size, UNICORN consumes more memory to store additional random variables sampled for sketch generation and update. Although with $|S| = 10,000$, memory consumption increases up to 2.5GiB (Table IX), the previous sections suggest that such large values are never a good option.

VII. Discussion & Limitations

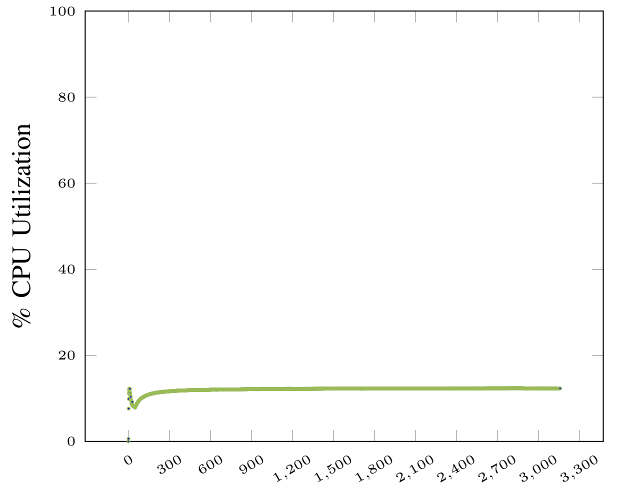
Anomaly-based Detection. UNICORN shares assumptions, architecture, and therefore limitations with other anomaly-based intrusion detection systems.

First, we assume there is a modeling period where system administrators can run their systems safely to capture system behavior (§ III). Second, we assume that there exists an exhaustive, finite number of system behavior patterns and most, if not all, patterns are observed during modeling [109]. UNICORN will raise a false alarm if it observes a new behavior that is, in fact, normal; such alarms will require human-in-the-loop intervention.

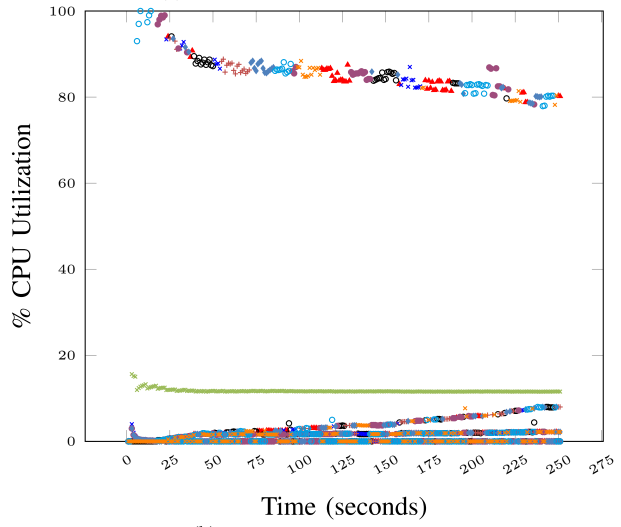
Adversaries may try to steer malicious behavior towards learned models to evade detection, similar to the well-known mimicry attacks [98]. However, conducting mimicry attacks on provenance graphs and/or UNICORN's graph sketches is more challenging than on sequences of system calls, because provenance graphs contain complex structural information that is difficult to imitate without affecting the attack. Additionally, UNICORN's consistent weighted sampling approach randomizes sketch generation, making it hard to guarantee that the low-dimensional projections of mimicry provenance graphs will be close to learned normal clusters. At the same time, this complexity will make it difficult to use models to identify the cause of an alarm. Fortunately, there exist tools that facilitate attack causality analysis on provenance graphs (§ VIII).

UNICORN, like other anomaly-based systems, requires sufficient benign behavior traces to learn behavior models. Our current modeling design and implementation assumes that UNICORN monitors a system from the same starting point as its model, tracking state transitions as the system executes. It can easily perform such continuous monitoring given its high performance and scalability (§ VI). However, UNICORN may raise false alarms if the current state does not match the modeled state as a result of, e.g., the system restoring to a saved state due to failure. One approach to addressing this issue is to integrate UNICORN more closely with the system so that it can save its model state at the same time the system creates snapshots; when the system restores a snapshot, UNICORN would restore the corresponding model state.

Fig. 5: Evaluation of UNICORN's CPU utilization.



(a): Average CPU utilization with the baseline configurations.



(b): Per virtual CPU and average CPU utilization.

× Average CPU ▲ vCPU0 ○ vCPU1 × vCPU2 + vCPU3 ● vCPU4 ♦ vCPU5 ○ vCPU6 × vCPU7

False Alarms. As briefly mentioned above, when normal system behavior changes, UNICORN might raise false positive alarms, since it does not dynamically adjust its model (to avoid attacker poisoning). The false alert problem is not unique to UNICORN. In fact, it is a major concern even for signature-based approaches used in practice, although in theory, these approaches are designed to generate few such alarms due to rigid attack matching [2]. UNICORN partially mitigates this issue using concept drift (§ IV-B), modeling system evolution. As we demonstrated in § VI, UNICORN improves precision by 24% compared to the state-of-the-art and achieves near perfect results on the DARPA datasets. However, UNICORN must also ensure stability to avoid adversarial manipulation, which inevitably increases the potential for false alerts. Therefore, system administrators might need to periodically retrain

UNICORN's model to stay up-to-date. Fortunately, updating UNICORN models is quick; the important caveat is to ensure that new training data is known to be devoid of malicious activity.

Graph Analysis. UNICORN enables efficient and powerful graph analysis, but similar to many IDS [8, 23, 83, 87], it also requires parameters that must be tuned to each system to improve detection performance (§ VI-D). Finding ideal parameters for a specific task is a well-known research topic in machine learning [119]. We used OpenTuner [11] to enable program autotuning. Although our tests are by no means exhaustive, it is encouraging that we were able to use the same settings on almost all of our evaluation datasets, even though they came from disparate sources and modeled rather different attacks. Due to time and space constraints, we leave the discussion of automatically tuning UNICORN to future work.

Heterogenous Host Activity. In testing, we observed that UNICORN performed extremely well in domains featuring homogenous normal activities. For example, we achieve near-perfect detection rates for the StreamSpot dataset (Table I). This makes UNICORN a promising security tool in datacenters and other production environments that perform well-defined tasks, which are frequently the target of attacks [22, 74, 106]. Hosts that exhibit more diverse behaviors, such as workstations [18], pose a greater challenge for IDS in general. Modern provenance capture systems (e.g., CamFlow) help mitigate this issue as they can separate provenance data based on, e.g., namespaces and control groups [100]. However, we acknowledge that endpoint security for workstations presents extra challenges that UNICORN does not attempt to address in this work, as it was originally designed to protect more stable environments.

Larger Cross-evaluation. We emphasize that comparing UNICORN with other existing IDS (most of which are syscall-based) is difficult for several reasons: A) many IDS are not open-source; B) existing public IDS datasets are either outdated [4, 85] or require a translation [28, 50, 51] from, e.g., syscall traces to data provenance, which is challenging and sometimes impossible (due to lack of information); C) systems that create their own private datasets only superficially describe their experimental procedures, making it difficult to fairly reproduce the experiments for provenance data. We believe that such a meta-study is a worthwhile endeavor that we plan to pursue in future work.

VIII. RELATED WORK

This work lies at the intersection of dynamic host-based intrusion detection, graph-based anomaly detection, and provenance-based security analysis. Therefore, we place UNICORN in the context of prior work in these areas.

Dynamic Host-based Intrusion Detection. Dynamic host-based intrusion detection (HID) was pioneered by Forrest et al.'s anomaly detection system [37] that used fixed-length sequences of syscalls to define normal behavior for UNIX processes. Debar et al. [29] and Wespi et al. [127] later generalized the approach to incorporate variable-length patterns. As attacks became increasingly sophisticated [117, 123], systems that modeled only syscall sequences suffered from low detection accuracy. Next generation systems added state to provide contextual information to the syscalls. Sekar et al. [109] designed a finite-state automaton (FSA) that modeled each state as the invocation of a system call from a particular call site. VtPath [36] extended this idea to avoid the impossible path problem, in which there may exist sequences of state transitions in the FSA that cannot happen in practice. VtPath performed more extensive call stack analysis to identify such impossible paths as anomalies. Jafarian et al. [62] addressed the impossible path problem by using a deterministic push-down automata (DPDA). Maggi et al. [81] extended these approaches by combining models of system call sequences with models for the parameters to those system calls. As automaton-based models approach their theoretical accuracy limit, which is equivalent to a linear bounded automaton (LBA) or a context-sensitive language model, their detection capacity increases; however, the non-polynomial complexity of such theoretical models makes it impossible to realize in practice [113]. Even a constrained DPDA model exhibits a polynomial time complexity [62]. Shu et al. [113] presented a formal framework that surveyed host-based anomaly detection, discussing in detail various dynamic and static approaches orthogonal to our work. Liu et al. [78] summarized the state-of-the-art host-based IDS [26, 68, 91] and discussed future research trends, indicating data as one of the decisive factors in IDS research.

UNICORN takes a completely different approach, because traditional system call approaches are not well-suited for APT attacks (§ II-A) [61, 125]. Our graph representation and analysis avoids costly control-flow construction and state transition automata, while accurately describing and modeling complex relationships among data objects in the system for contextualized anomaly detection. To the best of our knowledge, although some systems produce provenance-like graphs from audit logs [83], UNICORN is the first system to detect intrusions via runtime analysis of native whole-system provenance.

Graph-based Anomaly Detection. Akoglu et al. determine graph or subgraph similarity for anomaly detection by categorizing graphs based on their properties (i.e., *static* vs. *dynamic*, *plain* vs. *attributed*) [8].

Ding et al. [30] identified malicious network sources in network flow traffic graphs based on cut-vertices, using similarity metrics such as betweenness to detect cross-community communication behavior. Liu et al. [77] constructed a software behavior graph to describe program execution and used a support vector machine (SVM) to classify non-crashing bugs (e.g., logic errors that do not crash the program) based on closed subgraphs and frequent subgraphs. These systems and many other graph mining algorithms [39, 103] and graph similarity measures (e.g., graph kernels [122]) are designed only for static graphs and are difficult to adapt to a streaming setting.

Papadimitriou et al. [97] proposed five similarity schemes for dynamic web graphs, and NetSimile [17] used moments of distribution to aggregate egonet-based features (e.g., number of neighbors) to cluster social networks. Aggarwal et al. [6] used a structural connectivity model to define outliers and design a reservoir sampling approach that robustly maintains structural summaries of homogeneous graph streams. However, these and other streaming-oriented approaches [49, 88, 115] are either domain specific (e.g., bibliographic networks have a different structure than provenance graphs) or applicable primarily to homogeneous graphs.

In the realm of malware classification and intrusion detection, Classy [72] clusters streams of call graphs to facilitate malware analysis based on graph edit distance (GED) [41] of pairs of graphs using a modified version of simulated annealing. Although its runtime complexity is suitable for graph streams, the empirical evaluation was limited to graphs with no more than 3,000 vertices; real system execution yields graphs orders of magnitude larger [101].

StreamSpot [83] analyzes streaming information flow graphs to detect anomalous activity. However, StreamSpot's graph features are locally constrained while UNICORN's embody execution context. We show in § VI that contextualized graph analysis has great impact on detection performance. Furthermore, StreamSpot models only a single snapshot of every training graph, dynamically maintaining its clusters during test time. However, it results in a significant number of false alarms, which creates an opportune time window for the attacker. We also consider such an approach inappropriate in APT scenarios, where persistent attackers can manipulate the model to gradually and slowly change system behavior to avoid detection. UNICORN takes full advantage of its ability to continuously summarize the evolving graph, modeling the corresponding evolution of the system execution it monitors. FRAPpuccino [53] is another attempt at graph-based intrusion detection. It uses a windowing approach to allow for efficient graph analysis. Naturally, segmenting provenance graphs in this way produces a more limited view of system execution, unsuitable for long-term detection that spans windows.

Provenance-based Security Analysis A variety of security-related applications leverage provenance, mostly notably for forensic analysis and attack attribution [8]. BackTracker [69] analyzes intrusions using a provenance graph to identify the entry point of the intrusion, while PriorTracker [79] optimizes this process and enables a forward tracking capability for timely attack causality analysis. HERCULE [102] analyzes intrusions by discovering attack communities embedded within provenance graphs. Winnower [56] expedites system intrusion investigation through grammatical inference over provenance graphs, and simultaneously reduces storage and network overhead without compromising the quality of provenance data. NoDoze [55] performs attack triage within provenance graphs to identify anomalous paths. Bates et al. [16] were the first to use provenance for data loss prevention, and Park et al. [99] formalized the notion of provenance-based access control (PBAC). Ma et al. [80] designed a lightweight provenance tracing system ProTracer to mitigate the dependence explosion problem and reduce space and runtime overhead, facilitating practical provenance-based attack investigation. Pasquier et al. [101] introduced a generic framework, called CamQuery, that enables inline, realtime provenance analysis, demonstrating great potential for future provenance-based security applications.

Recently, as APT attacks become increasingly prominent, a number of systems leverage data provenance for APT attack analysis. Holmes [87] and Sleuth [58] focus primarily on attack reconstruction using information flow provided by data provenance. Their approach is similar to an architecture proposed by Tariq et al. [118] that correlates anomalous activity in grid applications using data provenance. The anomaly detection module itself uses a simple, pre-defined model that relies on expert knowledge of the existing APT kill-chain to match an *a priori* specification of possible exploits to localized components in the graph. Poirot [86] produces another form of attack reconstruction. It correlates a collection of compromise indicators (found by other systems) to identify APTs. Relying on expert knowledge from existing cyber threat intelligence reports and compromise descriptions to construct attack query graphs, Poirot performs offline graph pattern matching on provenance graphs to uncover potential APTs. For example, it uses the red team's attack descriptions to manually craft query graphs to correlate anomalies in the DARPA datasets used in § VI-B. This is a critical limitation given that composing a sufficiently detailed description of a new class of APT takes significant forensic effort [15].

UNICORN differs from these rule-based systems in that it is an anomaly-based system that requires no prior expert knowledge of APT attack patterns and behaviors. Although rule-based approaches are closely aligned with commercial practices today (i.e., they are essentially provenance-based versions of the Endpoint Detection and Response (EDR) tools offered by enterprise security vendors), prior research shows that rule-based EDR systems are the chief contributor to the "threat fatigue problem" [2]. Recent work on provenance analysis (e.g., NoDoze [55]) demonstrated that historical context is crucial for mitigating this problem; UNICORN instead investigates how to incorporate context as a first class citizen in HID systems, rather than as a secondary triage tool.

Gao et al. [40] leveraged complex event processing platforms and designed a domain-specific query language, SAQL to analyze large-scale, streaming provenance data. The system combines various anomaly models (e.g., rule-based anomalies) and aggregates data streams across multiple hosts, but it ultimately requires expert domain knowledge to identify elements/patterns to match against queries. We also note that our provenance graph analyses are able to incorporate implicitly (without domain knowledge) most of their anomaly models (e.g., invariant-based, time-series, and outlier-based). Barre et al. [15] mine data provenance to detect anomalies. The goal of their work is mainly to identify important process features that are likely to be relevant to APT attacks (e.g., a process' lifetime and path information). Using a random forest model with hand-picked process features, their system delivers a detection rate of only around 50%. Such low performance suggests that simple feature engineering on provenance graphs without considering graph topology is insufficient in detecting stealthy APT attacks. Berrada et al. [19] proposes score aggregation techniques to combine anomaly scores from different anomaly detectors to improve detection performance. Although their work targets provenance graphs for APT detection, it is orthogonal to UNICORN (or any other detectors) in that it functions only as an aggregator for existing anomaly detection systems.

IX. CONCLUSION

We present UNICORN, a realtime anomaly detection system that leverages whole-system data provenance to detect advanced persistent threats that are deemed difficult for traditional detection systems. UNICORN models system behavior via structured provenance graphs that expose causality relationships between system objects, taking into account the entirety of the graph by efficiently summarizing it as it streams into its analytic pipeline. Our evaluation shows that the resulting evolutionary models can successfully detect various APT attacks captured from different audit systems, including real-life APT campaigns, with high accuracy and low false alarm rates.

ACKNOWLEDGEMENT

We thank Dr. Robert N. W. Watson and his team at the University of Cambridge on the DARPA Transparent Computing Program. We also thank the anonymous reviewers and our shepherd Prof. Brendan Saltaformaggio who helped improve the paper. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC). Cette recherche a été financée par le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG). This research was supported by the US National Science Foundation under grants NSF 14-50277, 16-57534, and 17-50024. This research was also supported with promotional credits from the AWS Cloud Credits for Research program. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

REFERENCES

[1] "Causal, Adaptive, Distributed, and Efficient Tracing System (CADETS)," Accessed 21st January 2020, https://www.cl.cam.ac.uk/research/security/cadets/.

[2] "How Many Alerts is Too Many to Handle?" Accessed 21st January 2020, https://www2.fireeye.com/StopTheNoise-IDC-Numbers-Game-Special-Report.html.

[3] "Transparent Computing," Accessed 21st January 2020, https://people.csail.mit.edu/rinard/research/ComputerSecurity/.

[4] "University of New Mexico System Call Dataset," accessed 21st January 2020, https://www.cs.unm.edu/~immsec/systemcalls.html.

[5] C. C. Aggarwal and P. S. Yu, "On classification of high-cardinality data streams," in *International Conference on Data Mining*. SIAM, 2010, pp. 802–813.

[6] C. C. Aggarwal, Y. Zhao, and S. Y. Philip, "Outlier detection in graph streams," in *International Conference on Data Engineering (ICDE)*. IEEE, 2011.

[7] K. J. Ahn, S. Guha, and A. McGregor, "Graph sketches: sparsification, spanners, and subgraphs," in *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*. ACM, 2012, pp. 5–14.

[8] L. Akoglu, H. Tong, and D. Koutra, "Graph based anomaly detection and description: a survey," *Data mining and knowledge discovery*, vol. 29, no. 3, pp. 626–688, 2015.

[9] A. Alshamrani, S. Myneni, A. Chowdhary, and D. Huang, "A survey on advanced persistent threats: Techniques, solutions, challenges, and research opportunities," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1851–1877, 2019.

[10] J. P. Anderson, "Computer Security Technology Planning Study," Air Force Electronic Systems Division, Tech. Rep. ESD-TR-73-51, 1972.

[11] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 2014, pp. 303–316.

[12] Y. Bachrach, E. Porat, and J. S. Rosenschein, "Sketching techniques for collaborative filtering." in *IJCAI*, 2009, pp. 2016–2021.

[13] L. D. Baker and A. K. McCallum, "Distributional clustering of words for text classification," in *Conference on Research and Development in Information Retrieval*. ACM, 1998, pp. 96–103.

[14] M. Baquiran and D. Wren, "Endpoint Protection 2017 - Performance Testing," PassMark Software, Tech. Rep., 2017, https://www.symantec.com/content/dam/symantec/docs/reviews/endpoint-protection-2017-performance-testing-enterprise-Win10.pdf.

[15] M. Barre, A. Gehani, and V. Yegneswaran, "Mining data provenance to detect advanced persistent threats," in *11th International Workshop on Theory and Practice of Provenance (TaPP)*, 2019.

[16] A. M. Bates, D. Tian, K. R. Butler, and T. Moyer, "Trustworthy whole-system provenance for the linux kernel." in *USENIX Security Symposium*, 2015, pp. 319–334.

[17] M. Berlingerio, D. Koutra, T. Eliassi-Rad, and C. Faloutsos, "Netsimile: A scalable approach to size-independent network similarity," *arXiv preprint arXiv:1209.2684*, 2012.

[18] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, no. 3, pp. 81–84, 2014.

[19] G. Berrada and J. Cheney, "Aggregating unsupervised provenance anomaly detectors," in *11th International Workshop on Theory and Practice of Provenance (TaPP)*, 2019.

[20] L. Bilge and T. Dumitras, "Before we knew it: an empirical study of zero-day attacks in the real world," in *Conference on Computer and Communications Security*. ACM, 2012, pp. 833–844.

[21] L. Carata, S. Akoush, N. Balakrishnan, T. Bytheway, R. Sohan, M. Seltzer, and A. Hopper, "A primer on provenance," *ACM Queue*, vol. 12, no. 3, p. 10, 2014.

[22] E. P. I. Center, "Equifax Data Breach," available at https://www.epic.org/privacy/data-breach/equifax/.

[23] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, vol. 41, no. 3, p. 15, 2009.

[24] O. Chapelle, P. Haffner, and V. N. Vapnik, "Support vector machines for histogram-based image classification," *IEEE transactions on Neural Networks*, vol. 10, no. 5, pp. 1055–1064, 1999.

[25] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. ACM, 2002, pp. 380–388.

[26] Q. Chen, R. Luley, Q. Wu, M. Bishop, R. W. Linderman, and Q. Qiu, "Anrad: A neuromorphic anomaly detection framework for massive concurrent data streams," *IEEE transactions on neural networks and learning systems*, vol. 29, no. 5, pp. 1622–1636, 2018.

[27] O. Chum, J. Philbin, A. Zisserman *et al.*, "Near duplicate image detection: min-hash and tf-idf weighting." in *BMVC*, vol. 810, 2008, pp. 812–815.

[28] G. Creech and J. Hu, "Generation of a new ids test dataset: Time to retire the kdd collection," in *IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2013, pp. 4487–4492.

[29] H. Debar, M. Dacier, M. Nassehi, and A. Wespi, "Fixed vs. variable-length patterns for detecting suspicious process behavior," in *European Symposium on Research in Computer Security*. Springer, 1998, pp. 1–15.

[30] Q. Ding, N. Katenka, P. Barford, E. Kolaczyk, and M. Crovella, "Intrusion as (anti) social communication: characterization and detection," in *International Conference on Knowledge Discovery and Data Mining*. ACM, 2012, pp. 886–894.

[31] C. Doersch, "Tutorial on variational autoencoders," *arXiv preprint arXiv:1606.05908*, 2016.

[32] J. Drew, T. Moore, and M. Hahsler, "Polymorphic malware detection using sequence classification methods," in *Security and Privacy Workshops (SPW)*. IEEE, 2016, pp. 81–87.

[33] J. Fairbanks, D. Ediger, R. McColl, D. A. Bader, and E. Gilbert, "A statistical framework for streaming graph analysis," in *Proceedings of the International Conference on Advances in Social Networks Analysis and Mining*. ACM, 2013, pp. 341–347.

[34] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, "Advances in knowledge discovery and data mining," 1996.

[35] M. Fazzini, "Tagging and Tracking of Multi-level Host Events for Transparent Computing," Accessed 21st January 2020, https://smartech.gatech.edu/handle/1853/56510.

[36] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," in *Symposium on Security and Privacy*. IEEE, 2003, pp. 62–75.

[37] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Symposium on Security and Privacy*. IEEE, 1996, pp. 120–128.

[38] J. H. Friedman, "On bias, variance, loss, and the curse-of-dimensionality," *Data mining and knowledge discovery*, vol. 1, no. 1, pp. 55–77, 1997.

[39] J. Gao, F. Liang, W. Fan, C. Wang, Y. Sun, and J. Han, "On community outliers and their efficient detection in information networks," in *International Conference on Knowledge Discovery and Data Mining*. ACM, 2010, pp. 813–822.

[40] P. Gao, X. Xiao, D. Li, Z. Li, K. Jee, Z. Wu, C. H. Kim, S. R. Kulkarni, and P. Mittal, "Saql: A stream-based query system for real-time abnormal system behavior detection," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 639–656.

[41] X. Gao, B. Xiao, D. Tao, and X. Li, "A survey of graph edit distance," *Pattern Analysis and applications*, vol. 13, no. 1, pp. 113–129, 2010.

[42] M. R. Garey, "A guide to the theory of np-completeness," *Computers and intractability*, 1979.

[43] T. Garfinkel *et al.*, "Traps and pitfalls: Practical problems in system call interposition based security tools." in *NDSS*, vol. 3, 2003, pp. 163–176.

[44] A. Gehani and D. Tariq, "Spade: support for provenance auditing in distributed environments," in *Middleware Conference*. ACM/IFIP/USENIX, 2012, pp. 101–120.

[45] L. Georget, M. Jaume, F. Tronel, G. Piolle, and V. V. T. Tong, "Verifying the reliability of operating system-level information flow control systems in linux," in *International Workshop on Formal Methods in Software Engineering*. IEEE/ACM, 2017, pp. 10–16.

[46] I. Goldberg, D. Wagner, R. Thomas, E. A. Brewer *et al.*, "A secure environment for untrusted helper applications: Confining the wily hacker," in *USENIX Security Symposium*, vol. 6, 1996, pp. 1–1.

[47] D. Golunski, "Gnu wget 1.18 - arbitrary file upload / remote code execution," 2016, https://www.exploit-db.com/exploits/40064/.

[48] A. GReAT, "Operation shadowhammer," 2019, https://securelist.com/operation-shadowhammer/89992/.

[49] M. Gupta, C. C. Aggarwal, J. Han, and Y. Sun, "Evolutionary clustering and analysis of bibliographic networks," in *Conference on Advances in Social Networks Analysis and Mining*. IEEE, 2011, pp. 63–70.

[50] W. Haider, G. Creech, Y. Xie, and J. Hu, "Windows based data sets for evaluation of robustness of host based intrusion detection systems (ids) to zero-day and stealth attacks," *Future Internet*, vol. 8, no. 3, p. 29, 2016.

[51] W. Haider, J. Hu, J. Slay, B. P. Turnbull, and Y. Xie, "Generating realistic intrusion detection system dataset based on fuzzy qualitative modeling," *Journal of Network and Computer Applications*, vol. 87, pp. 185–192, 2017.

[52] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems*, 2017, pp. 1024–1034.

[53] X. Han, T. Pasquier, T. Ranjan, M. Goldstein, and M. Seltzer, "Frappuccino: fault-detection through runtime analysis of provenance," in *Workshop on Hot Topics in Cloud Computing (HotCloud'17)*. USENIX Association, 2017.

[54] D. J. Hand, "Principles of data mining," *Drug safety*, vol. 30, no. 7, pp. 621–622, 2007.

[55] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "Nodoze: Combatting threat alert fatigue with automated provenance triage." in *NDSS*, 2019.

[56] W. U. Hassan, M. Lemay, N. Aguse, A. Bates, and T. Moyer, "Towards scalable cluster auditing through grammatical inference over provenance graphs," in *Network and Distributed System Security Symposium, NDSS*, 2018.

[57] M. A. Hearst, S. T. Dumais, E. Osuna, J. Platt, and B. Scholkopf, "Support vector machines," *IEEE Intelligent Systems and their applications*, vol. 13, no. 4, pp. 18–28, 1998.

[58] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. D. Stoller, and V. Venkatakrishnan, "Sleuth: Real-time attack scenario reconstruction from cots audit data," in *USENIX Security Symposium*, 2017, pp. 487–504.

[59] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Symposium on Theory of Computing*. ACM, 1998, pp. 604–613.

[60] B. Jacob, P. Larson, B. Leitao, and S. Da Silva, "Systemtap: instrumenting the linux kernel for analyzing performance and functional problems," *IBM Redbook*, vol. 116, 2008.

[61] T. Jaeger, A. Edwards, and X. Zhang, "Consistency analysis of authorization hook placement in the linux security modules framework," *ACM Transactions on Information and System Security (TISSEC)*, vol. 7, no. 2, pp. 175–205, 2004.

[62] J. H. Jafarian, A. Abbasi, and S. S. Sheikhabadi, "A gray-box dpda-based intrusion detection technique using system-call monitoring," in *Annual Collaboration, Electronic messaging, Anti-Abuse and Spam Conference*. ACM, 2011, pp. 1–12.

[63] K. Jain and R. Sekar, "User-level infrastructure for system call interposition: A platform for intrusion detection and confinement." in *NDSS*, 2000.

[64] J. Ji, J. Li, S. Yan, Q. Tian, and B. Zhang, "Min-max hash for jaccard similarity," in *International Conference on Data Mining*. IEEE, 2013, pp. 301–309.

[65] L. Kaufman and P. J. Rousseeuw, *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, 2009, vol. 344.

[66] A. D. Keromytis, "Transparent computing engagement 3 data release," Accessed 21st January 2020, https://github.com/darpa-i2o/Transparent-Computing.

[67] S. Khandelwal, "Ccleaner attack timeline. here's how hackers infected 2.3 million pcs," 2018, https://thehackernews.com/2018/04/ccleaner-malware-attack.html.

[68] W. Khreich, S. S. Murtaza, A. Hamou-Lhadj, and C. Talhi, "Combining heterogeneous anomaly detectors for improved software security," *Journal of Systems and Software*, vol. 137, pp. 415–429, 2018.

[69] S. T. King and P. M. Chen, "Backtracking intrusions," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 223–236, 2003.

[70] R. Klinkenberg, "Learning drifting concepts: Example selection vs. example weighting," *Intelligent data analysis*, vol. 8, no. 3, pp. 281–300, 2004.

[71] R. Kondor, N. Shervashidze, and K. M. Borgwardt, "The graphlet spectrum," in *International Conference on Machine Learning*. ACM, 2009, pp. 529–536.

[72] O. Kostakis, "Classy: fast clustering streams of call-graphs," *Data mining and knowledge discovery*, vol. 28, no. 5-6, pp. 1554–1585, 2014.

[73] I. Koychev, "Gradual forgetting for adaptation to concept drift," in *Workshop on Current Issues in Spatio-Temporal Reasoning*. ICJAI/ECAI, 2000.

[74] G. Kurtz, "Operation Aurora Hit Google, Others," Jan 2010, available at http://securityinnovator.com/index.php?articleID=42948&sectionID=25.

[75] A. Kyrola, G. E. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a pc," in *Symposium on Operating Systems Design and Implementation*. USENIX, 2012.

[76] P. Li, "0-bit consistent weighted sampling," in *International Conference on Knowledge Discovery and Data Mining*. ACM, 2015, pp. 665–674.

[77] C. Liu, X. Yan, H. Yu, J. Han, and P. S. Yu, "Mining behavior graphs for backtrace of noncrashing bugs," in *International Conference on Data Mining*. SIAM, 2005, pp. 286–297.

[78] M. Liu, Z. Xue, X. Xu, C. Zhong, and J. Chen, "Host-based intrusion detection system with system calls: Review and future trends," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, p. 98, 2018.

[79] Y. Liu, M. Zhang, D. Li, K. Jee, Z. Li, Z. Wu, J. Rhee, and P. Mittal, "Towards a timely causality analysis for enterprise security." NDSS, 2018.

[80] S. Ma, X. Zhang, and D. Xu, "Protracer: Towards practical provenance tracing by alternating between logging and tainting." in *NDSS*, 2016.

[81] F. Maggi, M. Matteucci, and S. Zanero, "Detecting intrusions through system call sequence and argument analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 381–395, 2010.

[82] M. Manasse, F. McSherry, and K. Talwar, "Consistent weighted sampling," *Technical Report http://research. microsoft.com/en-us/people/manasse*, vol. 2, 2010.

[83] E. Manzoor, S. M. Milajerdi, and L. Akoglu, "Fast memory-efficient anomaly detection in streaming heterogeneous graphs," in *International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 1035–1044.

[84] ——, "Streamspot datasets," 2016, https://github.com/sbustreamspot/sbustreamspot-data.

[85] J. McHugh, "Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory," *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 4, pp. 262–294, 2000.

[86] S. M. Milajerdi, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, "Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting," in *Proceedings of the SIGSAC Conference on Computer and Communications Security*. ACM, 2019, pp. 1813–1830.

[87] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "Holmes: Real-time apt detection through correlation of suspicious information flows," in *Symposium on Security and Privacy*. IEEE, 2019.

[88] M. Mongiovi, P. Bogdanov, R. Ranca, E. E. Papalexakis, C. Faloutsos,

and A. K. Singh, “Netspot: Spotting significant anomalous regions on dynamic networks,” in *International Conference on Data Mining*. SIAM, 2013, pp. 28–36.

[89] J. Morris, S. Smalley, and G. Kroah-Hartman, “Linux security modules: General security support for the linux kernel,” in *USENIX Security Symposium*, 2002.

[90] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer, “Provenance-aware storage systems.” in *USENIX Annual Technical Conference*, 2006, pp. 43–56.

[91] S. S. Murtaza, A. Hamou-Lhadj, W. Khreich, and M. Couture, “Total ads: Automated software anomaly detection system,” in *14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014, pp. 83–88.

[92] D. Mutz, W. Robertson, G. Vigna, and R. Kemmerer, “Exploiting execution context for the detection of anomalous system calls,” in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2007, pp. 1–20.

[93] D. Namiot and M. Sneps-Sneppe, “On micro-services architecture,” *International Journal of Open Information Technologies*, vol. 2, no. 9, pp. 24–27, 2014.

[94] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, “graph2vec: Learning distributed representations of graphs,” *CoRR*, vol. abs/1707.05005, 2017.

[95] M. Neuhaus and H. Bunke, “Self-organizing maps for learning the edit costs in graph matching,” *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 35, no. 3, pp. 503–514, 2005.

[96] C. Nikos Virvilis, G. CISSP, C. Oscar Serrano, C. CISM *et al.*, “Big data analytics for sophisticated attack detection,” 2014.

[97] P. Papadimitriou, A. Dasdan, and H. Garcia-Molina, “Web graph similarity for anomaly detection,” *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 19–30, 2010.

[98] C. Parampalli, R. Sekar, and R. Johnson, “A practical mimicry attack against powerful system-call monitors,” in *Symposium on Information, Computer and Communications Security*. ACM, 2008, pp. 156–167.

[99] J. Park, D. Nguyen, and R. Sandhu, “A provenance-based access control model,” in *International Conference on Privacy, Security and Trust*. IEEE, 2012, pp. 137–144.

[100] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, “Practical whole-system provenance capture,” in *Symposium on Cloud Computing*. ACM, 2017, pp. 405–418.

[101] T. Pasquier, X. Han, T. Moyer, A. Bates, O. Hermant, D. Eyers, J. Bacon, and M. Seltzer, “Runtime analysis of whole-system provenance,” in *Conference on Computer and Communications Security (CCS’18)*. ACM, 2018.

[102] K. Pei, Z. Gu, B. Saltaformaggio, S. Ma, F. Wang, Z. Zhang, L. Si, X. Zhang, and D. Xu, “Hercule: Attack story reconstruction via community discovery on correlated log graph,” in *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 2016, pp. 583–595.

[103] B. Perozzi, L. Akoglu, P. Iglesias Sánchez, and E. Müller, “Focused clustering and outlier detection in large attributed graphs,” in *International Conference on Knowledge Discovery and Data Mining*. ACM, 2014, pp. 1346–1355.

[104] D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, “Hi-fi: collecting high-fidelity whole-system provenance,” in *Computer Security Applications Conference*. ACM, 2012, pp. 259–268.

[105] N. Provos, “Systrace-interactive policy generation for system calls,” 2006.

[106] T. Radichel, “Case Study: Critical Controls that Could Have Prevented Target Breach,” SANS Institute InfoSec Reading Room, September 2014, available at https://www.sans.org/reading-room/whitepapers/casestudies/paper/35412.

[107] E. Raff and C. Nicholas, “Malware classification and class imbalance via stochastic hashed lzjd,” in *Proceedings of the 10th Workshop on Artificial Intelligence and Security*. ACM, 2017, pp. 111–120.

[108] P. J. Rousseeuw, “Silhouettes: a graphical aid to the interpretation and validation of cluster analysis,” *Journal of computational and applied mathematics*, vol. 20, pp. 53–65, 1987.

[109] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, “A fast automaton-based method for detecting anomalous program behaviors,” in *Symposium on Security and Privacy (S&P)*. IEEE, 2001, pp. 144–155.

[110] N. Shervashidze, P. Schweitzer, E. J. v. Leeuwen, K. Mehlhorn, and K. M. Borgwardt, “Weisfeiler-lehman graph kernels,” *Journal of Machine Learning Research*, vol. 12, no. Sep, pp. 2539–2561, 2011.

[111] A. Shrivastava and P. Li, “In defense of minhash over simhash,” in *Artificial Intelligence and Statistics*, 2014, pp. 886–894.

[112] X. Shu, D. D. Yao, N. Ramakrishnan, and T. Jaeger, “Long-span program behavior modeling and attack detection,” *ACM Transactions on Privacy and Security (TOPS)*, vol. 20, no. 4, p. 12, 2017.

[113] X. Shu, D. D. Yao, and B. G. Ryder, “A formal framework for program anomaly detection,” in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2015, pp. 270–292.

[114] A. Somayaji and S. Forrest, “Automated response using system-call delay,” in *Usenix Security Symposium*, 2000, pp. 185–197.

[115] J. Sun, C. Faloutsos, S. Papadimitriou, and P. S. Yu, “Graphscope: parameter-free mining of large time-evolving graphs,” in *International Conference on Knowledge Discovery and Data Mining*. ACM, 2007, pp. 687–696.

[116] W. Symantec, “Advanced persistent threats: A symantec perspective,” *Symantec World Headquarters*, 2011.

[117] J. E. Tapiador and J. A. Clark, “Masquerade mimicry attack detection: A randomised approach,” *Computers & Security*, vol. 30, no. 5, pp. 297–310, 2011.

[118] D. Tariq, B. Baig, A. Gehani, S. Mahmood, R. Tahir, A. Aqil, and F. Zaffar, “Identifying the provenance of correlated anomalies,” in *Symposium on Applied Computing*. ACM, 2011, pp. 224–229.

[119] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, “Auto-weka: Combined selection and hyperparameter optimization of classification algorithms,” in *Proceedings of the 19th SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2013, pp. 847–855.

[120] A. Tsymbal, “The problem of concept drift: definitions and related work,” *Computer Science Department, Trinity College Dublin*, vol. 106, no. 2, 2004.

[121] E. van der Kouwe, D. Andriesse, H. Bos, C. Giuffrida, and G. Heiser, “Benchmarking crimes: an emerging threat in systems security,” *arXiv preprint arXiv:1801.02381*, 2018.

[122] S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt, “Graph kernels,” *Journal of Machine Learning Research*, vol. 11, no. Apr, pp. 1201–1242, 2010.

[123] D. Wagner and P. Soto, “Mimicry attacks on host-based intrusion detection systems,” in *Conference on Computer and Communications Security*. ACM, 2002, pp. 255–264.

[124] J. Wang, H. T. Shen, J. Song, and J. Ji, “Hashing for similarity search: A survey,” *arXiv preprint arXiv:1408.2927*, 2014.

[125] R. N. Watson, “Exploiting concurrency vulnerabilities in system call wrappers.” *WOOT*, vol. 7, pp. 1–8, 2007.

[126] B. Weisfeiler and A. Lehman, “A reduction of a graph to a canonical form and an algebra arising during this reduction,” *Nauchno-Technicheskaya Informatsia*, vol. 2, no. 9, pp. 12–16, 1968.

[127] A. Wespi, M. Dacier, and H. Debar, “Intrusion detection using variable-length audit trail patterns,” in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2000, pp. 110–129.

[128] W. Wu, B. Li, L. Chen, and C. Zhang, “Consistent weighted sampling made more practical,” in *International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2017, pp. 1035–1043.

[129] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, “How powerful are graph neural networks?” *arXiv preprint arXiv:1810.00826*, 2018.

[130] K. Xu, K. Tian, D. Yao, and B. G. Ryder, “A sharper sense of self: Probabilistic reasoning of program behaviors for anomaly detection with context sensitivity,” in *International Conference on Dependable Systems and Networks*. IEEE/IFIP, 2016, pp. 467–478.

[131] T. Yadav and A. M. Rao, “Technical aspects of cyber kill chain,” in *International Symposium on Security in Computing and Communication*. Springer, 2015, pp. 438–452.

[132] D. Yang, B. Li, L. Rettig, and P. Cudré-Mauroux, “Histosketch: Fast similarity-preserving sketching of streaming histograms with concept drift,” in *International Conference on Data Mining (ICDM)*. IEEE, 2017, pp. 545–554.

[133] J. Yang, Y.-G. Jiang, A. G. Hauptmann, and C.-W. Ngo, “Evaluating bag-of-visual-words representations in scene classification,” in *Workshop on multimedia information retrieval*. ACM, 2007, pp. 197–206.

[134] N. Ye *et al.*, “A markov chain model of temporal behavior for anomaly detection,” in *Systems, Man, and Cybernetics Information Assurance and Security Workshop*, vol. 166. West Point, NY, 2000, p. 169.

Appendix

A. Availability

The implementation described in § V and the material to reproduce the evaluation presented in § VI are available online under Apache License V2.0 and GPL v2 (see individual subcomponents for more details) at https://github.com/crimson-unicorn.

B. HistoSketch

Notation and Basic Concepts. Table X presents the notation we use in the rest of this section. Similarity-preserving data sketching, or locality sensitive hashing (LSH), allows us to efficiently compute the similarity between two graphs by projecting their high-dimensional histogram representations to a low-dimensional space while preserving their similarity [83].

ϵ	$\triangleq$	Set of histogram elements
h	$\triangleq$	Histogram element: $h \in \epsilon$
L	$\triangleq$	Cumulative, weighted histogram count vector: $L \in \mathbb{R}_{>0}^{\lvert\epsilon\rvert}$
L_h	$\triangleq$	Count of histogram element h
λ	$\triangleq$	Weight decay factor
w_t	$\triangleq$	Exponential decay weight: $w_t = e^{-\lambda\Delta t}$
S	$\triangleq$	Sketch of histogram vector L: $\lvert S\rvert \ll \lvert\epsilon\rvert$ and is fixed

TABLE X: Notation

Formally, we define LSH as follows (adopted from Charikar [25]):

Definition 1: A locality sensitive hashing scheme is a distribution on a family $\mathcal{F}$ of hash functions operating on a collection of objects, such that for two objects m, n,

$$P_{h\in\mathcal{F}}[h(m) = h(n)] = sim(m, n)$$

where $sim(m, n)$ is some similarity function defined on the collection of objects.

HistoSketch [132] uses *normalized min-max similarity* to measure the similarity between two histogram vectors:

Definition 2:

$$sim_{min-max}(H^a, H^b) = \frac{\sum_{h\in\epsilon} min(H_h^a, H_h^b)}{\sum_{h\in\epsilon} max(H_h^a, H_h^b)}$$

$$\sum_{h\in\epsilon} H_h^a = 1$$

$$\sum_{h\in\epsilon} H_h^b = 1$$

where the superscript a, b denotes the identity of a histogram.

Sketch Creation. HistoSketch uses a variation of consistent weighted sampling that takes as input positive real numbers to generate fixed-size sketches [76]. The size of the sketch $|S|$ controls the tradeoffs between information loss and computation efficiency for real-time detection (§ VI).

To generate one sketch element S_j, we first draw three random variables for each $h \in \epsilon$:

$$r_{h,j} \sim Gamma(2, 1)$$
$$c_{h,j} \sim Gamma(2, 1)$$
$$\beta_{h,j} \sim Uniform(0, 1)$$

and then follow the steps in Alg. 3. r, c, and β are fixed for each element in ϵ. The sketch element j is the element h in ϵ whose hashed value $a_{h,j}$ is the minimum in the j^{th} column of matrix A.

Algorithm 3: Creating Graph Sketch Using HistoSketch

```
Input : Histogram $L$, $r$, $c$, $\beta$
Output: Sketch $S$ and the corresponding hash values $A$
1 for $j \leftarrow 1$ to $|S|$ do
2     foreach $h \in \epsilon$ do
3         $y_{h,j} = exp(logL_h - r_{h,j}\beta_{h,j})$
4         $a_{h,j} = \frac{c_{h,j}}{y_{h,j}exp(r_{h,j})}$
5     $S_j = argmin_{h\in\epsilon} a_{h,j}$
6     $A_j = min_{h\in\epsilon} a_{h,j}$
```

Sketch Update. HistoSketch makes it possible to quickly update the sketch as new data arrives. At time $t + 1$, it incrementally updates the sketch $S(t + 1)$ based on the weighted histogram L, the previous sketch $S(t)$ and its corresponding hash values $A(t)$, the new data item x_{t+1}, and the weight decay factor λ. Alg. 4 describes this process in detail.

Algorithm 4: Updating Graph Sketch

```
Input : $L$, $S(t)$, $A(t)$, $x_{t+1}$, $\lambda$
Output: New sketch $S(t + 1)$ and the corresponding hash values $A(t + 1)$
1  foreach $h \in \epsilon$ do
2      $L_h = L_h(t) \cdot e^{-\lambda}$
3                                  /* exponential decay */
4  for $j \leftarrow 1$ to $|S|$ do
5      if $x_{t+1} \in \epsilon$ then $L_{x_{t+1}} = L_{x_{t+1}} + 1$
6      else
7          $\epsilon = \epsilon + \{x_{t+1}\}$
8          $L_{x_{t+1}} = 1$
9      Compute $a_{x_{t+1},j}$
10                                       /* see Alg. 3 */
11     if $a_{x_{t+1},j} < A_j(t) \cdot e^{-\lambda}$ then
12         $S_j(t + 1) = x_{t+1}$
13         $A_j(t + 1) = a_{x_{t+1},j}$
14     else
15         $S_j(t + 1) = S_j(t)$
16         $A_j(t + 1) = A_j(t) \cdot e^{-\lambda}$
```

C. Metrics

We denote false positives by fp, false negatives fn, true positives tp, and true negatives tn.

Definition 3: precision = $\frac{tp}{tp+fp}$

Definition 4: recall = $\frac{tp}{tp+fn}$

Precision and recall measure *relevance* in classification, where precision is a measure of *exactness* and recall *completeness*.

Definition 5: accuracy = $\frac{tp+tn}{tp+tn+fp+fn}$

Definition 6: F-score = $2 \times \frac{precision \times recall}{precision+recall}$

F-score combines precision and recall using their harmonic mean. In our measurement, recall and precision are evenly weighted.