Efficient Host Intrusion Detection using Hyperdimensional Computing

Yujin Nam
Dept. of Computer Science and Engineering
UC San Diego
La Jolla, USA
yujinnam@ucsd.edu

Rachel King
Dept. of Computer Sciences
University of Wisconsin-Madison
Madison, USA
rachelking@cs.wisc.edu

Quinn Burke
Dept. of Computer Sciences
University of Wisconsin-Madison
Madison, USA
qkb@cs.wisc.edu

Minxuan Zhou
Computer Science Department
Illinois Institute of Technology
Chicago, USA
mzhou26@iit.edu

Patrick McDaniel
Dept. of Computer Sciences
University of Wisconsin-Madison
Madison, USA
mcdaniel@.cs.wisc.edu

Tajana Rosing
Dept. of Computer Science and Engineering
UC San Diego
La Jolla, USA
tajana@ucsd.edu

***Abstract*—Modern host-based intrusion detection systems (HIDS) rely on querying provenance graphs—graph representations of activity history on a system—to detect and respond to security threats present on a system. However, as the complexity and number of applications running on a system increase, the size of provenance graphs also increase, and thus the latency to query them. State-of-the-art designs deliver query latencies that are impractical for modern threat detection. In this paper, we introduce a hyper-dimensional computing (HDC) approach to querying provenance graphs for HIDS. By encoding provenance graphs and attack patterns/signatures into hyper-dimensional vectors, we can implement a query engine using simple vector operations. Our approach is hardware accelerator compatible, providing further speedups under resource-constrained environments. Our evaluation on a real-world dataset shows that our approach achieves $> 90\%$ detection accuracy and up to $4,242\times$ speedups over the state-of-the-art. This shows that HDC-based approaches can effectively deal with scaling issues in modern HIDS.**

I. Introduction

Modern host-based intrusion detection systems (HIDS) rely on collecting and analyzing system-call logs to detect and respond to security threats present on a system [1], [2]. The logs are collected by kernel-level agents and used to generate *provenance graphs*, concise graph representations of activity history on the system. For example, provenance graphs capture what processes spawn other processes, access what files, and send and receive network traffic. The graphs are then queried offline to identify *behavioral patterns* that are consistent with malicious activity; this practice is commonly referred to as *threat hunting*. The behavioral patterns most often reflect the tactics, techniques, and procedures (TTPs) described in the industry-standard MITRE ATT&CK framework [3].

However, the increased complexity and number of applications running on modern computing systems has amplified the number of system-calls that are made, and thus the volume of data that must be collected and queried. Increased provenance graph size inevitably leads to increased query latency, which can negatively impact the ability of a security analyst to detect and respond to threats within a reasonable amount of time [1], [2], [4], [5]. It can also lead to increased power consumption, and it is important to consider how HIDS can be efficiently deployed in various contexts, such as on resource-constrained edge devices. While recent efforts have made strides towards developing compact, rich representations of provenance graphs [6], query latencies observed under these state-of-the-art approaches are still far from practical. Thus, designing faster methods to query provenance graphs is paramount to being able to detect threats as quickly as possible.

In this paper, we introduce an accelerated provenance graph query engine to detect malicious behavior. We explore provenance graph querying through the lens of hyperdimensional computing (HDC) [7]. Specifically, state-of-the-art designs rely on high-latency, backward- and forward-tracing algorithms to match known behavioral patterns (i.e., *signatures*) in provenance graphs. In contrast, we adopt a probabilistic approach by encoding provenance graphs and signatures into hyperdimensional vectors. This enables us to avoid high-latency traversal algorithms and instead perform queries using simple vector operations. Given the importance of intrusion detection on resource-constrained devices and that HDC is hardware-friendly, we also extend our HDC query engine to a state-of-the-art processing-in-memory accelerator with optimized data layout and processing flow, demonstrating significant performance improvements and power savings.

Our approach is comprised of three phases: signature generation, encoding, and threat detection. We first introduce a method to extract signatures from provenance graphs based on known TTPs. We then encode the graphs and signatures into hypervectors. Finally, we query the graphs to detect the presence of TTPs. Additionally, we demonstrate the compatibility of our approach with hardware accelerators to further improve latency. While using HDC requires balancing the trade-off

between accuracy (i.e., whether the signature is present and was detected) and latency, our approach still achieves $> 90\%$ accuracy with speedups of up to $4,242\times$ over state-of-the-art on CPU and $18,000\times$ on a hardware accelerator. Our HDC approach also consumes up to $5,300\times$ and 12 orders of magnitude less power with CPU and hardware acceleration than the baseline.

Our contributions. We contribute the following:

- The first HDC-based solution for querying provenance graphs for the presence of TTPs.
- Optimized data layout and processing flow of mapping the hardware-friendly HDC-based provenance graph to a state-of-the-art processing in-memory accelerator.
- An evaluation of our HDC approach on a real-world dataset showing that it can deliver sub-millisecond latency with high accuracy and low power consumption.

II. Background

A. Threat Hunting with Provenance Graphs

Threat hunting is the practice of proactively searching through host system logs (i.e., provenance graphs) to detect and isolate advanced threats that evade existing security solutions such as network firewalls. Threat hunting is recognized as a graph querying problem [2]. A query is typically structured as an attack signature (i.e., a subgraph) and posed against a provenance graph as a binary decision on whether or not the attack signature is present. Like traditional signatures in other contexts such as packet matching in network intrusion detection [8], queries are most commonly generated offline, but can also be designed and generated on-the-fly based on domain knowledge and situation-specific observations.

The graphs themselves are typically structured as trees, and there are two standard methods by which queries are posed against the graph: backward-tracing and forward-tracing. The former method begins at each node and traverses upwards until a node (event) with no parent is found. The latter method begins at each node and traverses downward until a node with no child is found. Consequently, backward- and forward-tracing suffer from the path-explosion problem, where the number of paths to be searched grows exponentially with the size of the tree. We focus specifically on backward tracing, which is the standard method to match attack signatures [2].

Many priors works have explored the use of provenance graphs for HIDS [5], [9]–[11]. These works have explored various techniques such as signature matching, machine learning, and others. In contrast to these works, we focus on encoding the graphs into vector space in the context of HDC, such that we can implement an HDC-based query engine to efficiently perform signature detection rather than compute anomaly scores. To the best of our knowledge, we are the first to propose the use of HDC for querying provenance graphs in HIDS.

B. SysFlow Telemetry Framework

We leverage the state-of-the-art SysFlow telemetry framework to collect system events and generate provenance graphs [6]. Similar to other widely used frameworks like Linux Auditd, SysFlow leverages tracepoints in the kernel to intercept system calls and log various attributes such as the syscall number, arguments, timestamps, etc. SysFlow provenance graphs contain three high-level *node types*: process, file, and network. Each node (of any type) is also classified in one of three classes: entities, events, and flows. This is shown in Figure 1. Entities represent nodes where code is currently being executed, such as virtual machines, containers, and processes. Entity behaviors are modeled as events and flows. Events are behaviors that represent a state change in the entity (e.g., the process is exited or is listening on a TCP port). Flows are aggregated sets of events that describe a higher-level behavior (e.g., connect, read, write, and close calls represent a normal Unix socket lifecycle). SysFlow also provides state-of-the-art semantic compression techniques that can parse the syscall logs to produce a compact, rich provenance graph representation.

C. Hyperdimensional Computing

Hyperdimensional computing (HDC) is a computational method that shows high efficiency and noise tolerance [12]. HDC encodes input data into a high-dimensional representation, or hyper vector. Unlike common learning methods, hyper vectors can serve as a model in HDC, making the model size small. Hyper vector representation is also holographic, making the model noise tolerant. HDC is also highly parallel and suitable for hardware acceleration. Because of the advantages, HDC has been widely used in applications including classification, graph prediction, and pattern recognition. Previous works have shown that it can significantly reduce resource consumption compared to neural networks and further improve the latency and efficiency using hardware implementations.

Encoding is a fundamental operation in HDC, where data is mapped into a high-dimensional vector of size D, known as hypervectors. The encoding method varies depending on the input data, to most accurately capture its characteristics. It is especially emphasized because in HDC, decision making is done by comparing hyper vectors. For example, in a pattern matching task, an encoded data would be compared to an encoded pattern that we are looking for. If the similarity between the two are close enough, it is decided that the pattern exists in the input data.

Some of the key operations for HDC tasks are as follows:

- Bundling ($+$): Element-wise addition between two hyper vectors. Bundling preserves the information from both inputs.
- Binding ($\odot$): Element-wise multiplication between two hyper vectors. Binding generates new information from the inputs.
- Similarity ($\delta(,)$): A metric to measure a similarity between two inputs. Commonly used metrics include Hamming distance, Euclidean distance and cosine similarity.

In this work, we apply HDC for host-based intrusion detection for the first time leveraging the advantages of HDC with provenance graph inputs. There are a few prior

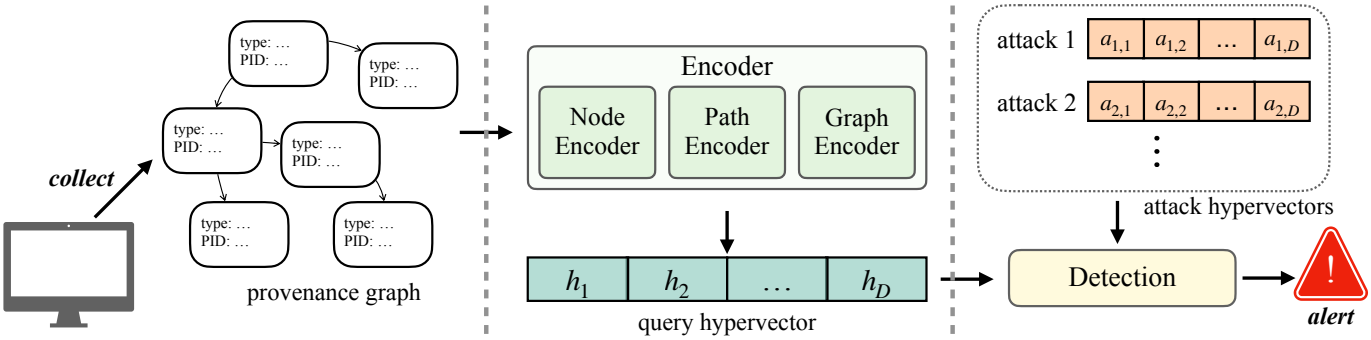


Fig. 1: Overview of our methodology.

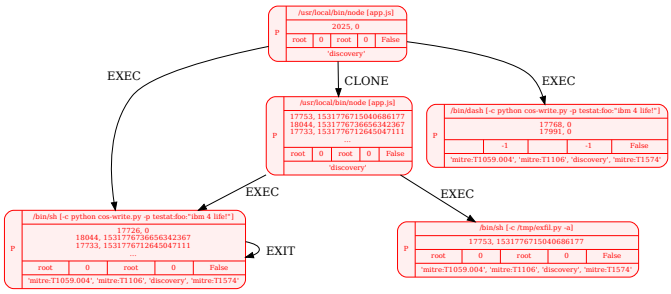


Fig. 2: TTP signatures 1059.004, 1106, and 1574.

works dealing with graph inputs or intrusion detection task. RelHD [7] proposed an encoding method for graph-like data to represent the relationship between the nodes. They took both different data types of nodes and the topology information into consideration. However, their model objective is to classify an input node while our objective is to classify an entire input graph. GraphHD [13] proposed a graph classification based on HDC, showing their scalability with real-world applications. However, their encoding focuses on a topology of a graph, which is not applicable for various applications that takes node information into consideration. HyperDetect [14] proposed the first HDC-based solution for network-based intrusion detection. They also improved the HDC learning method taking momentum into consideration. However, they did not suggest an encoding method for provenance graphs and it is nontrivial to extend to provenance graphs.

III. Methodology

The goal of our approach is to address the performance limitations of modern provenance graph querying solutions for HIDS. However, as the landscape of computing systems continues to grow, their provenance graphs do as well. With this, the latency of querying using state-of-the-art solutions is no longer acceptable. For example, querying SysFlow [6] provenance graphs generated from a single host across 1-hour of activity can take upwards of 10 seconds at the tail end. Querying for the presence of several TTPs can therefore take several minutes, at which point many attacks may have already successfully completed.

We consider the use of HDC in provenance graph querying for HIDS due to its lightweight schemes and high efficiency. Applying HDC, we encode both TTP patterns and a query provenance graph. Querying can be done by a simple vector comparison between the two encodings. This allows a timely detection of attacks. Furthermore, our approach is hardware acceleration friendly, as HDC operations consists of highly parallelizable vector operations. We introduce the details of the three phases in our methodology: signature generation, encoding, and threat detection, in the following sections.

A. Signature Generation

The first phase involves generating signatures matching known TTPs defined in the MITRE ATT&CK framework. The purpose of the signature is to capture the system-call pattern exhibited by a particular TTP, and use the pattern as the query on the provenance graph to detect the presence of the TTP. We note that this signature is different from traditional signatures that are used in network intrusion detection systems, which are based on regular expressions or byte sequences and do stateless matching. We aim to capture richer, inter-event relationships by matching paths (and subgraphs in general) in the provenance graph.

Sysflow does not natively support generating such signatures, but does support simple graph operations and a policy engine that provides facilities for defining higher-order logic expressions that are checked against each event. We can therefore specify the set of conditions that must be satisfied for a node (or path) in the graph to indicate the presence of a TTP. For example, to match TTP T1106 (Suspicious process spawned), a path in the graph must satisfy the following: (1) a privileged user process must be running, (2) the process must spawn a child process from an untrusted directory (e.g., */tmp*). An example is shown in Figure 2.

Given this, during construction of the provenance graph, we can use the policy engine to tag nodes that satisfy the conditions for a TTP. We can then retrieve a subgraph of the nodes that are tagged with a specified TTP, and the subgraph can be used as our attack signature. We note that attacks may manifest in different ways, and thus a single TTP may be represented by multiple, different signatures. For example, an attacker may access several other files before spawning a

suspicious child process. As discussed below, these signatures are then encoded into hypervectors for detection.

B. HDC Encoding

Given an input provenance graph, our HDC-based approach encodes graphs and signatures into hypervectors. The encoding process needs to accurately capture both the nodes' information as well as their relationships within the graph. Our encoding approach consists of three stages: node encoding, path encoding, and graph encoding. First, node encoder captures each node's collected syscalls. Then, path encoder integrates the node encodings to represent the relation between the nodes. Finally, graph encoder aggregates the path encodings.

1) Node Encoder: Each node holds various information, including the node type and collected log data. A few of the key log data includes the executable and its arguments that created the node. We propose a novel encoding method that takes these key data into consideration, utilizing bundling and binding operations between hypervectors. We start by assigning hypervectors randomly from $\{-1, 1\}^D$ for every feature value. For instance, node *type* features can take on one of three values: process, network, or file. We generate three hypervectors for type feature values. Similarly, we generate random hypervectors for executable features. Additionally, we use binding operations to create feature hypervectors that are closely associated with other features. For example, since arguments are related to the executable, we generate an argument feature hypervector by binding an executable feature hypervector with a randomly generated hypervector for each argument. Finally, we encode the node by bundling the feature hypervectors and applying min-max normalization to ensure consistency across nodes.

2) Path Encoder: With the generated node hypervectors, we proceed to encode paths within a graph. We extended the encoding method introduced in a prior work RelHD's [7] to capture the relations between nodes. We adopted the method to work with an arbitrary length of paths within the graph, enhancing the capability of our framework to efficiently identify attack patterns of any length within provenance graphs. We define a k-hop paths of node n as all routes within the graph that can be traversed by moving along k edges, starting at n. For each location of i-th hop, we generate a position hypervector $\mathbf{posHV}_i$ randomly. The encoding of the k-hop paths is then evaluated as the summation of the bindings of each sum of i-hop node hypervectors and its corresponding position hypervector, $\mathbf{pathHV}_n = \sum_i \mathbf{H}_n^i \odot \mathbf{posHV}_i$, where $\mathbf{H}_n^i$ is sum of i-hop node hypervectors of node n. We optimize this process by reusing the $\mathbf{H}_n^i$s. For example, $\mathbf{H}_n^{i+1}$ can be evaluated by $\sum_j \mathbf{H}_j^i$, where j includes the i-hop nodes of node n. Through the reuse of intermediate sums of hypervectors, we effectively reduce the computation and memory usage.

3) Graph Encoder: Finally, we encode the entire graph using the path hypervectors by bundling every path hypervector within the graph. This preserves every path information inside the graph as a single hypervector, allowing the detection of path patterns.

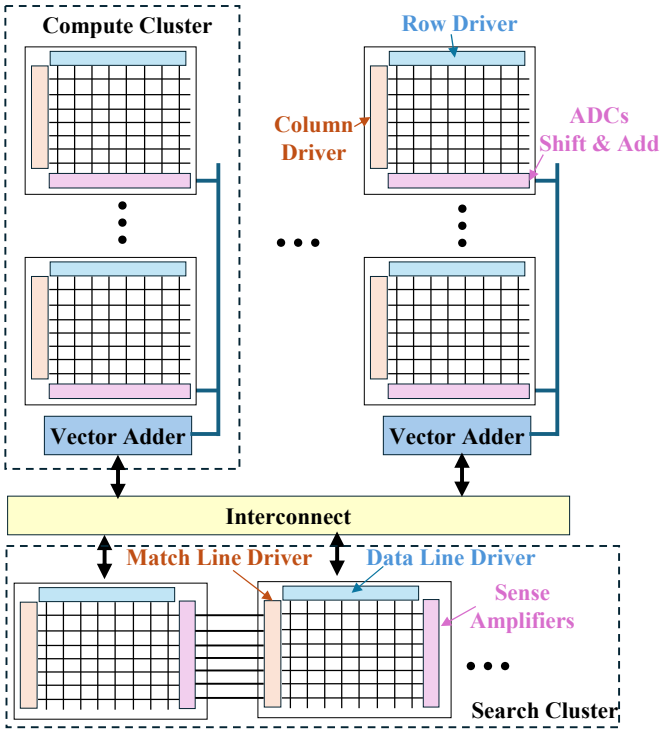


Fig. 3: FeFET-based PIM architecture for graph-based HDC.

C. Threat Detection

At runtime, syscall logs are collected at a specified timescale, the default being 60-second intervals for SysFlow. When a new set of records is saved to disk, we generate the provenance graph that is then queried against. In the standard Sysflow workflow, a query would effectively be performed by computing a graph intersection with the input provenance graph collected at runtime and the signature generated offline. If the intersection is exactly all the nodes in the signature, we conclude that the malicious events have occurred and have been detected.

For HDC, the TTP signatures are generated and encoded into hypervectors offline. At runtime, the input provenance graph is encoded into a hypervector. We then perform the query by computing the cosine similarity between the two hypervectors to evaluate the closeness of the vectors. If the similarity score is higher than a threshold th, we consider the threat detected and raise an alert.

IV. Hardware Acceleration

Prior works have noted the need for high-performance HIDS in edge-computing environments which are power constrained [15], [16]. HDC is also known as a hardware-friendly algorithm, for which customized accelerators can provide high performance and power/area efficiency. To demonstrate the applicability to resource-constrained environments, we therefore also propose an optimized data layout and processing flow to fully utilize the benefits of a state-of-the-art processing in-memory accelerator [7].

A. PIM Architecture

We exploit the FeFET technology, which has shown supreme efficiency on HDC acceleration, as the foundation of our PIM accelerator. We adopt the FeFET cluster design from previous work [7], which contains two types of FeFET blocks for computing and searching. Each compute cluster attaches an HD vector adder to a group of FeFET computing blocks to efficiently handle various HD vector operations (Figure 3). Specifically, each compute block can sum up hypervectors stored locally by activating corresponding rows, where each row can be applied a specific voltage to scale values before summation. The per-cluster adders handle cross-block vector accumulation. We adopt the compute cluster to calculate graph encoding. The search block conducts the max similarity search between the encoded graph and all encoded attack patterns.

B. Data Layout and Processing Flow

Figure 4 shows the data layout and processing flow of mapping HDC-based HIDS onto the PIM accelerator. An example provenance graph is shown in Figure 4(a). Based on the graph information, we first allocate a set of compute clusters to store feature hypervectors (HVs) for node encoding. Specifically, we allocate separate clusters for different executables, where each executable has a specific feature HV. Furthermore, we place HVs of all possible node types in the provenance graph (i.e., file, network, or process) along with each executable HV. We also allocate HVs for all possible features (i.e., command-line arguments) of a specific executable along with the corresponding executable HV. Such a layout strategy places HVs that are able to be bundled in the same memory clusters, encoding graph nodes with different executables simultaneously. However, graph nodes using the same executable can only be encoded sequentially. We improve the performance of node encoding by duplicating frequently used executables under memory constraints.

After generating all node HVs, we iteratively calculate path HVs for different hops, as shown in Figure 4(b). Unlike the strategy used in HDC-based graph learning acceleration [7], which sorts node HVs based on the edge distribution to maximize parallelization, HDC-based HIDS can fully utilize the memory using a straightforward layout because the provenance graph is essentially a set of trees. Therefore, we can keep a single instance of each node HVs (hop $==$ 0) or path HV (hop $>$ 0). To further improve the performance, we allocate different trees onto different memory blocks, maximizing the parallelism of in-memory HV bundling. We can alternatively use the memory for current path HVs and next path HVs when increasing the hops to save memory consumption. Finally, we bundle all final path HVs and calculate the maximum similarity score by comparing to all encoded attack patterns in FeFET CAM.

V. Evaluation

To evaluate our system, we compare our results against the Sysflow baseline query engine, which performs standard backward-tracing. For the CPU-based evaluation, we used a server equipped with an Intel Core i7-8700K processor and 64GB memory. Our experiments seek to answer the following research questions: *(1) What is the latency and power consumption of an HDC approach? (2) How accurate is our HDC approach at detecting threats?*

A. Experiment Setup

1) Dataset: A known limitation of intrusion detection systems is the lack of complete public datasets [4]. Many works, in response, choose to generate their own datasets for analysis [17]–[21]. However, due to privacy concerns these datasets are never published publicly, making it difficult to reproduce and compare results. With this in mind, we evaluate our system using a set of system traces made public by the Sysflow project that were collected from various research events [22]. This dataset includes 10 benign traces and 18 malicious traces with attacks from 15 known MITRE TTPs [3]. As mentioned in section III-A, a single TTP may represent multiple subgraphs. We generated a signature hypervector by first encoding every subgraphs and then evaluating an average of the encodings.

2) Simulation: We use NeuroSim [23] to model a FeFET memory with 32 clusters, each consisting of 2,048 memory blocks. Each memory block has 64 rows and 64 columns of 3-bit cells (i.e., 8 levels). We use the latency and energy consumption of various in-memory operations from NeuroSim in our in-house cycle-based simulation to evaluate the behaviors of the PIM accelerator.

B. Results

1) Performance Results: In Figure 5, we present a comparison of latency between our method and the baseline. We set the hop-length for our frame work as 3. Figure 5a shows the distribution of per-TTP latency. This shows the latency distribution involved in our graph encoding, detection with both CPU and PIM experiment and the baseline backward tracing for a single query of a single TTP. Since our detection is based on a threshold cosine similarity value, encoding consumes the majority of latency. Nevertheless, our encoding method is significantly faster than the baseline, showing speedups from 3 to $64\times$ on CPU and from $2,400$ to $18,000\times$ with PIM. Consequently, a security analyst can perform up to $18,000\times$ more queries within the same timespan using our method.

An additional advantage of our method is the ability to encode a data point just once and reuse it for multiple detections. This is because the detection only needs the encoded hypervector, not the original graph data. Therefore, although the encoding function is the slower operation in our method, the latency can be effectively amortized over time as the hypervector is reused for detecting multiple TTPs. This latency enhancement was achieved largely through efficient reuse of node encodings. This is demonstrated in Figure 5b, which compares the latency of detecting 15 TTPs using our method against the baseline. Our approach achieves 14 to $4,242\times$ speedup on CPU. Additionally, our PIM acceleration shows 4-6 orders of magnitude faster results.

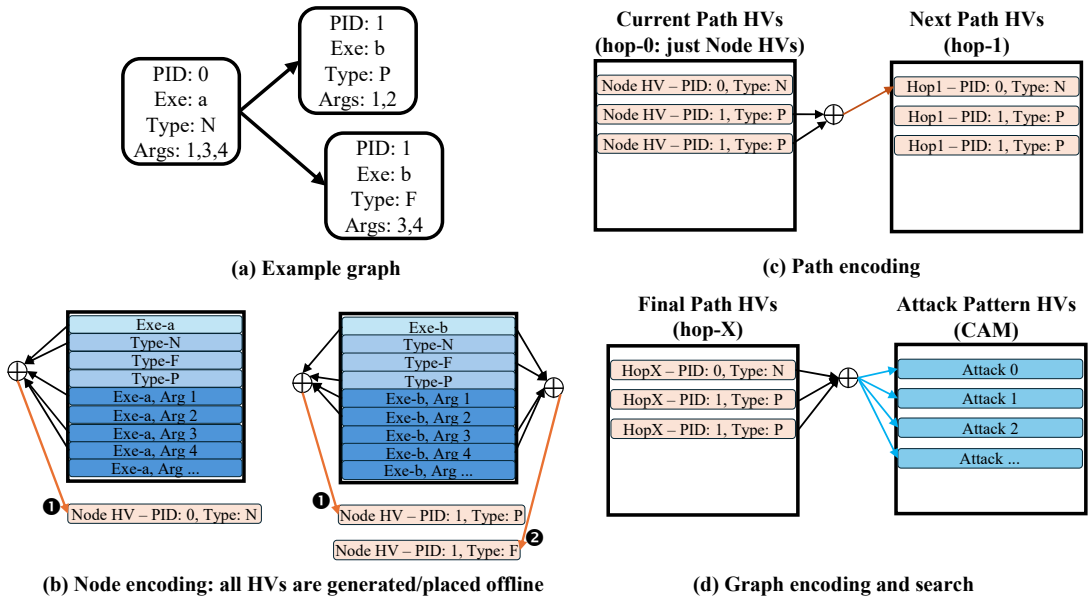
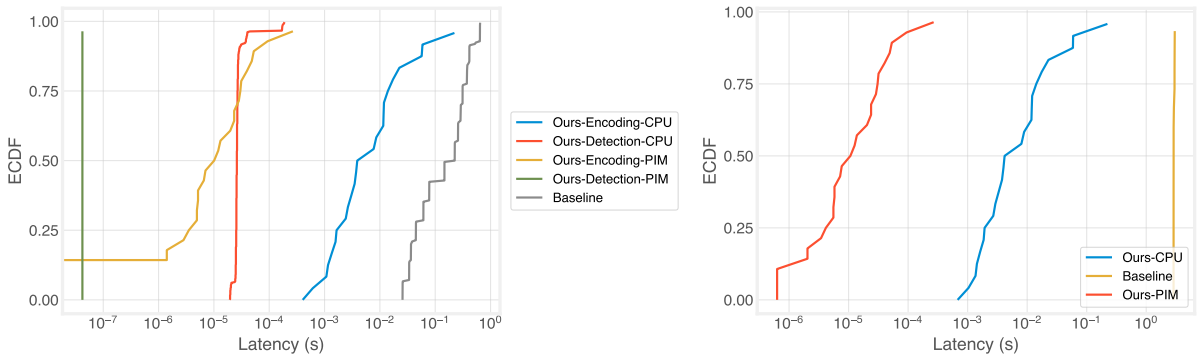


Fig. 4: Data layout and processing flow of HIDS on PIM.



(a) Per-TTP Latency

(b) Total Latency

Fig. 5: ECDF of query latency.

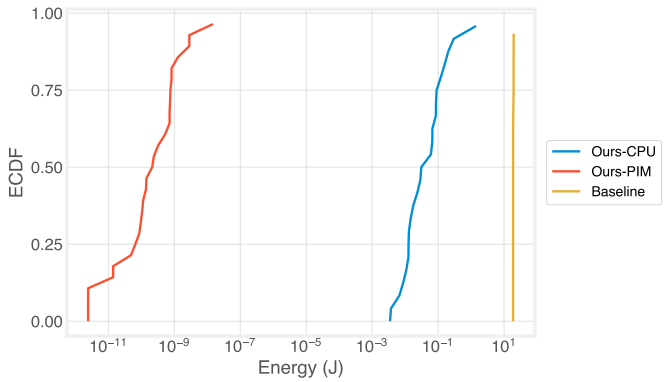


Fig. 6: ECDF of energy consumption.

Figure 6 illustrates the energy consumption distribution measured during single query and 15 TTPs detection experiments. Same as the latency result, our results includes energy consumption during encoding and 15 detections. Our framework surpasses the baseline in terms of energy efficiency, whether considering CPU or PIM acceleration. Specifically, our CPU results show 14 to $5,300\times$ lower energy consumption. Moreover, our PIM architecture demonstrates remarkable energy efficiency compared to the CPU baseline, consuming 9-12 orders of magnitude less energy.

2) Accuracy Results: To evaluate accuracy, we compute the cosine similarity using different hop lengths ranging from 0-3. The hop length l represents, for each signature, a subgraph comprised of all paths of length l. For example, a hop length of 1 would define a signature as a subgraph containing any two-node paths starting from node n. The larger the hop count, the more granular the signature becomes (i.e., the more descendants are included in the signature). We note that SysFlow already provides semantic compression of high-level behaviors (e.g., reading or writing to the same file is merged

	Hop-Length			
TTP	0	1	2	3
T1020	1	1	1	1
T1033	0.952	0.984	1	1
T1059.004	1	1	1	0.979
T1068	0.926	0.916	0.853	0.832
T1069.001	1	0.783	0.957	1
T1072	1	1	1	1
T1082	0.828	0.898	0.891	0.891
T1083	0.955	1	1	1
T1087	1	1	1	1
T1087.001	0.87	0.87	0.87	0.898
T1105	0.926	0.726	0.768	0.758
T1106	0.886	0.757	0.771	0.786
T1222.002	0.952	0.984	0.984	0.937
T1552.003	0.825	0.794	0.81	0.825
T1574	0.886	0.757	0.771	0.786
Average	0.934	0.898	0.912	0.913

TABLE I: ROC-AUC scores for each TTP and hop length.

into a single node).

As a result of this, attack patterns that can be specified with a smaller subgraph will naturally be able to be detected with smaller hop counts, and vice versa. This is reflected in our results in Table I. For example, the signature for T1059.004 is characterized by a depth of 2, and we observe an accuracy of 100% at a hop count of 2. We note that some TTPs like T1106 are more complex to express (i.e., have a higher depth with more complex interactions) and certain features have more significance than others. We are exploring alternative feature encodings to be able to better detect these.

Overall, our results have an average accuracy $> 90\%$ across all TTPs and hop lengths, demonstrating that HDC-based querying can provide substantial query latency speedups with high accuracy.

VI. Conclusion

We proposed a novel HDC-based HIDS query engine that provides the capability to query provenance graphs for the presence of attack signatures. Our framework first generates TTP signatures from a dataset of malicious system call traces. Then, we leverage HDC in the context of HIDS for the first time, introducing an encoding method to map provenance graphs and signatures into hyper vectors. We implement and evaluate our HDC method on a state-of-the-art accelerator to demonstrate the power cost savings. Our evaluation shows a detection accuracy of $> 90\%$, and up to a $4,242\times$ latency improvement on CPU and up to $18,000\times$ improvement on a hardware accelerator. Also, our method consumes up to $5,300\times$ and 12 orders of magnitude less energy than baseline with CPU and PIM accelerator, respectively.

Acknowledgment

This work was supported in part by PRISM and CoCoSys, centers in JUMP 2.0, an SRC program sponsored by DARPA.

References

[1] B. Nour, M. Pourzandi, and M. Debbabi, "A survey on threat hunting in enterprise networks," *IEEE Communications Surveys & Tutorials*, vol. 25, no. 4, pp. 2299–2324, 2023.

[2] X. Shu *et al.*, "Threat intelligence computing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. Toronto, Canada: ACM, 2018, pp. 1883–1898.

[3] MITRE, "Matrix - Enterprise | MITRE ATT&CK®," 2021. [Online]. Available: https://attack.mitre.org/matrices/enterprise/linux/

[4] M. Zipperle *et al.*, "Provenance-based Intrusion Detection Systems: A Survey," *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–36, Jul. 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3539605

[5] M. Liu *et al.*, "Host-based intrusion detection system with system calls: Review and future trends," *ACM computing surveys (CSUR)*, vol. 51, no. 5, pp. 1–36, 2018.

[6] T. Taylor *et al.*, "Towards an Open Format for Scalable System Telemetry," in *2020 IEEE International Conference on Big Data (Big Data)*. Atlanta, GA: IEEE, Dec. 2020, pp. 1031–1040. [Online]. Available: https://ieeexplore.ieee.org/document/9378294/

[7] J. Kang *et al.*, "Relhd: A graph-based learning on fefet with hyperdimensional computing," in *2022 IEEE 40th International Conference on Computer Design (ICCD)*, IEEE. Lake Tahoe, CA: IEEE, 2022, pp. 553–560.

[8] S. Northcutt and J. Novak, *Network intrusion detection*. Sams Publishing, 2002.

[9] M. N. Hossain *et al.*, "{SLEUTH}: Real-time attack scenario reconstruction from {COTS} audit data," in *26th USENIX Security Symposium*. Vancouver, BC, Canada: USENIX Association, 2017, pp. 487–504.

[10] W. U. Hassan *et al.*, "Tactical provenance analysis for endpoint detection and response systems," in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE. San Francisco, CA: IEEE, 2020, pp. 1172–1189.

[11] ——, "This is why we can't cache nice things: Lightning-fast threat hunting using suspicion-based hierarchical storage," in *Proceedings of the 36th Annual Computer Security Applications Conference*. Boston, MA: USENIX Association, 2020, pp. 165–178.

[12] M. Imani *et al.*, "Revisiting hyperdimensional learning for fpga and low-power architectures," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Seoul, South Korea: IEEE, 2021, pp. 221–234.

[13] I. Nunes, M. Heddes, T. Givargis, A. Nicolau, and A. Veidenbaum, "Graphhd: efficient graph classification using hyperdimensional computing," in *Proceedings of the 2022 Conference & Exhibition on Design, Automation & Test in Europe*, ser. DATE '22. Leuven, BEL: European Design and Automation Association, 2022, p. 1485–1490.

[14] J. Wang, H. Xu, Y. G. Achamyeleh, S. Huang, and M. A. A. Faruque, "Hyperdetect: A real-time hyperdimensional solution for intrusion detection in iot networks," *IEEE Internet of Things Journal*, vol. 11, no. 8, pp. 14 844–14 856, 2024.

[15] K. Alsubhi, "A Secured Intrusion Detection System for Mobile as Edge Computing," *Applied Sciences*, vol. 14, no. 4, p. 1432, Jan. 2024. [Online]. Available: https://papers.ssrn.com/abstract=4699806

[16] P. Singh *et al.*, "Edge-Detect: Edge-centric Network Intrusion Detection using Deep Neural Network," in *2021 IEEE CCNC*. Virtual: IEEE, Jan. 2021, pp. 1–6. [Online]. Available: http://arxiv.org/abs/2102.01873

[17] W. U. Hassan *et al.*, "NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage," in *NDSS 2019*. San Diego, CA: Internet Society, 2019.

[18] P. Gao *et al.*, "AIQL: Enabling efficient attack investigation from system monitoring data," in *2018 USENIX Annual Technical Conference*. Boston, MA: USENIX Association, Jul. 2018, pp. 113–126.

[19] Q. Wang, W. U. Hassan, D. Li, K. Jee, X. Yu, K. Zou, J. Rhee, Z. Chen, W. Cheng, C. A. Gunter, and H. Chen, "You Are What You Do: Hunting Stealthy Malware via Data Provenance Analysis," in *Proceedings 2020 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2020/02/24167.pdf

[20] X. Han *et al.*, "{SIGL}: Securing software installations through deep graph learning," in *30th USENIX Security Symposium*. Virtual: USENIX Association, 2021, pp. 2345–2362.

[21] Y. Xie, D. Feng, Z. Tan, and J. Zhou, "Unifying intrusion detection and forensic analysis via provenance awareness," *Future Generation Computer Systems*, vol. 61, pp. 26–36, 2016.

[22] IBM, "sysflow-telemetry/sf-lab," Feb. 2024, original-date: 2022-11-18T22:14:27Z. [Online]. Available: https://github.com/sysflow-telemetry/sf-lab

[23] X. Peng *et al.*, "Dnn+neurosim: An end-to-end benchmarking framework for compute-in-memory accelerators with versatile device technologies," in *2019 International Electron Devices Meeting (IEDM)*. San Francisco, CA: IEEE, 2019, pp. 32–5.