HDDroid: Federated Hyperdimensional Computing for Mobile Malware Detection

Article

Accepted version

A. Augello, A. De Paola, G. Lo Re, A. Menager

Proceedings of the Joint National Conference on Cybersecurity (ITASEC & SERICS 2025) - CEUR WORKSHOP PROCEEDINGS

It is advisable to refer to the publisher's version if you intend to cite from the work.

Publisher: CEUR-WS

HDDroid: Federated Hyperdimensional Computing for Mobile Malware Detection

Andrea Augello[1], Alessandra De Paola[1], Giuseppe Lo Re[1], and Antoine Menager[2]

[1] University of Palermo, Italy {andrea.augello01, alessandra.depaola, giuseppe.lore}@unipa.it

[2] Polytech Dijon, France antoine_menager@etu.u-bourgogne.fr

DRAFT

Abstract

Mobile malware poses significant security and privacy risks, hence effective detection methods are crucial. Graph-based representations of mobile applications have been shown to be well-suited for this task. However, traditional graph-based machine learning techniques are computationally expensive and unsuitable for on-device analysis. Nevertheless, off-device analysis raises privacy concerns, making on-device analysis combined with decentralized learning approaches like Federated Learning (FL) an attractive alternative. Hyperdimensional Computing (HDC) offers efficient graph classification on resource-constrained mobile devices. This work introduces HDDroid, an FL framework leveraging HDC to detect malicious software via function call graph analysis. HDDroid's novel online encoding strategy reduces memory usage, enabling large graph analysis on mobile devices. Additionally, HDDroid's improved model aggregation strategy enhances model robustness and classification accuracy, achieving state-of-the-art performance in distributed learning scenarios.

1 Introduction

The widespread diffusion of mobile devices, and the consequent increase in the number of applications available on those devices, has led mobile malware to become a significant threat to the security and privacy of users [1], thus making effective detection methods crucial to limit these risks. Raw features like bytecodes, opcodes, strings, permissions, and APIs are easy to extract and can be used to identify malware [2]. However, these low-level features are vulnerable to code obfuscation, considered that malware developers alter code features while maintaining functionality [3]. Higher-level structured features, such as multimodal features, API call reachability analysis, and function call graphs, are preferred as they are harder to modify while preserving the program's semantics [4]. Graph-based representations in particular effectively capture the semantics of the code and can be extracted by lightweight static analysis [4].

An additional challenge in mobile malware detection is the privacy of the users: a simple snapshot of the installed applications on a device is sufficient to infer sensitive information such as the users' location, interests, religious beliefs, marital status, medical conditions, and habits [5]. Hence, to preserve the privacy of the users, data should never leave their devices. Although graph-based representations can be obtained easily without requiring off-device dynamic analysis, traditional machine learning techniques for graph classification are computationally expensive and require multiple passes through the data. Thus, mobile devices might struggle to perform the necessary computations for on-device analysis, and real-time detection would not be feasible. Hyperdimensional Computing (HDC) is a brain-inspired computing paradigm that is recently gaining popularity in the machine learning community [6]. This paradigm is well suited for devices with limited energy and computational resources such as mobile phones and represents a promising alternative for on-device training and inference [6].

This work introduces HDDroid, a federated learning framework for mobile malware detection to efficiently and effectively detect malicious applications through the analysis of function call graphs. The main contributions of this work can be summarized as follows:

- HDDroid enables on-device training and analysis leveraging Hyperdimensional Computing;
- through online encoding, HDDroid reduces the memory requirements for function call graphs encoding, enabling the analysis of large graphs on resource-constrained devices;
- to integrate the contributions of multiple users, HDDroid employs a novel aggregation strategy for distributed learning that filters out irrelevant information and increases the classification accuracy.

Experimental results on the comprehensive real-world publicly available MalNet [7] dataset show that HDDroid achieves state-of-the-art performance in mobile malware detection, and maintains a high accuracy in distributed learning scenarios. The remainder of this paper is organized as follows. Section 2 reviews related works. Section 3 provides preliminary information on the adopted computing paradigms. Section 4 details the HDDroid architecture. Section 5 presents the experimental evaluation. Finally, Section 6 concludes the paper.

2 Related Works

The field of malware detection has been the subject of extensive research. The adoption of Machine Learning methods promises to be the best approach to enable the design of solutions that can automatically learn how to detect these types of threats [8]. Early automated malware detection systems detected malware by computing a signature summarizing the characteristics of the software and comparing it to known malware signatures. The signature of a software is constituted by a unique string of bytes generated through an analysis of the code. Several approaches to generate effective software signatures for malware detection have been proposed [9]. For instance, early approaches used a digest of the file to check if an executable is a known malware. More fine-grained approaches statically analyze the execution flow of the program trying to match sequences of operation in the code with known malware patterns. Signature-based approaches are fast and effective in detecting known malware but they are not reliable for detecting unknown malware types and can be eluded by polymorphic malware. Additionally, signatures need to be crafted through careful feature engineering, limiting versatility. A more flexible approach compared to signature-based detection is behavior-based classification. These techniques analyze the behavior of the software during its execution by monitoring system calls, file changes or network activity and build a feature vector that is then used to classify the software. This approach requires sandbox execution, which is computationally expensive for on-device analysis [10]. Thus, static analysis is more suitable for on-device feature extraction.

However, detection through low-level static features such as APIs, strings, permissions, and opcodes can be eluded through code obfuscation [4] since malware developers can alter the code to change its features while maintaining its functionality. Therefore, higher-level structured features are preferred. For instance, graph representations of software are robust to obfuscation [4] while being relatively easy to acquire from the code with a static analysis without requiring computationally expensive dynamic analysis [11]. Among the most used graph representations for malware analysis are: a) Control Flow Graphs [12] that model paths between blocks of code during the program execution; b) Function call and API dependencies graphs [13], which are Control Flow Graphs with the functions/API calls granularity; c) System call graphs [14] that model interactions between the program and the underlying operating system; d) System entity graphs [15] that model interactions between the program and the system entities such as files, network connections, and processes. Function call graphs extracted

DRAFT

through a static analysis of the API calls are the most popular choice for Android malware detection [13].

Traditional approaches to represent graphs for learning tasks often rely on hand-crafted summary statistics to encode the graph structure into feature vectors. For example, LDP [16] uses histograms of node degrees and their 1-hop neighbors, while NoG [17] focuses on node and edge attributes, ignoring topology. Other methods use random walks to describe node neighbors [18]. Graph kernels, such as Slaq-LSD and Slaq-VNGE [19], measure approximate distances between spectral graph representations. Recent works have focused on learning graph representations directly from raw graph data using deep learning models. Graph Neural Networks (GNNs) propagate information between neighboring nodes [20], learn the optimal aggregation function [21], and extending convolution operations to graph data [22]. While effective, these methods are computationally expensive and unsuitable for on-device analysis in constrained devices like mobile phones. A recent alternative for graph embedding is Hyperdimensional Computing, specifically the GraphHD algorithm [23]. Rather than learning a representation of the nodes, GraphHD assigns random high-dimensional vectors to nodes and combines them to represent edges and the entire graph. This approach is suitable for on-device analysis, as it is more computationally efficient with shorter training times compared to traditional GNNs.

3 Preliminaries

3.1 Federated Learning

Federated Learning (FL) is a decentralized machine learning approach that lets multiple independent data owners to contribute to a shared model without disclosing their data [24]. In FL, each participating client (data owner) has a local, private, dataset. A central server distributes a copy of a classification model to all clients. Each client trains the model on its local data for a set number of iterations. The clients then send the updated models to the server, which aggregates them into a global model through a weighted average. The new model is then redistributed to the clients for further refining, repeating this process for multiple rounds until model convergence. FL preserves the privacy of the clients' data, sharing only model parameters and keeping the data private, making it an attractive alternative to centralized learning for malware detection tasks [25].

DRAFT

3.2 Hyperdimensional Computing

Hyperdimensional computing (HDC), described in-depth in [26], is a brain-inspired computing paradigm that leverages high-dimensional vectors to perform computations. HDC performs classification tasks by representing data as hypervectors (HVs), whose dimensionality is typically in the order of tens of thousands, and associating these with class labels. An hypervector $\mathcal{H}$ is defined as $\langle h_1, h_2, \ldots, h_D \rangle$ where h_i is the value of the i-th dimension of the vector. The size of the vectors D is consistent throughout a HDC system: operations act on HVs of homogeneous length and do not modify their dimension. HDC is scalable, parallelizable, energy-efficient, and well-suited for few-shot learning tasks, requiring less training and inference time than neural networks of comparable accuracy [6].

Operations A useful property of random vectors in high-dimensional spaces is their quasi-orthogonality [26] (their cosine similarity is close to 0), which allows multiple HVs to be combined

through simple mathematical operations while maintaining the information encoded in each HV. Three major operations are supported by HVs and used in HDC:

- **Permutation** (Π) is an unary operation which rearranges the elements of the vector into a HV quasi-orthogonal to the original one. This operation is often used together with the binding operations to assign an order between HVs, and it typically implemented through a circular shift, moving all the coordinates of the HV clockwise:
$$\Pi(\mathcal{H}) = \langle h_D, h_1, \ldots, h_{D-1} \rangle.$$
- **Binding** ($\circledast$) two HVs yields an HV dissimilar to both. Binding associates information from two different HVs (e.g., assigning values to variables) through element-wise multiplication:
$$\mathcal{H}_1 \circledast \mathcal{H}_2 = \langle h_{1,1} \times h_{2,1}, \ldots, h_{1,D} \times h_{2,D} \rangle.$$
- **Bundling** ($\oplus$) combines two HVs into a new hypervector through element-wise addition, obtaining a HV similar to both the input HVs:
$$\mathcal{H}_1 \oplus \mathcal{H}_2 = \langle h_{1,1} + h_{2,1}, \ldots, h_{1,D} + h_{2,D} \rangle.$$

The high dimensionality allows these operations produce distinct and non-conflicting HVs. The operators act on HV elements independently and do not require loading the entire HV into memory, making them highly parallelizable and suitable for analysis on both resource-constrained devices such as low-end mobile phones and devices capable of parallel processing.

Training and classification In HDC, a classification task with k classes is performed by computing a class prototype HV for each class $\mathcal{C}_1, \ldots, \mathcal{C}_k$, storing them into an associative memory $\mathcal{M}$, and then comparing query HVs with the class prototypes to determine the predicted class.

Class HVs can be computed by bundling the HVs in the training set belonging to the corresponding class. Single-pass training is simple and efficient, but often leads to low accuracy, thus iterative training algorithms such as RefineHD [27] and OnlineHD [28] have been proposed to improve the classification performance. These algorithms bundle incorrectly classified query HVs to the correct class HV and subtract them from the wrongly predicted class HV.

At inference time, the similarities of a query hypervector $\mathcal{H}_{Query}$ and the class hypervectors in the associative memory $\mathcal{M}$ are computed. The class whose HV has the highest cosine similarity with the query is considered to be the predicted label:

$$pred(\mathcal{M}, \mathcal{H}_{Query}) = \arg\max_i \quad \delta(\mathcal{C}_i, \mathcal{H}_{Query}) = \arg\max_i \quad \frac{\mathcal{C}_i \cdot \mathcal{H}_{Query}}{||\mathcal{C}_i|| \times ||\mathcal{H}_{Query}||}.$$

4 HDDroid architecture

HDDroid leverages a collaborative learning framework to allow multiple independent data owners to contribute to a shared mobile malware detection model without disclosing their data. The system includes n clients, each with its own private dataset, and a central server that coordinates training. Each client uses its local data to train a hyperdimensional associative memory $\mathcal{M}^{(i)}$ for classification and sends it to the central server. The server aggregates the classifiers into a global classifier $\mathcal{M}^{(G)}$ and sends it back to the clients for further refinement over several iterations. A schematic representation of the HDDroid architecture is shown in Figure 1.

4.1 Mobile application encoding

HDDroid analyzes mobile applications using function call graphs obtained through static code analysis. Each function is a graph node with a unique identifier, and directed edges represent function calls, indicating possible execution paths. Hyperdimensional Computing directly encodes function call graphs of mobile applications into HVs for classification, without the need

DRAFT

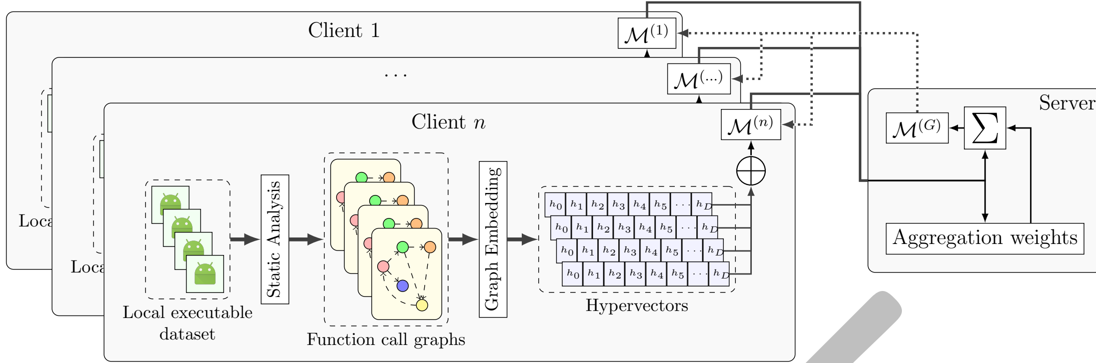

Figure 1: HDDroid architecture. Each client has a private dataset containing multiple mobile applications. Static analysis extracts function call graphs, which are then encoded as hypervectors. The HVs are used to train a local associative memory for classification which is sent to the central server for aggregation into a global classifier to be distributed back to the clients.

for feature engineering. The encoding process is hierarchical: vertices are encoded first, then they are combined into edges HVs, and edges are bundled to encode the entire graph.

Node encoding The first step in the encoding process is to assign a unique hypervector to each graph node. Since the function identifiers in an application are arbitrary, they cannot be encoded directly into HVs. To ensure consistent encoding across graphs, each node is labeled with its PageRank centrality [29], as proposed in [23]. PageRank measures a node's importance based on the stationary distribution of a random walk on the graph. Once the nodes receive a meaningful identifier consistent across different graphs, each identifier is mapped to an HV.

A common approach to encode discrete values into HVs is to precompute a set of random HVs, one for each possible value. However, this requires an HV for each value, causing the random basis size to grow linearly with the number of values, which can be impractical. For application call graphs, the number of possible function calls is unbounded, with some graphs containing almost 600 thousand nodes [7]. Encoding all nodes in HVs with dimensionality 10000 would require over 40 GB of memory, making it infeasible for resource-constrained mobile devices. Additionally, a fixed random base requires knowing the range of values to be encoded, which is impractical for function call graphs due to the arbitrary number of function calls in mobile applications. Thus, using a random basis is not viable in this target scenario.

Other approaches, such as Flocet encoding [27] and random projections [30], embed discrete values into HVs without precomputing a random basis. However, these methods preserve similarity between original values, which is undesirable here as consecutively ranked nodes correspond to different functions. To address these issues, HDDroid uses an online encoding strategy based on the Philox algorithm [31], a counter-based pseudo-random number generator. A digest is computed from the node label to seed the key and counter for Philox, which generates an arbitrary-length sequence of pseudo-random values as the HV encoding. This deterministic procedure ensures the same label is always encoded as the same hypervector, without storing a lookup table, and guarantees quasi-orthogonal HVs. Additionally, Philox can generate the values in parallel, making it suitable for encoding large graphs on multi-core devices.

DRAFT

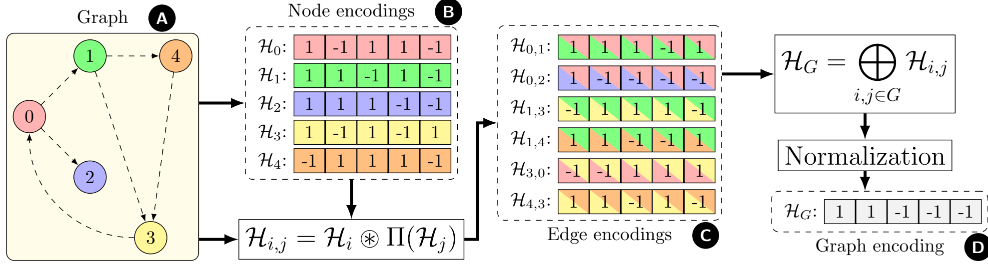

Figure 2: Graph embedding into a hypervector: nodes are numbered (a) and assigned unique pseudo-random HVs (b), edges are encoded by binding node hypervectors with a permutation on the destination node (c), and all edges are bundled to form the final graph hypervector (d).

Hypervector-based graph representation Given the HVs for all the nodes, they can be combined to compute the HV for the whole graph [23] as illustrated in Figure 2. Each edge (a, b) in the graph is encoded by binding (element-wise multiplication) the hypervectors $\mathcal{H}_a$ and $\mathcal{H}_b$ of the two nodes a and b, with the destination node HV permuted to indicate the direction of the edge. All the edges are then bundled (element-wise addition) into the graph hypervector $\mathcal{H}_{\text{graph}}$ as per Eq. 1. Each element in the graph HV is thus given by the sum, for each pair of connected nodes, of the products of the corresponding elements in the node HVs, with the elements in the arrival node HV rotated by one position:

$$\mathcal{H}_{\text{graph}} = \bigoplus_{(a,b)\in\text{edges}} \mathcal{H}_a \circledast \Pi(\mathcal{H}_b) = \langle \sum_{(a,b)\in\text{edges}} h_{a,1} \times h_{b,D}, \sum_{(a,b)\in\text{edges}} h_{a,2} \times h_{b,1}, \ldots, \sum_{(a,b)\in\text{edges}} h_{a,D} \times h_{b,D-1} \rangle. \quad (1)$$

Since all node HVs have the same dimensionality, the graph HV will have a fixed size regardless of the number of function calls. However, graphs with more edges will sum more quasi-orthogonal edge HVs, potentially increasing the vector magnitude. To ensure fair influence on the final classifier, graph hypervectors are normalized to have the same magnitude, regardless of the application size. The complete encoding algorithm is reported in Algorithm 1. At any given step of the algorithm, the server only needs to record at most three hypervectors: $\mathcal{H}_{graph}$, $\mathcal{H}_a$ and either $\mathcal{H}_b$ or $\mathcal{H}_{a,b}$. Excluding the PageRank computation, the memory requirement of the graph encoding algorithm is $O(D)$, independent of the graph size, while the time complexity is $O(|V| \cdot D)$. The graph information is spread across the HV, no HV element stores specific information about a node or an edge, granting robustness to noise and errors.

4.2 Few-shot collaborative learning

After encoding the function call graphs into HVs, each client computes a local associative memory $\mathcal{M}^{(i)}$ for classification. Through federated learning, clients share their local classifiers with the central server, which aggregates them into a global classifier $\mathcal{M}^{(G)}$. This process is repeated for several rounds, refining the global classifier. The HDC paradigm enables efficient distributed learning with minimal training iterations and communication cycles.

Local training As discussed in Section 3, the local training of the classifier is often performed by computing the class hypervectors through one-pass training. However, bundling all HVs of the same class can lead to model saturation, as common patterns overshadow specific characteristics crucial for classification. This is particularly problematic in mobile application classification,

DRAFT

Algorithm 1 Application Function Call Graph Encoding Algorithm

Require: Application, Hypervector size D
Ensure: Graph hypervector $\mathcal{H}_{\text{graph}}$
1: Graph $G = (V, E) = \text{CALL_GRAPH}(\text{Application})$
2: Initialize $\mathcal{H}_{\text{graph}}$ as an empty hypervector
3: $\mathbf{r} \leftarrow \{\text{PAGERANK}(G, a) \mid a \in V\}$ {*Compute unique ID for each node*}
4: **for** $a \in V$ **do**
5: $digest_a \leftarrow \text{HASH}(\mathbf{r}_a)$
6: $Key_a, Counter_a \leftarrow \text{LOWER_BITS}(digest_a), \text{UPPER_BITS}(digest_a)$
7: $\mathcal{H}_a \leftarrow \text{PHILOX}(Key_a, Counter_a, D)$ {*Generate start node HV*}
8: **for** $b \in V \mid (a, b) \in E$ **do**
9: $digest_b \leftarrow \text{HASH}(\mathbf{r}_b)$
10: $Key_b, Counter_b \leftarrow \text{LOWER_BITS}(digest_b), \text{UPPER_BITS}(digest_b)$
11: $\mathcal{H}_b \leftarrow \text{PHILOX}(Key_b, Counter_b, D)$ {*Generate destination node HV*}
12: $\mathcal{H}_{a,b} \leftarrow \mathcal{H}_a \circledast \Pi(\mathcal{H}_b)$
13: $\mathcal{H}_{\text{graph}} \leftarrow \mathcal{H}_{\text{graph}} \oplus \mathcal{H}_{a,b}$ {*Update graph HV with new edge*}
14: **return** $\frac{\mathcal{H}_{\text{graph}}}{|\mathcal{H}_{\text{graph}}|}$

where many apps share functionalities, and malware can mimic benign apps to elude detection. Instead of bundling all HVs of the same class, HDDroid adopts the weighted aggregation strategy in [27]. Each sample $\mathcal{H}$ is compared to the local associative memory. If the class hypervector most similar to $\mathcal{H}$ is not the one for its class $\mathcal{C}_{label}$, a weighted portion of $\mathcal{H}$ is added to $\mathcal{C}_{label}$ and subtracted from the most similar class hypervector $\mathcal{C}_{pred}$ to increase and decrease their similarity to $\mathcal{H}$, respectively. The weights depend on the similarity of the HV with the predicted and true label classes, δ_{pred} and δ_{label}, as per Eq. (2):

$$\begin{aligned}\mathcal{C}_{label} &= \mathcal{C}_{label} + (1 - \delta_{label})(1 - \delta_{label} - \delta_{pred})\mathcal{H} \\ \mathcal{C}_{pred} &= \mathcal{C}_{pred} - (1 - \delta_{pred})(1 - \delta_{label} - \delta_{pred})\mathcal{H}.\end{aligned} \tag{2}$$

When the sample has a high similarity with the correct class, only a small portion of the hypervector is added to the class hypervector. When the difference is significant, a larger portion of the sample hypervector is added to the correct class hypervector and subtracted from the incorrect one to better incorporate the sample's information. HVs with correct, but low confidence, predictions are still added to the correct class hypervector. In this case, the weight of the hypervector is computed as $\mathcal{C}_{label} = \mathcal{C}_{label} + (1 - \delta_{label} - \delta_{pred})\mathcal{H}$. A prediction is considered to have low confidence if the similarity with the correct class hypervector is below the average similarity of the wrongly classified samples with the correct class hypervector.

Model aggregation Once clients update their associative memories, they send the class HVs to the central server for aggregation into a global classifier [24]. This process leverages information from all clients, creating a more accurate model while reducing communication overhead and preserving client privacy compared to transmitting the raw HVs. Typically, HDC aggregates local classifiers by averaging class HVs from each client, similar to traditional federated learning. A representative example is Fed-HD [32], which computes the global classifier as $\mathcal{M}^{(G)} = \frac{\mathcal{M}^{(G)} + \sum_{i=1}^{n} \mathcal{M}^{(i)}}{n+1}$.

However, this aggregation process can lead to model saturation by repeatedly adding similar information from all clients. HDDroid weights class HVs from each client by the distinctiveness of their information, avoiding redundancy and improving learning outcomes. Given the class

DRAFT

c hypervector from client i, $\mathcal{C}_c^{(i)}$, the server evaluates the *intra-class redundancy* $I(c,i)$, which measures the average similarity of $\mathcal{C}_c^{(i)}$ with the hypervectors of other clients for the same class:

$$I(c,i) = \frac{1}{n}\sum_{j=1}^{n} \delta(\mathcal{C}_c^{(i)}, \mathcal{C}_c^{(j)}).$$

High intra-class redundancy means $\mathcal{C}_c^{(i)}$ is similar to other clients' hypervectors for the same class, adding little new information. The server also evaluates the *inter-class confusion* $U(c,i)$ of client i for class c by comparing $\mathcal{C}_c^{(i)}$ to hypervectors from other classes:

$$U(c,i) = \frac{1}{n}\sum_{j=1}^{n} \max \delta(\mathcal{C}_c^{(i)}, \mathcal{M}^{(j)} \setminus \mathcal{C}_c^{(j)}),$$

where $\mathcal{M}^{(j)} \setminus \mathcal{C}_c^{(j)}$ is the set of class HVs in the associative memory of client j excluding $\mathcal{C}_c^{(j)}$. High inter-class confusion indicates that $\mathcal{C}_c^{(i)}$ contains information not specific to class c.

The computation of U and I for all class hypervectors has a complexity of $O(n^2 \cdot k^2 \cdot D)$, where n is the number of clients, k is the number of classes, and D is the hypervector dimensionality. Since k and D are hyperparameters, the aggregation process scales quadratically with the number of clients. These computations are independent and can be parallelized for efficiency.

The normalized aggregation weight $\omega_c^{(i)}$ for the hypervector $\mathcal{C}_c^{(i)}$ is computed using the intra-class redundancy I and inter-class confusion U as per Eq. (3):

$$\omega_c^{(i)} = \frac{(1-I(c,i))(1-I(c,i)-U(c,i))}{\sum_{j=1}^{n}\sum_{l=1}^{k}(1-I(l,j))(1-I(l,j)-U(l,j))}. \tag{3}$$

A hypervector similar to others in the same class or to those of other classes is considered redundant or uninformative, respectively, and its weight is decreased. The class HVs' weights are used by the server to compute the updated global memory as:

$$\mathcal{M}^{(G)} = \left(\sum_{i=1}^{n} \omega_1^{(i)}\mathcal{C}_1^{(i)}, \ldots, \sum_{i=1}^{n} \omega_k^{(i)}\mathcal{C}_k^{(i)}\right).$$

Unlike traditional FL on neural networks, which requires many communication rounds to achieve good performance [33], HDDroid is experimentally shown to converge in few rounds.

5 Experimental evaluation

To evaluate HDDroid, experiments were conducted on the MalNet tiny 5K [7] and MalNet 61K [34] datasets, containing function call graphs of Android applications labeled as benign or malware. The MalNet tiny 5K dataset contains 5000 applications, each with up to 5000 nodes per graph. The larger 61K dataset includes 87430 applications, with graphs containing up to 599234 nodes. The 61K dataset's size and complexity make it a better testbed for evaluating HDDroid's scalability and encoding effectiveness. Most experiments were conducted on the 61K dataset, with the 5K dataset used for comparison with existing methods. In order to simulate a large number of devices, the experiments were conducted on a server with 32 Intel(R) Xeon(R) Gold 6242 CPU cores, 32 GB of RAM, and an NVIDIA A100 GPU. The HDDroid model was implemented using the TorchHD library [35].

Comparison with state-of-the-art approaches: First HDDroid is compared with other non-HDC existing approaches for graph classification in a centralized setting. Table 1 and Table 2 show the performance of HDDroid alongside state-of-the-art methods reported in the literature on the MalNet tiny 5K and MalNet 61K datasets, respectively.

DRAFT

Method	SIR-GN [34]	Feather [18]	LDP [16]	GIN [21]	GCN [22]	Slaq-LSD [19]	NoG [17]	Slaq-VNGE [19]	HDDroid
Accuracy	0.92	0.86	0.86	0.90	0.81	0.76	0.77	0.53	0.90

Table 1: Comparison of HDDroid on the MalNet tiny 5K dataset against several other graph-based methods reported in [7] and [34].

Method	**F1**			**Precision**			**Recall**		
	SIR-GN	ResNet18	HDDroid	SIR-GN	ResNet18	HDDroid	SIR-GN	ResNet18	HDDroid
	0.718	0.651	0.936	0.729	0.672	0.969	0.646	0.646	0.904

Table 2: Comparison of HDDroid on the MalNet 61K dataset against the ResNet18 model and the SIR-GN algorithm as reported in [34].

On the MalNet tiny 5K dataset, HDDroid achieves an accuracy of 0.90, closely matching the best method, SIR-GN, which has an accuracy of 0.92, demonstrating the viability of the HDC paradigm for malware detection. On the larger 61K dataset, HDDroid outperforms the SIR-GN algorithm, demonstrating its scalability and the effectiveness of the HDC paradigm in capturing relevant information in large function call graphs.

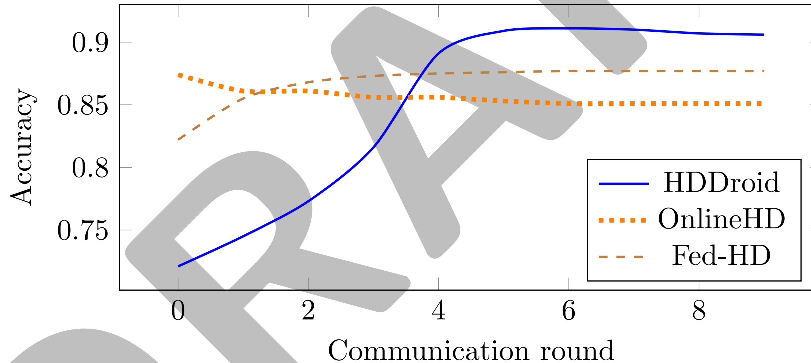

Figure 3: Accuracy achieved by HDDroid on the MalNet 61K dataset across training rounds compared with OnlineHD and Fed-HD.

Other HDC methods: HDDroid's training and aggregation strategies are assessed through a comparison with traditional HDC methods. The first compared approach utilizes OnlineHD [27] for local training, maintaining HDDroid's aggregation strategy. For the second, Fed-HD [32] is used for aggregation, keeping HDDroid's local training strategy. Figure 3 shows the accuracy of HDDroid on the MalNet 61K dataset across training rounds compared with OnlineHD and Fed-HD. OnlineHD starts with high initial accuracy but drops over time, likely due to local training saturation. Fed-HD shows initial improvement but quickly plateaus, possibly due to redundant information in the global classifier. HDDroid, on the other hand, starts with a lower initial accuracy, as the inclusion of new HVs is conservative, but quickly surpasses the other methods, capturing more of the relevant information in the HVs.

To better assess how well the obtained class hypervectors capture the information relative to their classes, the Receiver Operating Characteristic (ROC) curve for HDDroid, OnlineHD, and Fed-HD are compared, plotting the true positive rate against the false positive rate varying

DRAFT

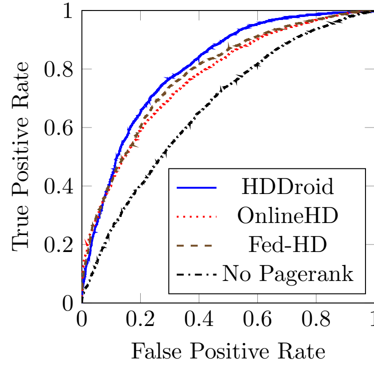

Figure 4: Receiver Operating Characteristic (ROC) curve for HDDroid, compared to Fed-HD, OnlineHD, and not ordering nodes before encoding.

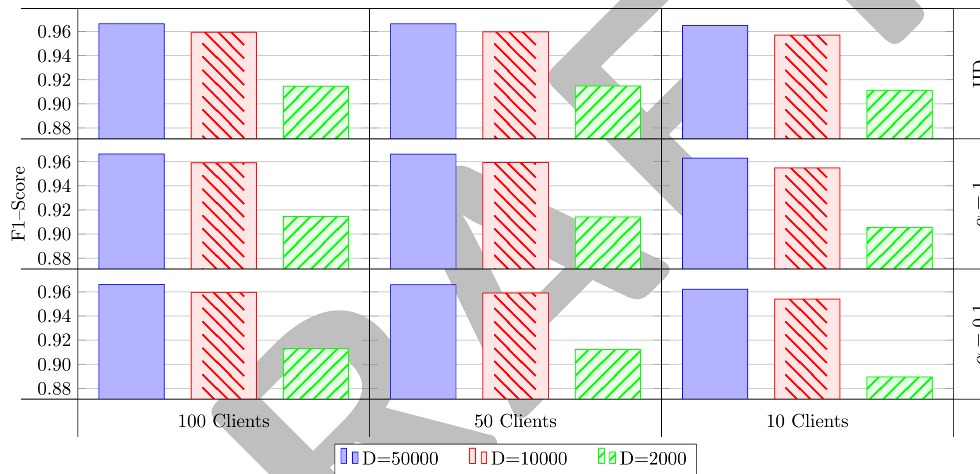

Figure 5: Model performance varying the number of clients, non-IID level, and hypervector size.

DRAFT

the classification threshold. In this analysis, a model that does not rely on the PageRank node ordering is also considered to gauge the impact of this procedure on the quality of the encoding. Figure 4 shows the ROC curve for the tested methods. Removing PageRank ordering significantly reduces model performance, highlighting its importance in encoding. As in the previous analysis, HDDroid outperforms the other approaches, confirming the effectiveness of the training and aggregation strategies.

Robustness to non-IID data distributions: In federated learning settings, clients often have heterogeneous data distributions, which can lead to a decrease in model performance [36]. To evaluate HDDroid's robustness to non-IID data distributions, the dataset labels are distributed among clients using a Dirichlet distribution. An IID setting and two non-IID settings are tested: a moderate non-IID setting ($\alpha = 1$), and a highly non-IID setting ($\alpha = 0.1$), where malware classes are almost partitioned between clients. As shown in Figure 5, larger hypervector sizes generally improve performance, with diminishing returns at higher sizes. With at least 50 clients,

non-IID data distribution does not significantly impact performance. Reducing the number of clients negatively affects the F1-Score, especially with smaller hypervectors. Larger hypervectors (e.g., 50000 dimensions) are more robust, maintaining accuracy above 96% across all settings.

6 Conclusions

Malware detection is a critical task to ensure the security of mobile devices, however collecting and sharing data for training models can raise privacy concerns, thus decentralized on-device analysis and training are desirable. This work presents HDDroid, a novel approach to mobile malware detection that leverages the Hyperdimensional Computing paradigm to allow lightweight and efficient federated training and local classification of mobile applications on resource-constrained mobile devices. The on-device analysis preserves the privacy of the participating users and allows for efficient and robust malware detection. The experimental evaluation shows that HDDroid achieves state-of-the-art graph-based performance for malware detection, remaining effective in non-IID scenarios, making it suitable for real-world applications where data distributions among devices can be highly heterogeneous. Future works will focus on improving the flexibility and adaptability of the model in handling variations over time in the data distribution of the clients and on enabling collaboration between devices with heterogeneous hardware capabilities.

Acknowledgements This work was partially funded by the European Union Next-Generation EU (PIANO NAZIONALE DI RIPRESA E RESILIENZA (PNRR) – MISSIONE 4 COMPONENTE 2, INVESTIMENTO 1.3) – Progetto "Future Artificial Intelligence - FAIR" PE00000013, CUP J73C24000060007

Declaration on Generative AI The authors have not employed any Generative AI tools.

DRAFT

References

[1] A. Qamar, A. Karim, and V. Chang, "Mobile malware attacks: Review, taxonomy & future directions," *Future Generation Computer Systems*, vol. 97, pp. 887–909, 2019.

[2] Y. Pan, X. Ge, C. Fang, and Y. Fan, "A systematic literature review of android malware detection using static analysis," *IEEE Access*, vol. 8, pp. 116363–116379, 2020.

[3] M. Chua and V. Balachandran, "Effectiveness of android obfuscation on evading anti-malware," in *Proceedings of the eighth ACM conference on data and application security and privacy*, pp. 143–145, 2018.

[4] T. Bilot, N. El Madhoun, K. Al Agha, and A. Zouaoui, "A survey on malware detection with graph representation learning," *ACM Computing Surveys*, vol. 56, no. 11, pp. 1–36, 2024.

[5] S. Seneviratne, A. Seneviratne, P. Mohapatra, and A. Mahanti, "Predicting user traits from a snapshot of apps installed on a smartphone," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 18, no. 2, pp. 1–8, 2014.

[6] L. Ge and K. K. Parhi, "Classification using hyperdimensional computing: A review.," *IEEE Circuits and Systems Magazine*, 2020.

[7] S. Freitas, Y. Dong, J. Neil, and D. Horng (Polo) Chau, "A large-scale database for graph representation learning," in *NeurIPS Datasets and Benchmarks Track*, November 2021.

[8] Z. Liu, R. Wang, N. Japkowicz, D. Tang, W. Zhang, and J. Zhao, "Research on unsupervised feature learning for android malware detection based on restricted boltzmann machines," *Future Generation Computer Systems*, vol. 120, pp. 91–108, 2021.

[9] P. Faruki, V. Ganmoor, V. Laxmi, M. S. Gaur, and A. Bharmal, "Androsimilar: robust statistical feature signature for android malware detection," in *Proceedings of the 6th International Conference on Security of Information and Networks*, pp. 152–159, 2013.

[10] Y. Cui, Y. Sun, and Z. Lin, "DroidHook: a novel API-hook based android malware dynamic analysis sandbox," *Automated Software Engineering*, vol. 30, no. 1, p. 10, 2023.

[11] B. Anderson, C. Storlie, and T. Lane, "Improving malware classification: bridging the static/dynamic gap," in *Proc. of the 5th ACM workshop on Security and artificial intelligence*, 2012.

[12] A. Narayanan, G. Meng, L. Yang, J. Liu, and L. Chen, "Contextual Weisfeiler-Lehman graph kernel for malware detection," in *2016 International Joint Conference on Neural Networks (IJCNN)*, pp. 4701–4708, 2016.

[13] C. Li, Z. Cheng, H. Zhu, L. Wang, Q. Lv, Y. Wang, N. Li, and D. Sun, "DMalNet: Dynamic malware analysis based on API feature engineering and graph learning," *Computers & Security*, vol. 122, p. 102872, 2022.

[14] T. S. John, T. Thomas, and S. Emmanuel, "Graph convolutional networks for android malware detection with system call graphs," in *2020 Third ISEA Conference on Security and Privacy (ISEA-ISAP)*, pp. 162–170, IEEE, 2020.

[15] N. V. Hung, P. N. Dung, T. N. Ngoc, V. D. Phai, and Q. Shi, "Malware detection based on directed multi-edge dataflow graph representation and convolutional neural network," in *2019 11th Int. Conf. on Knowledge and Systems Engineering*, pp. 1–5, IEEE, 2019.

[16] C. Cai and Y. Wang, "A simple yet effective baseline for non-attributed graph classification," *arXiv preprint arXiv:1811.03508*, 2018.

[17] T. Schulz and P. Welke, "On the necessity of graph kernel baselines," in *ECML-PKDD, GEM workshop*, vol. 1, p. 6, 2019.

[18] B. Rozemberczki and R. Sarkar, "Characteristic functions on graphs: Birds of a feather, from statistical descriptors to parametric models," in *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pp. 1325–1334, 2020.

[19] A. Tsitsulin, M. Munkhoeva, and B. Perozzi, "Just slaq when you approximate: Accurate spectral distances for web-scale graphs," in *Proc. of the Web Conf. 2020*, pp. 2697–2703, 2020.

[20] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2008.

[21] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?," in *International Conference on Learning Representations*, 2019.

[22] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations*, 2017.

[23] I. Nunes, M. Heddes, T. Givargis, A. Nicolau, and A. Veidenbaum, "GraphHD: Efficient graph classification using hyperdimensional computing," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1485–1490, IEEE, 2022.

[24] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial intelligence and statistics*, pp. 1273–1282, PMLR, 2017.

[25] A. Augello, A. De Paola, and G. Lo Re, "M2FD: Mobile malware federated detection under concept drift," *Computers & Security*, p. 104361, 2025.

[26] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors," *Cognitive computation*, vol. 1, pp. 139–159, 2009.

[27] P. Vergés, T. Givargis, and A. Nicolau, "Refinehd: Accurate and efficient single-pass adaptive learning using hyperdimensional computing," in *2023 IEEE Int. Conf. on Rebooting Computing (ICRC)*, pp. 1–8, IEEE, 2023.

[28] A. Hernández-Cano, N. Matsumoto, E. Ping, and M. Imani, "OnlineHD: Robust, efficient, and

DRAFT

single-pass online learning using hyperdimensional system," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 56–61, IEEE, 2021.

[29] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[30] J. Morris, R. Fernando, Y. Hao, M. Imani, B. Aksanli, and T. Rosing, "Locality-based encoder and model quantization for efficient hyper-dimensional computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 4, pp. 897–907, 2021.

[31] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw, "Parallel random numbers: as easy as 1, 2, 3," in *Proc. of 2011 int. conf. for high performance computing, networking, storage and analysis*, pp. 1–12, 2011.

[32] Q. Zhao, K. Lee, J. Liu, M. Huzaifa, X. Yu, and T. Rosing, "FedHD: Federated learning with hyperdimensional computing," in *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, pp. 791–793, 2022.

[33] Y. Park, D.-J. Han, D.-Y. Kim, J. Seo, and J. Moon, "Few-round learning for federated learning," *Advances in Neural Information Processing Systems*, vol. 34, pp. 28612–28622, 2021.

[34] M. Quebrado, E. Serra, and A. Cuzzocrea, "Android malware identification and polymorphic evolution via graph representation learning," in *2021 IEEE International Conference on Big Data (Big Data)*, pp. 5441–5449, IEEE, 2021.

[35] M. Heddes, I. Nunes, P. Vergés, D. Kleyko, D. Abraham, T. Givargis, A. Nicolau, and A. Veidenbaum, "Torchhd: An open source python library to support research on hyperdimensional computing and vector symbolic architectures," *Journal of Machine Learning Research*, vol. 24, no. 255, pp. 1–10, 2023.

[36] A. Augello, G. Falzone, and G. Lo Re, "DCFL: Dynamic clustered federated learning under differential privacy settings," in *2023 IEEE Int. Conf. on Pervasive Computing and Communications Workshops and other Affiliated Events*, pp. 614–619, 2023.

DRAFT