Mobile Applications Fingerprinting using Locality-Sensitive Hashing

Samer Saleh
Supervisor: Fatemeh Marzani
University of Twente
Email: samerashrafwiliamhanasaleh@student.utwente.nl, f.marzani@utwente.nl

***Abstract*—With the rapid growth in the number of mobile applications, monitoring their activity within a network has become increasingly challenging due to the substantial volume of traffic generated. Mobile application fingerprinting has emerged as a practical solution for identifying application behaviour through network traces, even when traffic is encrypted. This paper introduces an efficient approach using MinHash and Locality-Sensitive Hashing (LSH) to identify mobile application behaviours by comparing only traces with high similarity, which significantly reduces computational overhead while maintaining high accuracy and supporting the detection of previously unseen applications. We evaluated the proposed method through two experiments: application recognition and unseen application detection. In these experiments, we achieved an average accuracy of 83% on the ReCon dataset, while reducing comparison complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$.**

Keywords: LSH, Mobile Fingerprinting, Mobile Applications, MinHash

I. Introduction

A. Motivation

Mobile application fingerprinting refers to techniques for identifying which applications are generating observed network traffic [1]. This capability is vital for network management, quality-of-service (QoS) assurance, and intrusion detection [2]. For example, network operators need to know which applications traverse their infrastructure to allocate resources and detect malicious behaviour. Xin Wang et al. [3] note that the explosive growth of mobile device usage has made application identification "a crucial task for mobile network management and security." However, since most mobile traffic is encrypted [4] [5], fingerprinting must rely on metadata and behavioural features rather than payload content. This makes it possible to monitor applications just by looking at network traffic, but it also means that even encrypted traffic can unintentionally reveal personal details about how someone is using their applications[4]. Nevertheless, given billions of devices and millions of applications in ecosystems like Google Play, scalable application fingerprinting is essential to security and privacy-preserving monitoring.

B. Existing Mobile Fingerprinting Techniques

Traditional methods such as port- [6] or signature-based [7] inspection are ineffective on today's mobile traffic [8]. Instead, researchers have explored several classes of approaches:

Machine learning classifiers achieve high accuracy but require labeled data and frequent retraining [3][9].

Correlation-based methods, such as FlowPrint [4], discover application fingerprints from unlabeled traffic by finding sets of network endpoints that consistently communicate together. FlowPrint builds a graph where each node represents a cluster of destinations (IP/port pairs) and edges encode how often two destinations co-occur in time. It then finds groups of endpoints that frequently communicate together and uses the list of addresses in each group as an application's fingerprint. Once fingerprints are constructed, FlowPrint uses the *Jaccard similarity* between destination sets to label traffic. In the training phase, flows are grouped into fingerprints, and each fingerprint is assigned the application label that appears most often among its flows (a majority-vote). At runtime, a new flow's fingerprint (i.e. its set of destinations) is compared against all labeled fingerprints from training using Jaccard similarity; the flow is then given the label of the closest matching fingerprint. This unsupervised clustering approach allows FlowPrint to recognise applications without pre-existing signatures, but it can be computationally heavy (finding cliques in a graph) and may produce multiple overlapping fingerprints per application.

Automata-based approaches, like ML-NetLang [10], treat each application's traffic as a formal language of destination sequences. During training, ML-NetLang converts each application trace into a "word" by ordering its destination identifiers (e.g. IP addresses or service names). It then runs a grammar inference algorithm (k-TSS) to learn a *k-test vector* (a form of finite automaton) that captures all allowed substrings of length *k* in that word. Effectively, each application's fingerprint is a set of valid address subsequences (a k-TSS language) drawn from its training traces. For classification, an incoming trace is similarly turned into a word of destinations and compared against all learned languages. The system computes numerical features measuring how well the trace's substrings match each application's k-TSS fingerprint (for example, by counting common k-grams). These features (one per known application) are fed into a supervised classifier (e.g. logistic regression) which outputs the most likely application label. This method is robust to traffic encryption (since it only uses address sequences) and has shown high accuracy, but it relies on labeled training data for every application and cannot recognise *unseen* applications not in its training set.

More recently, Mashnoor et al. introduced LSIF, a lightweight fingerprinting method that uses the Nilsimsa (LSH-based method) hash to identify IoT devices based on

their encrypted network traffic. They combine all traffic from a device into one continuous stream and compute a single compact hash for it. Their approach does not require explicit feature extraction or model retraining. However, LSIF is fundamentally device-centric its goal is to recognise which device generated the traffic, not which specific application. In contrast, our approach shifts the focus to application-level fingerprinting, which is a more granular and practical need in mobile environments where a single device may run multiple applications simultaneously. Rather than aggregating all device traffic, we assume each trace originates from a single application and compute an LSH fingerprint for that application's behaviour. These fingerprints are then compared against a database of known application fingerprints to identify the source application. This distinction is crucial for network operators interested in application-level traffic analysis, for monitoring, access control, or anomaly detection. While both methods use LSH to enable fast, scalable comparisons on encrypted traffic, our approach distinguishes in the fact that it is able to identify and detect individual applications, including previously unseen applications. It avoids the need for hand-crafted features or supervised learning, and it supports open-set recognition, meaning that it can identify input (applications) that do not belong to any class (unseen applications) as compared to LSIF.

C. Problem Statement and Research Objectives

To evaluate the effectiveness and practicality of using LSH for mobile application fingerprinting, we propose the following research questions. These questions aim to assess the suitability of similarity metrics, the accuracy of estimation techniques, and the impact of configurable parameters on our approach:

- **RQ1**: Is Jaccard similarity an appropriate metric for comparing mobile application execution traces represented as sets of k-shingles?
- **RQ2**: How accurately does MinHash's estimated Jaccard similarity approximate the actual Jaccard similarity?
- **RQ3**: How do variations in key parameters, such as the number of hash functions, band size, and number of bands, affect accuracy?
- **RQ4**: How effective is the proposed LSH-based system in recognizing known applications and detecting previously unseen ones based on network behaviour traces?

By addressing these questions, the work aims to provide insights into the practical applicability of hash-based fingerprinting approaches.

BACKGROUND OF LSH

LSH is a technique used to efficiently identify similar items in large datasets without comparing all pairs. It begins with a process called Shingling. In simple terms, this is a way of representing the information within a document (or, in this case, an application's network trace) as a set of substrings of length *k*, known as *k*-shingles. For example, if a trace is represented by the string `abcdefg` and $k = 3$, the set of 3-shingles is `{abc, bcd, cde, def, efg}`. This lets us convert behavioural traces into sets, allowing similarity to be measured via set-based metrics like Jaccard similarity:

$$\hat{J}(A, B) = \frac{1}{n} \sum_{i=1}^{n} [h_i(A) = h_i(B)] \tag{1}$$

This formula is explained later in this section.

After generating these shingles, we apply a hash function to each shingle, rather than using the raw strings. This step compresses the data and allows for efficient comparison. For example, instead of storing each shingle as a full string, we can apply a hash function that converts it into a compact numerical value, such as a 32-bit integer. While this can lead to occasional collisions (different shingles mapping to the same value), it significantly reduces memory usage and allows for fast, constant-time comparisons.

The next step is MinHashing. We simulate this process using hash functions. For each hash function, we apply it to the indices of all shingles in a trace and record the minimum value. Repeating this across multiple hash functions yields a MinHash signature—a vector of integers that compactly represents the original set. The similarity between two traces can now be approximated by the fraction of components their signatures share, which estimates the Jaccard similarity between the original sets (1).

Example – MinHashing: Consider two traces A and B with shingle sets `{a, d}` and `{a, c, d}` respectively. Suppose we define a hash function such that:

`h(a) = 1, h(c) = 3, h(d) = 2`

The MinHash of A is `min{1, 2} = 1`, and for B it is `min{1, 3, 2} = 1`. Since both sets have the same MinHash under this function, we count this as one match. Repeating with multiple hash functions and counting the fraction of matches provides an estimate of Jaccard similarity.

To efficiently detect similar traces among large collections, we apply LSH bucketing to these MinHash signatures. A MinHash signature is split into *b* bands of *r* rows each. Each band (a short vector of hash values) is hashed into a bucket.

The relationship between the number of hashes, bands, and band size is given by the formula:

$$\text{Number of hashes} = \text{Number of bands} \times \text{Band size}$$

Each h-dimensional signature is split into b bands of size r, so that $b \cdot r = h$. Two signatures that match exactly in all r values of any band are placed in the same bucket. By construction, two signatures with true Jaccard similarity s collide in at least one band with probability

$$P(\text{candidate}) = 1 - (1 - s^r)^b$$

which creates a sharp threshold behaviour. In other words, similar traces (large s) are very likely to share a band, while dissimilar traces (small s) rarely do. Two traces that share the exact same band vector will land in the same bucket for that band and are marked as a candidate pair. This drastically reduces the number of comparisons needed.

Example – Bucketing: Assume signatures of length 12 are divided into 4 bands of 3 values each. Trace A has a band

`[0, 1, 3]`, and so does trace B. Since they match exactly in one band, they are assigned to the same bucket in that band and are flagged for detailed comparison. If two traces differ in all bands, they are never compared. This selective approach enables fast similarity search even on massive datasets. Note that these bands must also appear in the same position, meaning that if `[0, 1, 3]` appears in different places in both traces, then they are not assigned to the same bucket, as a bucket is identified by an index and the band, so for example assume trace A generated the following buckets: bucket 0 : `[0, 1, 3]`, bucket 1: `[1, 1, 3]`, bucket 3: `[0, 1, 0]`, and so on. If `[0, 1, 3]` does not appear in bucket 0 for trace B, then they are not put together in the same bucket, and therefore not considered similar.

After the buckets have been computed, traces that fall within the same bucket will then have all of there signatures compared with each other to find the similarity between them, which implies that exact equality between traces is not needed, but it will rather filter out dissimilar traces. If a trace shares similar signatures with multiple traces, then the trace with the highest similarity is considered to be the most similar trace. Thresholds are often used after computing the Jaccard similarity between pairs of traces. Pairs that have a similarity above the set threshold (e.g. 0.5) will be flagged as candidate pairs; those below the threshold are not considered to be candidate pairs.

In essence, LSH provides an efficient and scalable way to detect similar application behaviours by combining shingling, MinHashing, and probabilistic bucketing. This allows the system to quickly identify traces that are likely to be similar, without needing to compare every pair directly.

II. Methodology

A. Dataset

We use the ReCon dataset, which includes multiple versions of 512 popular Android applications from the Google Play Store, covering 7,665 application releases over eight years of version history [11, 12]. To collect network traffic, each application was installed and interacted with through automated and scripted scenarios on real mobile devices, simulating typical user behaviour. From this dataset, we randomly selected 100 applications and 10 execution traces for each, resulting in a total of $1,000$ traces used in our experiments.

The raw network traffic was converted into symbolic behavioural traces using the *Trace Generator* introduced in [10], which processes each application's traffic as follows:

1) Network flows are first extracted from packet captures.
2) For each flow, features such as timestamps, destination IP address, destination port, and TLS certificate are extracted.
3) All flows are sorted chronologically by the timestamp of their first packet.
4) Flows are categorised based on unique destination identifiers—either (IP, port) tuples or TLS certificates.
5) Each unique destination category is assigned a symbolic label using a mapping function. These labels follow the format "Sj," where j is a natural number representing the j-th unique destination observed. For example, "S6" denotes the 6th unique destination seen across all traces.
6) Each flow is then replaced with its corresponding symbol, forming a symbolic trace that captures the temporal order of destinations accessed by the application during execution.

These symbolic traces serve as abstract behavioural representations of each application's network activity and are the basis for our fingerprinting and similarity analysis.

B. Parameter selection

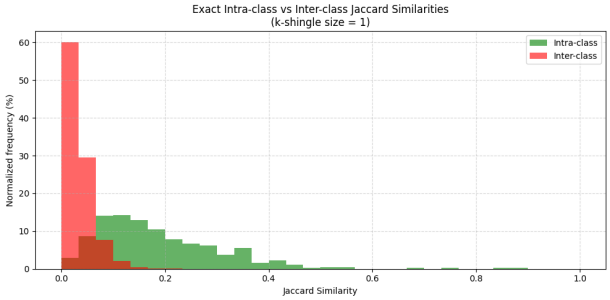
To evaluate whether Jaccard similarity is suitable for comparing mobile application execution traces, to answer RQ1, we analysed how well it distinguishes between traces from the same application (intra-class) and different applications (inter-class). This evaluation was carried out using 10 random applications. Each trace was then represented as a set of *k*-shingles, and the actual Jaccard similarity was then computed between all pairs:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2}$$

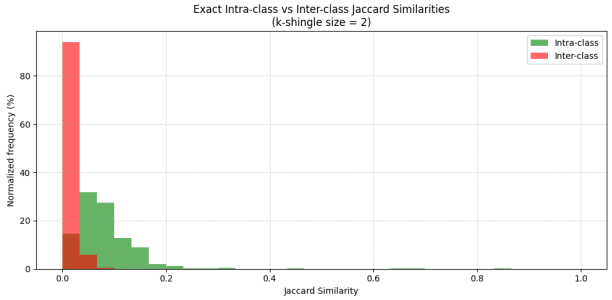
The difference between the actual Jaccard similarity (2) and estimated Jaccard similarity (1) is that it compares the k-shingles rather than the MinHash signatures, which involves more comparisons but is more accurate [13]. The normalised frequency of similarity values for both intra- and inter-class comparisons was then plotted. These plots reveal whether same-application traces consistently show higher similarity than different-application traces. A clear separation between the two distributions suggests that Jaccard similarity effectively captures behavioural similarity. By varying *k*, the plots also help identify the shingle size that best supports this distinction, directly answering the research question. To quantitatively assess how well different *k*-shingle sizes separate intra- and inter-class similarity distributions, Python's NumPy library was used to calculate the area of overlap between the two distributions. A lower overlap area indicates better separation and, therefore, a more effective similarity metric.

Across shingle sizes k = 1 to 4, the overlap areas were 0.234, 0.227, 0.234, and 0.399, respectively. The overlap areas can also be seen in each of the figures - it is the region in figures 1a, 1b, 1c and 1d colored in dark red. Based on these results, k **= 2** yielded the smallest overlap, and thus the clearest distinction between same-application and different-application traces, supporting it as the most appropriate shingle size for this task.

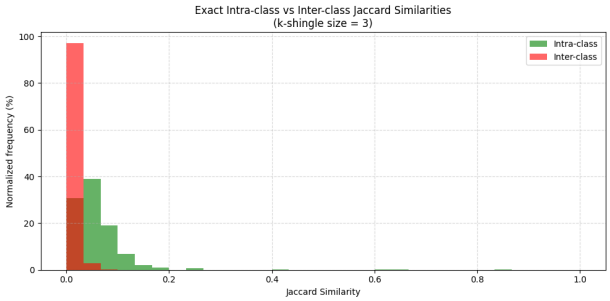
Then to establish how well MinHash's estimated Jaccard similarity (1) approximates the actual Jaccard similarity (2), to answer RQ2, the mean squared error (MSE) between them was plotted across different number of hash functions in order to find the optimal number of hashes where the difference between the actual Jaccard similarity and the estimated Jaccard similarity was considered negligible.



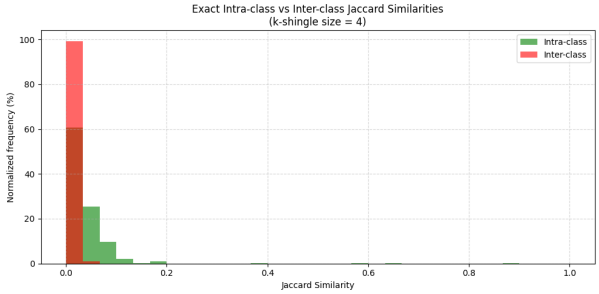
(a) 1-shingle



(b) 2-shingle



(c) 3-shingle



(d) 4-shingle

Fig. 1: Normalised frequency distributions of actual Jaccard similarity values for different k-shingle sizes. Lower overlap indicates better separation between same-application and different-application traces.

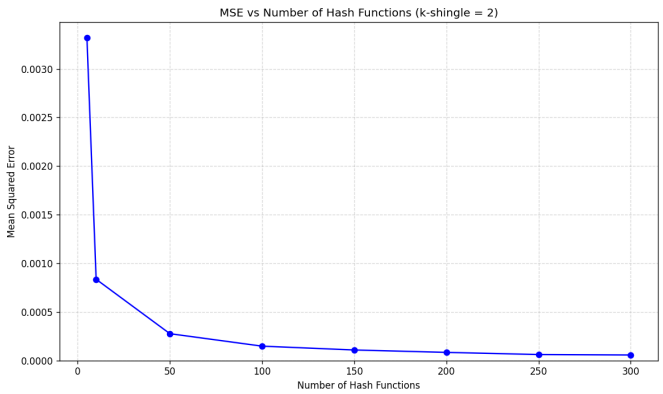


Fig. 2: MSE across different numbers of hash functions after setting the number of shingles to 2.

As shown in 2, the MSE decreases with more hash functions but plateaus around 150 hashes. Beyond this point, further increases in the number of hashes provide negligible improvement in estimation accuracy. Therefore, 150 hash functions were chosen as a balance between performance and computational efficiency.

C. System Overview

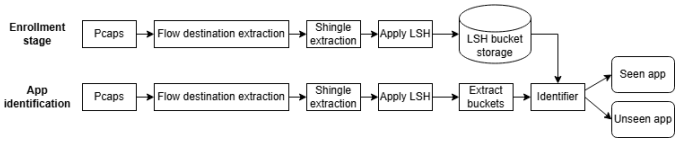


Fig. 3: System overview

The proposed system for application recognition and unseen application identification is structured into two main operational stages: the *enrollment stage* and the *application identification stage*, as illustrated in Fig. 3. Both stages follow a nearly identical processing pipeline, but serve distinct purposes within the fingerprinting framework.

Enrollment Stage. This stage is responsible for creating and storing application fingerprints. It begins with the collection of network packet captures (pcaps), which are preprocessed to extract flow-level destination information.

Once the destination flows are extracted, each trace is converted into a sequence of k-shingles, capturing short, overlapping subsequences that represent local behavioural patterns. These shingles are then passed through a MinHashing step to generate compact signature vectors. Each signature is divided into multiple bands according to the specified band size, and each band serves as an index into an LSH bucket.

The LSH bucketing process effectively groups traces that share a high degree of structural similarity. These buckets serve as the system's reference for application behaviour and are

stored persistently in the *LSH bucket storage* for use during identification.

Application Identification Stage. In this stage, the system processes new or incoming traces to determine whether they correspond to known applications or exhibit previously unseen behaviour. The trace undergoes the same preprocessing and fingerprinting pipeline as in the enrolment stage: flow destination extraction, shingle formation, MinHash signature generation, and LSH bucketing.

Once the trace is hashed into LSH buckets, the system invokes the *identifier*, which compares the newly generated buckets with those stored during the enrollment stage. If a matching bucket is found, the system assigns the trace to the corresponding enrolled application. The logic assumes that traces mapped to the same bucket are behaviourally similar and thus likely originate from the same application or a functionally similar one.

In case of unseen application detection, when there are no matching buckets found or the similarity falls below the defined threshold, the trace is flagged as belonging to an unseen application. This mechanism allows the system not only to classify known behaviours but also to detect novel patterns that deviate from all previously observed signatures.

III. Evaluation and Results

This section presents two key experiments designed to evaluate the ability of the LSH-based approach to classify mobile application traces. Both experiments use a dataset of 100 applications, each containing 10 execution traces extracted from recon-symbol logs. These traces were converted into k-shingles and then represented using MinHash signatures for efficient comparison. The core goal is to assess how effectively the model can identify whether a trace belongs to a previously seen (known) application or represents unseen behaviour, thereby supporting application fingerprinting in dynamic network environments.

To evaluate the effectiveness of the proposed approach, we report four widely used classification metrics: accuracy, precision, recall, and F1-score. These metrics are used for both application recognition and unseen application detection, and they are well-established in recent literature on similarity-based and LSH-driven fingerprinting systems.

Accuracy measures the overall correctness of classification, while precision and recall capture the trade-off between false positives and false negatives. The F1-score, as the harmonic mean of precision and recall, provides a balanced assessment of model performance. Their combined use is particularly relevant in similarity-based tasks where exact matches are not guaranteed, and in open-world settings where distinguishing known from unknown classes is essential.

Recent studies on mobile and IoT fingerprinting support the use of these metrics in LSH-based contexts. For instance, ML-Netlang [10], FlowPrint [4], and AppSniffer [14] report these metrics to evaluate application identification. These works emphasise that while accuracy gives a general performance view, precision and recall are crucial for understanding the system's ability to correctly classify or reject traces, particularly in scenarios involving encrypted or evolving traffic patterns.

In this work, we adopt the same metrics to ensure comparability with prior research and to capture both overall and class-specific performance, especially when evaluating robustness across seen and unseen application traces.

TABLE I: Band Size for Different Number of Bands (with 150 Hashes)

Number of Bands	Band Size
150	1
75	2
50	3
30	5

TABLE II: Results showing accuracy, precision, recall and F1-score of application recognition using different number of bands and band sizes.

Bands (b)	Accuracy	Precision	Recall	F1 Score
150	**0.8527 ± 0.0428**	**0.8144 ± 0.0733**	**0.7752 ± 0.0851**	**0.7772 ± 0.0778**
75	0.7795 ± 0.0570	0.7343 ± 0.0957	0.6777 ± 0.1001	0.6858 ± 0.0948
50	0.3186 ± 0.0708	0.4153 ± 0.0827	0.2877 ± 0.0710	0.3287 ± 0.0725
30	0.0645 ± 0.0341	0.0954 ± 0.0534	0.0629 ± 0.0335	0.0736 ± 0.0399

For Locality-Sensitive Hashing (LSH), each bucket corresponds to a band composed of a fixed number of hash values, known as the band size. In our experiment, the number of hashes was fixed at 150. The following table shows the resulting band size for each band count configuration:

According to Meira et al. [15], larger b (smaller r) increases collision probability for a given s (making the scheme more sensitive, at the cost of more false positives), whereas smaller b (larger r) makes the scheme stricter. We empirically tested these settings (see II and III) to balance the trade-off between false negatives and false positives.

A. *Application Recognition*

The first experiment evaluates the system's ability to recognise applications it has encountered during training. For each application, 8 of the 10 available traces are used for training, while the remaining 2 traces serve as the test set. This is implemented using 5-fold stratified cross-validation across all 100 applications, ensuring that each fold tests on a different subset of traces while preserving class balance. After generating the MinHash signatures for the training traces, LSH is used to bucketize them. During testing, each query trace is hashed, and candidates are retrieved from the corresponding LSH buckets. If the most similar candidate (based on Jaccard similarity) matches a training application, that application is predicted. In cases of tie on similarity across multiple applications, majority voting is applied to determine the final prediction. This experiment provides insight into the model's ability to correctly associate previously seen behaviours with their originating applications. For the classification of seen applications, our results show a sharp performance degradation as the number of bands decreases. In particular, the 150-band setting (band size 1) achieved the highest performance: 85.27% accuracy and an F1-score of 77.72%. In contrast, reducing the band count to 75, 50 or 30 yielded substantially

lower accuracy, precision, recall, and F1, reflecting the known effect that more hash functions (larger signature size) yield better LSH matching quality. In summary, our best seen-application configuration 150 bands and band size of 1 far outperformed all lower-band configurations, highlighting the trade-off between hash count and accuracy in LSH-driven classification.

B. Detection of Previously Unseen Applications

The second experiment introduces unseen applications into the evaluation pipeline. The dataset is split such that 80 applications are treated as "seen" and used for training and partial testing, while the remaining 20 are designated as "unseen." For each seen application, 8 traces are used for training and 2 for testing; all 10 traces of each unseen app are used in the test set. This setup also utilised a 5-fold stratified cross-validation across the 100 applications. This setup simulates real-world scenarios where new applications appear in the network. During inference, if a test trace's similarity to any training trace does not exceed a specified Jaccard threshold, it is labeled as "Unseen." Otherwise, the most similar application is selected based on majority voting among top-matching candidates. The experiment is repeated across five Jaccard thresholds: 0.05, 0.10, 0.15, 0.20, and 0.25 to evaluate robustness to similarity sensitivity. Performance is measured in terms of accuracy, precision, recall, and F1-score, specifically focusing on the ability to distinguish between seen and unseen applications. This experiment helps quantify the system's false positive and true negative rates in a realistic environment. Majority vote was also applied to this experiment.

TABLE III: Results showing accuracy, precision, recall and F1-score of unseen application detection using different number of bands and band sizes, and with the use of different similarity thresholds.

Threshold	Bands (b)	Accuracy	Precision	Recall	F1 Score
0.05	150	0.5783 ± 0.0067	0.9664 ± 0.0226	0.2500 ± 0.0141	0.3969 ± 0.0168
	75	0.6839 ± 0.0079	0.8813 ± 0.0274	0.4990 ± 0.0102	0.6369 ± 0.0070
	50	0.7083 ± 0.0107	0.6676 ± 0.0099	0.9470 ± 0.0068	0.7830 ± 0.0055
	30	0.5906 ± 0.0022	0.5757 ± 0.0013	1.0000 ± 0.0000	0.7307 ± 0.0011
0.10	**150**	**0.8011 ± 0.0111**	**0.8419 ± 0.0181**	**0.7910 ± 0.0128**	**0.8155 ± 0.0093**
	75	0.7883 ± 0.0116	0.8073 ± 0.0187	0.8140 ± 0.0116	0.8104 ± 0.0084
	50	0.6928 ± 0.0057	0.6482 ± 0.0049	0.9780 ± 0.0024	0.7796 ± 0.0029
	30	0.5906 ± 0.0022	0.5757 ± 0.0013	1.0000 ± 0.0000	0.7307 ± 0.0011
0.15	150	0.7433 ± 0.0080	0.6950 ± 0.0062	0.9590 ± 0.0037	0.8059 ± 0.0055
	75	0.7372 ± 0.0054	0.6894 ± 0.0040	0.9590 ± 0.0037	0.8022 ± 0.0038
	50	0.6544 ± 0.0028	0.6183 ± 0.0022	0.9880 ± 0.0040	0.7606 ± 0.0015
	30	0.5906 ± 0.0022	0.5757 ± 0.0013	1.0000 ± 0.0000	0.7307 ± 0.0011
0.20	150	0.6628 ± 0.0067	0.6237 ± 0.0045	0.9910 ± 0.0020	0.7656 ± 0.0038
	75	0.6628 ± 0.0067	0.6237 ± 0.0045	0.9910 ± 0.0020	0.7656 ± 0.0038
	50	0.6289 ± 0.0062	0.6000 ± 0.0040	0.9960 ± 0.0020	0.7489 ± 0.0032
	30	0.5906 ± 0.0022	0.5757 ± 0.0013	1.0000 ± 0.0000	0.7307 ± 0.0011
0.25	150	0.6161 ± 0.0054	0.5918 ± 0.0034	0.9960 ± 0.0020	0.7425 ± 0.0028
	75	0.6161 ± 0.0054	0.5918 ± 0.0034	0.9960 ± 0.0020	0.7425 ± 0.0028
	50	0.6117 ± 0.0067	0.5890 ± 0.0041	0.9960 ± 0.0020	0.7403 ± 0.0035
	30	0.5906 ± 0.0022	0.5757 ± 0.0013	1.0000 ± 0.0000	0.7307 ± 0.0011

For unseen applications, we assessed classification at multiple Jaccard similarity thresholds (0.05, 0.10, 0.15, 0.20, 0.25) using the same band settings. We found that 150 bands with a Jaccard threshold of 0.10 provided the most balanced results: this configuration yielded 80.11% accuracy and an F1-score of 81.55%, with consistently high precision and recall. Lowering the threshold (0.05) made the matching criterion less strict, which produced very high precision but at the cost of far lower recall and a reduced F1-score. Conversely, raising the threshold beyond 0.10 also unbalanced the tradeoff. These patterns reflect the standard precision–recall tradeoff in threshold-based similarity detection: a moderate similarity threshold (0.10) gives the best overall balance of metrics, whereas overly strict thresholds inflate precision at the expense of recall. Thus, our unseen-application experiments indicate that 150 bands and a band size of 1 with a threshold of 0.10 is the optimal setting, achieving high and stable values across all evaluation metrics.

The source code of the experiments are accessible at: [1].

IV. Discussion

Our experiments demonstrate that locality-sensitive hashing (LSH), when configured appropriately, is a practical and efficient strategy for generating behavioural fingerprints of mobile applications. By applying 2-shingle tokenisation, 150 hash functions, and a banding scheme with 150 bands (each of size 1), the system was able to effectively distinguish between known applications and reliably detect unseen ones. These results support the broader observation that approximate similarity detection using MinHash signatures is well-suited to capturing meaningful structural patterns in mobile traffic traces.

Nonetheless, there are several important considerations and limitations that merit further discussion.

Robustness and evasiveness. One potential limitation of LSH-based fingerprinting is its vulnerability to behavioural manipulation. While our approach does not rely on payload inspection or IP-level features—making it more resilient to encryption—it still depends on the consistency of trace-level structures. If a malicious application were engineered to inject noise or mimic another application's signature, it could potentially evade detection or produce ambiguous matches. That said, sustaining such mimicry without degrading the application's functionality or user experience may not be trivial, especially if the attacker does not control upstream network endpoints.

Ambiguity from generic traffic. Another challenge arises from applications whose traffic is dominated by interactions with common services such as advertising networks or analytics platforms. Since many applications share these dependencies, their trace structures can appear deceptively similar, leading to false matches. This limitation is not unique to our method—most flow-based and even packet-level fingerprinting systems struggle with traffic homogeneity. Future improvements might consider augmenting the signature with temporal or frequency-weighted components to better distinguish background services from core app behaviour.

Handling multiple active applications. Our current evaluation assumes that only one application is generating traffic at a time, which simplifies fingerprint assignment. However, modern mobile operating systems increasingly allow concurrent application execution, background activity, and multi-window interactions. These overlapping traces could compromise the fidelity of a single-application fingerprint. One

[1] https://gitlab.utwente.nl/s2888327/minhash-lsh.git

possible solution is to segment traffic based on temporal bursts or session contexts, but this requires richer metadata and further investigation.

Scalability and generalization. One of the key strengths of our method is that it remains lightweight and fast even as the number of known applications increases. The LSH structure supports sublinear-time lookup, and our fingerprinting step does not require retraining or updating large models. This positions the system well for deployment in real-time monitoring environments or resource-constrained devices. However, the accuracy and stability of the model may still depend on the diversity of the dataset used to generate reference signatures. Extending our evaluation to larger, more heterogeneous application sets remains a priority for future work.

Privacy considerations. Finally, while our method does not inspect payloads and operates effectively on encrypted traffic, it still has privacy implications. Being able to infer application usage from metadata alone—even in semi-supervised settings—raises concerns about surveillance and profiling. This underscores the need to carefully consider deployment contexts, ensure informed consent where applicable, and avoid misuse of such systems in ways that could undermine user autonomy.

In summary, our results affirm the value of LSH-based similarity detection for mobile application fingerprinting, particularly in settings where rapid identification, scalability, and encrypted traffic support are critical. At the same time, addressing the edge cases described above will be essential to making such systems more robust, trustworthy, and ethically sound.

V. Related Works

In contrast to our work, the surveyed works use very different techniques. Mashnoor et al. [2] also use LSH (the Nilsimsa hash) but for IoT device identification rather than mobile applications. They examine various Nilsimsa parameter settings and report about 94% device-ID accuracy (precision 94%, recall 93%, F193%). Like our method, this avoids explicit feature extraction, but Mashnoor's LSH is based on fixed-distance digests and tuned for device traffic, whereas our LSH operates on tokenised packet features. In practice, their LSH (LSIF-R) slightly outperforms prior ML methods by 12% (to 94% accuracy). By contrast, our LSH pipeline on application data achieved moderate accuracy (typically below these deep-learning baselines). Unlike our approach, Mashnoor et al. do not address unseen (new) applications (or in their case IoT devices), but rather only identify IoT devices that went through training. Wang et al. [3] propose App-Net, a supervised hybrid neural network that processes raw TLS traffic with parallel CNN and bi-LSTM paths. App-Net automatically learns features from the first data packet and the packet-length sequence, fusing them to recognise applications. On a dataset of 80 mobile applications, they report 93.2% overall accuracy and 91.2% macro-averaged F1 (Precision/recall values are not explicitly given, but accuracy and F1 are both above 90%). Unlike our LSH, App-Net requires training on labelled application traces and is far more complex (deep learning with millions of parameters). In our experiments, the LSH pipeline yielded lower accuracy than App-Net's 93% (85.37% for application recognition and 80.11% for unseen application detection), reflecting the advantage of learned features. App-Net, however, has no mechanism for unknown applications – it can only classify among the 80 trained classes. In summary, Wang et al.'s method is more accurate on the seen-application task but is a heavyweight, supervised model, whereas our LSH is simpler but less accurate. Van Ede et al. [4] introduce FlowPrint, a semi-supervised method focusing on unseen application detection. FlowPrint clusters destination IPs and finds temporal flow correlations to build application fingerprints, entirely without prior labels. On both Android and iOS traffic, they achieve 89.2% accuracy for recognising known applications. Crucially, FlowPrint can flag new applications: it reports a precision of 93.5% for unseen-application detection (and detects 72.3% of new applications within 5 minutes of activity). Our LSH method, by contrast, achieves 84.19% precision for unseen-application detection (takes ≈8 minutes with 5-fold stratified training and testing with different numbers of bands). In practice, FlowPrint's accuracy on known applications (89%) is comparable to our LSH (depending on settings), but its unseen-application capability (93.5% precision) is unique. Marzani et al. [4] propose ML-NetLang, combining automata learning with machine learning. They treat each application's flow-level behaviour as a formal language (k-TSS automaton) over destination-related symbols, then classify using a logistic regression or SVM. On a cross-platform dataset (mixed Android/iOS applications) they report around 95/97% accuracy (95% average F1). For example, ML-NetLang attains 96% accuracy (P≈95–96%, R≈96%, F1≈95%) on the combined Android/iOS test. This outperforms both FlowPrint (89%) and a traditional AppScanner baseline in their experiments. The approach is supervised (using one trace per application for training/testing) and cannot detect unseen applications. Compared to our LSH, ML-NetLang's accuracy is higher (96% vs our 85.37%), but it relies on handcrafted destination features and learning. Our LSH uses only raw packet ordering/hash similarity and is fully unsupervised after hashing; it is lighter but achieves somewhat lower accuracy than ML-NetLang. Finally, Liu et al. [9] present TransECA-Net, a hybrid CNN + Transformer model with channel attention for encrypted-traffic classification. They evaluate on a 12-class encrypted traffic dataset (ISCX). TransECA-Net achieves 97.7% overall accuracy, 98.2% precision, and 97.9% recall (implying F1 98.0%). This is state-of-the-art for their traffic categories, significantly higher than simple CNN or CNN+LSTM baselines. Note this is a general traffic classifier (VoIP, email, etc.), not specifically mobile applications. Nevertheless, it illustrates that deep transformer models can greatly exceed the accuracy of simple LSH or fingerprinting techniques. In relative terms, TransECA-Net's 98% accuracy far surpasses our LSH on application data, though it is a much more complex model and requires training. As with App-Net, there is no unseen-

application detection mechanism – it is pure classification over known classes.

TABLE IV: Performance Comparison on Application recognition

Method (Source)	Accuracy	Precision	Recall	F1 Score	Detects unseen applications?	Complexity
Our approach (LSH)	85.37%	81.44%	77.52%	77.72%	Yes	$\mathcal{O}(n \log n)$
App-Net	93.2%	–	–	91.2%	No	$\mathcal{O}(n^2)$
FlowPrint	94.47%	94.7%	94.47%	94.58%	Yes	$\mathcal{O}(n^2)$
ML-NetLang LR	97.0%	97.0%	97.0%	96.0%	No	$\mathcal{O}(n^2)$
TransECA-Net	$\approx 98.0\%$	$\approx 98.0\%$	$\approx 98.0\%$	$\approx 98.0\%$	No	$\mathcal{O}(n^2)$

VI. Conclusion and Future Work

This work introduced a scalable fingerprinting technique for mobile applications using Locality-Sensitive Hashing (LSH) over symbolic execution traces. By transforming behavioural traces collected via the ReCon framework into sets of k-shingles, then encoding them with MinHash and organizing them into LSH buckets, the system efficiently estimates similarity between traces without relying on deep packet inspection or fully supervised learning. Through careful parameter tuning—particularly using $k = 2$ shingles, 150 hash functions, and band size 1—the approach achieves a practical balance between detection accuracy and computational efficiency.

Empirical results on a dataset of 1,000 application traces (100 applications × 10 runs) validate the method's performance in both closed-world (seen applications) and open-world (unseen application detection) settings. The system attained up to 85.27% accuracy and 77.72% F1-score for known applications, and 80.11% accuracy with 81.55% F1-score for identifying unknown applications at a Jaccard threshold of 0.10. Compared to more complex models like FlowPrint [4] or neural approaches like ML-NetLang [10] or TransECA-Net [9], this method offers a lightweight and interpretable alternative that performs well under real-time constraints and encrypted traffic, making it especially suitable for network monitoring in constrained environments.

VII. Acknowledgements

In preparing this work, ChatGPT was used to correct grammatical errors, ensure consistent use of British English, and assist with debugging the source code.

References

[1] M. Jiang, Z. Li, P. Fu, W. Cai, H. Xu, K. Zhao, X. Zhang, and S. Xu, "Accurate mobile-app fingerprinting using flow-level relationship with graph neural networks," *Computer Networks*, vol. 217, p. 109309, 2022.

[2] N. Mashnoor, J. Thom, A. Rouf, S. Sengupta, and B. Charyyev, "Locality sensitive hashing for network traffic fingerprinting," in *Proceedings of the IEEE International Conference on Communications (ICC)*, 2023.

[3] X. Wang, S. Chen, and J. Su, "Automatic mobile app identification from encrypted traffic with hybrid neural networks," *IEEE Access*, vol. 8, pp. 182065–182077, Jan. 2020.

[4] T. van Ede, R. Bortolameotti, A. Continella, J. Ren, D. J. Dubois, M. Lindorfer, D. Choffnes, M. van Steen, and A. Peter, "Flowprint: Semi-supervised mobile-app fingerprinting on encrypted network traffic," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2020.

[5] E. Papadogiannaki and S. Ioannidis, "A survey on encrypted network traffic analysis applications, techniques, and countermeasures," *ACM Computing Surveys*, vol. 54, no. 6, pp. 1–42, 2021.

[6] T. Karagiannis, A. Broido, M. Faloutsos, and K. Claffy, "Transport layer identification of P2P traffic," *in Proc. 4th ACM SIGCOMM conference on Internet measurement*, pp. 121–134, 10 2004.

[7] Y. Wang, Y. Xiang, and S.-Z. Yu, "Automatic Application Signature Construction from Unknown Traffic," *in 24th IEEE International Conference on Advanced Information Networking and Applications*, pp. 1115–1120, 1 2010.

[8] G. Xu, M. Xu, Y. Chen, and J. Zhao, "A Mobile Application-Classifying Method Based on a Graph Attention Network from Encrypted Network Traffic," *Electronics*, vol. 12, p. 2313, 5 2023.

[9] Z. Liu, Y. Xie, Y. Luo, Y. Wang, and X. Ji, "Transecanet: A transformer-based model for encrypted traffic classification," *Applied Sciences*, vol. 15, no. 6, p. 2977, 2025.

[10] F. Marzani, F. Ghassemi, Z. Sabahi-Kaviani, T. Van Ede, and M. Van Steen, "Mobile app fingerprinting through automata learning and machine learning," in *2023 IFIP Networking Conference (IFIP Networking)*, pp. 1–9, 2023.

[11] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. R. Choffnes, "Demo: Recon: Revealing and controlling PII leaks in mobile network traffic," in *Proc. the 14th Annual International Conference on Mobile Systems, Applications, and Services Companion*, p. 117, 2016.

[12] J. Ren, M. Lindorfer, D. J. Dubois, A. Rao, D. R. Choffnes, and N. Vallina-Rodriguez, "Bug fixes, improvements, ... and privacy leaks - A longitudinal study of PII leaks across android app versions," in *Proc. 25th Annual Network and Distributed System Security Symposium*, 2018.

[13] A. Broder, "On the resemblance and containment of documents," 06 1997.

[14] Y. Zhu, Z. Zhang, H. Hu, Y. Chen, and L. Zhang, "Appsniffer: Mobile application fingerprinting over vpn using encrypted traffic features," in *Proceedings of the Web Conference 2023 (WWW)*, 2023.

[15] J. Meira, C. Eiras-Franco, V. Bolón-Canedo, G. Marreiros, and A. Alonso-Betanzos, "Fast anomaly detection with locality-sensitive hashing and hyperparameter autotuning," *Information Sciences*, vol. 607, pp. 1245–1264, 6 2022.