If At First You Don't Succeed, Trie, Trie Again: Correcting TLSH Scalability Claims for Large-Dataset Malware Forensics

Jordi Gonzalez [a,*]

[a]The MITRE Corporation, 7525 Colshire Drive, McLean, 22102, VA, USA

Abstract

Malware analysts use Trend Micro Locality-Sensitive Hashing (TLSH) for malware similarity computation, nearest-neighbor search, and related tasks like clustering and family classification. Although TLSH scales better than many alternatives, technical limitations have limited its application to larger datasets. Using the Lean 4 proof assistant, I formalized bounds on the properties of TLSH most relevant to its scalability and identified flaws in prior TLSH nearest-neighbor search algorithms. I leveraged these formal results to design correct acceleration structures for TLSH nearest-neighbor queries. On typical analyst workloads, these structures performed one to two orders of magnitude faster than the prior state-of-the-art, allowing analysts to use datasets at least an order of magnitude larger than what was previously feasible with the same computational resources. I make all code and data publicly available.

Keywords: Locality-sensitive hashing, Malware analysis

1. Introduction

The growing volume of both malicious and benign software presents a growing burden to malware analysts and security vendors. They must accurately identify connections between malware samples while avoiding false associations between innocuous files and malware, and between unrelated malware families.

Locality-sensitive hashing (LSH) [1, 2] helps analysts solve this problem by providing a precise [3] dimensionality reduction technique, whereby more similar pieces of software have higher spatial proximity. This capability enables at-scale nearest-neighbor searches and, in turn, clustering [4, 5], antivirus whitelisting [6], detection [7, 8], malware campaign tracking [9], and confidentiality-preserving threat information sharing [10].

Trend Micro Locality-Sensitive Hashing [3, 11] (TLSH) emerged as a standard locality-sensitive hash function for malware analysis. However, efficiently searching for similar hashes in large TLSH datasets remains a challenge: even though it is possible to compute TLSH hashes rapidly for a set of inputs, and even though it is possible to compare different pairs of hashes rapidly, finding nearest-neighbors is computationally demanding and scales poorly with corpora size. The root of this issue is in the TLSH distance function, which violates the triangle inequality and, therefore, limits the use of metric data structures for nearest-neighbor searches. No correct, non-approximate, publicly available algorithm was found that addressed this gap. This work makes several contributions toward closing it.

First, this work uses the Lean 4 theorem prover [12] to provide formal, tight bounds on the triangle inequality violations for the TLSH distance function and its various subcomponents. This exposes an error in prior research that underestimated the bounds by a factor of over 20 (see Table 1).

Second, based on those theoretical results, this work presents two TLSH-specific nearest-neighbor acceleration structures: one, based on tries [13], and another, based on vantage-point (VP) trees [14, 15].

Third, this work evaluates the performance of these structures on real-world and synthetic data, demonstrating >10x throughput on common workloads compared to corrected versions of the prior state-of-the-art.

Finally, all code is made available at `https://github.com/mitre/fast-search-for-tlsh`.

2. Background

2.1. Design of TLSH

The TLSH whitepaper [3] accurately describes TLSH behavior. This subsection summarizes the aspects of the whitepaper most relevant to this work.

TLSH maps arbitrary byte streams to fixed-length hashes. The hashes are split into a "header" component and a "body" component. As illustrated in Figure 1, these two components have several subcomponents, or "features." The precise semantics of these features are unimportant to this work, but their layout is illustrated by Figure 1.

*Corresponding author
Email address: jgonzalez@mitre.org (Jordi Gonzalez)

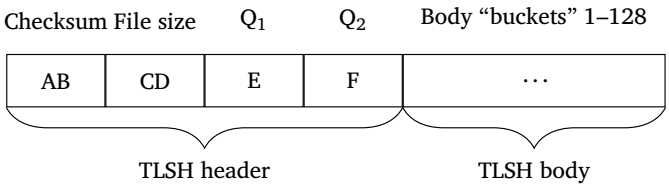


Figure 1: Structure of a TLSH hash.

Computing the distance between two samples using TLSH is a three-step process:

First, TLSH hashes are computed for both files. Second, each feature in one TLSH hash is compared against the corresponding feature in the other TLSH hash, and the feature distances are recorded. Finally, these distances are summed to produce the final TLSH distance. Figure 2 presents this visually.

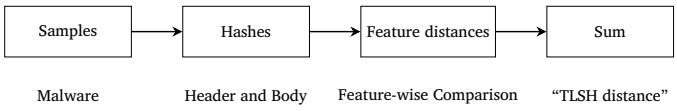


Figure 2: TLSH File Comparison Pipeline

The formulae used to compute feature distances are specific to the features being compared. For example, the formula for checksum (the first feature in Figure 1) distance can only output distances of zero or one.

Not mentioned in the whitepaper, but deeply relevant to this work, is that most of these formulae are not metrics in the mathematical sense because they do not all obey the triangle inequality. However, they are semi-metrics because they are all symmetric, non-negative, and only yield distances of zero when two features are the same.

The proofs of these claims are omitted for reasons of triviality: for example, all TLSH formulae accumulate distance by either adding modular distances (a counting-based distance metric [3]), positive constants, or absolute values of expressions [3]. As these all constitute natural numbers, and as the naturals are closed under addition, TLSH distances must also be naturals; and proof that TLSH distance violates the triangle inequality follows trivially from the proof that TLSH can violate the triangle inequality by 430 distance units, which I include in Appendix A, complete with constructive examples. Others have also made constructive (if not maximal) proofs available [16].

2.1.1. TLSH for malware forensics

TLSH possesses several extremely valuable attributes for malware forensics and analysis: TLSH is open and permissively licensed [11], so unlike proprietary LSH schemes, there are no limitations or costs associated with its access or usage. TLSH also performs well in comparative evaluations, where–relative to other high-uptake, fully general LSH schemes–it is both accurate, and robust against adversarial attacks [17]. Finally, as a corollary of doing well amongst high-uptake LSH schemes, TLSH benefits from strong network effects, having been adopted by platforms like VirusTotal.

These features are of unique and substantial value to analysts as they facilitate the sharing and broader use of TLSH hashes. For example, because VirusTotal adopted TLSH, analysts can conduct TLSH-similarity queries on the entire VirusTotal corpus [18].

2.1.2. TLSH limitation

Unfortunately, a TLSH technical limitation disrupts the viability of large-corpora TLSH nearest-neighbor queries. Indeed, “due to performance reasons”, VirusTotal throttles TLSH queries to a rate of 15 per minute for paying users and prohibits their use entirely for non-paying users. Moreover, VirusTotal’s TLSH indexing and querying algorithms are private, and a review of the literature found no public alternatives. This leaves no straightforward way for those in the industry to bear the cost of self-hosting a similar service. The lack of such tooling necessarily constrains analysts’ ability to use TLSH with large malware datasets.

This limitation stems from the fact that TLSH is only a semi-metric, not a metric: it violates the triangle inequality.

In a metric space, the triangle inequality suggests that if Alice and Bob are close, and if Bob and Eve are close, then one can use their closeness to bound the distance between Alice and Eve. If TLSH were a metric, VP trees [14] could enable efficient nearest-neighbor searches, because TLSH hashes could be laid out in such a way that these bounds can direct a search, and, in turn, enable pruning of large areas of the search space.

Because TLSH does not obey the triangle inequality, the use of metric-based techniques for nearest-neighbor searches is limited. Analysts ostensibly must either conduct *exhaustive linear scans* of the entirety of a dataset, every time that a query is conducted; or analysts must use *approximate nearest-neighbor search* and sacrifice precision, recall, or both.

2.2. Related Work

2.2.1. Research into TLSH scalability

The official TLSH documentation contradicts the claims of this paper, noting that “TLSH is very fast at nearest-neighbor search at scale [.] being a distance metric (as per the mathematical definition) and hence has logarithmic search times [.] and in particular [obeys] the triangle inequality” [19]. However, this is misleading, as the authors now publicly acknowledge that “the [TLSH] distance function does not obey the triangle inequality” [20] and is only “an approximate distance metric” [21].

Prior work to improve TLSH’s scalability has had varying degrees of applicability. For instance, Trend Micro Research documented one improvement to *the use* of TLSH for clustering, centered around the parallelization of a clustering algorithm [22]. While this can improve throughput on a system in (at best) direct proportion to the amount

of available system parallelism, it does not solve the high computational costs of TLSH queries themselves; rather, it just allows that cost to be distributed across more CPU cores.

Other Trend Micro Research work presented two different acceleration structures for TLSH nearest-neighbor search: a random forest and a VP tree [4]. This work avoids the former because it only gives approximate results, as noted in the prior work. The latter, as was also noted, is a path toward an exact algorithm, making it highly relevant. This work builds off their VP tree concept.

A careful reader may note that VP trees were previously implied to be incompatible with non-metric distance functions. The prior work attempts to address this based on the premise that because TLSH is still "within a constant of a metric" [21], and because "distance functions that are within a constant [factor] of a metric" can still be accelerated, a special VP tree can still work [4, 21] if it considers additional nodes. The flaw with that work is in the implementation: the authors assume that the constant factor is 20 [23], which is incorrect. To be correct, the authors should have used 430 (see Table 1). Unfortunately, applying this correction reduces the performance of their implementation to that of unaccelerated, linear scanning (see Figure 3).

2.2.2. Non-TLSH alternatives

Several mainstream LSH schemes, like ssdeep [24] and sdhash [25], can be used as TLSH alternatives; however, these fail to solve the underlying scalability problem and, in comparative evaluations, perform much worse than TLSH in terms of accuracy and resistance to adversarial manipulation [3, 26–28].

Other LSH schemes map to proper metric spaces [29, 30], theoretically enabling efficient nearest-neighbor search. However, these have not gained widespread adoption, limiting opportunities for their use.

Within digital malware forensics specifically, several LSH schemes exist as specialized algorithms, like PermHash for Chromium extensions [31], or peHash for PE files [32]. These specializations can be highly constraining: peHash, for example, will never identify relationships in code similarity between Mac, Windows, and Linux payloads, as ELF files are not PE files, and neither are Mach-O files.

While many non-LSH techniques exist [2], none were identified in the literature with the compactness [3], throughput [2, 3, 33], generality [3], and wide uptake of LSH schemes.

3. Methods

This research is divided into three sections: theoretical analysis to formalize TLSH's triangle inequality violations, algorithm design to exploit the theoretical results, and comparison of the devised algorithms to alternatives.

3.1. Theory

I first proved bounds on the degree to which TLSH subcomponents could violate the triangle inequality, formalizing the proof using the Lean v4.14.0 [12] proof-assistant and mathematical library [34].

Specifically, letting H_1, H_2, H_3 be any three distinct TLSH hashes, and letting $d(x, y)$ represent the contribution of a feature of TLSH to the total TLSH distance, I solved for the tightest bounds on c in the triangle inequality, $d(H_1, H_3) \leq d(H_1, H_2) + d(H_2, H_3) + c$.

Building on those results, I showed bounds for the TLSH distance function and its constituent header and body components. For the proof itself, see Appendix A.

3.2. Algorithm Design

I implemented and tested the following algorithms. For each, Appendix B includes pseudocode.

3.2.1. VP tree

I implemented a VP tree [14, 15], inspired by prior work [21].

Unlike existing methods, this VP tree exclusively indexes the TLSH header component, which, in theory, confers two advantages over indexing the entirety of a TLSH hash: faster VP tree index construction because body feature distance never needs to be computed; and faster VP tree queries because header component distance violates the triangle equality to a much lesser degree than body component distance, which allows for more aggressive pruning during searches.

Because body component distance still matters, it is checked before a candidate is added to the resulting nearest-neighbor list; if the sum of header and body distance for two hashes is outside the "cutoff" for what defines a nearby-neighbor, it is pruned.

3.2.2. Trie with schema-learning

I implemented a novel, trie-based [13] approach to nearest-neighbor queries. Rather than search for nearby neighbors in the order that TLSH features are laid out in a hash, it uses a greedy, randomized algorithm to find an ordering, or "schema," that maximizes the number of nodes pruned at shallow levels of the search tree.

Trie search follows the ordering laid out in the schema. Helpfully, schemas can be saved and reused on different datasets. In trie search, the proofs of each TLSH feature's contributions to the total triangle inequality violation are used to constrain the search radius.

3.2.3. Replication of prior work

I implemented and evaluated the VP tree design described by Trend Micro Research [23] in Rust in order to evaluate performance relative to prior work.

Concerning the correctness issue outlined in Section 2.2.1, I tested "corrected" and "uncorrected" versions of their algorithm.

3.2.4. Linear scanning

Linear scanning served as a control.

Though largely outside the scope of the paper, the source code for this work includes a parallel, Tensorflow-accelerated [35] Python library for working with TLSH on large datasets. Appendix C covers the performance of its linear scanning routine.

3.3. Empirical Evaluation

3.3.1. Datasets

I evaluated performance on two datasets: randomly generated synthetic TLSH hashes, and a convenience sample of 1,263,016 TLSH hashes sourced from the VirusTotal metadata for a local repository of VirusShare [36] data.

The latter dataset represents the entirety of an employer-maintained collection of data. Data was not filtered prior to or during the evaluation. Furthermore, and in the interest of transparency, the data is provided in the source code associated with the paper (see Appendix A).

3.3.2. Workloads

This work examined three groups of nearest-neighbor search workloads, based on common analytical tasks.

"Near" neighbors were defined as those with a TLSH distance of at most 30, as this is a standard industry and academic choice [37, 38] with TLSH. Larger cutoffs trade precision for recall [3].

1. All-to-all workloads:
   An analyst may receive a set of samples to triage and want to cluster them.
   To approximate inefficient clustering algorithms, I do pairwise comparisons of every sample in corpora of sizes 5,000 and 10,000.
2. Fixed-query-size, variable-corpora-size workloads:
   An analyst may receive a set of samples to triage and want to check which are novel.
   To approximate this task, I measured the time it took to conduct 1,000 queries on various-sized subsets of a larger corpus. The queries were chosen randomly from the larger corpus rather than its subset.
3. Variable-query-size, variable-corpora-size workloads:
   An analyst may use TLSH and a specialized clustering algorithm requiring very few comparisons.
   To approximate more efficient clustering algorithms, I measured the time it took to query a random 10% of corpora of various sizes. The sampling procedure is the same as for the fixed-query-size, variable-corpora-size workload.

To assess whether the choice of 30 as a distance cutoff biased the results and to capture alternative workloads, I evaluated cutoffs from 1 to 1,000. Note that at cutoffs above 200, TLSH may exceed a false-positive rate of 50% [3], so measurements past that point are of questionable utility.

To assess whether aspects of these workloads were "shifted" from query-time to preprocessing-time in a way that might prove too computationally demanding or wasteful for specific tasks, I also evaluated the time spent building the acceleration structures.

Note that because a single schema may be reused with different corpora, the choice to do schema-learning (rather than reuse a single schema) represents a trade-off between data structure efficiency and preprocessing time. Although this has little influence on results, I show measurements of trie preprocessing with and without schema-learning.

3.3.3. Benchmark environment

I ran all benchmarks 11 times on a 12-core M2 Max MacBook Pro using high-performance mode on macOS 15.2. Reported results represent median execution times. The testing harness and associated algorithms were built with Rust 1.83.0. The benchmark suite recorded CPU clock speeds using the sysinfo crate [39]. The records showed no indications of throttling.

4. Results

4.1. Theory

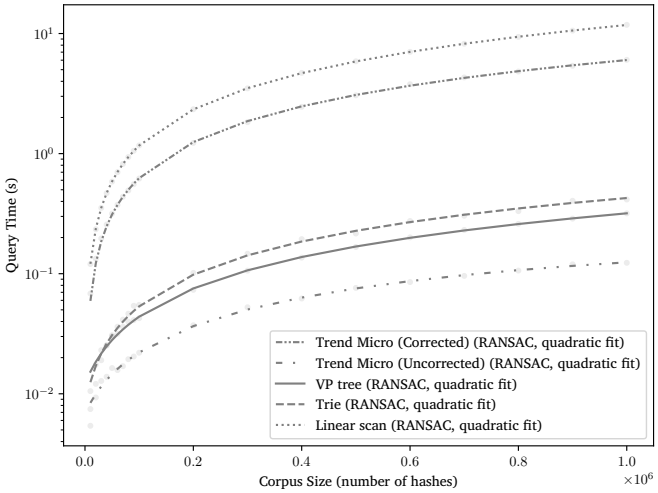
I quantified the contribution of different TLSH features to the total violation of the triangle inequality. For information regarding the proof, please see Appendix A.

Component	Violation
Header Distance Violations	
Checksum-distance	0
L-distance	22
q_1-distance	12
q_2-distance	12
Total Header Distance	**46**
Body Distance Violations	
Per-bucket body distance	3
Total Body Distance (128 buckets)	**384**
Overall Violation	**430**

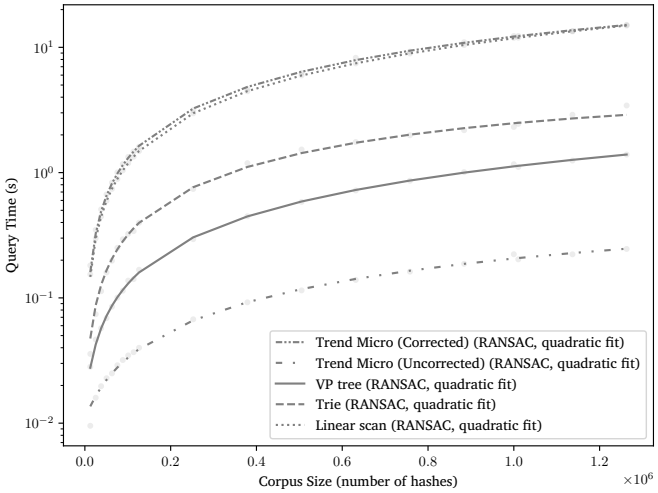
Table 1: TLSH triangle inequality distance violations

4.2. Scaling by Workload and Data Source

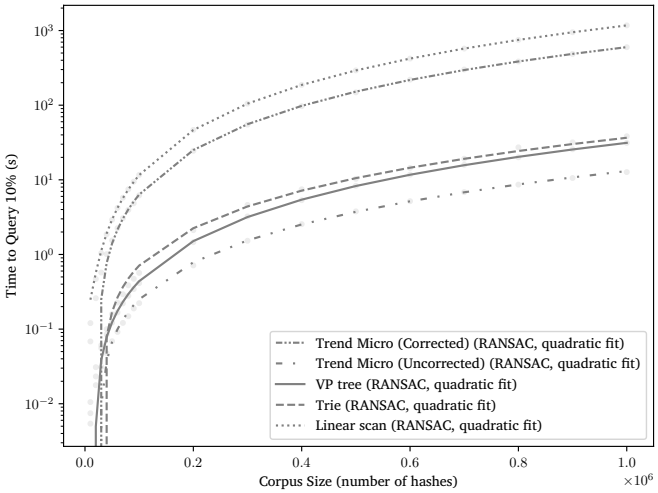
Figure 3 shows algorithm performance when the cutoff for similarity is 30 and corpora sizes are varied. Figure 4 shows the performance of the algorithms when only the cutoff for similarity is varied.



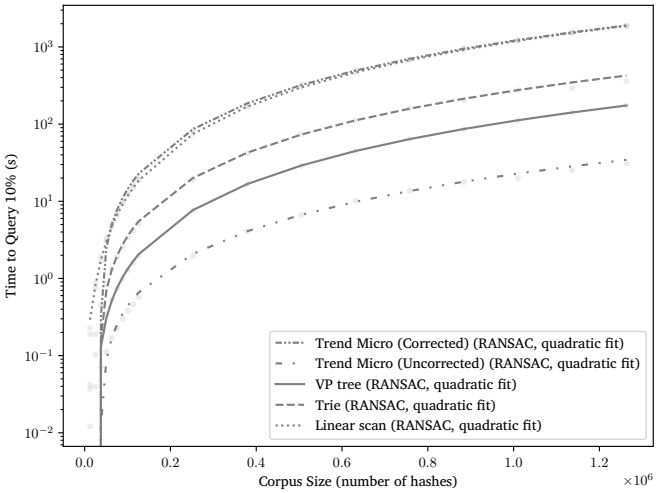
(a) Relationship between corpus size and time to query 1,000 hashes in randomly generated hash corpora.



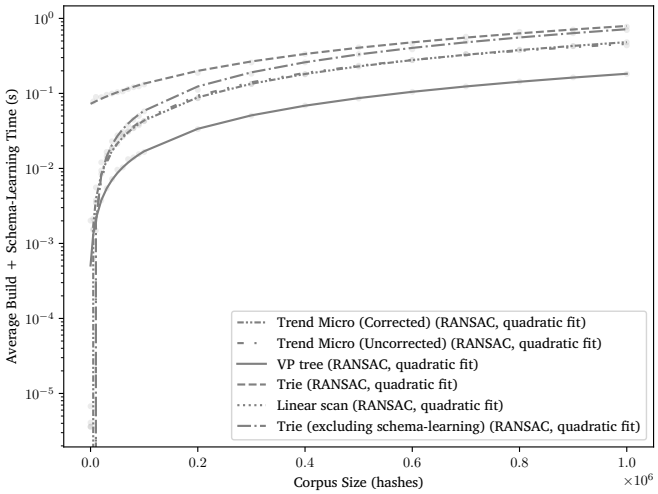
(b) Relationship between corpus size and time to query 1,000 hashes in VirusShare-derived hash corpora.



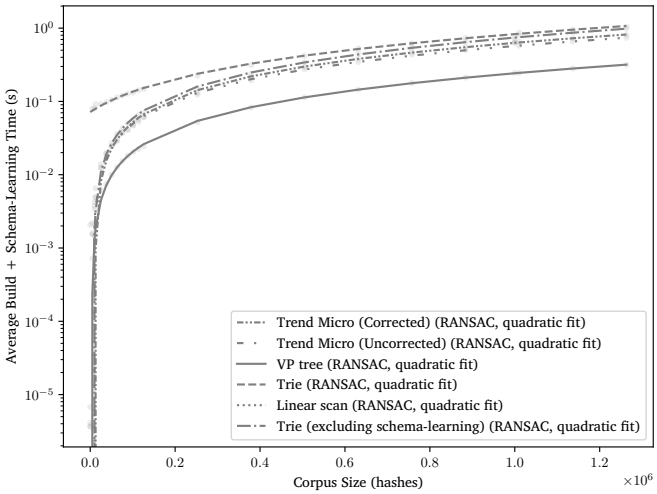
(c) Relationship between corpus size and total build and query time, when querying 10% of randomly generated hash corpora.



(d) Relationship between corpus size and total build and query time, when querying 10% of VirusShare-derived hash corpora.

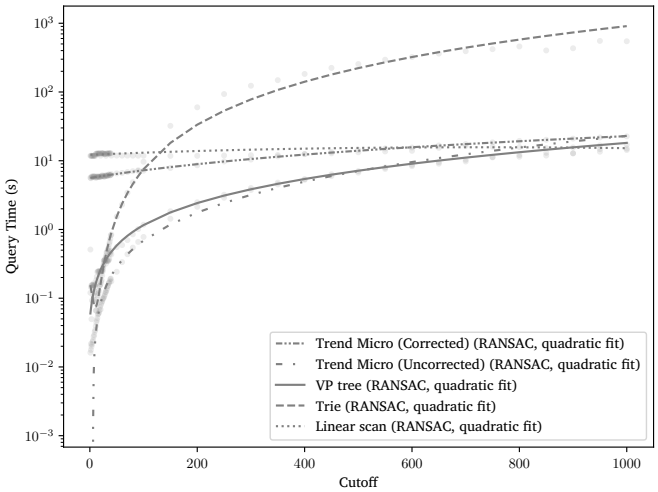


(e) Relationship between corpus size and index build time for randomly generated hashes.

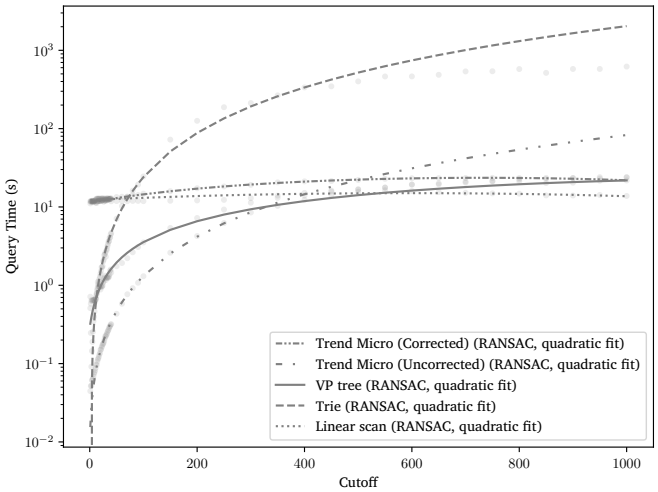


(f) Relationship between corpus size and index build time for VirusShare-derived hashes.

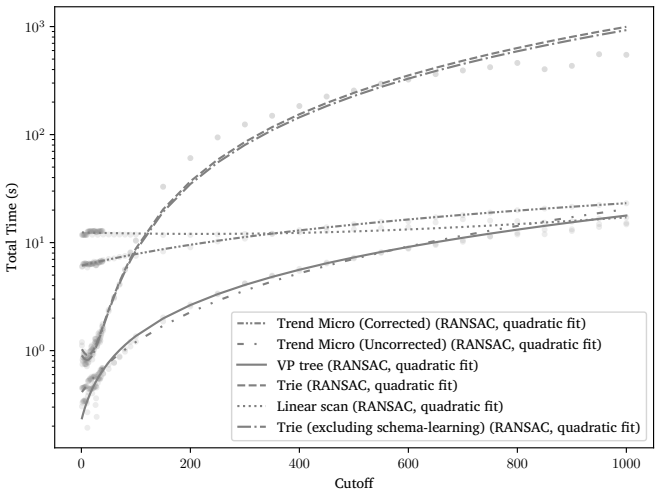
Figure 3: Assorted performance metrics for the various algorithms with cutoff of 30.



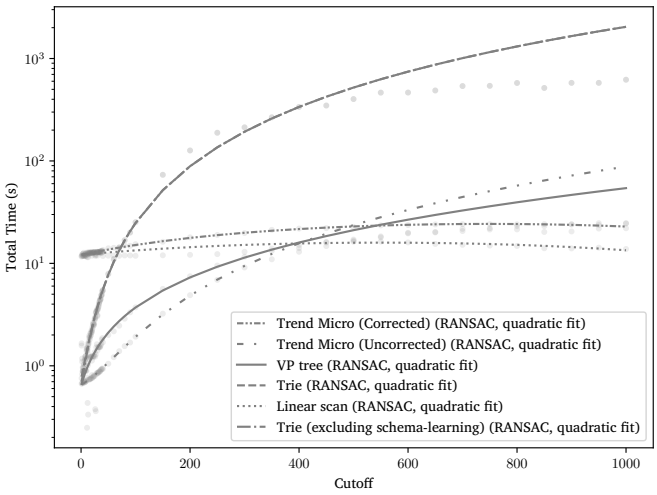
(a) Relationship between cutoff and time to query 1,000 of 1,000,000 random hashes.



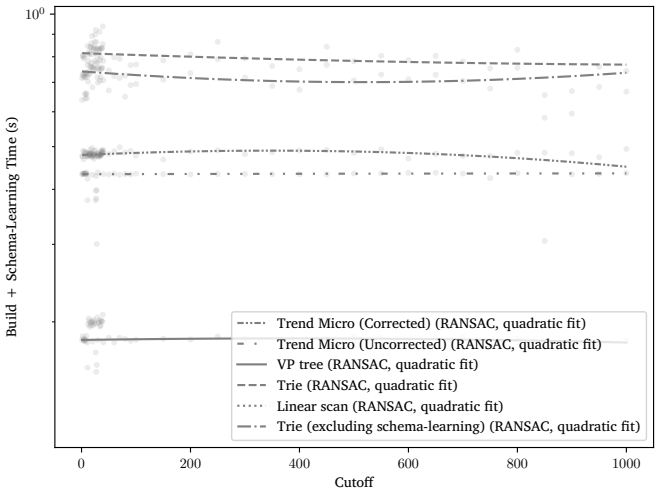
(b) Relationship between cutoff and time to query 1,000 of 1,000,000 VirusShare-derived hashes.



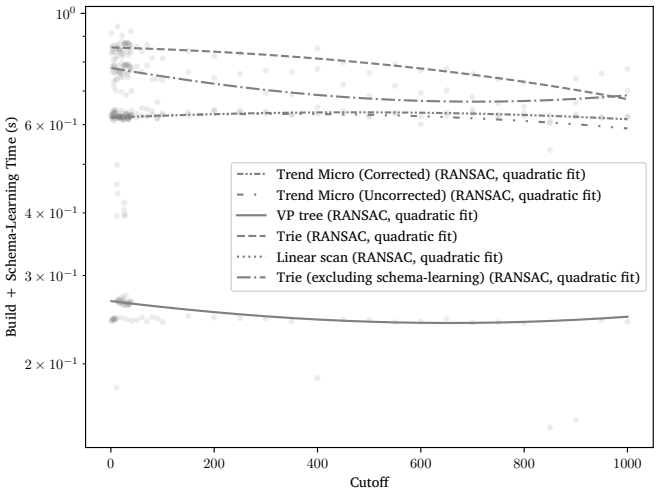
(c) Relationship between cutoff and total build and query time for 1,000 queries of 1,000,000 random hashes.



(d) Relationship between cutoff and total build and query time for 1,000 queries of 1,000,000 VirusShare-derived hashes.



(e) Relationship between cutoff and build time (with and without schema-learning) for 1,000,000 random hashes.



(f) Relationship between cutoff and build time (with and without schema-learning) for 1,000,000 VirusShare-derived hashes.

Figure 4: Effect of different cutoff levels on algorithm performance.

4.3. Fixed-Workload Results

Both indices outperformed linear scanning on the 10,000-to-10,000, cutoff-of-30, clustering-like workloads. The VP tree was strictly faster than the trie on real-world data.

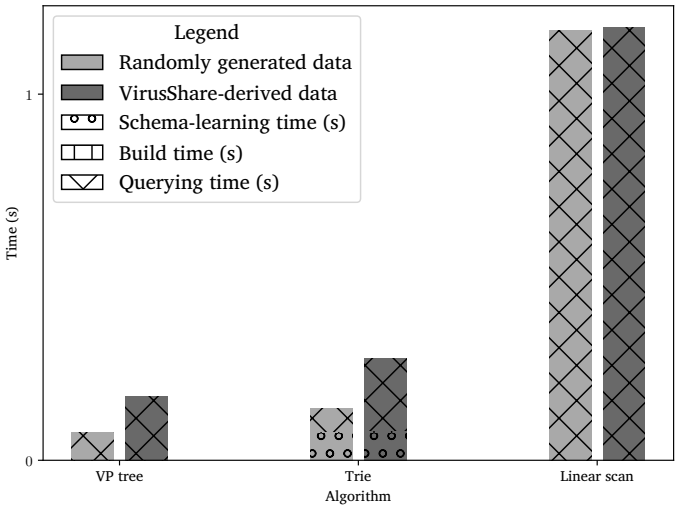


Figure 5: Median profile of an all-to-all workload involving a 10,000-hash corpus, by data source.

Results were similar on 1,000-to-1,000,000 workloads.

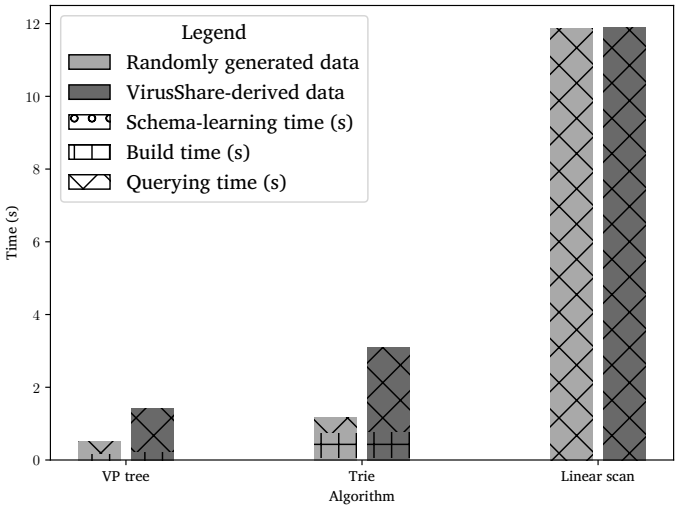


Figure 6: Median profile of a 1,000-to-1,000,000 workload, by data source.

5. Discussion

The results make for several substantial contributions to the TLSH scalability literature.

The analysis of TLSH's triangle inequality violations was the most relevant to prior literature because it demonstrated errors in prior work. Table 1 showed that the triangle inequality could be violated by a constant factor of 430, far exceeding the prior estimate of 20 [23]. After correcting this error in the prior work, its performance degraded to that of linear scanning (see Figure 3b).

The results on synthetic data were much better for the acceleration structures than the results on real-world data. This discrepancy likely stems from the tighter distribution of TLSH features in real-world data. For example, file length–a key TLSH feature–has much lower variance in real-world datasets than in randomly generated data. This tighter clustering means hashes have more plausibly nearby neighbors, reducing opportunities for pruning during queries. This analysis focuses on real-world data to avoid drawing biased conclusions.

5.1. VP Tree

Compared to the corrected prior work and linear scanning, I see much better performance–over an order of magnitude greater throughput at the standard cutoff of 30–with the presented VP tree. The uplift was in both index construction (Figure 3f) and querying (Figure 3d), due to reduced computational overhead during construction (as body distance does not get used) and more aggressive pruning during searches, respectively.

The VP tree demonstrated performance advantages during index construction at all cutoff levels and against all tested data structures (see Figure 4f).

Although the VP tree underperformed linear scanning at extremely wide similarity cutoffs, at every cutoff below 200–when $\geq 50\%$ false positive rates are known to occur [3] and broader cutoffs are likely to be impractical–the VP tree was the fastest technique (see Figure 4d).

The advantage was particularly pronounced in Figure 4b, which represents cases where index construction times amortize away; here, the VP tree was the best performing up to cutoffs of approximately 400.

5.2. Trie

I diverged from prior work by introducing a novel, trie-based VP tree alternative for TLSH nearest-neighbor search. At very strict cutoffs, the trie outperforms the VP tree and the prior work by an order of magnitude, but this advantage quickly vanishes at higher cutoff values. At cutoffs above 10, the VP tree shows a consistent performance advantage, and at cutoffs above 100, the trie underperforms linear scanning. This stark regression is likely because, with each query, the trie performs an exhaustive breadth-first search. It is only because of aggressive pruning, which requires tight cutoffs, that trie search is performant.

5.3. Linear scanning

Because linear scanning performance is unaffected by cutoff (see Figure 4d), linear scanning may be advantageous in certain ultra-high-cutoff use cases. On smaller corpora, even when linear scanning is slower than querying indexed structures, highly demanding workloads like all-to-all queries can still be completed in seconds (see Figure 5). Consequently, there may be cases where linear scanning is preferable to indexed searches for speed or convenience.

Nevertheless, linear scanning performance degrades quickly with larger corpora and more queries. Figure 6 demonstrates this limitation, showing a large performance gap between the index-based algorithms and linear scanning.

5.4. Implications

The results were highly pronounced on clustering workloads, where the best algorithm–the VP tree–delivered a 10× performance uplift compared to the state-of-the-art (see Figure 3b). Though difficult to see on a log scale, performance *scaling* was also much better with the VP tree than with the prior work.

Unlike clustering workloads, where the number of nearest-neighbor queries grows with the size of the dataset, malware corpus search APIs like VirusTotal's "Advanced Corpus Search" [18] represent workloads that consist entirely of a single nearest-neighbor query. For these APIs and workloads, performance improvements for queries directly translate to increased API or analyst capacity: at a cutoff of 30, compared to the prior state-of-the-art, a nearest-neighbor search API using the VP tree can either dispatch ten times as many queries or query a dataset ten times the size, with the same compute budget.

6. Conclusion

I used a formal proof assistant to demonstrate fundamental limitations in TLSH and in prior TLSH nearest-neighbor search implementations. I then leveraged the formal results to design two data structures that could overcome these limitations: one based on a vantage-point tree, and another based on a trie-like structure.

I found that on real-world data, for nearest-neighbor querying tasks and associated clustering workloads, these algorithms provided one to two orders of magnitude greater throughput relative to the state of the art. Except at the most stringent cutoffs for what constitutes a "near neighbor," the VP tree was the fastest of the two data structures. Accordingly, this performance improvement allows analysts to process datasets an order of magnitude larger than what was previously feasible with the same resources.

7. Acknowledgments

I want to acknowledge Corvus Forensics for maintaining the VirusShare dataset, which made it viable to test on real-world data. I would also like to thank MITRE and MITRE's sponsors for sponsoring this research, particularly Brian Shaw, Christopher Andersen, Dan Perret, Dr. Justin Brunelle, Frank Posluszny, Laurence Hunter, Morgan Keiser, and Tim McNevin.

I want to give additional thanks to Brian Shaw, Christopher Andersen, Dr. Justin Brunelle, and Tim McNevin for their invaluable constructive criticism; Frank Posluszny, for his feedback and for providing me with the malware metadata from which the real-world TLSH digests were extracted; and Laurence Hunter, for his mentorship, generosity, and feedback, without which this could not have been written.

This software (or technical data) was produced for the U.S. Government and is subject to the Rights in Data-General Clause 52.227-14, Alt. IV (May 2014) – Alternative IV (Dec 2007).

References

[1] P. Indyk, R. Motwani, Approximate nearest neighbors: Towards removing the curse of dimensionality, in: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, STOC '98, Association for Computing Machinery, New York, NY, USA, 1998, pp. 604–613. doi:10.1145/276698.276876.

[2] I. U. Haq, J. Caballero, A Survey of Binary Code Similarity, ACM Comput. Surv. 54 (3) (2021) 51:1–51:38. doi:10.1145/3446371.

[3] J. Oliver, C. Cheng, Y. Chen, TLSH – A Locality Sensitive Hash, in: 2013 Fourth Cybercrime and Trustworthy Computing Workshop, 2013, pp. 7–13. doi:10.1109/CTC.2013.9.

[4] J. Oliver, M. Ali, H. Liu, J. Hagen, Fast Clustering of High Dimensional Data Clustering the Malware Bazaar Dataset (2021). URL https://tlsh.org/papersDir/n21_opt_cluster.pdf

[5] M. Bak, D. Papp, C. Tamás, L. Buttyán, Clustering IoT Malware based on Binary Similarity, in: NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium, 2020, pp. 1–6. doi:10.1109/NOMS47738.2020.9110432.

[6] Smart Whitelisting Using Locality Sensitive Hashing (Mar. 2017). URL https://www.trendmicro.com/en_us/research/17/c/smart-whitelisting-using-locality-sensitive-hashing.html

[7] M. T. Intelligence, Combing through the fuzz: Using fuzzy hashing and deep learning to counter malware detection evasion techniques (Jul. 2021). URL https://www.microsoft.com/en-us/security/blog/2021/07/27/combing-through-the-fuzz-using-fuzzy-hashing-and-deep-learning-to-counter-malware-detection-evasion-techniques/

[8] N. Naik, P. Jenkins, N. Savage, A Ransomware Detection Method Using Fuzzy Hashing for Mitigating the Risk of Occlusion of Information Systems, in: 2019 International Symposium on Systems Engineering (ISSE), 2019, pp. 1–6. doi:10.1109/ISSE46696.2019.8984540.

[9] N. Naik, P. Jenkins, N. Savage, L. Yang, Cyberthreat Hunting - Part 2: Tracking Ransomware Threat Actors using Fuzzy Hashing and Fuzzy C-Means Clustering, in: 2019 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), IEEE, New Orleans, LA, USA, 2019, pp. 1–6. doi:10.1109/FUZZ-IEEE.2019.8858825.

[10] A. Almahmoud, E. Damiani, H. Otrok, Hash-Comb: A Hierarchical Distance-Preserving Multi-Hash Data Representation for Collaborative Analytics, IEEE Access 10 (2022) 34393–34403. doi:10.1109/ACCESS.2022.3158934.

[11] J. Oliver, GitHub - trendmicro/tlsh (Apr. 2024). URL https://github.com/trendmicro/tlsh

[12] L. de Moura, S. Ullrich, The Lean 4 Theorem Prover and Programming Language, in: A. Platzer, G. Sutcliffe (Eds.), Automated Deduction – CADE 28, Springer International Publishing, Cham, 2021, pp. 625–635. doi:10.1007/978-3-030-79876-5_37.

[13] E. Fredkin, Trie memory, Commun. ACM 3 (9) (1960) 490–499. doi:10.1145/367390.367400.

[14] P. N. Yianilos, Data structures and algorithms for nearest neighbor search in general metric spaces, in: Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '93, Society for Industrial and Applied Mathematics, USA, 1993, pp. 311–321.

[15] J. K. Uhlmann, Satisfying general proximity / similarity queries with metric trees, Information Processing Letters 40 (4) (1991) 175–179. doi:10.1016/0020-0190(91)90074-R.

[16] D. Baggett, TLSH distance metric appears to violate triangle inequality (Mar. 2023).
URL https://github.com/trendmicro/tlsh/issues/130#issue-1623514292

[17] J. Oliver, J. Hagen, Designing the Elements of a Fuzzy Hashing Scheme, in: 2021 IEEE 19th International Conference on Embedded and Ubiquitous Computing (EUC), IEEE, Shenyang, China, 2021, pp. 1–6. doi:10.1109/EUC53437.2021.00028.

[18] VirusTotal, Advanced corpus search (Jul. 2024).
URL https://docs.virustotal.com/reference/intelligence-search

[19] J. Oliver, TLSH - Technical Overview (Apr. 2021).
URL https://tlsh.org/papers.html

[20] J. Oliver, TLSH distance metric appears to violate triangle inequality (Jan. 2024).
URL https://github.com/trendmicro/tlsh/issues/130#issuecomment-1906886178

[21] J. Oliver, M. Ali, J. Hagen, HAC-T and Fast Search for Similarity in Security, in: 2020 International Conference on Omni-layer Intelligent Systems (COINS), 2020, pp. 1–7. doi:10.1109/COINS49042.2020.9191381.

[22] M. Ali, J. Hagen, J. Oliver, Scalable Malware Clustering using Multi-Stage Tree Parallelization, in: 2020 IEEE International Conference on Intelligence and Security Informatics (ISI), IEEE, Arlington, VA, USA, 2020, pp. 1–6. doi:10.1109/ISI49825.2020.9280546.

[23] J. Oliver, Tlsh/tlshCluster/pylib/hac_lib.py (Sep. 2021).
URL https://github.com/trendmicro/tlsh/blob/96536e3f5b9b322b44ce88d36126121685e45a77/tlshCluster/pylib/hac_lib.py#L143

[24] J. Kornblum, H. Grohne, T. OI, Ssdeep (Nov. 2017).
URL https://github.com/ssdeep-project/ssdeep

[25] V. Roussev, Data Fingerprinting with Similarity Digests, in: K.-P. Chow, S. Shenoi (Eds.), Advances in Digital Forensics VI, Springer, Berlin, Heidelberg, 2010, pp. 207–226. doi:10.1007/978-3-642-15506-2_15.

[26] J. Oliver, S. Forman, C. Cheng, Using Randomization to Attack Similarity Digests, in: L. Batten, G. Li, W. Niu, M. Warren (Eds.), Applications and Techniques in Information Security, Springer, Berlin, Heidelberg, 2014, pp. 199–210. doi:10.1007/978-3-662-45670-5_19.

[27] F. Pagani, M. Dell'Amico, D. Balzarotti, Beyond Precision and Recall: Understanding Uses (and Misuses) of Similarity Hashes in Binary Analysis, in: Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 354–365. doi:10.1145/3176258.3176306.

[28] A. Azab, R. Layton, M. Alazab, J. Oliver, Mining Malware to Detect Variants, in: 2014 Fifth Cybercrime and Trustworthy Computing Conference, 2014, pp. 44–53. doi:10.1109/CTC.2014.11.

[29] X. Gu, Y. Zhang, L. Zhang, D. Zhang, J. Li, An improved method of locality sensitive hashing for indexing large-scale and high-dimensional features, Signal Processing 93 (8) (2013) 2244–2255. doi:10.1016/j.sigpro.2012.07.014.

[30] C. Oprişa, M. Checicheş, A. Nändrean, Locality-sensitive hashing optimizations for fast malware clustering, in: 2014 IEEE 10th International Conference on Intelligent Computer Communication and Processing (ICCP), 2014, pp. 97–104. doi:10.1109/ICCP.2014.6936960.

[31] Jared Wilson, Permhash — No Curls Necessary (May 2023).
URL https://cloud.google.com/blog/topics/threat-intelligence/permhash-no-curls-necessary

[32] G. Wicherski, peHash: A Novel Approach to Fast Malware Clustering, in: USENIX Workshop on Large-Scale Exploits and Emergent Threats, 2009.
URL https://www.semanticscholar.org/paper/peHash%3A-A-Novel-Approach-to-Fast-Malware-Clustering-Wicherski/a52ddc15377bc9f2ef1f237afa41d324f321bb9b

[33] Y. Li, J. Jang, X. Ou, Topology-Aware Hashing for Effective Control Flow Graph Similarity Analysis, in: S. Chen, K.-K. R. Choo, X. Fu, W. Lou, A. Mohaisen (Eds.), Security and Privacy in Communication Networks, Springer International Publishing, Cham, 2019, pp. 278–298. doi:10.1007/978-3-030-37228-6_14.

[34] The mathlib community, The Lean Mathematical Library, in: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020, 2020, pp. 367–381. doi:10.1145/3372885.3373824.

[35] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, Xiaoqiang Zheng, TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems (2015).
URL https://www.tensorflow.org/

[36] VirusShare.com (Sep. 2024).
URL https://virusshare.com/

[37] P. Hutelmyer, R. Borre, Implementing TLSH Based Detection to Identify Malware Variants (Dec. 2024).
URL https://tech.target.com/blog/implementing_TLSH_based_detection

[38] R. J. Joyce, T. Patel, C. Nicholas, E. Raff, AVScan2Vec: Feature Learning on Antivirus Scan Data for Production-Scale Malware Corpora, in: Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security, AISec '23, Association for Computing Machinery, New York, NY, USA, 2023, pp. 185–196. doi:10.1145/3605764.3623907.

[39] Sysinfo - crates.io: Rust Package Registry (Dec. 2024).
URL https://crates.io/crates/sysinfo

[40] Py-tlsh: TLSH (C++ Python extension) (Sep. 2024).
URL https://github.com/trendmicro/tlsh

[41] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T. E. Oliphant, Array programming with NumPy, Nature 585 (7825) (2020) 357–362. doi:10.1038/s41586-020-2649-2.

Appendix A. Supplemental Material

All supplemental material can be found at https://github.com/mitre/fast-search-for-tlsh.

Appendix B. Pseudocode

All algorithms take in a data structure holding a corpus, a query, and a radius used as the cutoff for similarity search.

The VP tree is constructed as any other: assume every node in the tree is a vantage-point with radius of size `node.threshold`. Little improvement was observed with different vantage-point selection strategies. The approach described as "prior literature" differs only in 430 being used as `MAX_HEADER_VIOLATION`, and with the total distance being used instead of header distance.

Algorithm 1 Query logic for VP tree

```
function RANGE_QUERY(tree, query, radius)
    results ← [ ]
    stack ← [tree.root]
    new_radius ← radius + MAX_HEADER_VIOLATION
    while stack not empty do
        node ← stack.pop()
        header_dist ← HEADER_DIST(node.point, query)
        body_dist ← BODY_DIST(node.point, query)
        if header_dist + body_dist ≤ radius then
            append node.point to results
        if header_dist - new_radius ≤ node.threshold then
            append node.left to stack
        if header_dist + new_radius ≥ node.threshold then
            append node.right to stack
    return results
```

The trie has both a searching component and a schema-learning component.

Algorithm 2 Trie schema-learning logic

```
function LEARN_SCHEMA(data, cutoff)
    sample_n ← MIN(64, |data|)
    sampled_data ← randomly sample sample_n from data
    schema ← [ ]
    max_cutoffs ← 0
    loop
        best_feature ← null
        best_num_cutoffs ← max_cutoffs
        best_feature_by_sum ← null
        best_sum ← 0
        feature_range ← 0..36 ▷ 36 = # TLSH features
        for each i in feature_range do
            if schema contains i then
                continue
            trial ← schema + [i]
            num_cutoffs ← 0
            total_sum ← 0
            for each v in sampled_data do
                pair_cutoffs ← 0
                pair_sum ← 0
                for each u in sampled_data do
                    diff ← FEATURE_DIST(u, v, trial)
                    if diff ≥ cutoff then
                        pair_cutoffs ← pair_cutoffs + 1
                    pair_sum ← pair_sum + diff
                num_cutoffs ← num_cutoffs + pair_cutoffs
                total_sum ← total_sum + pair_sum
            if total_sum > best_sum then
                best_sum ← total_sum
                best_feature_by_sum ← i
            if num_cutoffs > best_num_cutoffs then
                best_num_cutoffs ← num_cutoffs
                best_feature ← i
        if best_feature not null then
            append best_feature to schema
        else if best_feature_by_sum not null then
            append best_feature_by_sum to schema
        else
            break
        max_cutoffs ← best_num_cutoffs
    return schema
```

Algorithm 3 Trie query logic

```
function TRIE_SEARCH(node, query, radius, schema)
    if node is a leaf then
        return FILTER(node.points, query, radius, schema) ▷ Filter out would-be false-positives using remaining features
    results ← [ ]
    feature ← first feature in schema
    value ← GET_FEATURE_VALUE(query, feature)
    max_diff ← COMPUTE_MAX_DIFF(feature, radius)
    for all possible candidate value s.t. candidate diff ∈ [0, max_diff] do ▷ diff > max_diff → dist > radius
        dist ← FEATURE_DIFF(value, candidate, feature)
        if dist ≤ radius then
            child ← node[candidate]
            radius_budget ← radius - dist
            schema' ← TAIL(schema)
            sub_results ← TRIE_SEARCH(child, query, radius_budget, schema')
            for each sub_result in sub_results do append sub_result to results
    return results
```

For completeness, `linear_scan` is as follows:

Algorithm 4 Linear scan

```
function LINEAR_SCAN(corpus, query, radius)
    results ← [ ]
    for digest in corpus do
        header_dist ← HEADER_DIST(digest, query)
        body_dist ← BODY_DIST(digest, query)
        if header_dist + body_dist ≤ radius then
            append digest to results
    return results
```

Appendix C. Tensorflow-accelerated scanner

I authored a Tensorflow-accelerated linear scanning routine to facilitate using TLSH on large datasets, within Python notebooks. I benchmarked it against a linear scanning routine powered by py-tlsh [40], the official C++ Python extension for fast TLSH operations.

The benchmarks used Rust 1.83.0, Python 3.12.7, Tensorflow 2.18.0 [35], numpy 2.0.1 [41], and py-tlsh 4.7.2 [40].

I included Rust linear scanner performance for illustrative purposes only, as the Rust and Python benchmark harnesses differ, limiting result comparability.

The benchmark results were as follows:

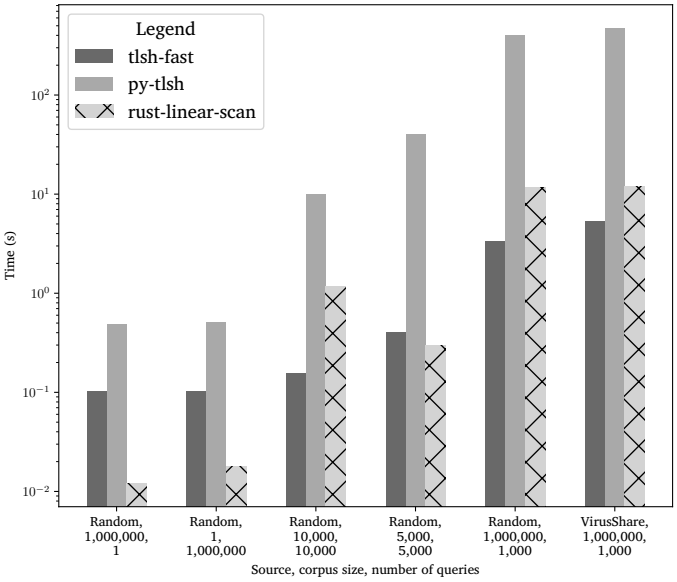


Figure C.7: Performance of linear-scan implementations.

The Tensorflow-enhanced Python library outperformed the py-tlsh-based scanner across all measured workloads.

I attribute a roughly 10x performance uplift, relative to the py-tlsh-based scanner, to parallelism; and the residual performance uplifts to two factors: that Tensorflow optimizes memory access patterns, particularly on all-to-all tasks; and that Tensorflow batches queries, which reduces the time spent in the Python interpreter.

The benchmark code is available in Appendix A.