Towards Application of Cuckoo Filters in Network Security Monitoring

Jan Grashöfer, Florian Jacob, Hannes Hartenstein
Institute of Telematics
Karlsruhe Institute of Technology
jan.grashoefer@kit.edu, florian.jacob@student.kit.edu, hannes.hartenstein@kit.edu

Abstract—**In this paper, we study the feasibility of applying the recently proposed cuckoo filters to improve space efficiency for set membership testing in Network Security Monitoring, focusing on the example of Threat Intelligence matching. We present conceptual insights for the practical application of cuckoo filters and provide a cuckoo filter implementation that allows runtime configuration. To evaluate the practical applicability of cuckoo filters, we integrate our implementation into the Bro Network Security Monitor, compare it to traditional data structures and conduct a brief operational evaluation. We find that cuckoo filters allow remarkable memory savings, while potential performance trade-offs, caused by introducing false positives, have to be carefully evaluated on a case-by-case basis.**

I. Introduction

Set membership testing is a common operation in Network Security Monitoring (NSM). Determining whether a given datum is known to be of special relevance or has been seen before are prevalent use cases.

A prominent example of set membership testing is intrusion detection: Despite recent advances in machine learning based anomaly detection, intrusion detection in multi-purpose networks relies on misuse detection. The patterns to be watched for have evolved from simple byte signatures to higher-level Indicators Of Compromise (IOCs) like hashes of transferred files. The effectiveness of this approach depends on the quality of the input data. Hence, the term *Threat Intelligence* was coined to emphasize that proper misuse detection requires refined, high quality data. While the effectiveness of misuse detection relies on data quality, the efficiency of matching IOCs is key for practical applications. Consequently, data structures that maximize lookup performance and minimize space costs are of particular interest in this domain.

The use of probabilistic data structures, like the well-known Bloom filter [1], is a common approach to reduce space costs and improve performance of set membership tests. This is achieved by relaxing the task of set membership testing to so called *approximate set membership testing*, allowing a bearable number of false positives. That means, the data structure might consider an element part of the set that was never added. Unfortunately, traditional probabilistic data structures do not support deletion of elements without introducing significant overhead, leaving them impractical for many real-world scenarios. In 2014, Fan *et al.* introduced *cuckoo filters* [2], which promise to overcome these shortcomings.

Given the huge and still growing scales of today's computer networks, the efficient matching of Threat Intelligence is a key challenge in network security. In this paper, we study the feasibility of cuckoo filters to improve space efficiency in network security monitoring, focusing on the example of Threat Intelligence matching.

II. Related Work & Fundamentals

In this section, we will discuss related work (II-A) and explain the basic concepts behind cuckoo filters (II-B).

A. Related Work

Probabilistic data structures have a comprehensive history regarding applications in networking [3]. A recurring pattern in their use is to establish a filtering step prior to a complex processing tasks. For example, Dharmapurikar *et al.* make use of Bloom filters for matching byte signatures in network traffic [4]. False positives are mitigated by a downstream[1] analyzer, which applies a computational complex deterministic algorithm to verify matches. Upstream filtering to reduce the pressure on downstream processing is a common approach in context of the wide-spread manager-worker-pattern as shown in Figure 1. But, in case of Threat Intelligence matching, todays IOC sets are rapidly evolving. Hence, appropriate probabilistic data structures are required to support manipulating operations, i.e. insertions and deletions, without impeding continuous operation.

In 2014, Fan *et al.* introduced *cuckoo filters* [2], a probabilistic data structure for approximate set membership testing. Contrary to Bloom filters, cuckoo filters allow item removal without rebuilding the whole data structure or introducing false negatives. Interestingly, for many common applications cuckoo filters perform even better than Bloom filters and their removal-supporting alternatives in terms of space cost and lookup performance [2]. Consequently, cuckoo filters have been brought up in the networking context [5].

B. Cuckoo Filters

Cuckoo filters are based on a variant of *cuckoo hash tables* [6]. Like traditional hash tables, a cuckoo hash table consists of buckets that may contain several keys each. Usually, each key is associated to a value, which can be omitted

[1]Please note that we use the terms *upstream* & *downstream* to refer to the order of processing steps.

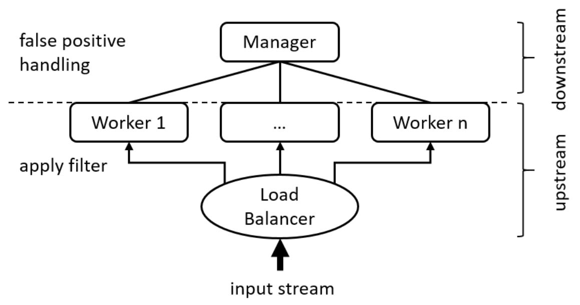

Fig. 1. Manager-worker-pattern for scaling processing of streamed data (e.g., network traffic).

for set membership testing. In contrast to traditional hash tables, cuckoo hash tables use independent hash functions to determine multiple candidate buckets for a given key. On insertion, if the first candidate bucket is already occupied, the next bucket can be used. If all candidate buckets are filled, the key to insert pushes an already inserted key out of its bucket (hence the name cuckoo hashing). In this case, the insertion algorithm continues by trying to insert the displaced key into one of its alternative buckets, which might trigger further relocations. After a specified number of relocations, the insertion is considered to be failed. The lookup of a given key can be performed in constant time, by checking all candidate buckets.

Cuckoo filters reduce space costs by storing only a short hash $p = f(x)$ called fingerprint, instead of the complete key x. Mapping the set of keys to a significantly smaller target set introduces the possibility of false positives. Furthermore, in case of relocation the original key is not available for calculating the corresponding alternative bucket, as the key cannot be reconstructed from the stored fingerprint. To enable relocation, Fan *et al.* make use of *partial-key cuckoo hashing*, by defining the two hash functions to obtain the candidate buckets as follows:

$$h_1(x) := hash(x)$$
$$h_2(x) := h_1(x) \oplus hash(f(x))$$

Figure 2 shows the insertion of an item x at index 2 using h_1 and the calculation of h_2 based on the stored fingerprint. As the *xor* operation is an involution, i.e. it is its own inverse, the alternative bucket for a fingerprint p stored in bucket i can be universally calculated as $h_{alt}(i, p) = i \oplus hash(p)$.

The false-positive rate ϵ of a cuckoo filter is influenced by the size of fingerprint s_f [bits], the bucket capacity e [entries] and the load factor α. By increasing the size of the fingerprints, the probability of fingerprint collisions and thus false-positives decreases. By increasing the number of entries per bucket, a fingerprint collision gets likelier; and obviously, the more items are inserted, i.e. the higher the load factor, the higher the chance to see fingerprint collisions[2]. The achievable load factor and the false-positive probability are conflicting

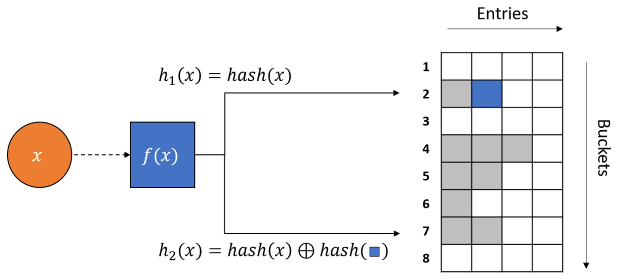

Fig. 2. The insertion of an item x into a cuckoo filter stores x's fingerprint. $hash()$ is used to obtain the bucket hash and $f()$ is used to generate fingerprints. The alternative bucket can be calculated based on the bucket index and the stored fingerprint value.

optimization goals. In [2], Fan *et al.* focus on cuckoo filters using two candidate buckets and four entries per bucket, as this configuration performs best in terms of space efficiency for target false-positive rates between 0.002 and 0.00001. In the following, we will focus on this configuration as well and use the fingerprint size to adjust false-positive rates.

III. Conceptual Insights

While Fan *et al.* provide a reference implementation of cuckoo filters[3], we chose to implement cuckoo filters ourselves to allow **runtime configuration** of the false-positive rate influencing parameters. The reference implementation uses a fixed bucket size $e = 4$ and a C++-template parameter to specify the fingerprint size. Although a cuckoo filter is theoretically able to hold the same item multiple times [2], we assume that every element is stored only once. Given that we use the filter to lower the pressure on downstream data structures, these can keep track of element counts if needed. Our C++ implementation, called *cuculiform*[4], is available under an open source license. In this section we will discuss conceptual insights we gained during implementation.

Dimensioning constraints: As the number of buckets varies for different cuckoo filter instances, the larger target set of h_1 and h_2 has to be mapped to the actual number of buckets. Mapping the set to the corresponding residue class (*mod m* where m is the number of buckets) is not generally applicable, as *xor* is not well-defined and therefore does not represent an involution for all residue classes:

$$\exists\ a, b, m : [a \oplus b]_m \neq [a]_m \oplus [b]_m$$

Intuitively this can be explained by the fact that *xor* operates bitwise. In case $a \oplus b > m$, the modulo operation might change the resulting bit-pattern in different ways on both sides of the inequation above. If $m = 2^k$, the modulo operation corresponds to a bitwise *and* of $2^k - 1$, which keeps the remaining bit-pattern stable. Thus, partial-key cuckoo hashing with *xor* requires the number of buckets to be a power of two. Regarding space costs, this might result in a worst case scenario of a filter that is over-provisioned by a factor of nearly

[2]Note that the number of entries per bucket also influences the achievable load factor [2].

[3]https://github.com/efficient/cuckoofilter
[4]https://github.com/kit-dsn/cuculiform

two. On the other hand, the smaller load factor reduces the filter's false-positive rate.

Impact of choice of hash function: While Fan *et al.* suggest to use CityHash[5], their reference implementation implements a much simpler universal hashing algorithm explored by Dietzfelbinger [7]. Our benchmarks revealed that Dietzfelbinger's algorithm, if used to obtain the bucket hash, causes a high variance in the filter's false positive rate across runs, while achieving the expected mean rate. This can be explained by the probabilistic nature of universal hashing. As a high variance is undesirable for practical applications, we recommend to use deterministic hash functions. Furthermore, the performance of a cuckoo filter is dominated by the applied hash functions. For example, using Highwayhash[6], which provides even stronger quality guarantees than CityHash, does not improve the filter's load factor. However, lookup performance degraded due to its increased complexity. Although this is an expected behavior, as it represents a core property of cuckoo hashing [6], we would like to emphasize the fact due to its operational relevance.

Representation of empty cells: The fact that there is no element at a given position in a bucket has to be represented. Naturally, this is done by setting all bits to zero, a value that is also part of the fingerprint hash function's codomain. A straight-forward solution to avoid collisions would be to add a bit for each element that indicates the cell's occupancy state. But, in the context of minimizing space costs, an additional bit is disadvantageous. Hence, the reference implementation maps zero-fingerprints to the value one. This yields a deliberately imbalanced distribution of fingerprint hash values, which, however, does not restrain practical application. Note that the effect becomes relevant for smaller fingerprint sizes.

Finally, the actual capacity of a cuckoo filter depends on the inserted elements. As described in section II-B, the filter is considered full after an insertion triggered the relocation threshold. In this case, the element to insert is incorporated into the filter in the first step and further relocations displace arbitrary elements. To prevent the filter from removing an element unpredictably, the reference implementation maintains an *eviction cache*, which stores the remaining element after the maximum number of relocations has been performed.

IV. Application in NSM

For many applications in Network Security Monitoring, even a small number of false-positives might be unacceptable. Nevertheless, cuckoo filters can be extremely valuable for processing high bandwidth data streams: A common pattern is to use probabilistic data structures upstream, to filter out a vast amount of irrelevant queries. The remaining false-positives can be handled by traditional data structures used in the backend.

To study applications of cuckoo filters in Network Security Monitoring, we integrated cuckoo filters into *Bro* [8], a popular open-source network monitoring software. In the following, we will introduce Bro and its Threat Intelligence monitoring capabilities (IV-A), compare three cuckoo filter implementations to the conventional data structures used in Bro (IV-B) and provide a brief operational evaluation (IV-C).

A. The Bro NSM

The Bro Network Security Monitor serves as a flexible platform for the analysis of network traffic. Incoming packets are processed by an *event engine* that utilizes protocol analyzers to parse the traffic and generate a high-level event stream. The event stream gets processed by a *policy script interpreter* that executes scripts written in the Bro scripting language. The Bro scripting language is a domain-specific, Turing-complete language, tailored to fit the needs of NSM. Bro ships with a comprehensive set of scripts, organized in frameworks.

Bro's architecture is designed to scale horizontally by supporting a clustered setup [9], following the manager-worker-pattern depicted in Figure 1. Traffic is load-balanced per connection across multiple workers, which perform per-connection analysis. A central manager node is used to distribute information and aggregate results. As Bro implements a multi-process design, each node represents a dedicated Bro instance. This approach applies in particular to multi-core setups in single-machine deployments.

In 2012, Amann *et al.* introduced the means to implement Threat Intelligence matching with Bro [10]. Since version 2.2, Bro ships with the *Intelligence Framework*, a collection of scripts for managing intelligence data. In a clustered setup, the manager node maintains an in-memory representation of IOCs and their corresponding meta data using hash tables. To perform the matching, worker nodes only need to keep the IOCs, which is realized using more space-efficient hash sets. In case a worker detects an IOC, the hit is reported to the manager, who looks up the associated meta data and logs the match enriched with context information and IOC meta data.

B. Comparison of Different Implementations

In preparation of operational testing, we evaluate our implementation in the context of the intended area of application. We compare *cuculiform* to the reference implementation and a reimplementation in Rust[7], as well as to the built-in Bro data types *table* and *set*. Our measurements are performed using a Bro-script and Bro in version 2.5.3. To provide script-level access to the cuckoo filter implementations, we utilize Bro's plugin interface. All measurements have been performed on HP ProLiant™ DL160 G6 machines, equipped with two 2GHz Intel® Xeon® E5504 processors (4 cores each) and 24 GB DDR3 memory.

Table I shows the results of our measurements and parameterization details. For each cuckoo filter we add items until it is filled. All cuckoo filter implementations achieve similar load factors of about 96%. The false-positive rate is determined by querying 1 Mi items that have not been added, whereas lookup-time calculation is based on twice as many queries, half of which are guaranteed to succeed. Under the given

[5]https://github.com/google/cityhash
[6]https://github.com/google/highwayhash
[7]https://github.com/seiflotfy/rust-cuckoofilter

TABLE I
COMPARISON OF CUCKOO FILTER IMPLEMENTATIONS

Implementation	Structure Size [KiByte]	FP-Rate [%]	Lookup-Time [μs]
Bro Hashtable	211 202.8	-	1.7589
Bro Hashset	145 666.8	-	1.9281
Reference	1 024.0	2.9794	0.9637
Rust	1 024.0	2.9789	1.0847
Cuculiform	1 024.0	2.9787	1.3746

Fingerprint size 8 bit, 4 elements per bucket, 1 Mi elements capacity, mean of 1 000 runs (confidence intervals negligible)

parameterization, the examined cuckoo filter implementations differ only slightly in lookup-time. Cuculiform is the slowest, trading runtime-configuration flexibility for performance, but still faster than Bro's built-in table and set types. The main advantage of cuckoo filters over the traditional data structures manifests in their size of 1 MiB compared to at least 145 MiB in case of the hashset.

C. Operational Evaluation

The essential advantage of cuckoo filters over Bro's traditional data structures is the reduced memory consumption. In IV-B, we have shown that cuckoo filters outperform Bro's hash sets by a factor of about 140 in terms of memory costs[8]. Considering the Intelligence Framework, this effect applies per worker, as each worker node requires a copy of the structure to match IOCs. As recent Bro deployments scale up to 50 workers and more [11], memory savings in practical operation might exceed 10 GB for this single use case. Given the genericness of the manager-worker-pattern, we expect comparable improvements for further use cases in Bro and beyond.

The memory savings come at the cost of introducing false positives, which have to be handled. To prove the practical applicability of cuckoo filters, we conducted a brief operational evaluation, investigating the overhead imposed on the cluster. To this end, we integrated cuckoo filters into Bro's Intelligence Framework and compared the approach to the traditional one. For our tests we established a link between two of the previously described machines, one of the machines serving as traffic generator, the other hosting a single-worker Bro cluster distributing worker and manager across different cores. We ingested 16 000 IPs as IOCs into the Intelligence Framework, which roughly corresponds to filling a filter with a capacity of 2^{14} in case of using a cuckoo filter. Finally, we created an artificially extreme scenario by designing the experiment to yield a true positive rate of 10% and an intended false positive rate of 3% for the cuckoo filter variant.

Interestingly, the false positives introduced by cuckoo filters do not significantly influence the CPU load of the manager. This can be explained by the fact that the manager immediately detects false positives and omits any further processing, which

[8]Note that the actual space efficiency depends on an accurate forecast of the total number of elements to insert.

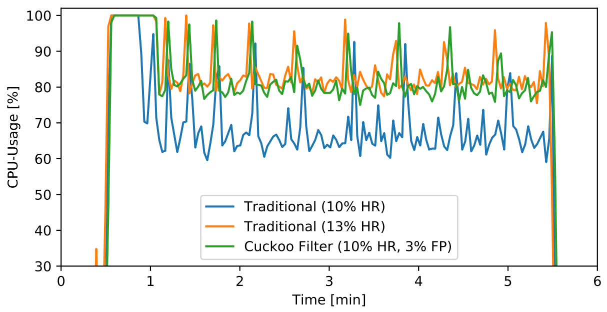

Fig. 3. CPU load of the worker processing $\sim 270\,000$ connections per minute for cuckoo filter variant with 3% false positive rate (FP), traditional variant and traditional variant with increased hit rate (HR).

would be required for true positives. In contrast, the worker exhibits a higher CPU load in case of the cuckoo filter variant. Figure 3 shows that the increased load correlates to the number of hits, which have to be processed. The high processing costs of hits on the worker can be explained as the worker sends a considerable amount of meta data related to each hit to the manager. Hence, the overhead caused by false positives affects Bro's workers rather than the manager. All in all, the application of cuckoo filters trades space costs for processing time as a function of processed traffic. As cuckoo filters easily achieve false positives rates below 0.1% and the overall overhead is distributed across all workers, we consider cuckoo filters a promising alternative to Bro's built-in data types.

V. CONCLUSION & FUTURE WORK

In this paper, we investigated the practical applicability of cuckoo filters in Network Security Monitoring. We provided insights for the practical application of cuckoo filters not covered by the original paper. In particular, we would like to emphasize the dimensioning constraints of cuckoo filters, whose size has to be a power of two, and the impact of the hash function choice. Both aspects substantially influence practical operation.

By integrating our cuckoo filter implementation into Bro, we demonstrated the feasibility of applying cuckoo filters to Network Security Monitoring. We verified that the implementation can compete with traditional data structures in terms of computational performance and allows a remarkable reduction of space costs. In addition, we found that the overhead introduced by false positives affects workers rather than the manager in case of intelligence matching with Bro. While the conceivable memory savings underline the massive impact in case of typical NSM architectures, we showed that the performance trade-offs have to be carefully evaluated case by case.

For future work, we plan to conduct a case study on real-world deployments to deepen the understanding of practical requirements and establish parameterization strategies. In this context we want to evaluate the recently proposed adaptive cuckoo filters [12], which allow to mitigate the effect of recurring false positives. Furthermore, we will address the dimensioning constraints of cuckoo filters.

Acknowledgment

The authors would like to thank Matthias Vallentin for pointing out cuckoo filters as a notable Bloom filter alternative.

References

[1] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1, 1970, ISSN: 00010782. DOI: 10.1145/362686.362692.

[2] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14, 2014, pp. 75–88, ISBN: 978-1-4503-3279-8. DOI: 10.1145/2674005.2674994.

[3] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, Jan. 2004, ISSN: 1542-7951, 1944-9488. DOI: 10.1080/15427951.2004.10129096.

[4] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep packet inspection using parallel bloom filters," in *11th Symposium on High Performance Interconnects*, 2003, pp. 44–51, ISBN: 978-0-7695-2012-4. DOI: 10.1109/CONECT.2003.1231477.

[5] M. Al-hisnawi and M. Ahmadi, "Deep packet inspection using cuckoo filter," in *2017 Annual Conference on New Trends in Information Communications Technology Applications (NTICT)*, Mar. 2017, pp. 197–202, ISBN: 978-1-5386-2962-8. DOI: 10.1109/NTICT.2017.7976111.

[6] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004, ISSN: 01966774. DOI: 10.1016/j.jalgor.2003.12.002.

[7] M. Dietzfelbinger, "Universal hashing and k-wise independent random variables via integer arithmetic without primes," in *Annual Symposium on Theoretical Aspects of Computer Science (STACS 96)*, vol. 1046, Springer Berlin Heidelberg, 1996, pp. 567–580, ISBN: 978-3-540-60922-3. DOI: 10.1007/3-540-60922-9_46.

[8] V. Paxson, "Bro: A system for detecting network intruders in real-time," *Computer Networks*, vol. 31, no. 23, pp. 2435–2463, 1999.

[9] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney, "The NIDS cluster: Scalable, stateful network intrusion detection on commodity hardware," in *Recent Advances in Intrusion Detection (RAID)*, vol. 4637, Springer Berlin Heidelberg, 2007, pp. 107–126, ISBN: 978-3-540-74320-0. DOI: 10.1007/978-3-540-74320-0_6.

[10] J. Amann, R. Sommer, A. Sharma, and S. Hall, "A lone wolf no more: Supporting network intrusion detection with real-time intelligence," in *Research in Attacks, Intrusions, and Defenses (RAID)*, vol. 7462, Springer Berlin Heidelberg, 2012, pp. 314–333, ISBN: 978-3-642-33338-5. DOI: 10.1007/978-3-642-33338-5_16.

[11] V. Stoffer, A. Sharma, and J. Krous, "100g intrusion detection," Lawrence Berkeley National Laboratory (LBL), 2015. [Online]. Available: https://commons.lbl.gov/download/attachments/120063098/100GIntrusionDetection.pdf.

[12] M. Mitzenmacher, S. Pontarelli, and P. Reviriego, "Adaptive cuckoo filters," in *2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2018, pp. 36–47. DOI: 10.1137/1.9781611975055.4.