REX: Resilient and Efficient Data Structure for Tracking Network Flows

Dinil Mon Divakaran¹, Li Ling Ko, Le Su, Vrizlynn L. L. Thing

Department of Cyber Security and Intelligence, Institute for Infocomm Research (I²R), Agency for Science, Technology and Research, Singapore.
1 Fusionopolis Way, #21-01 Connexis (South Tower), Singapore 138632

Abstract

One of the important tasks for most network security solutions is to track network flows in real-time. The universe of flow identifiers being huge, hash tables with their fast operations are well suited for this task. In order to overcome the limitations of traditional hash tables, the research community have come up with different improved variants; two of the well-knowns being Cuckoo and Peacock hash tables. Yet, network flows have interesting characteristics that can be exploited for tracking flows more efficiently. Besides, the existing hash tables are vulnerable to attacks. In this context, we design, develop and evaluate REX, a resilient and efficient data structure for tracking of network flows. REX is designed to make good use of, both, the characteristics of Internet traffic, as well as the different memory technologies. REX stores most commonly updated flows in the faster and smaller SRAM, while storing the rest in DRAM. We conducted extensive experiments using real network traffic to evaluate and compare REX, Cuckoo and Peacock hash tables. The results demonstrate, under both normal and attack scenarios, that REX not only rejects the least number of packets, but also significantly reduces the total time taken for the important hash table operations.

Keywords: Network, Resilience, Flows, Hash tables, Security

1. Introduction

Real-time tracking of flows in a network has wide variety of applications, such as attack and anomaly detection, attack defence and mitigation, enforcement of firewall rules, application protocol classification, flow control, accounting, flow routing, load balancing, etc. With the advent of software-defined networking (SDN), there has been a renewed interest in flow-based solutions for different applications [1, 2]. In the context of security solutions, flow related information such as states (‘SYN sent’ and ‘SYN+ACK received’ are examples of TCP states), packet size, inter-arrival times between packets, (ongoing) flow size, etc. are important features that often need to be tracked in real-time.

Hash tables, known for their fast $O(1)$ operations, have been in use in network elements and middlewares like routers, intrusion detection systems and firewalls, for purposes such as routing, connection state tracking, QoS provisioning, etc. As traditional hash tables become inefficient with increasing load, the research community has come up with a number of improved variants, notable examples being Cuckoo hashing [3] and Peacock hashing [4]. The objectives behind these new variants have been different—Cuckoo table achieves high space efficiency, while Peacock table aims to provide high determinism (in worst-case scenarios).

¹Email addresses: divakaran@i2r.a-star.edu.sg (Dinil Mon Divakaran), kol@i2r.a-star.edu.sg (Li Ling Ko), lsu@i2r.a-star.edu.sg (Le Su), vriz@i2r.a-star.edu.sg (Vrizlynn L. L. Thing)

¹Corresponding author
Yet, existing hash table structures are vulnerable to attacks [5], with performance degrading in such adverse scenarios. In [6], researchers demonstrated attacks on the Bro intrusion detection system [7]. When performance degrades, a hash table (increasingly) rejects and thereby stops tracking flows even when there is free space within the data structure. This in turn affects solutions deployed for network security, such as defense, detection and mitigation. In this context, our work aims to design a hash table for tracking network flows\(^2\) (a use case where Cuckoo and Peacock tables can be applied) with the following objectives:

1. Resilience: the data structure should be resilient when faced with adverse scenarios such as DDoS attacks;
2. Efficiency: improve efficiency in tracking flows in comparison to the existing data structures.

Without resilience, a flow-tracking hash table loses its function and utility in assisting security solutions. Both Cuckoo and Peacock hash tables are prone to attacks [8]. Under attacks, hash tables can fill up quickly, leading to performance degradation, and consequently resulting in the discard of a large number of packets (and flows). Peacock hash table even suffers from post-attack damage, such that rebalancing of the entire table will not help to solve the problem. We stress the importance of the resilience property in our design.

Though efficiency is related to resilience, it has relevance on its own, specifically when we consider challenges in networks. Link capacities are increasing at a rate that leaves time in the order of a few nanoseconds to process packets flowing through a network node. An unavoidable action on packet arrival is memory lookup, performed prior to insert, update and delete operations. Another time-consuming and important factor is the number of computations required to be performed on the arrival of a packet; for example, hash computations. For efficient lookup, hash table designs should utilize a combination of different memory technologies. SRAM is much faster than DRAM in accessing data stored in memory. Yet, due to the high cost difference, SRAM memory is much smaller in size compared to DRAM. Using a combination of SRAM and DRAM can bring significant savings in the time to perform different hash table operations.

Our interest lies in real-time tracking of flows in a network. Depending on the applications, a number of flow-related features may be of interest. Table 1 gives an example set of such flow features, where flow ID is the identifier of a flow. A flow is identified by the (hash of the) common five-tuple of \{source IP address, destination IP address, source port, destination port, protocol\}. The ongoing size of the flow is the number of packets (or bytes) of this particular flow seen until the current time. For security analysis, knowing the application protocol (app. protocol) is also useful, as attacks might be specific to a protocol (e.g., HTTP). While not shown, the transport layer protocol is tracked. For TCP flows, the state of the flow is useful for security purposes (say, by checking whether the flow conforms to TCP’s finite state machine) as well as for QoS delivery. In this work, we maintain one entry for (packets of) a bidirectional flow.

<table>
<thead>
<tr>
<th>Flow ID</th>
<th>ongoing size</th>
<th>app. protocol</th>
<th>state</th>
<th>start time</th>
<th>end time</th>
<th>mean IAT</th>
</tr>
</thead>
</table>

Though inherently UDP does not maintain state, depending on the application protocol, one can define state for UDP flows too. For example, a DNS query can be defined as state 0 and the corresponding response as state 1. Such state representation is helpful in detecting bots that try to reach out to their bot masters using DNS protocol. In addition, different aspects of time, such as start time, end time, and mean inter-arrival time (mean IAT) of packets in the given flow also need to be tracked.

\(^2\)A flow is generally defined as a set of packets localized in time and having the same five-tuple of source and destination IP addresses, source and destination ports, and protocol.
In this work, we design and develop a resilient and efficient data structure, called REX, for tracking connection flows in real-time. Inspired by previous works (for example [9]), we exploit the well-known heavy-tailed nature of Internet traffic to design REX, while also borrowing architectural ideas from Cuckoo and Peacock hash tables. The objective of the design is to make REX not only resilient to attacks and anomalous traffic spikes, but also to perform hash table operations efficiently. We evaluate REX and compare with Cuckoo and Peacock hash tables using real network traffic, in four scenarios varying in load, traffic type and table structure. Our results show that, REX not only rejects the least number of packets, but also brings significant savings in terms of the total time for hash table operations.

In the following section, we briefly discuss the related works. In Section 3, we motivate the concept and present the design of the data structure REX, while also comparing with Cuckoo and Peacock hash tables. Subsequently, in Section 4, we develop the algorithms for the operations in REX, in the context of tracking network flows. We further discuss the resilience property of REX in Section 5. Section 6 performs extensive performance evaluations, comparing REX with Cuckoo and Peacock hash tables.

2. Related works

2.1. Hash tables

Hash tables are widely used in high speed network environment for a broad range of applications and services, such as IP address lookup, packet/flow classification, packet/flow filtering, flow accounting, flow state keeping, QoS measurement, heavy-hitter identification, intrusion detection, QoS measurement, virus signature scanning, etc. However, by design, hash tables inevitably face collisions; the traditional approaches to resolve collisions is to resort to chaining (using a linked list of buckets dynamically allocated) or linear probing (open addressing). Therefore, while the common operations of insert, delete and lookup take constant time on average, the worst case performance becomes non-deterministic due to collisions at high load.

Over the years, researchers focused on improving the performance of hash table performance under worst-case scenarios. A particular direction that drops the probabilistic worst-case performance significantly is the use of multiple hash functions [10, 11, 12]. In a general form, the multi-level hash table (MHT), consisting of $n$ buckets, is divided into $d$ subtables with each sub-table consisting of $\frac{n}{d}$ buckets. To insert an item, $d$ hashes (corresponding to the $d$ subtables) are computed on the item; and the item is stored in the least occupied bucket. Ties can be broken randomly, or asymmetrically as is done in the well-known $d$-left scheme [12]. One drawback of multiple-choice hashing schemes is that lookup operation takes a fixed $d$ hash computations and memory accesses.

Among the numerous design variants of the MHTs, an interesting variant that significantly improves the space utilization over standard MHTs is Cuckoo hashing [3]. Standard Cuckoo hash table computes two hash functions ($d = 2$) on the item to be inserted. Assume for simplicity that a bucket holds not more than one item. If either of the hashed buckets is empty, the item is successfully inserted. If there is collision at both bucket locations, one of the two buckets is randomly selected, and the corresponding item stored in the selected bucket is kicked out. In essence, the stored item is relocated to another bucket within the hash table. The process then works recursively, such that the kicked out item is considered for insertion (at its only alternate location), which may result in the displacement of another item. This process, of the resident item making way for the item being inserted, goes on until an unoccupied bucket is discovered. In a practical implementation, however, the insert operation consists of (1) finding a path of occupied buckets ending in an unoccupied bucket, and subsequently (2) carrying out the kickout process. The length of the path is limited by a predefined threshold, so that the search is bounded. The insertion fails if no such path is found (during the first step). A number of variants of Cuckoo table were subsequently proposed, devoting to improvements and analysis by varying and generalizing different parameters, such as the number of choices ($d$) [13] and the number of items per bucket [14] (refer [15] for empirical analysis varying both parameters).
A popular MHT for network systems, called the Peacock hashing [4], explores the possibility of improving the worst-case hash table performance in another way. Peacock hash table is a sequence of subtables with size following a decreasing geometric progression (say, with a common ratio of 0.1). An item is stored only in one of the subtables. The length of the probe sequence in any table is limited by a threshold. For insertion, the subtables are probed sequentially, progressing from larger subtable to smaller subtable, only if no unoccupied bucket is found in the larger subtable. The first subtable, which is also the largest in size, is called the main table. Every subtable except the main subtable has an associated on-chip Bloom filter that maintains the summary of items stored. For lookups, the Bloom filters are first queried; and only if none of them returns a positive response, is the main subtable queried. Peacock hash table is designed with the aim of providing guarantees on the worst-case performance; hence the number of lookups is bounded by the number of subtables and the threshold for collisions (probe sequence length) in a subtable.

Peacock hashing is one of the hash tables that makes use of SRAM. Though SRAM is fast, it is expensive and limited in size in comparison to the much slower DRAM. Therefore, it is impractical to store the entire hash table in SRAM for network systems. The main subtable of Peacock hash table is stored in DRAM, and the Bloom filters and possibly some of the smaller subtables (depending on the sizes of the subtables) reside in the on-chip SRAM. There are also other works that consider combining both SRAM and DRAM technologies for improved performances. For example, [16] analyzes the trade-off between the hash table load and its expected lookup time for an MHT that utilizes both SRAM and DRAM; however, this is based on the assumption that the hash table is built offline.

2.2. Bloom filters

We briefly discuss Bloom filters, a widely used data structure in network devices and applications, which we also employ in our design of REX. Standard Bloom filter [17] is designed for fast and probabilistic membership testing; and it supports insertion and lookup operations. Whenever an element is queried for membership, the data structure will return either a definite negative response, or a probabilistic positive response. While there can be false positives, there are no false negatives. To represent a set of \( n \) elements, a Bloom filter uses a bit array of size \( m \) (bits) and a set of, say, \( k \) hash functions with range \([1, \ldots, m]\). For insertion, the \( k \) hash functions are applied on the element, and the corresponding positions in the bit array are set to one. Upon query for lookup, the \( k \) hash functions are computed on the given element. If any of the hashed locations is zero, then the response is negative; if all locations have value one, then the response is positive. A false positive response might be obtained in the case when all the \( k \) bits are set (due to different elements), but the item was not inserted. A Bloom filter brings considerable space savings at the cost of introducing probability of false positives. There is an optimal \( k \) for the ratio \( \frac{m}{n} \) that gives the minimum false positive probability; therefore the false positive probability can be reduced by increasing \( m \).

There are a number of variants of Bloom filters, such as, cuckoo filter [18] (inspired by cuckoo hash table), \( d \)-left counting Bloom filter [19], blocked Bloom filter [20], and so on. We touch upon counting Bloom filter (CBF), which is of interest to our work here. The CBF, proposed by Fan et.al [21], supports deletion in addition to the standard operations of insertion and membership query. Instead of of using a bit at every location, a CBF uses a counter. Each time an element is inserted, the counters corresponding to the hashed locations are incremented. For deleting an element, the counters in the locations where the element is hashed to, are decremented. Therefore, only if all hashed locations have positive values, is the element considered to be in the CBF. The additional cost to support deletion operation is the space used to represent counters. Though overflow of the counter leads to false negatives, a counter size of four bits, for optimal number of hash functions, is sufficient for negligible overflow probability (refer [21] for details). This means that, a CBF usually needs four times as much space as standard Bloom filter, to retain the same false positive probability. We also note that, a variant of the CBF lowers both the false positive probability and counter overflow probability using variable increments of counters [22].
2.3. Exploiting Internet traffic characteristics

There is a class of research proposals that exploit the heavy-tail distribution of Internet traffic (explained in the next section) to improve flow-based algorithms in routers, of which works on efficient caching are related to our work. We discuss some of them here.

Observing the mice-elephant phenomenon in Internet traffic, Estan and Varghese developed multi-stage (counting) filters for efficiently estimating the heavy hitters in the traffic [9]. As discussed, Bloom filter, although provides extremely fast membership testing, its basic form lacks the capability of storing important flow information for further analysis. Therefore, like other Bloom filters, this multi-stage counting filter on its own, is used only for caching or tracking limited number of flows, so as to meet goals such as efficient accounting of heavy hitters. Specifically, they are not used for tracking a number of features (as exemplified in the previous section) of each flow.

In [23], authors proposed a variant of LRU (least recently used) algorithm, called $S^3$-LRU. The proposed algorithm uses a naive hash table for tracking elephant flows. The collisions are not resolved, instead they result in false positives, leading to measurement errors. Furthermore, the LRU-based approach is prone to discard flows that are already stored, which may result in tracked flows being discarded during attacks. Two works [24, 25], leveraging on a technique called ALFE (adaptive least frequently evicted), segregate the flows according to their “evict distance” in a stream of flow records. By defining such a distance and combining the use of thresholds, the data structure stores flows of different sizes into different segments, giving priority to the elephant flows.

All the above mentioned works have their own importance and merit. While each of them exploits the heavy-tailed nature of Internet traffic, which REX also does, our design of REX differs from them in the following way. REX falls under the category of hash tables (such as Cuckoo and Peacock tables) where collision resolution is employed. On the other hand, the use of naive hash tables (without collision resolution) for caching large flows, introduces false positives. Therefore, multiple flows hash into the same location, inevitably leading to measurement and tracking errors. That said, REX also has measurements errors, but the errors are not due to false positives. As discussed in Section 3.2.5, REX rejects only the first few packets of a flow. Therefore, one important feature of REX, that Cuckoo and Peacock hash tables also share is: once a flow is inserted into the data structure (and thus tracked), it is always tracked. This is in contrast with the above designs, wherein flows may potentially go through (zero or more) cycles of eviction and insertion. We elaborate this property of REX in Section 3.2.5.

3. REX: motivation, design and properties

The basic idea behind the design of REX is to exploit the inherent characteristics of Internet traffic. The flow-size distribution of the Internet traffic is known to exhibit heavy-tail behaviour—a small percentage of flows, called elephant flows, contributes to a large percentage of the traffic in volume, and vice-versa [24, 26, 27]. The large number of small-size flows are usually called the mice flows. Similar traffic characteristics are also observed for data center networks [28]. The knowledge of the so-called mice-elephant phenomenon had been previously used for different purposes, ranging from accounting [9] to scheduling [29] to queue management [30]. The design of REX draws on this important property of the Internet traffic to make good use of the different memory technologies (SRAM and DRAM).

In this section, we design and develop REX, a hash-based resilient and efficient data structure for tracking connection flows in a network. First, we provide the motivation for the design.

3.1. Motivation

To motivate the design of REX, we performed analysis of a 15-minute MAWI traffic dataset obtained from the WIDE project available online [31]. We filtered out TCP connection flows. For each flow, we recorded the TCP state transitions (for example, from ‘SYN received’ to ‘SYN+ACK sent’), to differentiate ‘illegitimate’ TCP flows from the ‘legitimate’ ones: any flow that follows a sequences of valid state transitions is tagged as legitimate; and we refer to the rest as illegitimate
network flows. For example, a flow that establishes connections (using the three-way handshake \texttt{SYN} → \texttt{SYN+ACK} → \texttt{ACK}) and transfers data, and finally terminates as per the TCP’s finite state machine (FSM) is legitimate. A TCP flow that has only one packet (say, \texttt{SYN} packet) is an example of illegitimate flow, and so is any flow with a sequence of states not beginning with the connection establishment phase. Note that different operating systems and servers might have slightly different implementations, potentially for making the protocol efficient for communications, and might therefore not follow the FSM strictly. While such changes can also be incorporated into this deterministic classification approach, we do not delve into such nuances here.

The trace we analyzed consisted of traffic flows during a 15-minute interval, collected on 10th December 2014. We extracted TCP flows from this dataset; they were around 360,000 TCP flows, constituting around 16 million packets. In the traffic trace, \approx 10\% of all flows contributed to more than 60\% of traffic volume (in packets), and \approx 25\% of all flows accounted for around 72\% of all packets. The illegitimate traffic volume constituted around 18\% of the total traffic. Fig. 1(a) plots the empirical CDFs of the sizes of the TCP flows in the dataset. Besides tracking the termination of TCP flows based on its FSM, we also consider a flow as terminated if it is inactive for a time period of two minutes. In Fig. 1(b), we plot the same data but only for small-size TCP flows, for better visualization. The plot clearly demonstrates the mice-elephant phenomenon, with more than 80\% of all traffic flows (comprising of legitimate and illegitimate flows) having size less than 30 packets. If we use a size of 30 packets to differentiate between small and large flows, i.e., a flow is small if its size is less than 30 packets and large otherwise, we notice that, around 78\% of legitimate flows are small in size. The mice-elephant phenomenon is even more pronounced for the illegitimate flows—more than 91\% of all illegitimate traffic flows are less than or equal to 25 packets in size.

From the above analysis (that confirms the mice-elephant phenomenon reported in literature), we highlight a few important insights that lead us to the design of REX: (i) most of the packets during a given duration belong to flows that are large in size; (ii) the number of large flows is small in number, and (iii) illegitimate flows are often small in size. We now proceed to present the design of REX that builds on this particular mice-elephant phenomenon of the Internet traffic.

3.2. Design

In this section, we develop the design of REX. Detailed descriptions of the operations in REX are presented in Section 4.

---

3We estimated the size of a flow by the number of packets it carries, instead of using the sum of the sizes of the packets in the flow. The difference is negligible in this context.
3.2.1. Structure of REX

Our proposed design of REX consists of a hierarchy of subtables where each subtable is in itself a Cuckoo hash table with $d$-ary hashing. The reason to choose Cuckoo hashing for the subtables is because it is highly space efficient [13], while having only constant lookup time ($d$ lookups for each stored item). Fig. 2 illustrates the overall structure of REX, consisting of multiple subtables of increasing size from top to bottom. The topmost subtable $T_0$ is also the smallest of all the subtables in the REX, and $T_n$ is the largest subtable; we will discuss further on this below. The number of subtables is dependent on the different memory technologies available for use as well as their capabilities and limitations. At any point in time, a flow will be present in only one of these subtables. To avoid memory lookups on all the subtables for each packet arrival, much like the Peacock hash table structure, REX has one Bloom filter (BF) for each subtable except for the topmost subtable $T_0$. BFs are small in size, and by design $T_0$ will also be small in size, therefore they can all fit in SRAM.

3.2.2. Lookup on packet arrivals

A packet is part of a flow. For an arriving packet, queries are made to all Bloom filters as well as to the smallest subtable $T_0$, using its flow identifier. As these queries can be made in parallel, we assume responses are received at the same time. There are three possible response scenarios:

1. If no positive response was returned for the queries, the flow is new and not present in any of the subtables.
2. If the smallest subtable returned a positive response, then, irrespective of what the Bloom filters returned, we know for sure that the flow is indeed in the smallest subtable $T_0$.
3. If none of the above scenarios happened, and one or more Bloom filters responded positively, then one of the subtables corresponding to the Bloom filters that responded positively might contain the flow. In such a scenario, the corresponding subtables need to be further probed to find out the exact subtable which stores the flow, if any.

3.2.3. Subtables of REX

A new flow, or to be precise the first packet of a new flow, with ongoing size of one packet, is always inserted in the largest subtable ($T_n$ in Fig. 2), or equivalently the table of small-size flows\(^4\). As the ongoing size of a resident flow increases due to the arrival of packets, the flow moves to the upper subtables in the hierarchy. In our illustration, $\theta_i$’s are the flow-size thresholds for subtables. As more packets of a resident flow arrive and the flow size goes beyond $\theta_n$, this particular flow is removed from the subtable $T_n$ and inserted in $T_{n-1}$, a smaller subtable up in the hierarchy. Similarly, if the size of the flow, while residing in the subtable $T_1$, becomes greater than $\theta_1$, the flow is moved to the topmost subtable $T_0$. Therefore, with this design, REX segregates flows of different sizes into various subtables that correspond to (possibly) different memory technologies. Moreover, as REX moves the flows from larger to smaller subtables with increasing flow-sizes, the Bloom filters associated with corresponding subtables need to be updated. Therefore, the BFs in REX are counting Bloom filters [21] that support deletion. As a flow moves from $T_i$ to $T_{i-1}$, the flow is deleted from the BF of $T_i$ and inserted into the BF of $T_{i-1}$. The only exception is when a flow moves from $T_1$ to $T_0$, in which case there is only one BF to be updated (which is BF-1 of subtable $T_1$); $T_0$ being small in size, does not have a BF.

A flow that moves from a larger subtable (lower one) to smaller subtable (upper one) in the hierarchy will be treated as a new flow in the smaller subtable. Therefore, the insert operation is the same and applies to all subtables. When no new packet of a flow arrives for a given expiry duration, the flow is considered expired and removed.

\(^4\)Flow size is measured in number of packets, and not in bytes.
3.2.4. Relevance of flow size

In the context of our work, we define a flow as large if its size, in number of packets, is greater than the threshold defined for the subtable $T_1$; this threshold is $\theta_1$ in Fig. 2. Similarly, all flows with sizes less than or equal to the threshold for the largest table $\theta_n$ are considered small. In the case where there are only two subtables, there is only one threshold differentiating small and large flows, and hence there are only two categories of flows based on size (small and large). As the number of subtables increases, obviously the number of categories also increases. However, the important point of observation is that, REX segregates the largest flows into one subtable, $T_0$. Since such large flows constitute a small percentage of the total number of flows, the size of this subtable can be much smaller than that of a traditional hash table designed to store all flows. At the same time, the large flows contribute to 80–90% of traffic in volumes (as demonstrated in Section 3.1). For REX, this means, for most of the arriving packets, lookups and updates will be performed on the subtable for large flows $T_0$. Therefore, storing the smallest subtable comprising only of large flows (as well as the Bloom filters for the other larger subtables) in the fastest RAM is expected to bring significant savings in time during table operations. Such a hierarchical design allows REX to scale with the number and sizes of memory technologies available.

3.2.5. An important property: once tracked, always tracked

An important property that REX shares with Cuckoo and Peacock tables is that, once a flow is allocated space in REX, it is tracked until it expires. This is different from many works in the literature, where a flow once tracked can go through (zero or more) cycles of eviction and insertion. Such cycles lead to lossy information during the lifetime of the flow. Besides, evicting and inserting the same flow is an inefficient process, that can be exploited by an attacker. In REX, under high load, one or more packets of a new flow may be rejected until space is found in the REX table. However once a packet of a flow is inserted, no more packet of the same flow is discarded.

Assume that the probability of finding space for a packet of a new flow in REX is a constant $p$ (during a specific time interval). Then the probability that a flow of size $s$ packets is not tracked by REX (or equivalently, the probability that the first $s$ packets of a flow are not tracked), is exponential in $s$, and can be approximated by $(1-p)^s$. Even for a very low value of $p = 0.2$ (at high load), the probability of not tracking a flow of 20 packets is 0.012, and that for a flow of 40 packets is a negligible 0.00013. Therefore, at high load, REX will drop packets of new flows.
allowing more flows to move from the corresponding subtable when the traffic intensity changes. Observe that, by decreasing the value of the threshold, say $\theta$, changes the value of the threshold (corresponding to a subtable) by a constant value when the thresholds adapt to varying traffic intensity. To achieve this, we use a simple linear function that changes the value of a subtable $T_i$ from $l_i$ to $l_i - 1$ based on the load $l_i$ of subtable $T_i$ within a target range. (It is impossible to control the load of all subtables, as incoming traffic intensity is outside the control of the system.) Note that $\theta_i$ controls the number of flows in both $T_i$ and $T_{i-1}$. As the load of $T_0$ is left uncontrolled, the threshold $\theta_0$ is varied depending on the load of the subtable $T_{i-1}$. The variable $l_{i-1}$ represents the current load of subtable $T_{i-1}$. The algorithm aims to keep the load $l_i$ of subtable $T_i$ within a given target load range of $[l_i^{\text{min}}, l_i^{\text{max}}]$. To give an example, the algorithm controls $\theta_1$, based on the load $l_0$ of the subtable $T_0$, with the aim of keeping $l_0$ within a desired predefined range. Observe that a high target load range for $T_0$ will ensure that the SRAM is left uncontrolled, the threshold $\theta_0$ is varied depending on the load of the subtable $T_{i-1}$. The variable $l_{i-1}$ represents the current load of subtable $T_{i-1}$. The algorithm aims to keep the load $l_i$ of subtable $T_i$ within a given target load range of $[l_i^{\text{min}}, l_i^{\text{max}}]$. To give an example, the algorithm controls $\theta_1$, based on the load $l_0$ of the subtable $T_0$, with the aim of keeping $l_0$ within a desired predefined range. Observe that a high target load range for $T_0$ will ensure that the SRAM
Algorithm 1 adaptiveThreshold($\theta$, $l$)

**Input:** Threshold: $\theta$, Current load: $l$

1. if $l > l_{\text{max}}$ then
2. \hspace{1em} if $(\theta + C) < \theta_{\text{max}}$ then
3. \hspace{2em} $\theta \leftarrow \theta + C$ \hspace{1em} $\triangleright$ increase by a constant
4. \hspace{1em} end if
5. \hspace{1em} else if $l < l_{\text{min}}$ then
6. \hspace{2em} if $(\theta - C) > \theta_{\text{min}}$ then
7. \hspace{3em} $\theta \leftarrow \theta - C$ \hspace{1em} $\triangleright$ decrease by a constant
8. \hspace{2em} end if
9. \hspace{1em} end if

is maximally utilized as $T_0$ resides in SRAM. To avoid instability due to frequent updates, the threshold is checked and changed (i.e., the function `adaptiveThreshold` is invoked) only once in every $\eta$ new packet arrivals.

Let $\theta_{\text{min}}$ and $\theta_{\text{max}}$ denote the minimum and maximum values, respectively, of $\theta_i$. For readability, in Algorithm 1, we have dropped the subscripts indicating the subtables—the threshold $\theta$ and the current load $l$, as well as their minimum and maximum values are all table specific. The constant, $C$, by which we increment or decrement $\theta$, is also specific to each subtable. The value of $C$ should be such that the flow-size threshold would only change gradually. Since the largest subtable has the threshold in a few tens of packets, the corresponding $C$ should be small, say one (packet). Similarly, when the threshold is in thousands, $C$ should be in tens of packets.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_i$</td>
<td>subtable $i$</td>
</tr>
<tr>
<td>$\theta_i$</td>
<td>flow-size threshold for $T_i$</td>
</tr>
<tr>
<td>$f$</td>
<td>a flow ID</td>
</tr>
<tr>
<td>addr</td>
<td>address of a bucket in a (sub)table</td>
</tr>
<tr>
<td>$l_i$</td>
<td>load of subtable $T_i$</td>
</tr>
<tr>
<td>$\eta$</td>
<td>interval between two consecutive calls to Algorithm 1, in number of packets arrivals</td>
</tr>
<tr>
<td>$b$</td>
<td><code>findPathDFS</code> parameter: max no. of paths to explore in a tree</td>
</tr>
<tr>
<td>$h$</td>
<td><code>findPathDFS</code> parameter: depth of the paths explored in a tree</td>
</tr>
</tbody>
</table>

The commonly used notations in this work are listed in Table 2 for quick reference. In the next section, we develop the algorithms for the different operations in REX.

4. Operations in REX

The actions performed by REX on each packet arrival are given in Algorithm 2. The most important operations are `lookup`, `insert`, and `update`. For a new flow, the `insert` operation is performed on the bottommost table in the hierarchy ($T_n$) consisting of only small-size flows. As we shall see, the `insert` and `update` operations form important steps in segregating flows in specific subtables based on their sizes.

Before proceeding to describe the operations, we present an example, given in Fig. 3, that will be revisited to illustrate the working of REX. In the example, REX consists of two Cuckoo hash tables $T_0$ and $T_1$. In REX (as is the case with other hash tables), each bucket has a fixed number of slots, and usually more than one; the operations and algorithms below work on this common generalization. But for the example here, bucket size is assumed to be one; i.e., a bucket has just one slot. The size, in number of slots, of $T_0$ is two, and of $T_1$ is eight. For simplicity, the flow
Algorithm 2 REX(P: packet)

1: packetInfo $\leftarrow \text{retrieveInfo}(P)$
2: addr $\leftarrow \text{lookup}(\text{flowID(packetInfo)})$ \hspace{1cm} \triangleright$ address of a bucket
3: if addr $> -1$ then
4: update(packetInfo, addr)
5: else
6: insert(packetInfo, $T_n$) \hspace{1cm} \triangleright$ T_n$: the bottommost and the largest subtable
7: end if

information tracked is just flow ID and flow size (i.e., the ongoing size of an active flow is updated as when a new packet belonging to the flow arrives). In reality, the protocol, the state, etc. are relevant information that are tracked for security analysis. The figure shows an instant of time when there are packets that arrived sequentially at the system. While $T_0$ is empty, $T_1$ has only one free slot. One of the flows, the flow with ID 31, is indicated as being expired. For Cuckoo hashing, assume $d = 2$; that is, two hash computations are performed for lookups. The flow-size threshold $\theta_1$ for $T_1$ is 10 packets.

![Figure 3: Example: REX has two tables $T_0$ for large flows and $T_1$ for small flows](image)

4.1. Lookup

The most common operation is lookup. Given a flow identifier $f$, the lookup operation will return an address (a non-negative value) if the flow is already stored in REX; otherwise a negative value is returned. The steps are listed in Algorithm 3. In the first line, subtable $T_0$ is probed; if the flow exists in $T_0$, the address is returned. The address corresponds to that of a bucket where the flow resides. If the flow is not present in $T_0$, then the Bloom filters corresponding to all other subtables are probed (line 3). In lines 5-8, the subtables that are likely to contain the flow (based on the BFs that returned positive responses) are queried, and upon success, the address of the corresponding bucket is returned.

Observe that, subtable $T_0$ and the BFs can be probed in parallel; and subsequently, the probing of the subset of the remaining subtables can also be done in parallel. Recall that, to perform an operation in a BF (say, membership query), traditionally $k$ hash functions are computed; where $k$ is the optimal value based on the desired false positive rate. But, a BF can also be implemented using just two hash functions. The two hash functions can generate the $k$ hash functions using simple standard hashing technique, and yet not affect the performance (false positive rate) [32]. However, if there is limitation in performing the queries in parallel, there is a definite querying
Algorithm 3 lookup($f$: flow ID)

1: addr ← queryTable($T_0$, $f$)  \(\triangleright \) address of a bucket in $T_0$
2: return addr if addr > $-1$
3: tables ← membershipQuery($f$)
4: if tables $\neq \emptyset$ then  \(\triangleright \) at least one BF gives positive response
5: for $i \in$ tables do
6: addr ← queryTable($T_i$, $f$)  \(\triangleright \) address of a bucket in $T_i$
7: return addr if addr > $-1$
8: end for
9: end if
10: return -1  \(\triangleright \) flow does not exist in REX

order that is preferred among all possible sequences. As the distribution of packets follows a heavy-tail distribution (see Fig. 1 and Section 3.1), most of the packets are part of large flows. Therefore, if a serial order is mandated due to limitations, the first subtable to be queried is the smallest one $T_0$. Subsequently, the subtables of increasing sizes can be queried (via their corresponding BFs).

4.2. Delete

In REX, deletion happens as part of insert operation; for clarity, we describe it first. A flow entry is removed when it expires due to inactivity; that is, no packet corresponding to the flow arrived for a predetermined constant duration.

The insert operation in REX supports lazy deletion. This is in contrast to periodic deletion where at every time interval a search for inactive flows is performed over the entire table. Lazy deletion, however, takes places during the search operation triggered by the insert operation on the arrival of a new flow. Therefore, in lazy deletion, an inactive flow is removed only when a search for a free slot hits an occupied slot with expired flow. As a trade-off, expired flows might remain in the table indefinitely unless and until they are discovered by the search operation. Therefore, for practical reasons, we need to have periodic deletion to remove expired flows that have remained undiscovered in the table for too long; but the time interval for this periodic deletion can be much longer (by at least an order of magnitude) than in the case where there is only periodic deletion.

4.3. Insert

The steps for insertion of a new flow is listed in Algorithm 4. On a packet arrival, if the corresponding flow is not found in REX, it is considered a new flow and the operation to insert it in (the largest subtable in) REX is invoked. However, insert operation is not guaranteed to succeed, as a slot to store the new flow might not be found; this results in the discard of the packet belonging to the new flow. Therefore, a flow is always new until it is inserted into the structure, even if a number of packets of the same flow might have arrived earlier and got rejected.

For convenience, we refer to a bucket or a slot as suitable, if it is either empty or has an expired flow. A bucket (or slot) that does not fit either of these conditions is referred to as unsuitable. In line 1 of Algorithm 4, the hash functions are computed on the flow ID, and the bucket addresses are retrieved. The subsequent line searches for a suitable slot in these buckets. The search performed by emptyExpiredSlot gives priority to a slot with expired flow in case multiple suitable slots are found in one bucket. If one such suitable slot exists, in lines 3-10 the new flow is inserted, storing also the size, state and other relevant information of the new flow. If the bucket has other slots with expired flows, those flows are ‘summarized’ and then deleted (line 8). With summarize, we mean that the flow details are stored in a database for further and various kinds of analyses related to QoS provisioning, anomaly and attack detection, etc.

The subsequent lines of the algorithms works similar to the Cuckoo table—as the insert operation hashes into unsuitable buckets, a kickout is initiated. There are multiple ways to explore the search for suitable bucket, where the sequence of buckets visited by this search process constitutes a particular path. We implement the depth-first-search (DFS) for exploring the paths
Algorithm 4 \text{insert}(packetInfo, T; table)

1: addr ← buckets(flowID(packetInfo), T) \Comment{address of a bucket in subtable $T$}
2: slot ← emptyExpiredSlot(addr)
3: if slot > -1 then \Comment{either empty or has expired flow}
4: \hspace{1em} if $\neg$ isEmpty(T[slot]) then \Comment{slot not empty}
5: \hspace{2em} summarizeDeleteExpired(T[slot]) \Comment{summarize and delete expired flow in slot}
6: \hspace{1em} end if
7: \hspace{1em} T[slot] ← packetInfo \Comment{store flow, and related features of interest}
8: \hspace{1em} summarizeDeleteExpired(addr) \Comment{summarize and delete expired flows in bucket, if any}
9: return SUCCESS
10: end if
11: path ← findPathDFS(addr, b, h) \Comment{find path to suitable slot}
12: if path $\neq \emptyset$ then \Comment{kick out and insert}
13: \hspace{1em} kickout(concat(slot_f, path), packetInfo)
14: \hspace{1em} return SUCCESS
15: end if
16: return FAIL

(function \text{findPathDFS}, given the addresses of buckets to start with. Function \text{findPathDFS} has two arguments, $b$ and $h$; $b$ is the maximum number of paths to explore in a tree, where the length of each path is limited to $h$. The \text{findPathDFS} function searches for a path of unsuitable slots ending with a suitable slot (line 11). The \text{kickout} procedure on such a discovered path (line 13) will relocate each flow to its alternate location and insert the packet in the only suitable slot in the path. Note that, the \text{kickout} might also involve summarizing and removing an expired flow from REX, if the suitable slot has an expired flow. A successful insertion results in initialization of flow-features being tracked; for example, the size of flow $f$ ($\text{size}_f$). If no such path is discovered, the insertion fails, and the arriving packet (and the corresponding new flow) is not stored in the table.

![Figure 4: Illustration of insert operation](image-url)

Fig. 4 illustrates the insertion of a new flow in REX. Let the ID of this flow be seven. When the first packet of this flow arrives, the lookup logic learns that the flow is not in REX, and therefore computes the two hash functions to determine the two locations in $T_1$ where the flow can possibly be stored. Recall that, a new flow is always inserted in the largest table, $T_1$ here. Fig 4(a) depicts the case where both the locations are occupied. Since $d = 2$, two lookups are done initially; these are denoted as 1 and 1’ in the figure, under the assumption that they can be performed in parallel. Since both the locations are occupied with active flows, REX executes depth-first-search to find a suitable slot (function \text{findPathDFS}), selecting one of the two locations randomly as the root
of the search operation. In our example, the selected root node is the one pointed by the lookup (arc) 1, which holds flow 17. The alternate location for flow 17 is the last slot in T1, which is occupied by another active flow, with ID 53. This search goes on, taking the path as indicated in the figure, until the slot of flow 31 is looked up. As this flow is an expired flow, the search ends successfully; and a kickout process is initiated. Observe in Fig. 4(b), the flows \{17, 53, 94\}, that are in the DFS path of the successful search, are relocated to their respective alternate locations. The expired flow (flow ID 31) is summarized into a database, and then removed from REX.

4.4. Update

Algorithm 5 performs the update operation. In line 2, important features related to the flow are updated, such as state, flow size (sizef, in number of packets), so on. The next step is where REX differs from other hash tables. The algorithm checks the updated flow size; if the size is greater than the threshold for the subtable, the flow is removed from the current subtable and inserted into the next subtable in bottom-up order. The threshold \(\theta_T\) is \(\theta_2\) if the flow is in subtable \(T_2\), and will be moved to subtable \(T_1\) when the flow’s size exceeds \(\theta_2\). For flows in the smallest subtable \(T_0\), the threshold \(\theta_0\) is set to \(\infty\); i.e., they are never moved to another subtable but only removed on expiry.

```
Algorithm 5 update(packetInfo, addr)
1: T: table corresponding to addr
2: updateDetails(packetInfo, addr)
3: if sizef > \(\theta_T\) then \(\triangleright\) updated flow-size is greater than the threshold for the subtable
4: move\((f, T_{i-1})\) \(\triangleright\) move the current flow to the next subtable up in the hierarchy
5: end if
```

The move operation basically deletes the flow from the current subtable, after inserting it into the upper subtable. Yet it performs an important function. The upper subtable where the flow is being moved to, might not always be able to find a free slot for the flow. Though normally such a scenario leads to the flow being rejected, and therefore discarded, rejecting a flow due to an internal move operation is inefficient. The partial information collected on the flow is lost, while having allocated a slot for it all along. Hence, an important property that REX is designed to have is thus: once a flow is inserted into REX, it is always tracked. Therefore, when the insertion to the upper subtable fails, instead of rejecting and thus dropping the resident flow, the flow continues to reside in the current subtable. Besides, this also indicates that the threshold for the subtable has to be changed (increased). Hence, flows in a subtable may not always respect the size-threshold; it is quite possible that a flow of size greater than the threshold of, say, \(\theta_2\) resides in \(T_2\), though only temporarily. As the load of the upper subtable decreases, the probability of finding free slot increases, at which time a move operation on a flow crossing threshold would be successful. Alternately, it is also possible that the flow-size threshold increases (due to Algorithm 1), and consequently the resident flows respect the threshold. Though such wasted lookup consumes time (due to search), this is drastically reduced by having a boolean flag that is set once a move operation fails. As long as this flag is set, no move operation is carried out (on the corresponding subtable); and the flag is reset only when the load decreases.

The simpler case of the update operation, where a flow is not moved from one subtable to another, is observed for flow with ID 7 in figures 4(b) and 5(a). However, on the arrival of the next packet, one belonging to flow 4 (Fig. 5(a)), REX finds that the flow exists in \(T_1\); the updated size of the flow now becomes greater than the threshold \(\theta_1\) for the subtable \(T_1\). Therefore, the flow is moved from \(T_1\) to \(T_0\), as depicted in Fig. 5(b).

5. Resilience in REX

The previous section defined and developed the common hash table operations for REX. In this section, we describe the operation that makes REX resilient in the face of attacks. As the load
increases during an attack phase, leading to an increase in rate of operations on the data structure, we need to build resilience into the data structure to ensure limited effect on the performance.

5.1. Characteristics of attack traffic

Many attacks that overwhelm a flow-tracking data structure are in the form of large number of small flows. It is the high rate of flow arrivals that causes the hash tables to fill up quickly. As link capacity—which can be expressed as a product of flow-rate and flow-size—is fixed, it is easy to note that higher flow-rate and smaller flow-size form a natural attack behaviour. In network too, many malicious activities consist of small-size flows. Examples of anomalies that manifest as small flows are TCP SYN flooding, TCP RESET attack, DNS amplification attack, NTP amplification attack, SSH brute force attack, buffer overflow attack, etc., as well as suspicious activities such as network scans and port scans. All such anomalous traffic are made of flows that are small in size. Recall that, by design REX segregates all small flows into one subtable; thus, since most attack flows are small in size, they will concentrate in the table for small flows ($T_n$).

Going further, we observe that different types of flows have different expiry times. Established TCP flows are most often normal connections, and such connections become inactive (with no packet transmitted in either directions) mostly in the termination phase when they are in the waiting states. While this waiting duration is dependent on the configuration of the operating systems, it is usually in minutes. Therefore lazy deletion has expiry set in minutes. Inactivity periods are also likely in keep-alive connections, but such connections usually see keep-alive probes being sent periodically.

Attack traffic has different characteristics. For example, consider TCP SYN flooding. An attack flow consists of SYN packet in one direction and a SYN+ACK from the victim in the reverse direction. As the SYN+ACK goes to a spoofed and non-existent IP address, no ACK is returned. During the connection establishment phase, the timeout mandated by RFC 6298 is one second [33]. In the absence of acknowledgment, the sender attempts a few fixed number of times, with the RTO (retransmission timeout) doubling each time. We can safely assume that an inactivity of 30 seconds during connection establishment phase is a definite sign of connection timeout; and timeouts in the early phase of connection are suspicious. This can be generalized, even more so, for UDP-based connections. A DNS response to a query is expected in less than a second, the response time being dominated mostly by the RTT (round-trip time, which ranges from tens to hundreds of milliseconds). To highlight the relevance, a DNS reflection and amplification attack consists of large number of incomplete ‘connections’ at both the attacker and the victim—attacker sends requests, but responses go to the victim with the spoofed IP address.
5.2. Load-based proactive deletion

Indeed, expiry based on the flow sizes can be generalized as follows. We maintain two timers: if the ‘ongoing’ size of a flow is less than, say, \( \kappa \) number of packets, use the smaller value for timeout, otherwise use the higher value of timeout. We call the deletion based on the smaller expiry timeout as proactive deletion. The deletion based on the larger value of expiry time is the lazy deletion (as described in Section 4.2). For REX, proactive deletion is an inexpensive operation, as it does not have to explicitly search for flows of small sizes for (proactive) deletion; instead, deletion happens during the search performed at the time of insertion. In addition, we employ smaller expiry time only on the largest subtable, as this is the subtable with the small size flows. In fact, we go one step further, by applying what we call load-based proactive deletion. Since proactive deletion is most useful during attacks and traffic spikes, REX invokes proactive deletion on the largest subtable \( T_n \) only when the load of that subtable is higher than a threshold.

Using different timers for removing old flows is common in routers. For example, Cisco allows users to configure a few timers, based on the aging of flows (for example, if a flow was inactive for a given time duration, or have had less than a specified number of packets in the given duration, etc.) [34, Ch. 3]. While currently such timer-based removals are usually applied to all kinds of flows, in REX, we apply them only on the largest subtable, when the load is high, and during the kickout process.

6. Performance evaluation

In the context of tracking network connection flows, we compare the performance of REX against that of Cuckoo and Peacock hash tables. In the process, we evaluate all three data structures for both efficiency and resilience. To this end, we developed an emulation environment, where all the three hash tables and the corresponding operations are implemented. We carried out the experiments on a standard server configuration of the day—Intel Xeon Processor E5-2640 (6 cores, 2.50GHz) with 64 GB RAM, and running Ubuntu OS.

We use real network traces obtained from the MAWI traffic dataset of the WIDE project [31] for experiments. Based on the timestamps in the pcap file, we run the experiments for two hours of network activity. For each hash table, our evaluation system processes packets in the order they appear in the pcap file, which in turn is in chronological order. Processing is nothing but the execution of different operations on the data structures. The data structures are all initialized just once, at the start of the network activity. We present the results in sections below; in Section 6.2, experiments are evaluated on normal traffic flows, thereby testing the efficiency of the hash tables. Section 6.3 focuses on evaluating the resilience of the hash tables, and therefore the experiments are run over data that has malicious traffic at different levels of intensities. Before we present the scenarios and the results in detail, we describe the experiment settings.

6.1. Settings

The implementation of all the table operations for Cuckoo hash table are the same as those for the subtables of REX. Recall that, the subtables of REX are indeed Cuckoo tables. The number of hash functions used for hashing a flow identifier is two. Therefore, for Cuckoo hash table and REX, two hash functions are computed on each packet, to determine the addresses of the corresponding two buckets where the flow can reside. For the findPathDFS function used in the insert operation (refer Section 4.3), the maximum number of paths searched \( b \) is set to three, and the length of each path \( d \) is limited to five.

Peacock hash table is implemented using three subtables in the size ratio of 100:10:1 (that is, a common ratio of 0.1, as in the original work [4]). The maximum probe sequence length for each subtable is set to five. Thus, the maximum number of buckets probed while searching for a slot to insert a new flow is 15 in all the hash tables (Cuckoo, Peacock and REX). In [4], the authors also define rebalancing for Peacock hashing, a process aimed to keep most elements in the larger subtables, by periodically searching for and transferring elements in the smaller subtables to larger ones. But, we do not implement the computationally expensive rebalancing; to be fair,
we neither implement rehashing in Cuckoo table and REX. However, we highlight that, in our implementation of the Peacock hashing, the two smaller subtables reside in the SRAM, and only the main table resides in the DRAM, making it efficient to do operations on smaller subtables.

We have two implementations of REX, named REX-0 and REX-1 in all the scenarios below. The two implementations have lazy deletions in the all subtables; but REX-1 performs proactive deletions of flows in the largest subtable, when the load of the largest subtable goes beyond 0.9 (i.e., under high load). Both Cuckoo and Peacock hash tables employ lazy deletion. The expiry durations for lazy deletion and proactive deletion are, 120 seconds and 30 seconds, respectively.

We carry out experiments for four different scenarios, varying in load\(^5\), traffic type and number of subtables in REX. The scenarios are summarized in Table 3, and presented in the following sections. In addition, parameter values in each scenario are provided in Table 4. In the first three scenarios, REX has two subtables—\(T_0\) and \(T_1\), with \(T_0\) being the smaller table for ‘large’ flows. In these scenarios, there is one flow-size threshold, \(\theta\), that distinguishes between ‘small’ and ‘large’ flows at a given point in time. The threshold \(\theta\) adapts dynamically as described in Algorithm 1 (we set \(\eta\) to a reasonably large value 1000; which means, Algorithm 1 will be invoked once every 1000 packet arrivals). The lower bound for \(\theta\) is set to five packets, and upper bound is infinite, such that \(\theta\) has wide range of values to adapt to. The target load range for \(T_0\), depending on which \(\theta\) varies, is \([0.9 - 1.0]\); this is to ensure that \(T_0\), which resides in the fastest and smallest memory, is highly utilized. Next, we come to the sizes of the hash tables. In the first three scenarios, the ratio of sizes of subtables (\(T_0 : T_1\)) in REX is 1:10. Therefore, the subtable \(T_0\) of each REX implementation is assumed to be stored in SRAM, whereas \(T_1\) would be in DRAM. The smaller two subtables of Peacock hash table are also stored in SRAM, with its main table being stored in DRAM. Since Cuckoo table is just one large table, it can reside only in DRAM. The bucket size for all the data structures is set to four slots.

Table 3: Experiment scenarios. # denotes the number of subtables, ‘size ratio’ refers to the ratio between different subtables (with respect to the smallest subtable), ‘access time ratio’ is with respect to the memory technologies where the subtables reside.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Traffic Type</th>
<th>Size of hash tables</th>
<th>Load</th>
<th>REX-0 &amp; REX-1 Subtables</th>
<th>#</th>
<th>size ratio</th>
<th>access time ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (Sec. 6.2.1)</td>
<td>Normal</td>
<td>80,000</td>
<td>low</td>
<td>2</td>
<td>1:10</td>
<td>1:5 ((T_0 : T_1))</td>
<td></td>
</tr>
<tr>
<td>2 (Sec. 6.2.2)</td>
<td>Normal</td>
<td>45,000</td>
<td>high</td>
<td>2</td>
<td>1:10</td>
<td>1:5 ((T_0 : T_1))</td>
<td></td>
</tr>
<tr>
<td>3 (Sec. 6.3.1)</td>
<td>Normal + Malicious</td>
<td>40,000</td>
<td>high</td>
<td>2</td>
<td>1:10</td>
<td>1:5 ((T_0 : T_1))</td>
<td></td>
</tr>
<tr>
<td>4 (Sec. 6.3.2)</td>
<td>Normal + Malicious</td>
<td>40,000</td>
<td>high</td>
<td>3</td>
<td>1:10:100</td>
<td>1:2:10 ((T_0 : T_1 : T_2))</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: REX parameter settings for different experiment scenarios.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Range for size threshold</th>
<th>Target load ranges for subtables</th>
<th>Constant C</th>
<th>(\eta)</th>
<th>findPathDFS parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (Sec. 6.2.1)</td>
<td>(\theta_1 = [5, \infty])</td>
<td>(T_0 : [0.9 - 1.0])</td>
<td>(C_1 = 1)</td>
<td>1000</td>
<td>(b = 3) (h = 5)</td>
</tr>
<tr>
<td>2 (Sec. 6.2.2)</td>
<td>(\theta_1 = [1000.0, \infty]) (\theta_2 = [5, 1000])</td>
<td>(T_0 : [0.95 - 1.0]) (T_1 : [0.75 - 1.0])</td>
<td>(C_1 = 10) (C_2 = 1)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the first scenario (Section 6.2.1), each data structure can hold up to 80,000 flows; this size is set to have low load at all the hash tables. In the second scenario (Section 6.2.2), we reduce the table size to 45,000 flows, to allow the hash table load to go beyond a high value of 80% most

\(^5\)Load of a hash table is computed as the sum of the occupied slots over all subtables divided by the sum of the sizes of all subtables.
of the time. In the last two scenarios (presented in Section 6.3), to increase the load of the hash tables further, we bring down the size of each data structure to 40,000 flows. Each slot can store one flow. The REX implementations have three subtables in the last scenario; the modifications for this scenario from the above settings are presented in Section 6.3.2.

The unit of time we use for comparison purposes below (for insert and update operations), is the time to access a bucket in memory. The access time for a subtable depends on the memory that it resides in. As our experiments are in an emulated environment, we count the number of read and write operations on each subtable and multiply the same by the corresponding access time. The access time of DRAM is much higher than that of SRAM; we assume a conservative ratio of 5:1 for the (relative) access times of DRAM to SRAM [16, 35]. Specifically, we assume access times of 50ns for DRAM and 10ns for SRAM.

6.2. Experiments using normal traffic

This section aims to evaluate the hash tables using normal traffic flows. As mentioned previously, we obtained a MAWI dataset; this is a two-hour dataset that starts at around 2:30 AM on 10th December 2014. The dataset is processed to remove all non-TCP flows (e.g., UDP flows) as well as those TCP flows that do not conform to TCP’s finite state machine. Using such a method, we could remove not only TCP related attacks (such as SYN flood), but also TCP port scans and network scans. Furthermore, by identifying signatures and commonly repeated patterns in the traffic, we also detected and removed SSH brute force attacks from the dataset. The final processed data comprises of a total of more than three million flows and 100 million packets. More than 90% of these TCP flows had less than 30 packets.

6.2.1. Scenario 1: Low load; REX with two subtables

In this scenario, the sizes of the data structures were set such that, the load in all tables were low—between 0.4 and 0.6 at all times of the experiment. There are only two subtables in REX. The total number of packets dropped by each hash table was negligible (less than 10). Fig. 6(a) plots the cumulative memory access time to insert new flows in each of the tables, until a given time. Recall that, insert operation involves searching (and performing lookups of) buckets; and the insert time is the sum of the time required to access all buckets in a search path. For both REX-0 and REX-1 implementations, the time to move flows from $T_1$ to $T_0$ is also included in the estimated time here. However, only flows greater than the flow-size threshold $\theta$ move from $T_1$ to $T_0$; besides the search time in $T_0$ is much less due to the significantly smaller access time of SRAM in which $T_0$ resides. Therefore, even though REX hash tables are seen to have higher insertion times than Cuckoo and Peacock tables, the difference is negligible.

Fig 6(b) plots the cumulative time taken to update resident flows (the plots for Cuckoo and Peacock tables are overlapping, and so are the plots for REX-0 and REX-1). Observe that both implementations of REX bring in significant reduction in the time to update resident flows, in comparison to Cuckoo and Peacock hash tables. This is because, most of the updates in REX are on the flows stored in the subtable $T_0$ which resides in SRAM. In the case of Cuckoo hash table, as it has one large table residing in DRAM, the updates take much more time. The interesting case is with Peacock hash table—even though its smallest two subtables are stored in SRAM, it is not able to make good use of the fast memory. Recall that, unlike in REX, in Peacock hashing there is no guarantee that the flows with the largest number of updates are stored in the subtables that reside in SRAM. Besides, for REX and Cuckoo table implementations, the update operation involves just two memory lookups; whereas for the Peacock hash table, even if there are Bloom filters to reduce the number of subtables being searched, within one subtable it still has to perform a linear probe. Due to the fact that the number of packets is always higher than the number of flows, the performance improvement gained in update operation affects and dominates the total memory access time taken to process packets. This is reflected in Fig. 6(c). We observed that, at the end of the experiment, REX hash tables bring about $37 - 38\%$ reduction in both update time as well as total time, in comparison to Cuckoo and Peacock tables. Another notable aspect is that, the load of subtable $T_0$ is maintained within a small range as aimed in the design, thanks
(b) Performance comparison of Cuckoo, Peacock, REX-0 and REX-1 hash tables. First three figures plot the cumulative time for the mentioned operations.

to the adaptive threshold algorithm (Algorithm 1). Fig. 6(d) presents the loads of the subtables of REX-1. The corresponding figure for REX-0 was similar, and hence not plotted.

6.2.2. Scenario 2: High load; REX with two subtables

For this scenario, we decreased the size of the hash tables to increase the effective load. The incoming traffic intensity at all the hash tables (Cuckoo, Peacock and REX tables) are the same; but at a high incoming traffic intensity, due to the inherently different designs of each hash table, the load values are unlikely to be the same. Hash tables start discarding incoming packets when they cannot find slots for inserting new flows. Note that, there is a limit to the number of searches in REX and Cuckoo hash tables (due to Cuckoo hashing); therefore, packets may be discarded even when there are vacant slots, and this failure increases at higher traffic intensities.

Fig. 7(a) plots the load of each hash table over the experiment duration. From the figure, we observe that for all the hash tables, the load is almost always higher than 0.75. Hence, we get to evaluate the hash tables at varying high loads. Observe that, Peacock table has the least load. As seen in Fig. 7(c) (and ignoring the initial transient period), this is due to the high number of packets Peacock hashing rejected, or equivalently, the number of times it failed to insert new flows. While most of the time, REX-1 has similar load values as Cuckoo hash table and REX-0, there are two time periods when the difference is more explicit—at around 900s-1500s and 3700s-4100s. REX-1 proactively deleted stale flows during these periods, as the load in $T_1$ increased beyond the predefined limit of 0.9. Fig. 7(b) plots the load of both subtables of REX-1. Observe that the load of subtable $T_0$ is maintained steadily at around 0.9, ensuring the SRAM is highly utilized by REX. Consequently, the load of $T_1$ fluctuates depending on the incoming traffic intensity.

Fig. 7(c) plots, for all three data structures, the cumulative number of packets rejected as a function of time, due to failed insertions. After the initial transient period, all the hash tables are seen to reject packets of new flows. We see that, the number of packets rejected by Peacock hashing is much higher than that of Cuckoo and REX hash tables. For the given pcap file, REX-1 discarded the least, around 81% less than Peacock hashing, and around 30% less than both Cuckoo and REX-0 hash tables. Observe that the abrupt increases in rejects, which is most visible for Peacock hashing, correspond to the times of high load. While Peacock hashing relies on the ‘backup’ subtables when collision occurs (at all buckets in one subtable), this is evidently not as good as the kickout process of Cuckoo hashing. Due to the kickout process, collisions get resolved; and therefore Cuckoo and the REX hash tables successfully find spaces for new flows. Correspondingly, the time to insert new flows is also higher in Peacock table, as is seen in Fig. 7(d). The differences in insert times are negligible among the remaining tables.

Fig 7(e) plots the cumulative update time as the traffic flows arrive. Similar to previous scenario, both implementations of REX take the least time to update existing flows. Cuckoo hash table is seen to perform the worst. In this high-load scenario, all subtables of Peacock table are occupied with flows; therefore, for the flows that reside in the smaller two subtables of Peacock table, the time to access a bucket is smaller than that of Cuckoo table. But yet, unlike REX, Peacock table is not designed to store more frequently updated flows, i.e, large flows, in the smaller subtables. This is reflected in the performance gain attained by the REX-0 and REX-1 hash tables.
Finally, in Fig. 7(f) we observe that, the total time for insert and update operations is the least for REX hash tables. Though REX-1 removes stale flows faster than REX-0, the effect was not significant enough to reduce the total time for insert and update operations. Flow-tracking data structures will face significantly large number of updates in comparison to inserts during normal scenarios, simply because a flow most often has multiple packets. Therefore, both REX implementations are able to perform better than the other two hash tables in the total time taken for the important table operations. Comparing the total time for operations at the end of the experiment, we see that the REX implementations reduce the time by $\approx 34\%$ relative to Cuckoo hash table, and by $\approx 30\%$ relative to Peacock hash table.

6.3. Experiments using malicious traffic

In this section, normal traffic is mixed with real malicious traffic, to evaluate the performance of the hash tables at different intensities of malicious activities. Into the two-hour normal traffic we had processed for the previous section, we injected malicious traffic at four different time points, with each malicious activity lasting at least 30 seconds. The intensities of malicious traffic are 10%, 20%, 30%, and 40%, in order; where the intensity is measured as the number of malicious flows that are injected as a percentage of the normal flows during the duration of the attack. The time between two consecutive malicious activities is fixed at $\approx 1440$ seconds. The first such activity is injected at $\approx 740$ seconds from the start of the experiment. The malicious activities, all gathered from MAWI dataset of real network flows, consist of network scan, port scan, TCP SYN floods, and SSH brute force. Below, we present results of two experiment scenarios, one in which REX implementations have two subtables, and another in which REX-0 and REX-1 have three subtables.

6.3.1. Scenario 3: Malicious traffic; high load; REX with two subtables

The results are plotted in Fig. 8, wherein we have indicated the points in time when the malicious traffic spikes are introduced as well as the intensity levels. Fig. 8(a) plots the time-varying total load of the hash tables. It depicts that, all hash tables, with the exception of REX-1,
show spikes in their loads at these marked points in time; the spikes are highest for Cuckoo table and REX-0. From Fig. 8(b), we observe that the load in $T_0$ of REX-1 becomes stable over time, even when there are spikes of malicious traffic.

In Fig. 8(c), Peacock table is seen to discard the most number of packets (leading to its relatively lower load reflected in Fig. 8(a)). Both REX-0 and Cuckoo hash tables also rejected large number of packets, with the counts increasing at every spike of malicious traffic. But REX-1, with the implementation of load-based proactive deletion on $T_1$, rejected far less—an order of magnitude fewer than the rest. At a snapshot soon after the fourth spike, Peacock table discarded more than double the number of packets than REX-0 and 30 times more than REX-1. By analyzing the flows removed by the hash tables, we observed that REX-1 always deleted more number of malicious flows than the other three, due to both lazy and proactive deletions (recall that the load-based proactive deletion is performed on the flows in $T_1$ for REX-1 only when the load is greater than 0.9). In comparison to Peacock, Cuckoo and REX-0 hash table, REX-1 deleted (approximately) 87,000, 24,000, and 29,000 more flows, respectively.

Fig. 8(d) reveals that, for Cuckoo, Peacock and REX-0 hash tables, the time to insert flows increased at each of the traffic spikes. Different from previous scenarios, REX-1 now performs significantly better than the rest, when it comes to the time taken to insert new flows. However, as update operations dominate, and because both REX implementations perform similarly when analyzed using update time (see Fig. 8(e)), there is no perceivable performance difference between the two in terms of the total time (Fig. 8(f)). This reduction in insert time achieved by REX-1 is due to proactive deletion, which enables REX-1 to discard short malicious flows, not only earlier than the other hash tables, but also while searching for suitable slots during insert operation. The trend for update time, as shown in Fig. 8(e), is similar to the previous scenario, with both REX implementations taking the least time. Consequently, as can be observed from Fig. 8(f), the relative reduction in total time brought by both REX-0 and REX-1 hash tables is $\approx 29\%$ with respect to Peacock table, and $\approx 34\%$ with respect to Cuckoo table.
6.3.2. Scenario 4: Malicious traffic; high load; REX with three subtables

Finally, we also conducted experiments using three subtables for REX. The total table size was kept at 40,000 to have high load. The ratio of subtable sizes in REX ($T_0 : T_1 : T_2$) was set to 1:10:100. Correspondingly, we assume three memory technologies (for example, DRAM, off-chip SRAM, and on-chip SRAM); and set the ratio of access times to 10:2:1. As in the previous scenarios, the slowest memory (DRAM) is assumed to have an access time of 50ns. Therefore, the access times of $T_0$, $T_1$, and $T_2$ are 5ns, 10ns, and 50ns, respectively. Similarly, the three subtables of Peacock hashing are also assumed to reside in the three different memories; as the ratio of sizes is the same as for REX, the largest subtable of is assumed to reside in DRAM, the next subtable in off-chip SRAM, and the smallest in on-chip SRAM. The Cuckoo table resides in DRAM.

Fig. 9 presents the results. In Fig. 9(a), we observe that, different from the 2-subtable scenarios, both REX-0 and REX-1 take higher time for insert operations than Cuckoo and Peacock hash tables. With both the REX implementations now having three subtables, the number of insert operations is higher than in previous scenarios. Each flow in $T_0$ has undergone three insert operation, one each in $T_2$, $T_1$, and $T_0$. Similarly, every flow in $T_1$ has been inserted first in $T_2$ and then in $T_1$. Besides, failed attempts to move flows from one subtable to an upper subtable, are accounted for in the insert time. In Fig. 9(d), we see that the load of $T_0$ was maintained high and close to the configured value of 0.95 most of the time. However, the subtable $T_1$ was always almost full, leading to increased insert times. Yet, the increase in insert time (of REX) has negligible effect on the total time for operations (see Fig. 9(c)). As in previous scenarios, REX-0 and REX-1 show noticeable reduction in time to update flows (see Fig. 9(b)). We also observed that, the number of packets discarded by REX-1 was minimum, and was less than that of Cuckoo and REX-0 tables by an order of magnitude; Peacock hashing discarded the highest. This result is similar to Fig. 8(c) and hence not plotted here.

6.4. Discussion

To summarize, we analysed the performance of the data structures in different scenarios, ranging from low to high load, using normal as well as malicious traffic, and with different number of subtables. We observed that, both REX-0 and REX-1 always significantly reduce the time to update flows stored in the tables, in comparison to Cuckoo and Peacock hash tables. Since flows have multiple packets, update operations occur more frequently than insert operations. Interestingly, even while Cuckoo hashing takes just two lookups for an update operation, on average, Peacock hashing takes less time than Cuckoo hashing for updating flows. Evidently, this is due to Peacock table’s design of using smaller but faster memory technologies. REX improved this performance gain a step further, by exploiting the heavy-tailed nature of Internet traffic.

Between REX-0 and REX-1, the latter clearly performed better due to proactive deletion. For scenarios in which REX-1 had two subtables, the gain in insert time was noteworthy in the presence of malicious traffic. In all scenarios, REX-1 discarded minimum number of packets, and with increasing load, the difference between REX-1 and other hash tables was very high.

In the scenario where REX consisted of three subtables, both REX-0 and REX-1 incurred higher insert time. This is one disadvantage of having higher number of subtables—flows are
moved from one subtable to another, resulting in higher insert time. Besides, we also observed very high load in the intermediate table. A more sophisticated algorithm to adjust the threshold is required when REX has three or more subtables. These experiments also indicate that REX performs better on all metrics when configured with two subtables rather than three (subtables). Nevertheless, we highlight that REX with three subtables outperforms both Cuckoo and Peacock hashing (with three subtables).

REX is resilient to flooding attacks, as by design, REX does not evict resident flows; instead, REX drops (first few) packets of non-resident flows during high load. On the other side, if there a parallel attack (for example, an exploit) of just a few packets during flooding, there is no guarantee that REX will insert and track the attack flow. However, such a capability is hard to achieve, given there is no prior knowledge of attacks.

7. Conclusions

In this work, we designed and developed REX, a resilient and efficient data structure for tracking network flows. The design of REX exploits, both, the different memory technologies and the inherent mice-elephant phenomenon of the Internet traffic. We carried out extensive experiments in four different scenarios, to compare REX with two well-known data structures—Cuckoo and Peacock hash tables. In the experiments, we had two implementations, REX-0 and REX-1, the latter to separately evaluate the performance gain due to load-based proactive deletion. The experiments demonstrated that both implementations of REX perform better than Peacock hash table in reducing not only the number of packet discards, but also the total time taken for the important hash table operations of insert and update. While the performances of REX-0 and Cuckoo table were similar in terms of the number of packets discarded, the total time for operations was significantly lower in REX-0 in comparison to Cuckoo table. With proactive deletion, REX-1 (with two subtables) is able to outperform Cuckoo hash table on all metrics evaluated, under scenarios of both normal traffic and high intensity malicious traffic.

One interesting aspect to explore in future is on reducing the number of kickouts performed by REX during insert operation. This, nonetheless, is also an important problem in Cuckoo hashing. More specific to REX, is to reduce the total number of displacements of large flows starting from its first location in the largest subtable $T_n$ to its final location in the smallest subtable. Yet another question is: can we reduce the number of displacements of large flows, by directly inserting into $T_0$ (instead of $T_n$) when the load is low? We leave these questions for future work.

Acknowledgment

This material is based on research work supported by the Singapore National Research Foundation under NCR Award No. NRF2014NCR-NCR001-034.

References


