Caesar: High-Speed and Memory-Efficient Forwarding Engine for Future Internet Architecture

Mehrdad Moradi[†] Feng Qian [‡] Qiang Xu[×] Z. Morley Mao[†] Darrell Bethea[⋆] Michael K. Reiter[⋆]
University of Michigan[†] Indiana University [‡] NEC Labs[×] University of North Carolina [⋆]

ABSTRACT

In response to the critical challenges of the current Internet architecture and its protocols, a set of so-called clean slate designs has been proposed. Common among them is an addressing scheme that separates location and identity with self-certifying, flat and non-aggregatable address components. Each component is long, reaching a few kilobits, and would consume an amount of fast memory in data plane devices (e.g., routers) that is far beyond existing capacities. To address this challenge, we present *Caesar*, a high-speed and length-agnostic forwarding engine for future border routers, performing most of the lookups within three fast memory accesses.

To compress forwarding states, Caesar constructs scalable and reliable Bloom filters in Ternary Content Addressable Memory (TCAM). To guarantee correctness, Caesar detects false positives at high speed and develops a blacklisting approach to handling them. In addition, we optimize our design by introducing a hashing scheme that reduces the number of hash computations from k to $log(k)$ per lookup based on hash coding theory. We handle routing updates while keeping filters highly utilized in address removals. We perform extensive analysis and simulations using real traffic and routing traces to demonstrate the benefits of our design. Our evaluation shows that Caesar is more energy-efficient and less expensive (in terms of total material cost) compared to optimized IPv6 TCAM-based solutions by up to 67% and 43% respectively. In addition, the total cost of our design is approximately the same for various address lengths.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network communications; C.2.6 [**Internetworking**]: Routers

Keywords

Future Internet Architecture; Bloom Filters; Border Routers

1. INTRODUCTION

Aiming at providing a more secure, robust, and flexible Internet, the networking research community recently has focused on developing new architectures for the next-generation Internet. For example, AIP [12] introduces accountability at the IP layer, thus enabling simple solutions to prevent a wide range of attacks. XIA [21] supports an evolvable Internet by providing the capability to accommodate potentially unforeseen diverse protocols and services in the future. MobilityFirst project aims at developing an efficient and scalable architecture for emerging mobility services [9].

All of these proposals share two important features of their addressing schemes: each address is decoupled from its owner's network location and permits its owner to cryptographically prove its ownership of the address. The separation feature enables improved mobility support and multihoming. The cryptographic aspect facilitates authentication and authorization of control and data messages. However, on the down side, both features require addresses to be inherently *long* and thus take up significant memory space due to a lack of hierarchical structure to support aggregation. For instance, in the design of MobilityFirst [9], each address component can be a few kilobits in size. Not surprisingly, it is expected to have forwarding tables on the order of gigabytes in future Internet architecture designs [21]. Such addressing schemes make the design and implementation of high-speed *border routers* challenging as detailed below.

First, memory provisioning becomes more difficult compared to existing network elements. The future Internet will experience a tremendous surge in the number of addressable end-points. Recent studies [8, 6] have predicted that the number of connecting devices and active address prefixes will jump to 50 billion and 1.3-2.3 million, respectively, by the end of 2020. On the other hand, the current rapid growth of the number of address prefixes (i.e., about 17% per year) is the root of many existing problems for operators, who have to continuously shrink the routing and forwarding tables of their devices or upgrade to increasingly more expensive data planes [13].

Second, power consumption of border routers is expected to increase substantially. Most of high-speed routers and switches utilize a specialized fast memory called Ternary Content Addressable Memory (TCAM) due to its speed and in particular its parallel lookup capabilities. TCAM is the most expensive and power-hungry component in routers and switches. It requires 2.7 times more transistors per bit [11] and consumes an order of magnitude more power [35] compared with the same size of SRAM. Therefore, increased address length imposes substantial cost and power consumption in particular on high-speed border routers with TCAM. Although software-based solutions might seem viable, their forwarding speed cannot compete with TCAM-based routers that can support up to 1.6 billion searches per second under typical operating conditions [3].

Third, the critical-path fast memory components of high-speed routers are small in size, and their capacity does not increase at a rate that would accommodate the large addresses of future Internet designs in the foreseeable future. Moore's law is only applicable to slow memories (i.e., DRAM) but not to fast memories [4]. As a matter of fact, we have observed that the TCAM capacity of the state-of-the-art high-speed routers has remained mostly unchanged for several years. As a result of limited memory, network operators still have difficulties in dividing the memory space between IPv4 and IPv6 addresses [7].

To address these challenges, recent research has offered scalable routing tables [29] and forwarding engines (e.g.,

storage-based [25] and software-based [21]) for the new addressing schemes. Unfortunately, these solutions have limited performance due to the approach of storing addresses into slow memories. Also, due to a lack of address compression, efficiency and scalability of their proposed schemes are inversely proportional to the length of addresses. The same limitation also makes a large body of research in IP lookup [27], which optimizes longest prefix matching, ill-suited for flat and non-aggregatable addresses.

This paper presents *Caesar*, a high-speed, memory-efficient, and cost-effective forwarding and routing architecture for future Internet border routers. At a high level, Caesar leverages Bloom Filters [14], a probabilistic and compact data structure, to group and compress addresses into flexible and scalable filters. Filters[1] have been used in designing routers for both flat (e.g., [33, 20]) and IP (e.g., [15, 31]) addresses. These designs are optimized for small-scale networks (e.g., layer two networks) and do not provide *guaranteed* forwarding speed and *full* correctness. Therefore, Caesar focuses on improving performance, memory footprint, energy usage, and scalability of routers deployed at future Internet domain borders. In particular, we make the following contributions:

- We propose a new method for grouping self-certifying addresses into fine-grained filters. The grouping scheme minimizes route update overhead and supports diverse forwarding policies. We also design the first high-speed forwarding engine that can handle thousands of filters and forward almost all incoming packets within three fast memory accesses.
- We design a backup forwarding path to ensure the correctness of forwarding. Our approach leverages the multi-match line of TCAM to detect false positives at high speed. We also introduce a blacklisting mechanism that efficiently caches RIB lookup results to minimize the frequency of accessing slow memory. In contrast, previous work either accesses slow memory several times per packet [15] or randomly forwards packets [33] when false positives occur.
- We strategically leverage counting filters [18] to support address removal while keeping the memory usage benefits of standard filters for high-speed forwarding. To achieve the best of both worlds, for each standard filter in TCAM, we construct a "shadow" counting filter in slow memory and *always* keep standard filters highly utilized in address removal and insertion procedures.
- Based on hash coding theory [36], we propose a hash computation scheme for filters to reduce the number of computations from k to $\log(k)$ per lookup (k is the number of hash functions for a filter). We show that the lookup processing overhead can be reduced by up to 70% compared to the flat scheme. Also, our scheme requires at most $1.16\log(k)$ hash computations for finding k different positions in a small filter while the flat scheme needs up to $1.5\,k$ computations.
- We perform analysis and extensive simulations using real routing and traffic traces to demonstrate the benefits of our design. Caesar is more energy-efficient and less expensive (in terms of total material cost) compared to optimized IPv6 TCAM-based solutions (e.g., [34]) by up to 67% and 43% respectively. In addition, the cost of our design remains constant for various address lengths.

[1]We use "filter" as a shorthand for Bloom Filter throughout this paper.

2. BACKGROUND AND MOTIVATION

Caesar's focus is on the *generalized future Internet architecture*. As illustrated in Fig 2a, the generalized architecture is comprised of a set of *independent accountable domains (IADs)*. An IAD represents a single administration that owns and controls a number of small *accountable domains (ADs)*. For example, an AD can be a university or an enterprise network. In this model, each end host uses a global identifier (GID) to attach to ADs. In addition, a logically centralized name resolution service stores GID $\longleftrightarrow$ AD mappings to introduce new opportunities for seamless mobility and context aware applications.

Packet forwarding at borders? The architecture has different routing and forwarding mechanisms compared to today's Internet. In particular, borders routers sitting at the edge of ADs build *forwarding states* or mappings between *destination ADs and next-hop ADs*. Formally, when a border router of AD_i receives a packet destined to $AD_d : GID_d$, it forwards the packet, through a physical port, to a next hop AD on the path to AD_d. The same procedure occurs until the packet reaches a border router of AD_d. Finally, based on GID_d, it is sent to an internal router where the destination end host is attached. In this procedure, AD addresses are cryptographically verifiable and thus they are long and non-aggregatable. The length of addresses is typically between 160 bits [12] and a few kilobits [9] leading to forwarding tables on the order of gigabytes [21]. In the future, larger address lengths are expected to counter cryptanalytic progress.

Why Bloom filters? Caesar employs filters to compress the forwarding states, i.e., AD to next hop AD mappings, in the *border* routers. However, Caesar can be extended to support forwarding schemes with more components in its other pipelines (e.g., XIA [21]). It also supports various standard forwarding policies (e.g., multi-path and rate-limiting). A filter is a bitmap that conceptually represents a group. It responds to membership test queries (i.e., "Is element e in set E?"). Compared to hash tables, filters are a better choice. First, they are length-agnostic, i.e., both long and short addresses take up the same amount of memory space. Second, a filter uses multiple hash values per key or address, thus leading to fewer collisions. In the insertion procedure, a filter computes k different and uniform hash functions $(h_1, h_2, \ldots h_k)$ on an input and then sets the bits corresponding to the hash values to 1. In a membership test, a similar procedure is followed; if all the bits corresponding to hash results have the value of 1, it reports the element exists otherwise the negative result is reported.

2.1 Caesar Design Goals and Challenges

Using filters for minimizing fast memory consumption poses several design challenges that are unique to the future Internet scale and Caesar's role as a high-speed border router, which make our work different from previous designs using similar techniques (e.g., [20, 33, 31, 17]).

Challenge 1: Constructing scalable, reliable and flexible filters. Compared to the future Internet scale, a data center or enterprise network is very small in size with orders of magnitude fewer addresses. In such *single-domain, small-scale* networks, designing filters to compress forwarding states of flat addresses (e.g., layer two (MAC) addresses) is straight-forward. One widely used approach is to construct multiple filters in each switch, each storing destination addresses reachable via the same next-hop port on the shortest

path (e.g., see [33, 20, 17]). Based on this approach, each switch generates and stores a few very *large* filters in terms of bit length and constituent members (addresses) since the number of ports on a switch is limited.

We argue this filter construction is very coarse-grained and thus not sufficiently scalable and flexible to be used in Caesar, because our target network consists of *multiple independent domains* and has a *higher scale*. First, there can be millions of AD addresses in the future Internet, putting tremendous pressure on the forwarding plane. It is neither scalable nor reliable to store hundreds of thousands of AD addresses into each filter. This is because even a single bit failure in the filter bitmap can risk correctness by delivering a large portion of traffic to wrong next-hop ADs. Second, AD addresses are from various administrative domains, each of which can publish extensive routing updates. Because of storing many addresses into a few large filters, the above approach interrupts or "freezes" packets in the forwarding pipeline at a higher rate in response to each update. This is because modifying a filter requires inactivating the entire bitmap for consistency. For these reasons, the design of Caesar benefits from fine-grained filter construction with higher scalability and flexibility.

Challenge 2: Providing guaranteed high-speed forwarding. Caesar's goal is to achieve a forwarding rate similar to that of high-speed border routers (e.g., 100s of millions of packets per second). However, compressing addresses into filters creates a bottleneck in the processing pipeline. To run a membership test on a filter, we need to compute k hash functions and access the memory k times in the worst case. Previous designs do not provide hash computation optimization and also access filters naively (e.g., [15, 20]). Thus they have limited peak forwarding speeds, on the order of a few hundred kpps (e.g., [33]), even for fewer than a hundred filters. This is orders of magnitude smaller than Caesar's objective. Also, instantiating more filters to support fine-grained policies makes existing designs more inefficient.

Challenge 3: Avoiding Internet-wide false positives. One key limitation of compression using filters is occasional false positives; that is, a filter incorrectly recognizes a non-existing address as its member due to hash collisions. In this case, all positions that correspond to hash values of the address have been set to 1 by insertions of other addresses. For a filter, there is an inherent tradeoff between the memory size and false positive rate. A filter naturally generates fewer false positives as memory footprint increases. For Caesar, false positives can result in Internet-wide black holes and loops, thus disrupting essential Internet services. To address this problem, multiple solutions have been proposed (e.g., [24, 33]) that either are very slow, incur domain-level path inflation or offer partial correctness. Caesar cannot borrow them because, as a border router, it must provide *deterministic correctness* at *high speed*.

Challenge 4: Updating filters and maximizing their utilization. Routing and forwarding tables might need to be updated. Supporting updates poses two challenges to Caesar. First, a routing message can lead to address withdrawal from filters. However, removing an address from a standard filter inevitably introduces false negatives. An address is mapped to k positions, and although setting any of the positions to zero is enough to remove the address, it also leads to removing any other addresses that use the same position. Second, even with supporting address removal, the total utilization of filters and the compression rate can be negatively impacted if many addresses are removed from a filter and distributed into other filters.

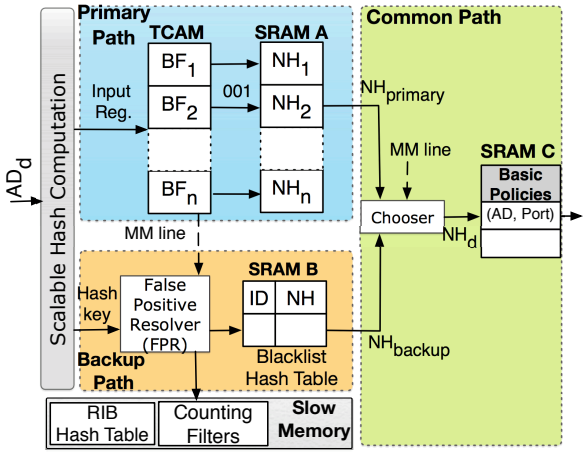


Figure 1: Caesar Architecture. The backup path result is selected when MM (multi-match) flag is high.

2.2 Caesar Architecture Overview

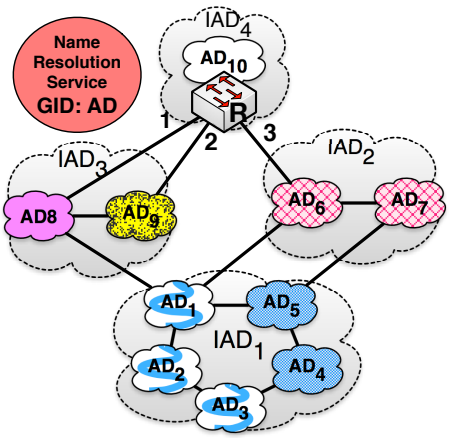
Caesar benefits from two logical data structures: a routing information base (RIB) and a forwarding information base (FIB). The RIB maintains all paths to destinations ADs; the FIB is used to match ingress packets to outgoing links. Similar to modern hardware routers, Caesar implements the RIB and FIB using slow and fast memories respectively.

Caesar has a novel FIB design as illustrated in Fig 1, which consists of two *forwarding paths* or *pipelines*. Each pipeline performs a different series of actions on the input packet, but they both run in parallel. The vast majority of packets go through the *primary path* that leverages our scalable and flexible filters constructed in TCAM (Sec 3). The *backup path* is built from the fast memory and handles uncommon cases where the primary path is not reliable due to false positives in the filters thus *rarely* is less efficient when it accesses the RIB (Sec 4). In other words, the primary path ensures the common-case high-speed forwarding while the backup path guarantees the correctness.

Caesar minimally extends the RIB to support routing updates and keep filters of the primary path highly utilized in such events; it also optimizes the computational overhead of hash functions to remove a potential processing bottleneck (Sec 5). Our design provides a practical solution that can be implemented by existing hardware (e.g., SDN switches) with guaranteed performance. More importantly, our design can be replicated to support specific future forwarding schemes (e.g., XIA [21] having more address components and the backward compatibility feature).

3. PRIMARY FORWARDING PATH

We first describe our design of the primary forwarding path. A simple approach to compressing forwarding states is to group all destination addresses reachable through the same outgoing interface into a filter (e.g., Buffalo [33]). In this section, we first discuss how our high-speed filters minimize data path interruptions, improve the reliability, and allow rapid false positive handling compared to the simple method (Sec 3.1). Then we describe how we dynamically instantiate filters and perform parallel membership tests (Sec 3.3)



(a) Generalized future Internet architecture

Filter #	Members	Next Hop Pairs
IAD_{11}	{AD_1, AD_2, AD_3}	(AD_8,1)
IAD_{12}	{AD_4, AD_5}	(AD_6,3), (AD_9,2)
IAD_{21}	{AD_6, AD_7 }	(AD_6,3)
IAD_{31}	{AD_8}	(AD_8,1)
IAD_{32}	{AD_9}	(AD_9,2)

(b) Caesar approach. Filter IAD_{ij} denotes j^{th} filter assigned to IAD_i

Filter #	Members	Next Hop Pair
1	{AD_1, AD_2 , AD_3 , AD_8 }	(AD_8,1)
2	{AD_4, AD_5 , AD_9 }	(AD_9,2)
3	{AD_4, AD_5 , AD_6 , AD_7}	(AD_6,3)

(c) Simple approach (e.g., Buffalo [33]). Filter i is assigned to outgoing port i

Figure 2: Caesar's scalable and reliable filter construction in border router R.

3.1 Scalable and Reliable Filters

As shown in Fig 2b, Caesar's control logic stores forwarding states into multiple *fine-grained* filters in the data path, presenting a new abstraction. Each filter encompasses a group of destination AD addresses and is mapped to forwarding actions or instructions. To forward an incoming packet, the data path in parallel performs membership tests on filters and then learns how to deliver the packet to outgoing ports. At the control plane, Caesar introduces two *primary* properties to group and store destination AD addresses into filters. AD addresses that have the same properties at the same time get an identical group membership, and consequently are encoded into the same logical filter. Caesar's control plane is also flexible to define additional properties to form various groups. The primary properties are as follows (the design rationale will be clarified in subsections 3.1.1 and 3.1.2):

- **Location property** separates destination ADs that are advertised and owned by the same IAD from the others.
- **Policy property** separates destination ADs that are under the same forwarding policy, which is determined by the Caesar's control plane, from the others.

Caesar's control logic continuously determines the FIB entries, and forms groups and constructs filters based on the local properties. Then, it couples each filter to *the forwarding policy* of the group. For simplicity, we focus on a basic forwarding policy below, even though Caesar supports more complex policies (e.g., rate-limiting). For a destination AD address, Caesar's *basic policy or next-hop information* includes *all* (next-hop AD, outgoing port) pairs that are selected by the control plane for forwarding ingress traffic destined for the AD. For multi-path forwarding, the next-hop information simply consists of multiple such pairs.

Example. In a multi-path scenario, filters of border router R are shown in Fig 2b. Based on the Caesar's control logic outputs, destination ADs with the same policy and location properties are filled with the same pattern, each representing an address group (Fig 2a). Then the groups are stored into five filters in the Caesar's data path (Fig 2b). In this example, traffic to each of the addresses AD_4 and AD_5 is desired to be forwarded on multiple paths. For input packets, the data path runs parallel membership tests on the filters to retrieve the next-hop information at high speed (Sec 3.3). We save memory from two aspects. First, we hash each long address into a few positions within a small filter. Doing so consumes significantly less memory than storing the original address does. Second, we reduce the memory usage of the next hop information by decreasing the number of FIB entries. Caesar further minimizes the overhead of maintaining next-hop information (Sec 3.4).

3.1.1 *Why Separation by Forwarding Policy?*

At the high level, the *policy property* isolates destination AD addresses under the same forwarding actions from the others, and allows us to guarantee data path correctness. For any action or policy supported in the data path of Caesar routers (e.g., rate limiting, ACLs, or next-hop information), the policy property ensures each address is only inserted into *one* group and thus leads to disjoint filters. This is a key design decision that allows our false positive detection procedure to work at high speed (will be detailed in Sec 4.1).

Multi-match uncertainty problem. Existing address grouping approaches used in previous filter-based routers mostly store an address into multiple filters and inevitably make the reasoning about membership tests both hard and slow (e.g., [20, 31, 15, 33]). For example, Fig 2c shows how Buffalo [33] establishes a simple approach to construct one filter per outgoing port, which is referred as "simple grouping method" in this paper. Buffalo suffers from an uncertainty in its data path operations in multi-path forwarding scenarios as follows. Assume we are interested in splitting incoming traffic destined for an AD address into multiple outgoing links. The simple grouping method installs the AD address into multiple filters, each assigned to one of the egress links. For example, Buffalo inserts AD_4 and AD_5 into filters 2 and 3 in Fig 2c to perform load balancing. This potentially equivocates the lookup operation output. If there are multiple matching filters, it is impossible to immediately distinguish between two states: 1) true multiple matches in the multi-path forwarding; and 2) multiple matches due to one or more false positives.

Current solutions. There are two solutions in the literature for mitigating the multi-match uncertainty problem in filter-based routers and switches. The first category of solutions accesses the RIB stored in slow memory and checks all candidates sequentially [15, 31] when multiple matches happen in a lookup. The other category of solutions forwards packets randomly without further checking or randomize fil-

ters [33, 28]. Because of insufficient performance and poor correctness, Caesar constructs disjoint filters, each of which is coupled and mapped to the *entire* forwarding actions of the group (e.g., all specified next-hop pairs in multi-path scenarios) in its data path. For instance, in Fig 2b, Caesar stores AD_4 and AD_5 only into the filter IAD_{12} that is associated with both next-hop pairs as its forwarding actions. Therefore, Caesar expects exactly one matching filter from the lookup operation. Note if there are other policies or actions in addition to next-hop pairs, we can aggregate them in a similar way to build up disjoint filters.

3.1.2 Why Separation by Location?

As shown in Fig 2, the *location property* isolates destination AD addresses of different IADs into separate logical groups and makes constructed filters flexible and reliable. It minimizes processing interruption and performance degradation when the control plane updates a forwarding state in the FIB once it receives route updates or locally enforces new forwarding policies.

First, given there can be millions of AD addresses in the future Internet, the location-based isolation *systematically* ameliorates the reliability challenges by making defined groups small in size and shrinking filters in width substantially. Therefore a small portion of the Caesar's FIB becomes "frozen" when a desired filter is inactivated during its bitmap update, or when a bit failure occurs in a bitmap. However, existing designs that use the simple filter construction method (e.g., Buffalo [33]) can disrupt traffic forwarding to many destinations and are prone to more failure. This is because they store millions of addresses into a few very large filters.

One can use other properties to make groups more specific and smaller, but the location-based separation also limits side effects of *Route flapping* events, which have been identified by other work [16]. In the future Internet context, these events occur because of a hardware failure or misconfiguration in a border router of an IAD. In this case, the router advertises a stream of fluctuating routes for ADs in its owner IAD into the global routing system. However, Caesar's data plane keeps the majority of filters protected from any bitmap modification in response to such route updates, except those filters built for that problematic IAD.

Second, the location-based isolation allows Caesar to enforce business-specific policy efficiently. For example, Caesar's control plane can dynamically stop forwarding traffic to ADs in a specific IAD (e.g., due to political reasons [32]) without interrupting traffic forwarding to other AD addresses.

3.2 Memory Technology for Filters

In practice, the number of filters generated based on the two primary properties can be high. This is because Caesar constructs more specific and fine-grained address groups. We approximate the worst case number of filters that might be constructed in our forwarding engine. To achieve our performance requirement, the approximation is used to find the best memory technology for filter implementation.

Let d denote the total number of IADs throughout the Internet. Also, let p be the total number of different forwarding policies that can be defined by the control plane of a Caesar router. Then the number of filters is $O(dp)$. For example, 2M filters are generated for $p = 20$ and $d = 10^5$ in the worst case. This poses performance challenges because Caesar must test filters very fast to achieve a high forwarding rate (e.g., 100s of millions of packets per second (Mpps) [2]).

SRAM is the fastest memory technology in terms of access delay (about $4ns$ based on Table 1). However, it can provide high-speed forwarding rates only when it stores a small number of filters, and the performance dramatically degrades to a few kpps even when a few hundred filters are tested in a lookup [33]. This is because the memory bandwidth is limited (even for multi-port SRAMs) and there is a lack of parallelism in accessing multiple filters, each requiring k memory accesses in a membership test in the worst-case (k is the number of hash functions). Therefore serialization and contention become intensified as many fine-grained filters are instantiated.

To overcome the above limitations, we propose to realize filters using TCAM due to its three advantages over SRAM. First, it supports parallel search operation that can be used to lookup filters in one clock cycle (Sec 3.3). Second, we can intelligently leverage one of its flags to handle false positives (Sec 4). Third, it has less implementation complexity compared to the approach of using distributed SRAM blocks [23].

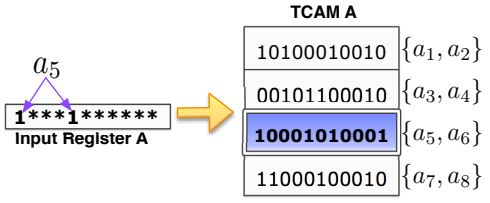
3.3 Parallel Lookup of Filters

As shown in Fig 1, Caesar encodes filters that are heterogeneous in bit width and constituent members in TCAM data entries to attain its desired forwarding rate. TCAM is an associative memory that is made up of a number of entries. All data entries have an identical width, which is statically configurable by combining multiple entries. For example, a TCAM with a base width of 64 bits can be configured to various widths such as 128, 256, and 512 [10]. As shown in Fig 3, each bit of the memory and input register can be set to either 0, 1, or * (don't-care). To search a key, TCAM in parallel compares the content of the input register, which contains a search key, with all memory entries in one clock cycle. When there are multiple matches, it returns the index of the matching entry with the highest priority. Typically, an entry at a lower address has higher priority.

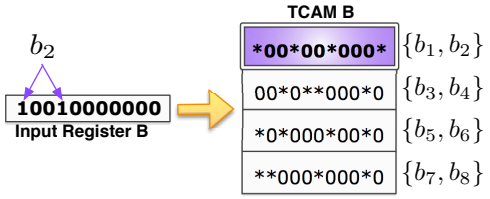
Heterogeneous filters. Since each IAD manages a different number of ADs, top tier IADs that form the core of the future Internet can own a lot more ADs compared to others. Hence the grouping properties can produce heterogeneous filters from three aspects: for a filter, the number of inserted addresses, the filter bit width, and the number of hash functions (in the insertion and test procedures) can be different from the others. We can show it is possible to store such heterogeneous filters in TCAM. For example, we can extend short-width filters by filling the don't-care values into different positions, but the memory space is wasted and the filter management becomes complex in terms of the insertion and membership test procedures.

Caesar memory allocation strategy. To avoid the low utilization and memory management overhead, we construct equal-sized filters that have identical bit width (w) and use the same k-hash functions. Caesar defines a global maximum capacity for filters by which it restricts the number of ADs in them. This maximum capacity, n_{max}, can be configured by the router's bootstrap program. By defining the maximum capacity, we can limit false positives in practice, and theoretically calculate an upper bound on their rate. Instead of storing all addresses of a group into a large filter, Caesar allocates, releases, and combines equal-sized filters depending on the size of a group that might change over

time due to address insertion and removal into the data path (details in Sec 5.2). In the evaluation, we conduct experiments to study how different n_{max} and w values affect the trade-off between the filter utilization and false positive rate (Sec 6.2). Below, for simplicity, we focus on the lookup procedure in the primary path when n_{max} and w values are given.



(a) Option 1



(b) Option 2

Figure 3: Two options for a parallel membership test in TCAM when there is no false positive and k is 2

Caesar's parallel filter lookup. Assume Caesar's TCAM contains a set of w-bit disjoint filters, each storing at most n_{max} destination AD addresses. Each equal-sized filter occupies a memory entry, as shown in Fig 3. We design two options in Caesar to perform parallel filter lookups. Suppose we would like to retrieve the basic forwarding policy or the next-hop pairs of an incoming packet destined for AD_{key}. First, k hash functions (H=$\{h_1, h_2, \ldots h_k\}$) are computed on AD_{key}. In the first option (Fig 3a), we set all the positions of the input register that do not correspond to H (i.e., not set by any of the k hash functions) to the don't-care value, and set all other positions to 1. When the search is issued, the TCAM locates the target filter by matching 1s in one clock cycle. In the second option (Fig 3b), we set all the positions of the input register that correspond to H to 1, and set all other positions to 0. When we issue the search, the TCAM locates the filter whose positions that correspond to H have the don't-care value. Finally, we retrieve the next-hop information mapped to the matching filter to continue the packet processing.

Design implications. The first difference between the two options is how filters are represented in the TCAM. In the second option, all 1s within standard filters need to be changed to don't-cares. The second difference is about the number of writes to the input register bits. Assume the input register has the default value of 0, the second option requires setting only k positions in the input register while the first option needs to modify all w bits to 1s or *s. Although the power to toggle the memory and input register bits can be very small in practice, one can benefit from one of the options to perform hardware-specific optimizations for write-intensive workloads. Note that the encoding options do not change the false positive rate or incur false negatives. Also, unlike IP routers that keep address prefixes sorted in decreasing prefix-length order to implement the longest prefix matching algorithm, the order of entries does not matter Caesar, except in uncommon cases where there are matches of multiple entries due to false positives. In this case, there is no unique ordering with which we can deterministically mask all false positives, so the backup forwarding path is triggered (Sec 4).

3.4 Reducing Next-Hop Fast Memory

Because routers usually have a limited number of (next-hop AD, outgoing port) pairs, often many filters with different location properties are mapped to the same next-hop information (e.g., AD_8 and AD_1 in Fig 2). We can eliminate the memory redundancy in storing next-hop information and make Caesar's data path more agnostic to the address length. At a cost of an extra fast memory access per lookup, we can store all different next-hop information into a separate fast memory space (i.e., SRAM C in Fig 1), and then map each filter to a pointer (NH) pointing that memory. Each NH-pointer can be realized by using only one byte in most cases because of limited number of ports on a router. This approach minimizes the fast memory overhead, in particular when TCAM contains a large number of filters. A similar technique can be applied when other forwarding actions or policies are supported in addition to (next-hop AD, outgoing port) pairs for filters.

Primary path performance implication So far, we have encoded AD addresses into the primary path such that the next-hop information can be retrieved in at most three memory access, each taking about $4ns$ delay based on Table 1. With using faster TCAMs [3], the primary path can support up to 1.6 billion filter searches per second under typical operating conditions

4. BACKUP FORWARDING PATH

We now describe the backup forwarding path and introduce a blacklisting mechanism to handle false positives as shown in Fig 1.

4.1 High-Speed False Positive Detection

The grouping properties lead to disjoint filters in the presence of different forwarding policies in a Caesar router. Therefore, we expect one matching filter in each parallel lookup and thus do not need to deal with the multi-match uncertainty problem (Sec 3.1). Caesar deterministically interprets any multiple matching filters as an event that indicates the primary path is no longer reliable, and the processed packet might be forwarded to an incorrect next hop. Caesar intelligently detects such events using *Multi Match (MM) line* of state of the art TCAMs, a flag indicating that there are multiple matching entries. In every lookup, the primary path is used if and only if the MM line output is low (see Fig 1). We describe the details as follows.

- The low MM line ensures that the index of the matching entry reported by the TCAM is the (only) correct filter. In other words, the destination address of the incoming packet is encoded in the TCAM without ambiguity. Therefore, the packet is processed based on the correct next-hop pointer ($NH_{primary}$) through the primary path.
- If the MM line is high, the true matching filter is at the i^{th} position and there is at least a filter at position $j \neq i$ that has returned false positive. If $j < i$, the reported index is not correct, otherwise the error is masked by the true matching filter that has higher priority (lower address).

However, distinguishing between these two cases is not possible, so the backup forwarding path is triggered.

4.2 Blacklisting Mechanism

The backup forwarding path delays *at most* the first packet in a flow that is destined for an AD on which the primary path encounters multiple matches. There are two components in the backup path (see Fig 1):

- The *Blacklist Memory* is a very small, high-speed, and SRAM-based hash table that maps the hash value of an AD to its correct next-hop pointer (NH). In addition, each entry has an *expiration or idle* time that helps keep the hash table small and minimize potential collisions. An entry is deleted from the blacklist memory if it is not used for a predetermined period of time.
- The *False Positive Resolver (FPR)* is a component that creates entries in the Blacklist memory. It accesses the RIB that is stored in a hash table in slow memory, and retrieves the correct next-hop information, i.e., all (next-hop AD, port) pairs, in constant time.

Backup path. Given the above components, the backup path works as follows (Fig 1). In parallel to the primary path, the backup path proactively retrieves the next-hop pointer NH_{backup} from the blacklist memory for every incoming address AD_d. If the primary path activates the MM line, *Chooser* picks NH_{backup} from the backup path, otherwise it selects $NH_{primary}$ from the primary path. If the MM line is high and NH_{backup} does not exist in Blacklist, the backup path delays forwarding and waits for *FPR* to retrieve the mapping from the RIB. *FPR* then updates *Blacklist* to avoid delaying subsequent packets belonging the same flow as well as future flows to the same destination AD. Finally, the NH_{backup} is sent to *Chooser* in this case.

Backup path performance implication. The backup path is as performant as the primary path almost always for three reasons. First, for optimally configured filters, the backup path result is rarely used because the multi-match rate is very small in theory and for actual workloads. We show this by analysis and evaluating two extreme cases of blacklisting (Sec 6). Second, when a multi-match rarely occurs, the results of the backup and primary paths are ready at the same time since the Blacklist memory mostly hits. Third, a Blacklist miss occurs for the first packet of a flow in the worst case (when the idle time is minimum). In such a case, the processing takes more time due to accessing slow memory. Caesar can minimize this by employing two known techniques. Given only next-hop pointers are needed to be retrieved, we can implement an efficient and summarized RIB to minimize the delay to one slow memory access (about $20ns$). Also, we can minimize the miss rate by establishing a multi-level Blacklist approach similar to hierarchical caching schemes.

Security implication. From the security perspective, it is difficult for an attacker to trigger the backup path and to infer what ADs use this path for two reasons. The inference of k hash functions and filter organization, changing over time, is very hard. Second, the *observed delay* caused by the slow memory access is very small and happens infrequently.

5. FORWARDING OPTIMIZATIONS

The parallel paths can process almost all packets within three fast memory accesses and offer deterministic correctness. However, the hash computation must be optimized to guarantee the entire performance. Although there are solutions for building uniform hashing (e.g., [26]), the computation overhead is still an unsolved issue [22]. Related designs (e.g., [33, 15, 20, 17]) have not taken into account this overhead. Our key idea is to exponentially minimize the number of hash computations, and then run them in parallel similar to state-of-the-art routers (Sec 5.1). Caesar also handles route updates and optimizes the filter utilization (Sec 5.2).

5.1 Scalable Hash Computation

We leverage a simple but effective technique to reduce the number of hash computations. Our approach is based on hash coding theory [36] with its basic property. If we have two different and uniformly distributed hash values $f(x)$ and $g(x)$ for input key x, we can construct hash value $h(x) = f(x) \oplus g(x)$ that is also from a uniform distribution. The main intuition is that XOR is a uniform operation that generates both 0 and 1 with the same probability for random inputs (i.e., 0 and 1 are equally likely to appear). This property is also applied to n-bit inputs in practice. For example, SSL computes the MD5 and SHA-1 of its inputs and combines them to avoid cryptanalytic attacks.

Hierarchical hash computation. Caesar recursively employs the property to faster generate hash values in the lookup and update procedure of *small* filters (e.g., 288-bit), which have specific characteristics compared to large filters (will be clarified below). Given k_1 different hash values from uniform distributions, $H = \{h_1, h_2, h_3, \cdots h_{k_1}\}$, any non-empty subset of H is a candidate for constructing a new uniform hash value by performing the XOR operation among its members. Because H has $2^{k_1} - 1$ non-empty subsets, we can build $2^{k_1} - k_1 - 1$ new uniform hash values. This dramatically improves the hash computation performance. For instance, four different uniform hash values in $H = \{h_1, h_2, h_3, h_4\}$ give us $G = \{h_1 \oplus h_2, h_1 \oplus h_3, h_1 \oplus h_4, \ldots h_1 \oplus h_2 \oplus h_3 \oplus h_4\}$ that consists of 11 uniform hash values by performing only 11 XOR operations.

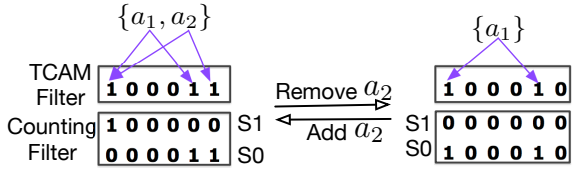
Internal correlation. Theoretically, the correlation between recursively-constructed hash values does not lead to more false positives as long as the seed set satisfies the uniformity and diversity requirements. In practice, even cryptographic hash functions might not completely satisfy the uniformity. In this case, we have observed negligible (positive and negative) difference values between the flat and hierarchical hash computation schemes in terms of false positive and multi-match rate, making this scheme practically useful.

Small filters and internal collision. Caesar's focus is on very small filters, which is different from a similar usage of the core property in previous work [31]. Particularly, our results show small filters require k hash values of an address to be different in contrast to large filters. However, k hash functions might generate fewer than k different hash values in practice. Therefore, we proactively compute sufficient extra hash values for each address, which we refer it as internal collision avoidance. In this case, we have observed the hierarchical scheme has substantially lower computational overhead compared to the standard flat scheme (complete details in Sec 6.2.4).

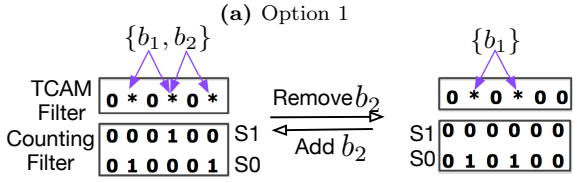
5.2 Optimized Route Update Support

To handle control plane messages that change forwarding states, Caesar should be able to remove any address from an old filter, and insert it into a new filter. However, a standard filter does not support graceful address removal.

We leverage counting filters [18] to realize this operation. In a counting filter, each position is extended from a single-bit (as in a standard filter) to an s-bit counter. To insert/remove an address, the value of each of the k positions, each corresponding to a hash value of the address, is incremented/decremented instead of being set/unset. Similarly, the membership test checks the positions to see if all of them have non-zero values or not. To integrate counting filters into Caesar, a trivial solution is to directly put them into TCAM, but this increases the TCAM memory usage as well as the complexity of the parallel lookups. Instead, Caesar keeps a counting filter in slow memory for each standard filter in TCAM. Caesar does not access the counting filters to perform forwarding, but only uses them to assist updating standard filters.



(a) Option 1



(b) Option 2

Figure 4: Address removal and insertion in filters when k and n_{max} are 2, and thus $s = \lfloor \log_2(n_{max}) \rfloor + 1 = 2$

Insertion and zero overflow. To store an address in the data path, Caesar first learns its group based on the primary properties (Sec 3.1). Then it determines the insertion position of the address in the TCAM based on its group. If the specified group is old, multiple filters already have been assigned to old members of the group in the TCAM. In this case, Caesar balances the load among existing filters of the group with a simple greedy approach. It selects an available position or filter with the minimum utilization ratio, n_i/n_{max} (n_i is the number addresses in the filter). Otherwise Caesar assigns a new position to the group and the address. The specified position is also recorded in the RIB to support removing the address without false negatives later.

After determining the position, we first insert the address into the corresponding counting filter and then the standard filter in a simple procedure (as shown in Fig 4). When a counter changes from zero to non-zero, we change the corresponding bit in the standard filter from zero to one for option 1 (Fig 4a) and from zero to the don't care (*) for option 2 (Fig 4b). Unlike previous usage of counting filters (e.g., [18]), our counting filters can be configured to avoid overflows by setting $s = \lfloor \log_2(n_{max}) \rfloor + 1$ because each standard filter contains at most n_{max} addresses (Sec 3.3).

Removal and high filter utilization. To remove an address, we first retrieve the its position (in TCAM) from the RIB, which is necessary to avoid generating false negatives. As shown in Fig 4, we then remove the address from the counting filter at the position by decrementing the counters, each of which maps to a hash value of the address. If any of the counters becomes zero, we set the corresponding bit in the standard filter to zero in the both options. In the address removal procedure, Caesar checks the utilization ratio of the affected standard filter. If the ratio is below a predetermined threshold, Caesar tries to combine the filter with other filters allocated to the same group. This is necessary to reduce the number of filters in the TCAM (We adjust the counting filters accordingly).

6. EVALUATION

In this section, we first perform a cost-accuracy analysis (Sec 6.1) and then do extensive simulations using multiple workloads (Sec 6.2) to study Caesar from different aspects.

6.1 Cost-Accuracy Analysis

We provide a simple *approximation* showing the inherent trade-off between false positives of filters in the primary path and the total cost of Caesar. Note Caesar correctly processes all packets and false positives only affect the amount of traffic handled in the backup path. Similar to AIP and XIA [12], we assume each AD corresponds to an IP prefix in today's Internet but with *larger* size. The current number of active address prefixes is about 481k and we envision 1M *ADs* to accommodate a reasonable growth rate [6].

False positive estimation. Intuitively, the false positive rate in the parallel filter lookup procedure for a destination *AD* address depends on two factors. First, the position of the true filter containing the address. Second, the fill factor (i.e., the ratio of bits with the value 1) of all the filters above the true filter. The fill factor of filter i is a function of the number of inserted addresses in the filter (n_i), the width of entries (w), and the number of hash functions (k). Caesar does not insert more than n_{max} addresses in each filter, so we can derive a theoretical upper bound on the *maximum false positive rate (FP)* of filter i, given n_i and w are fixed, by $FP(i) \leqslant \left(1 - e^{\frac{-kn_{max}}{w}}\right)^k$. In this equation, k can be computed optimally for given n_{max} and w, $k_{opt} = 9w/13n_{max}$. We omit the detailed derivation for simplicity. Assuming all addresses are accommodated into E entries and a given packet matches any entry with the probability of $1/E$, *the maximum expected false positive rate* in parallel filter lookups can calculated by Eq. 1.

$$\mathbb{E}[\text{false positive rate}] \leqslant \frac{(E-1)}{2}\left(1 - e^{\frac{-kn_{max}}{w}}\right)^k \quad (1)$$

The same approach can be used to derive *the maximum expected multi-match rate* in parallel filter lookups, which results in Eq. 2.

$$\mathbb{E}[\text{multi-match rate}] \leqslant (E-1)\left(1 - e^{\frac{-kn_{max}}{w}}\right)^k \quad (2)$$

Table 1: Fast memory reference price

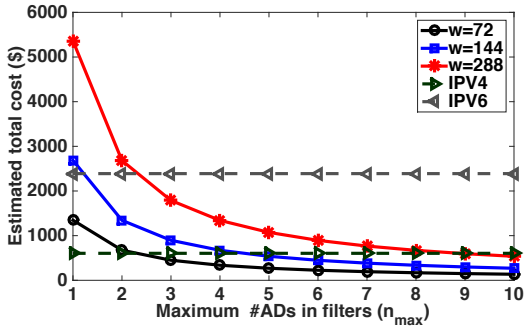
Memory	Capacity	Delay	Price	Company	Cost/ MB
SRAM	9MB	3-4ns	$90	Cypress	$10
TCAM	2.5MB	3-4ns	$390	Broadcom	$156

Cost estimation. Now, we turn into estimating the total material cost. Our prices have been quoted by Cypress and

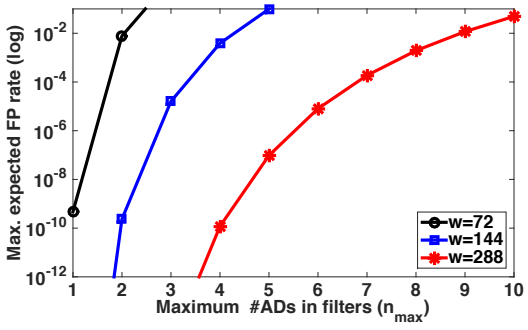
Broadcom for a TCAM and an SRAM working at 250 MHz as shown in Table 1. As expected, the TCAM is more expensive compared to the SRAM. Let C_{TCAM} and C_{SRAM} denote the cost per-bit of TCAM and SRAM respectively. Assume Caesar has h next-hop pairs and addresses are q-bit long. Then, Eq. 3 gives an estimation of the total cost of a Caesar router excluding the blacklist memory which is expected to be small, while also ignoring the RIB and counting filters which use inexpensive DRAM. The first term is the TCAM cost, and the second term includes the cost of SRAM A and C (see Fig 1).

$$Total\,Cost = EwC_{TCAM} + (log(h)E + hq)\,C_{SRAM} \quad (3)$$

Based on Table 1, Fig 5a illustrates the total cost of Caesar for $h = 64$, $q = 1kb$, and variable n_{max} and w. We assume filters are fully utilized (i.e., $E = \#ADs/n_{max}$). We also estimate the total costs of optimized TCAM-based IPv4 and IPv6 routers (e.g., [34]) to be \$604 and \$2,389 respectively. For the same parameters and the optimal k values, Fig 5b depicts the maximum expected false positive rate in parallel filter lookups.



(a) Estimated cost for one million 1kb addresses; IP addresses are 32b and 128b.



(b) Expected false positive rate in parallel lookups; k is optimal.

Figure 5: Cost-accuracy analysis

Finding 1. Caesar can be substantially less expensive compared to TCAM-based IPv6 routers. We observe there are several interesting options that provide a reasonable accuracy for the filters with a feasible total cost. For $n_{max} = 4$ and $w = 288$, the maximum expected false positive rate is around 10^{-10} and the total cost is \$1,340. In this case, Caesar is about 43% less expensive than the IPv6 router while our addresses are 8X longer than the IPv6 addresses.

Finding 2. The total cost of Caesar is constant for very long addresses. To analyze the sensitivity of our design to the AD address length, we compare the cost of Caesar router with TCAM-based IP router. Fig 6 shows that

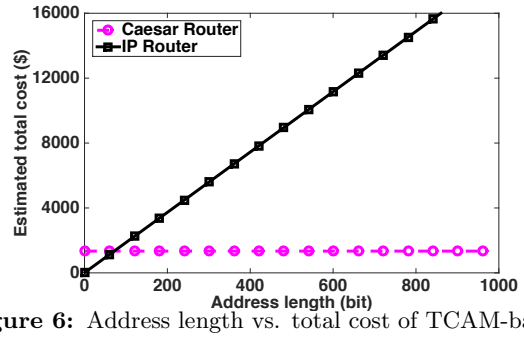


Figure 6: Address length vs. total cost of TCAM-based IP routers and Caesar routers with $w = 288$, $n_{max} = 4$

the total cost of our design is roughly constant even for very long addresses. In contrast, the total cost of IP routers increases linearly as addresses become longer (assuming future IP addresses can be longer to support new services).

6.2 Extensive Trace-Driven Simulation

We now evaluate Caesar under real workloads. Because existing prototyping platforms such as NetFPGA lack sufficient TCAM, we implement an accurate packet-level simulation framework in C/C++ (with 2500 LoCs) and "mix and match" various datasets. We measure the multi-match rate, percentage of delayed flows, effects of grouping properties on the memory utilization, efficiency of the hierarchical hash computation scheme, and energy consumption.

Table 2: Experiment statistics

Experiment /Snapshot	Total # ADs	# forwarded packet	Monitoring duration(s)
Jan 1, 2008	149842	12481172	25
Jan 1, 2009	162102	12849066	31
Jan 1, 2010	180470	16516240	41
Jan 1, 2011	205361	25459705	52
Jan 1, 2012	234483	28460868	55
Jan 1, 2013	255424	16662799	42
Total		**112 M**	**246s**

6.2.1 *Dataset and Methodology*

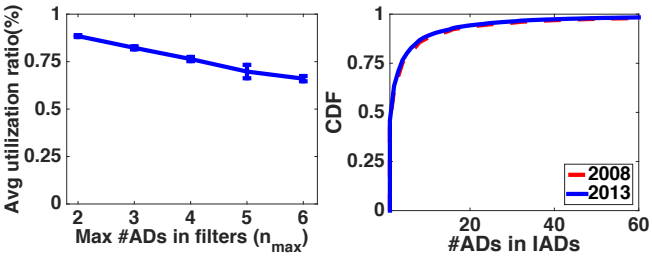
We simulate the future Internet architecture using public datasets in six snapshots between 2008 and 2013 to consider the growth rate of IADs and ADs. In our simulation, IAD and AD correspond to today's AS and IP prefix respectively. We select only address prefixes of length 24 to feed experiments with flat addresses. To consider high entropy of future addresses, we replace them with their corresponding hash value, which is computed using the SHA1 algorithm. To represent the business agreements among IADs, we utilize CAIDA inferred relations between ASes [1] and leverage RIBs and update traces of Route Views [5] to generate the FIBs based on the path length metric. We replay the packets in traffic traces collected from backbone links [1] and use a recent power model [10] to measure the dynamic power consumptions in the experiments. For each pair of n_{max} and w, we compute the optimal k and then construct filters based on the primary properties. For the space reason, we show the results for a single Caesar at the border of IAD 7726 when w is 144 and n_{max} is between 2 and 6. The other Caesar routers and other settings follow a similar trend. Table 2 lists the number of ADs and forwarded packets by this Caesar router, and the duration of traces in each snapshot. Although the monitoring duration is relatively short, the coverage of

destination addresses is high and sufficient for our evaluation purpose. In total, the Caesar router forwards 112M packets, and upon receiving each route update message, it runs the best route selection procedure and updates filters if necessary. The average rate of route update messages varies between 88.2 and 277.1 across different snapshots.

6.2.2 Multi-Match Rate and Memory Consumption

We first measure the multi-match rate in the primary path. This rate indicates the amount of traffic that is forwarded by the backup path regardless of whether it is delayed by a slow memory access to the RIB due to the blacklist memory miss or it is delivered without any delay. Table 3 presents the multi-match results and the memory usage of filters in 30 configurations.

Finding 3. Caesar can forward most of the traffic through the primary path within three fast memory accesses. As expected, Table 3 shows the multi-match rate in each snapshot exponentially increases as n_{max} increases. Although predicting the exact multi-match rate is impossible, we observe different snapshots have the same order of magnitude of the multi-match rate for a fixed n_{max}. This indicates we can practically control the order of magnitude of the multi-match rate in Caesar routers. In the case that n_{max} is 2, we observe that the multi-match rate is zero thus the MM line never goes high. This means all the packets are forwarded through the primary path, mainly due to using more hash functions.



(a) (b)

Figure 7: Determining n_{max}. (a) Average filter utilization ratio of filters for $w = 144$ across all snapshots. (b) Distribution of ADs in IADs in the first and last snapshots.

Finding 4. To find a reasonable n_{max}, both the memory utilization and multi-match rate are determining factors. Based on Table 3, two important reasons suggest that we should keep n_{max} smaller than 5 for $w = 144$. First, we observe the multi-match rate increases several orders of magnitude, from -2 to 0, when n_{max} changes from 4 to 5. Second, the memory utilization rate (i.e., the difference between the memory footprint of two consecutive n_{max} values) becomes smaller as n_{max} increases. Now the question is why the memory utilization rate decreases. Fig 7b shows the distribution of ADs in IADs between five years. Also, Fig 7a illustrates the average utilization of filters in TCAM, which is defined as $\sum_{i=1}^{E} n_i/(n_{max}E)$, across all snapshots for each n_{max} value. From these two figures, we observe the memory utilization rate reduces because the average utilization of filters goes below 75% for n_{max} values larger than 4 that is because about 77% of IADs own fewer than 5 ADs.

6.2.3 Energy Consumption Breakdown

We now measure the total dynamic energy consumption of Caesar and compare it to optimized TCAM-based IPv4 and IPv6 solutions (e.g., [34]). TCAM is the most power consuming component among different memory technologies in routers by several orders of magnitude. Therefore, we can ignore the energy consumptions in the other memory components of Caesar. The dynamic energy used in each search operation depends on many factors and parameters but is used in three high level architectural components [10]:

- **Match lines** that are charged in every lookup, and then except the lines that match input address, the others are discharged. The energy for this operation is proportional to the sum of match lines capacitance (i.e., the ability to store charge).
- **Select lines** that are driven to allow the comparison between input address and entries to happen. The energy used to drive select lines usually increases as the size of TCAM increases.
- **Priority encoder** that needs some power to work and the required energy depends on the number of filters (E), and is independent of the width of filters (w).

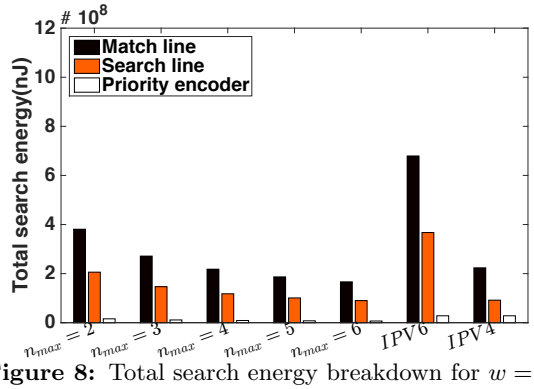


Figure 8: Total search energy breakdown for $w = 144$.

Finding 5. Caesar consumes 67% less total energy compared to TCAM-based IPV6 routers while the addresses are substantially longer. Fig 8 illustrates the total energy consumption break down across the TCAM components for the snapshot 2012 (others have similar results) for $w = 144$, variable n_{max}, and $65nm$ CMOS technology. We repeat the similar experiments for the IP routers. We observe the total energy consumption of Caesar for $n_{max} = 4$ is only 1% higher than the IPV4 router and 67% less than the IPV6 router while addresses in Caesar are very longer.

6.2.4 Hierarchical Hash Computation Scalability

We now compare the hierarchical and flat hash computation schemes from two aspects.

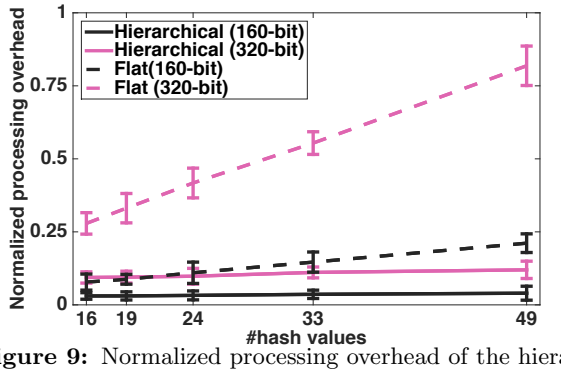


Figure 9: Normalized processing overhead of the hierarchical and flat schemes

Finding 6. The hierarchical scheme needs smaller number of hash computations for handling internal

Table 3: Multi-match rate and TCAM memory consumption for $w = 144$ and variable n_{max}.

n_{max}	$k_{optimal}$	k_{Caesar}	Multi Match Rate(%)[w=144]						TCAM Memory Footprint(MB)[w=144]					
			2008	2009	2010	2011	2012	2013	2008	2009	2010	2011	2012	2013
2	49	6	0	0	0	0	0	0	1.46	1.58	1.75	1.98	2.25	2.46
3	33	6	0.0002	0	0.0001	0	0.0001	0.0003	1.05	1.13	1.26	1.42	1.6	1.76
4	24	5	0.033	0.05	0.032	0.035	0.032	0.001	0.85	0.92	1.02	1.14	1.29	1.41
5	19	5	1.84	1.35	4.47	6.06	6.78	9.34	0.73	0.79	0.88	0.98	1.1	1.2
6	16	5	14.69	14.22	18.71	25.68	29.94	38.82	0.66	0.71	0.79	0.88	0.98	1.08

Table 4: Effects of permanent and per-flow blacklisting approaches

n_{max}	#Delayed Flows[Per-flow Blacklisting, w=144]						#Delayed Flows[Permanent Blacklisting, w=144]					
	2008	2009	2010	2011	2012	2013	2008	2009	2010	2011	2012	2013
2	0	0	0	0	0	0	0	0	0	0	0	0
3	3	0	23	0	2	1	1	0	2	0	1	1
4	923	903	764	1371	885	2067	141	30	145	261	245	115
5	32209	20282	25670	45888	110984	47815	3359	628	3096	6303	6040	2978
6	271902	175575	284606	656360	457918	308102	21366	4559	28789	47024	50039	21231

collisions. Although filters use k different hash values to insert and test an address, k hash functions might generate fewer than k different hash values in practice, which we call it internal collisions. In small filters, we have observed such collisions can increase the multi-match rate by up 50%. This is because the number of buckets in small filters is limited, and the correlation among the k hash values computed for a given address is stronger in practice. To control internal collisions, we proactively compute extra hash values for each address in the insertion and test procedures. In this way, we obtain sufficient different hash values in both the flat and hierarchical schemes. We measure the computational overhead of the two schemes in terms of number of hash computations, and aggregate the results across all snapshots and configurations. For a given k, the hierarchical scheme requires at most $1.16\log(k)$ hash computations while the flat scheme needs at most $1.5\,k$ hash computations to generate k different values. This indicates our scheme performs well for small filters because an extra different hash value in the seed set can double the total number of different hash values.

Finding 7. The hierarchical scheme substantially reduces hash computation processing overhead. We also measure the computational overhead of the two schemes in terms of CPU time when multi-threading is enabled for the same number of threads. We plot the aggregate results in Fig 9. To study the effect of address length, we consider 160-bit and 320-bit AD addresses. For a fair comparison, we do not enable internal collision avoidance that generates extra hash values. For a fixed k, we observe our scheme in average incurs by up to 18% and 70% smaller processing overheads for 160-bit and 320-bit addresses compared to the flat scheme. Note the overall number of XOR operations in the hierarchical scheme is $2^{k_{Caesar}} - k_{Caesar} - 1$.

6.2.5 Blacklisting and backup path delay

Finally, we evaluate the blacklisting in two extreme cases: *per-flow* and *permanent*. Due to blacklisting (Sec 4.2), only a small amount of packets activating multi-match line and going through the backup path are delayed. In the per-flow case, we store the destination AD address of a flow that leads to a multi-match until the flow completes. In the permanent case, we permanently store AD addresses that lead to a multi-match, upon the first detection.

Finding 8. Very few flows are delayed by both the permanent and per-flow blacklisting schemes. Table 4 shows the difference between these two cases. The permanent case reduces the number of delayed flows by an order of magnitude compared with the per-flow case. For example, for $n_{max} = 4$, fewer than 261 flows are delayed by the permanent blacklisting while the per-flow blacklisting approach delays up to 17.93X more flows. Note when a flow is delayed in both approaches, except its first packet, the other packets are forwarded at high speed. In both approaches, the blacklist memory footprint is insignificant because each next-hop pointer is only one-byte.

7. RELATED WORK

In the past few years, filters have been used in designing routing and forwarding engines. These efforts mostly have targeted improving the performance and simplifying traffic processing in enterprise switches and routers.

Much of the previous work focuses on memory-efficient IP routers [15, 31]. Unlike Caesar, these designs optimize the longest prefix matching algorithm to minimize the fast memory consumption. In particular, they store all address prefixes of the same length into a very wide filter. Given there can be multiple matches and the lookup uncertainty problem, these systems mostly test all candidates against a very large hash table located on slow memory to find the length of the longest match. Due to limited performance, these approaches cannot be used in high-speed border routers. Also, their coarse-grained filter construction is not reliable in practice.

Some other work focuses on designing low-cost and scalable switches handling flat addresses in small-scale and single-domain enterprise networks [20, 33]. The primary technique in such designs is constructing a very large filter per outgoing interface, each containing flat addresses reachable from the port. Upon facing the multi-match uncertainty problem, these designs mostly randomly choose among matching candidate filters, and thus impose significant delays and path inflations. Also, they require many memory accesses per lookup, and therefore have limited peak performance.

In contrast to the above techniques, Caesar is designed for high-speed border routers in the future Internet. Caesar constructs fine-grained filters that are more reliable and scalable. Caesar does not need to compute separate hash values for accessing each filter. It tests all the filters in parallel in one clock cycle and does not waste many memory accesses to check all the matching filters as it can detect false positives

at high speed using a hardware flag. Caesar minimizes hash computation overheads by recursively combining hash values.

Our idea of using filters in TCAM entries is similar to previous work [19]. However, the authors focus on the case that input register is filled by a set of elements instead of one to solve multiple string matching and virus detection problems. Although the authors propose a theoretical upper bound on the maximum false positive rate, still they do not provide any mechanism for detecting false positives at high speed. In contrast to this work, we reduce the hash computation overhead, design parallel forwarding paths, cleanly detect false positives, manage memory entries, and design an element removal procedure.

To optimize hash table operations in network processors, prior work [30] employs counting filter per table bucket. Instead, we use an expiration timer per address to minimize the size of the Blacklist memory and avoid occasional collisions. We can improve the robustness of the backup path by benefiting from such techniques.

Our idea of using small counting filters to support route changes is similar to some proposals (e.g., [33, 15, 18]). In contrast to existing approaches, Caesar constructs equal-sized filters in terms of bit width and the maximum number of constituent members. We dynamically allocate counting and standard filters while maintain them highly utilized. Also, our counting filters never experience overflow, and we do not modify the filters in the critical forwarding path during updates.

8. CONCLUDING REMARKS

Existing future Internet architecture proposals share an addressing scheme that requires addresses to be substantially long. These proposals lack a high-speed and memory-efficient border router design. In this paper, we propose a practical solution for high-speed routers of the future Internet architecture, called Caesar. We design scalable and reliable filters to compress long addresses. Our design can be extended to more address components. Due to poor performance of storing filters into SRAMs, we design a forwarding engine to search all the filters in parallel. To avoid forwarding loops and black holes, the engine uses two forwarding paths. We offer a novel blacklisting mechanism for accelerating the performance of the backup path. Caesar supports routing updates and performs intelligent memory management by utilizing counting filters in slow memory. For minimizing the computational overhead of hash functions, we propose a hierarchical hash computation scheme for our small filters. Our evaluation results indicate our design is memory-efficient, energy-efficient, and high-performance in practice.

Acknowledgments

We would like to thank the anonymous reviewers for providing valuable feedback on our work, and Amir Rahmati and Prabal Dutta at University of Michigan, and Patrick Soheili at eSilicon for helpful discussions on SRAM and TCAM technologies. This research was supported in part by the National Science Foundation under grants CNS-1039657, CNS-1345226, and CNS-1040626.

9. REFERENCES

[1] CAIDA Datasets. http://goo.gl/kcHlQf.
[2] Cisco Routers. http://goo.gl/6CWpLA.
[3] Esilicon TCAMs. http://goo.gl/O3rloQ.
[4] Internet Architecture Board. http://goo.gl/cnMyY9.
[5] Route Views Dataset. http://goo.gl/cn8sT6.
[6] Scaling Issues. http://goo.gl/SAnq28.
[7] TCAM Memory Challange. http://goo.gl/OGYyKn.
[8] Predictions. http://goo.gl/qmnVDy, 2012.
[9] MobilityFirst Project. http://goo.gl/sD64f9, 2014.
[10] B. Agrawal and T. Sherwood. Modeling TCAM Power for Next Generation Network Devices. In *Proc. IEEE ISPASS*, 2006.
[11] M. Akhbarizadeh, M. Nourani, and D. Vijayasarathi. A Nonredundant Ternary CAM Circuit for Network Search Engines. *IEEE Trans. VLSI*, 2006.
[12] D. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet Protocol. In *Proc. ACM SIGCOMM*, 2008.
[13] H. Ballani, P. Francis, T. Cao, and J. Wang. Making Routers Last Longer with ViAggre. In *Proc. USENIX NSDI*, 2009.
[14] B. H. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. In *ACM CCR*, 1970.
[15] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest Prefix Matching Using Bloom Filters. In *Proc. ACM SIGCOMM*, 2003.
[16] A. Ermolinskiy and S. Shenker. Reducing transient disconnectivity using anomaly-cognizant forwarding. In *Proc. Workshop on Hot Topics in Networks*, 2008.
[17] C. Esteve, F. L. Verdi, and M. F. Magalhães. Towards a new generation of information-oriented internetworking architectures. In *Proc. ACM CoNEXT*, 2008.
[18] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE Trans. Networking*, 2000.
[19] A. Goel and P. Gupta. Small Subset Queries and Bloom Filters Using Ternary Associative Memories, with Applications. In *Proc. ACM SIGMETRICS*, 2010.
[20] B. GrtinvaU. Scalable Multicast Forwarding. In *ACM CCR*, 2002.
[21] D. Han, A. Anand, F. R. Dogar, B. Li, et al. XIA: Efficient Support for Evolvable Internetworking. In *Proc. USENIX NSDI*, 2012.
[22] C. Henke, C. Schmoll, and T. Zseby. Empirical evaluation of hash functions for multipoint measurements. In *ACM CCR*, 2008.
[23] C. E. LaForest and J. G. Steffan. Efficient multi-ported memories for fpgas. In *Proc. ACM/SIGDA*, 2010.
[24] M. Mitzenmacher. Compressed bloom filters. *IEEE/ACM Transactions on Networking (TON)*, 2002.
[25] S. C. Nelson and G. Bhanage. GSTAR: Generalized Storage-Aware Routing for MobilityFirst in the Future Mobile Internet. In *Proc. ACM MobiArch*, 2011.
[26] A. Ostlin and R. Pagh. Uniform Hashing in Constant Time and Linear Space. In *Proc. ACM STOC*, 2003.
[27] M. Á. Ruiz-Sánchez, E. W. Biersack, and W. Dabbous. Survey and taxonomy of ip address lookup algorithms. *Network, IEEE*, 2001.
[28] M. Sarela, C. E. Rothenberg, T. Aura, and Zahemszky. Forwarding anomalies in Bloom filter-based multicast. In *Proc. IEEE INFOCOM*, 2011.
[29] A. Singla, P. Godfrey, K. Fall, G. Iannaccone, and S. Ratnasamy. Scalable Routing on Flat Names. In *Proc. ACM CoNEXT*, 2010.
[30] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. In *ACM CCR*, 2005.
[31] H. Song, F. Hao, M. Kodialam, and T. Lakshman. IPv6 lookups using distributed and load balanced bloom filters for 100gbps core router line cards. In *Proc. IEEE INFOCOM*, 2009.
[32] W. Xu and J. Rexford. MIRO: Multi-Path Interdomain ROuting. In *Proc. ACM SIGCOMM*, 2006.
[33] M. Yu, A. Fabrikant, and J. Rexford. BUFFALO: Bloom Filter Forwarding Architecture for Large Organizations. In *Proc. ACM CoNEXT*, 2009.
[34] F. Zane, G. Narlikar, and A. Basu. Coolcams: Power-efficient tcams for forwarding engines. In *Proc. IEEE INFOCOM*, 2003.
[35] K. Zheng, C. Hu, H. Lu, and B. Liu. A TCAM-Based Distributed Parallel IP Lookup Scheme and Performance Analysis. *IEEE Trans. Networking*, 2006.
[36] A. L. Zobrist. A New Hashing Method with Application for Game Playing. Technical report, 1970.