Effcient and Realistic Generation of IP Addresses

Joel Sommers
Colgate University
jsommers@colgate.edu

John Raffensperger
Colgate University
jraffensperger@students.colgate.edu

ABSTRACT

Network simulation and emulation environments play a crucial role in evaluating proposed protocols, applications, and networked systems. In such settings, the ability to scalably and efficiently generate traffic that has characteristics similar to those measured in the live Internet is of great importance. A key aspect of generating realistic traffic is to assign source and destination IP addresses to traffic flows such that the statistical structure of the addresses is similar to what would be seen in a live Internet setting.

In this paper, we propose and evaluate an algorithm and data structure for efficient and realistic generation of IP addresses. We describe our new method and compare it with existing and prior work, while also showing that our technique is far more efficient — both in terms of memory consumed and computation time required. We also show that the statistical structure of the generated addresses is similar to what would be measured in the live Internet. Our results show that it is possible to efficiently generate addresses over the entire IPv4 address space, and that it is feasible to generate addresses from a /64 IPv6 subnet.

Categories and Subject Descriptors

C.2.5 [**Local and Wide-Area Networks**]: Internet (*e.g.*, TCP/IP); C.4 [**Performance of Systems**]: Modeling Techniques; I.6.7 [**Simulation Support Systems**]

General Terms

Algorithms, Design, Experimentation

Keywords

Network traffic generation, IP addresses, Tries, Network simulation, Network emulation

1. INTRODUCTION

The Internet has seen massive expansion over the past decades. Because of its decentralized and dynamic nature, gaining an understanding of the Internet's behavior and properties has posed serious challenges to the research community. As a result, network simulation and emulation environments [1–3, 9, 11, 13] have been relied upon for yielding insights into different aspects of the Internet. Simulation and emulation settings are of critical importance because they are controllable, experiments can be made repeatable, and virtually any characteristic of interest can be measured.

A key requirement of any simulation or emulation environment is that it be *realistic*. Specifically *how* that realism is embodied in a given testbed or simulation experiment generally depends on the research question at hand, but a common need is to create network traffic conditions that are representative of what would be observed in the live Internet. Tools for generating Internet-like traffic can generally be classified as either *model-based* [12, 17] or *replay-based* [8, 15]. In model-based systems, a structural or behavioral model forms the basis of traffic generation, and the model is often parameterized with measurements that have been collected from a live environment. In contrast, in a replay-based system a packet trace collected in a live environment is re-emitted (perhaps with some minimal modification of source and destination IP addresses) in the test environment. In either case, the goal is to create traffic conditions that are in some way comparable to those that would be experienced in the wild.

An important aspect of realism with respect to traffic generation tools is for the source and destination addresses observed in flows to reflect the distributional characteristics of flows observed in the live Internet. For example, one might want the series of IPv4 destination addresses seen at a given router in a simulator to be similar in distribution to those that might be observed in the Internet. Realistic generation of addresses is important for assessing IP forwarding (longest prefix match) performance, for creating a representative mix of origin-destination flows in a simulated or emulated network, and in security applications such as anomaly detection in which source and destination addresses of flows are critical to the algorithm's performance. Two key challenges for generating realistic addresses are that (1) a new address must be generated quickly, since, in the context of a simulation or emulation environment, traffic generation should not be slowed down due to selection of a source or destination address, and (2) the generation technique must be frugal in its use of memory, since there may be multiple, separate address spaces that are used by different traffic generation instances in the simulator.

One possible approach to generating IP addresses might be to simply choose an address at random from a network prefix. We do not consider this approach further in this paper, since it would result in a series of addresses that do not reflect the complex multifractal characteristics observed in live Internet traces [10].

Two other basic approaches have been used in prior work to address the problem of generating representative network addresses in a simulation or emulation setting. The first is exemplified by the Harpoon traffic generator [12], which can be configured with a list of IP prefixes for source and/or destination addresses. The tool selects a prefix at random from the configured set, then selects at random a given address from the prefix. To generate addresses that conform to a given distribution (*i.e.*, to ensure realism), it is incumbent on the experimenter to construct the prefix list such that random selection of any address in the list will yield the desired characteristics. The Swing traffic generator takes a similar approach [17].

Another approach has been to create a complete structure that models the address space from which addresses are to be generated. Building on the earlier work of Kohler *et al.* [10], Barford *et al.* [5] developed such an approach using a *multiplicative, multiscale innovations model*. Their approach considers a density function defined on the interval $[0,1]$, and directly maps this to a given IP address prefix. A unit mass is assigned to the entire interval, then the prefix is recursively subdivided and the mass is correspondingly reallocated in the resulting subintervals. How the mass is subdivided is based on a parametric distribution function that is fitted to data collected in a live setting. Once the interval is subdivided and mass is reallocated to the granularity desired (*e.g.*, dividing a prefix into a series of IP addresses that make up the prefix), addresses can be selected at random, weighted by the mass that has been allocated.

Unfortunately, although the first (Harpoon-like) approach is computationally fast, it is clearly inefficient in space (the list of prefixes may potentially be quite long), and configuration can be a serious burden on the experimenter. The second approach (*i.e.*, Barford *et al.*) is also problematic, since the *entire* structure from which to generate addresses must be pregenerated. Thus, it is inefficient in computational time, and if the address space is reasonably large, it is also inefficient in memory usage.

The contribution of this paper is an algorithm for generating a realistic distribution of IP addresses that is efficient in both memory and computation time required. Building on the work of [5] to generate addresses that exhibit the multifractal characteristics observed in live Internet traces, we design and evaluate a new trie-based algorithm for generating addresses. We focus in particular on generating IPv4 source and destination addresses, though we also comment on generating IPv6 addresses. Our approach is designed to be highly efficient and to impose minimal cost on a large simulation or emulation setting in which many addresses need to be generated.

To examine the performance of our new trie-based approach, we describe a series of experiments in which we generated addresses for different sized prefixes, up to the entire IPv4 address space. We show that our method preserves the multifractal characteristic that has been observed in live Internet traces [10]. Our results also show that our approach scales well to large address spaces, and that addresses are generated quickly and with parsimonious use of memory. For example, generating 1 million new random addresses from the full IPv4 address space takes less than 2 microseconds on average per address, while consuming an average of about 50 MB. Moreover, our results suggest that generating addresses from much larger address spaces, such as an IPv6 /64 prefix, is feasible.

The remainder of this paper is organized as follows. In Section 2 we discuss work related to ours. Following that, in Section 3 we describe the design and implementation of our trie-based approach to address generation. In Section 4 we describe the results of experiments designed to examine the performance and behavior of our approach. Finally, in Section 5 we summarize our work and discuss future directions.

2. RELATED WORK

Most closely related to this paper are the works by Kohler *et al.* [10] and Barford *et al.* [5]. In [10], the authors establish that the structure of observed IPv4 addresses is multifractal. Using a Cantor dust model, they show how the observed address structure could be generated. The authors hypothesize that the reason for the observed multifractality has to do with the nature of address allocation. In that work, the authors do not discuss *generating new addresses* based on the observed characteristics.

In Barford *et al.* [5], the authors build on the insights of Kohler *et al.* and develop a model for generating addresses that have the same multifractal characteristics as those seen in a live environment. Their approach is based on a multiplicative, multiscale innovations model, which we describe in the next section. While this technique can effectively generate random IP addresses that have the desired distributional characteristics, it is costly both in terms of memory and computation time. We discuss this issue below.

Generating realistic IP addresses is important both in emulation environments in which commodity workstations and routers are used, as well as in simulation settings. In emulation environments such as Emulab [1] and WAIL [3], traffic generation tools such as Harpoon [12], Swing [17], and others [4, 8, 14] have been employed. The source and destination IP addresses assigned to generated flows depend on the design of the traffic generator. In the simplest case, addresses from a collected packet trace are directly reemitted, *e.g.*, in tcpreplay [15]. In other cases, basic distributional characteristics of addresses can be reproduced, given a suitable configuration, as in Harpoon [12]. None of these tools are able to generate addresses with realistic distributional characteristics from a large address space and with low computational and memory overhead.

Similarly, network simulators typically have various traffic generation capabilities, *e.g.*, the PackMIME and t-mix generators in ns-2 [6, 18], that are able to create flows with characteristics similar to those that would be measured in a live environment. Although there are rather sophisticated capabilities in various network simulators [2, 7, 9, 11, 13], we are not aware of any simulation traffic generator that is able to efficiently generate realistic network addresses.

3. ADDRESS GENERATOR DESIGN

In this section we describe our method for efficiently generating IP addresses. We first describe in more detail the method of Barford *et al.* [5] on which our work is based. We then describe our new technique and discuss its properties.

3.1 Detailed Background

As discussed above, the work in [5] proposes a random cascade model for generating a random IP address distribution with multifractal characteristics similar to those found in measured data (*cf.* [10]). The idea in [5] is to consider a density function with unit mass assigned to the interval $[0,1]$, and to map this interval to a prefix from which to generate addresses. The interval is recursively subdivided in two, and the mass is divided among the two smaller intervals. The address prefix length determines the number of times this subdivision takes place. Once the prefix has been divided into the series of IP addresses that comprise the prefix, addresses can be selected at random, weighted by the mass that has been allocated to each address.

Figure 1 depicts this process: at the top level, we have mass 1. This mass is subdivided into two intervals. Each interval at the second level is then subdivided into two more intervals, and so on. In this example, addresses are generated from a /29 IPv4 prefix (*i.e.*, 3 bits are generated and appended onto the 29 bit prefix). Thus, the recursive subdivision takes place 3 times. At the lowest level, the mass in each of the 8 bins represents the probability of generating that address.

In Barford *et al.*, the mass redistribution function is based on the Beta distribution. The Beta distribution takes two parameters β and β'; in [5], a single value $\beta = \beta'$ is used. This parameter can be fitted to measured data from the live Internet. Once the estimate of β is obtained, it can be used to *generate* new addresses. The authors show that for different aggregations of live traces, the fitted value of β leads to generating addresses that have sparse, bursty (multifractal) characteristics similar to those measured in live traces.

In this paper, we do not attempt to improve on the fundamental ideas of this approach. Our focus is rather on *how it is carried out*. Specifically, the authors in [5] describe modeling the address space and generating addresses in the context of the Haar wavelet decomposition. While the Haar wavelet computation can be done quickly, it requires creating a structure that represents the *entire* address space (*i.e.*, the bottom row in Figure 1). This approach is wasteful for two reasons. First, because of the sparsity observed in IP address distributions measured in live traces, there may be significant memory wasted by creating a structure that considers addresses that will potentially *never* be generated. Moreover, the up-front cost of computing the entire structure may be significant for larger addresses spaces. For example, if one wanted to generate addresses from the entire IPv4 address space, a structure on the order of 2^{32} elements would need to be created prior to any address generation. If each element consumes 4 bytes, the memory required would be 16 GB: clearly too much.

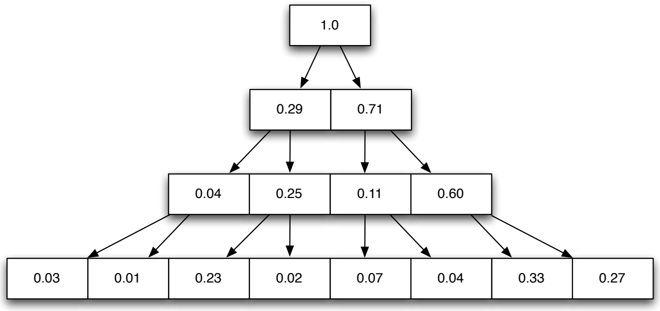


Figure 1: Depiction of the method in Barford *et al* [5] for generating addresses for a /29 IPv4 prefix (3-bit host address).

3.2 A Trie-based Approach

There are two key ideas with our approach. First, we exploit the sparsity observed in measured IP address distributions in order to reduce the memory footprint of address generation. Second, we construct the in-memory structure required to generation address *dynamically*, as new addresses are needed. This approach amortizes the computational cost of address generation over time, while still imposing low overhead to generate a single address.

At the heart of our method for generating addresses is the well-known *trie* data structure. In particular, we use a unibit trie. Tries provide an efficient way to store and retrieve a set of k-bit strings. A unibit trie is simply a tree in which each node contains two pointers: a 0-pointer and a 1-pointer [16]. Variations on the trie are used in standard algorithms to implement longest prefix match lookup for IP routing. We augment each node to contain not only pointers to the next bit, but also to include a *link transition probability*. As in [5], this probability is drawn from the Beta distribution.

The process of generating addresses starts with an empty trie. For each new address to generate we produce one bit at a time, starting with the most significant bit. To produce the first bit of the first address, we must first create a root node. We draw a value p from the Beta distribution with parameter β and add this to the node. This probability p represents the likelihood of generating a 0 as the first bit; $1 - p$ is the likelihood of generating a 1 as the first bit (thus, we only store one additional value in a node beyond the 0- and 1-pointers). We then draw a uniformly distributed random number p' from the interval $[0, 1]$. If $p' < p$, we generate a 0, otherwise we generate a 1. We then create a new node which is linked to the root. If we generated a 0 at the root, we link this new node to the 0-pointer, otherwise we link it to the 1-pointer. We repeat this process by drawing a new value p from the Beta distribution (adding it to the most recently created node), followed by another uniformly distributed value p'. We again test $p' < p$ to decide whether to generate a 0 or 1, then create a new node and link it to the pointer corresponding to the bit generated. We continue this process for the remaining bits in the host address.

An example of this process is depicted in Figure 2. In this example we generate a 4-bit host address for a 28-bit IPv4 prefix. The figure depicts how the trie evolves as three addresses are generated in succession. The subgraph on the left shows the state of the trie after generating one address (0100), the subgraph in the middle depicts the trie after generating a second address (0110), and the right subgraph is the state of the trie after generating a third address (1011). Note that when a new address is generated, a particular node may already have been constructed. In that case, we simply draw a new uniformly-distributed random number to determine which bit to generate (and which link to follow). Note also that no explicit link needs to be made from the leaf nodes since they contain enough information (namely, p) to generate the last bit of an address. As a result, the same structure as in Figure 2 can be used to generate the addresses 010*, 011*, and 101*.

Lastly, note that the function of the transition probabilities (Beta-distributed random numbers) at each level of the trie is equivalent in effect to the redistribution of mass from one level to another in [5]. Thus, given an appropriate parameter β, we should expect to generate addresses that exhibit the sparse, bursty distributional characteristics observed in live Internet traces. In the results of our experiments, below, we show that our new method indeed preserves these important characteristics.

There are a number of optimizations that can be made for single-bit tries [16]. One common technique used when implementing a single-bit trie to represent prefixes or addresses is to use some kind of compression to conserve memory. In addition to a simple unibit trie, we also implemented a compressed version that coalesced a series of nodes that each had a single branch. Our compressed trie functions somewhat differently from a standard level-compressed trie primarily because the structure is not used for IP longest-prefix match lookups. Rather, since we must retain a transition probability at each node, a compressed node must be able to store an array of such probabilities. Moreover, as new addresses are generated, each probability (even in compressed nodes) must be consulted. Thus, any expected potential savings of this approach would be in memory usage rather than computation. We compared a compressed trie implementation to a basic unibit trie and found that, in general, the additional complexity of the compressed trie is not offset with any significant reduction in memory consumption. While IP longest prefix match lookup workloads can take effective advantage of various forms of trie compression, the probabilistic construction of the trie in our address generation method does not lead to very many

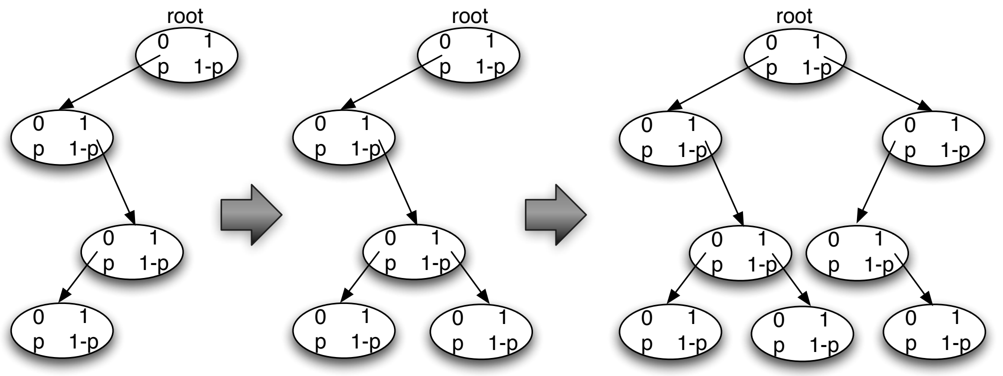


Figure 2: A depiction of how the in-memory trie is constructed as the addresses 0100, 0110, and 1011 are generated. Note that only the pointers to the next bit and the value p must be stored in a node. $1-p$ is shown in each node for conceptual completeness. Note also that the actual values p are drawn from the Beta distribution, thus are different for each node.

opportunities to compress a path. As a result, we do not consider trie compression further in this paper.

4. EVALUATION

In this section we evaluate our trie-based method for generating IP addresses. The primary motivation behind these experiments is to evaluate the suitability of our proposed method for generating addresses for a network simulation or emulation environment. In such a setting it is important to generate addresses very quickly and with low memory overhead. We also compare our technique with the prior work of [5] but do not explicitly compare against other techniques. We note that all other methods described previously in this paper would either require significant memory resources (*e.g.*, using the technique in [5], as we show below) or difficult manual configuration (using the technique in Harpoon [12]), making them problematic for our target settings.

4.1 Experiment Setup

The experiments described in this section were performed using a C implementation of our trie-based address generation method. We used a commodity workstation running a 64-bit build of FreeBSD 8.0. The machine was equipped with 4 GB of RAM and a quad-core Intel Xeon E3120 processor running at 3.16 GHz. During the experiments there were no non-OS-related user processes running, and the server was not running any X Window services.

4.2 Realism

First, we examine how well our trie-based method can generate a bursty, sparse distribution of IP addresses similar to those in a live setting. We do this by examining a plot of the multifractal spectrum of a set of generated addresses. For this experiment we set $\beta = 0.61$ (a value that Barford *et al.* found to fit their data), and generated 1 million addresses from a /16 prefix. Figure 3 was generated based on the histogram method of Kohler *et al.* [10] (we refer the reader to that work for a description of the method). As with both [5] and [10], sampling effects tend to dominate the analysis at finer scales. Over medium scales, however, the figure shows a range of values for different scaling exponents, consistent with multifractal behavior. Highly similar results are shown in [5], which should not be surprising given that our trie-based approach is functionally equivalent to their method.

We also examined the rank-frequency plots and histograms of

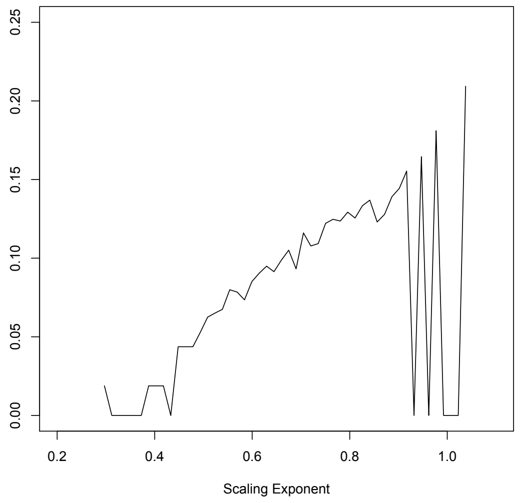


Figure 3: The multifractal spectrum of a set of generated addresses. Plot generated using the histogram method of [10].

generated addresses. While we do not show those results in this paper, they exhibit similar qualitative features as addresses from measured traces shown in [5, 12].

4.3 Performance

We now turn to examining the performance of our trie-based method. The baseline parameters we used in experiments described below are to generate 1 million addresses from a /8 prefix, with $\beta = 0.61$. In the results discussed below, we vary the number of addresses generated, the prefix length, and the value β to examine the performance of our address generation method. For each data point in the plots below, we computed the average over 10 runs.

4.3.1 *Varying the number of addresses generated*

We first examine the runtime and memory consumption of our method while varying the number of addresses generated. Since the trie is dynamically constructed, generating additional addresses results in a larger memory footprint. In particular, as the number of addresses generated increases, the number of chances for the trie to expand (*i.e.*, for new nodes to be created) increases. As an upper limit, there could be $2^{k-1} \times b$ bytes consumed by the trie, where k is the prefix length and b is the number of bytes consumed per node. We never came close to this maximum in testing, even when generating 10 million addresses.

Figure 4 shows results of generating up to 10 million addresses while fixing the prefix length at /8 and setting β to 0.61. The figures show runtime (seconds) and memory consumed by the trie. Error bars indicate one standard deviation above and below the mean. Note that the average time to generate an address actually *decreases* as we generate more addresses. The reason for this is that after a large number of addresses have been generated, fewer nodes must be created in the trie, which speeds address generation. We also see that memory consumption rises modestly over the range of number of addresses generated, peaking at around 18 MB for 10 million addresses.

4.3.2 Varying the prefix length

Next, we examine the effect of varying the prefix length. Figure 5 shows the run time and memory consumption of our trie-based method while generating 1 million addresses and varying the prefix length from 0 to 30. Generating addresses with a 0-length prefix results in generating *entire* 32-bit IPv4 addresses, while generating addresses with a 30-bit prefix can only result in four distinct addresses. We see from the figure that for prefixes of (around) length 16 and longer, the memory consumes by our trie-based algorithm is minimal. We also see that the average time required to generate decreases as we generate fewer bits, which should be expected.

As prefix length shrinks from 16 bits (*i.e.*, as more bits are generated), the time required to generate an address increases modestly, up to an average of just under 2 microseconds. The memory consumption also increases, though somewhat more significantly as the prefix shrinks from 16 bits to the full IPv4 address space. At worst, our trie-based method consumed about 50 MB to generate 1 million addresses from the full IPv4 address space. We note that this is significantly smaller than a method that would require some amount of storage for every unique address (*i.e.*, 2^{32} storage locations).

4.3.3 Varying β

Finally, we turn to examining the effect of different values of β on the performance of our trie-based method. The parameter β affects the structural characteristics of addresses generated, including the sparsity of addresses generated. As noted in [5], one of the reasons for using the Beta distribution is that it is a two-parameter distribution, thus provides significant modeling flexibility.

Indeed, the choice of β has a great impact on the structure of the trie and is the determining factor in distributional characteristics of the addresses that are generated. Specifically, as $\beta \to \infty$, the branching probabilities, p, approach $1/2$. As a result, it becomes more and more likely for the entire trie to be built because each branch at each node has an equal chance to be taken. Conversely, as $\beta \to 0$, the result is a bimodal distribution in which p is either 0 or 1. As a result, the trie will be extremely sparse because the probability that one branch will be taken goes to zero, meaning that the other branch is almost always taken. This results in very little new construction of paths as addresses are generated.

Figure 6 shows the results of experiments in which the value of β was varied between 0.1 and 5. A /8 prefix is used for these experiments (24 address bits generated). We see that for very small values, the memory footprint of the trie is quite small (as expected, the tree is quite sparse). As β increases, both runtime and the memory footprint increase quite significantly. With a large β value like 5, the memory footprint is just over 50 MB, still suggesting some sparsity in the tree (consider that a complete tree would contain 2^{24} nodes, which would be approximately 256 MB, assuming a node that consumes 16 bytes). We note that in prior work [5], it is suggested that values of β fitted from live traces will likely be less than 1, which would result in rather sparse tries. Thus, we expect the performance of our method to be very good when generating a realistic set of addresses for a simulation or emulation setting.

4.4 Comparison with Prior Work

We wrote another C program to generate addresses using the prior method of [5], also setting β to 0.61. Using and initializing a structure to do address generation with this prior method can be expensive both in CPU time and memory, depending on the prefix length.

Table 1 shows the memory required and initialization time for a range of prefix lengths using the address generation method of [5]. First, we see that for a /16 prefix, relatively little memory is needed (1 MB). For a /8 prefix, however, 256 MB are required to hold the data structure. (Each of the 2^{24} elements in the structure required 16 bytes in our implementation; some optimization may be possible, but at minimum 4 bytes would be necessary per element, *i.e.*, the size of a single-precision floating point number.) We did not have sufficient RAM in our test machine to handle shorter prefixes, thus rendering it impossible to generate addresses from the full IPv4 address space (64 GB would be needed!). Note that with our proposed technique, we only required about 80 MB to generate addresses from the *full* IPv4 address space and less than 10 MB to generate addresses from a /8 prefix (*cf.* Figure 5).

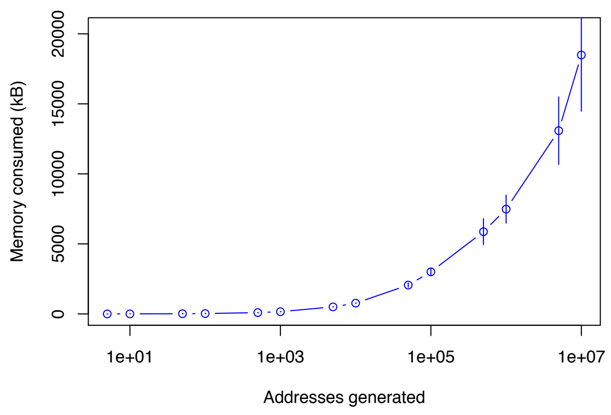
With respect to time required to generate addresses, we see in Table 1 that little time is required to create the necessary data structure for prefixes longer than /8. For a /8 prefix, about 7 seconds are required to create the data structure, which must be done *before* any addresses are actually generated. Once the structure is created, addresses can be generated quickly and with essentially fixed CPU cost. Thus, as more addresses are generated, the average cost per address goes down. (Clearly, the worst case with the approach of [5] is that the structure must be created to generate a single address.) When we generated 10 million addresses from a /8 prefix, the average time to generate one address was reduced to about 2 microseconds, which is similar to the performance of our proposed technique. The benefit of our proposed method, however, is that there is no up-front cost of creating an entire data structure. Thus, the CPU time required to generate *any* address is quite small (*cf.* Figure 4).

Table 1: Memory requirements and initialization time for generating addresses using the method of [5].

Prefix Length	Memory Required (kB)	Initialization Time (sec)
16	1024	0.029
12	16384	0.455
8	262144	7.285
4	4194304 (not feasible)	—
0	67108864 (not feasible)	—

4.5 Discussion

Our results above show that our trie-based method for network address generation is a good fit for network simulation and emulation environments. In particular, our approach produces addresses



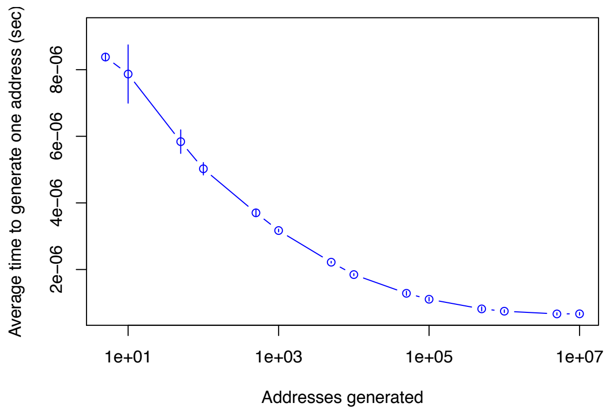
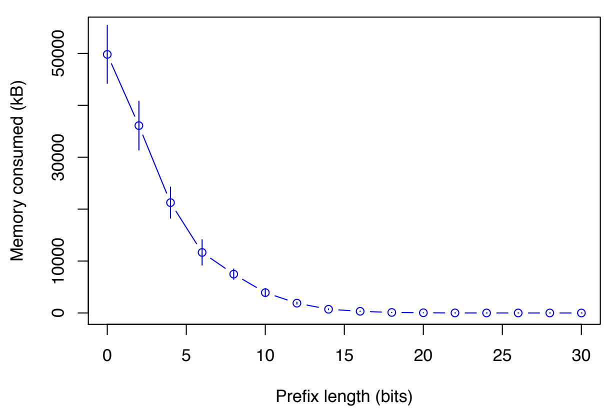


Figure 4: Memory consumption (left) and run time (right) of the trie-based address generation for a range of number of addresses generated. The prefix length is fixed at /8 and β is fixed at 0.61 for this set of experiments. Error bars show one standard deviation above and below the mean. (Note that the x axis is on log scale.)



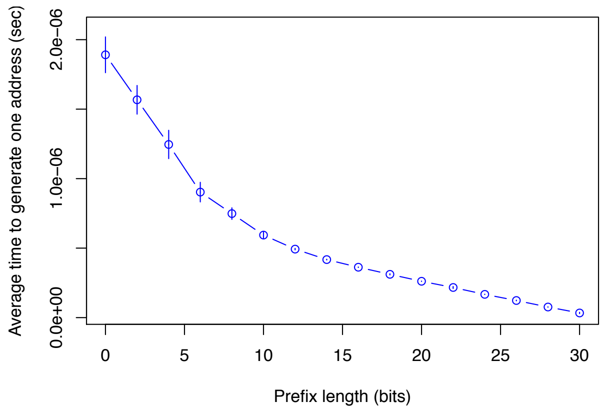


Figure 5: Performance data on memory consumption (top) and runtime (bottom) when varying the prefix length. The value of β is fixed at 0.61 and 1 million addresses are generated for each prefix. Error bars show one standard deviation above and below the mean.

very quickly and with low memory overhead, and the resulting addresses exhibit the important multifractal qualities found in real traces. While our experiments have been done in the context of generating IPv4 addresses, the specific results in Section 4.3.2 suggest that our approach may scale to larger addresses, *e.g.*, IPv6 /64 prefixes, assuming sufficient sparsity in address space allocation, which we expect to be the case. However, it is yet unclear whether our technique can scale effectively to generate full 128-bit IPv6 addresses. Perhaps more importantly, it is presently unknown whether distributional characteristics of IPv6 addresses are similar to IPv4 addresses. We intend to investigate these issues in future work.

5. SUMMARY AND CONCLUSIONS

In this paper we propose and evaluate a new method of generating IPv4 addresses for traffic generators in network simulation and emulation environments. At the core of our technique is a unibit trie structure that is dynamically constructed as requests are made for new addresses to generate. We evaluate the performance of our address generator and show that our method results in a set of addresses that have similar sparse, bursty (multifractal) distributional characteristics as addresses observed in live settings. Our results show that our technique is well-suited for fast, low-overhead generation of realistic addresses for traffic generation in network simulators and emulators.

In future work, our goals are to investigate further the structure of IPv6 addresses and to evaluate how well our method scales to that address space. We also intend to release the C-based implementation of our address generation method to the research community so that our technique can be incorporated into new and existing simulation and emulation tools.

Acknowledgments

This work was supported in part by NSF grant CNS-1054985 and Colgate University. Any opinions, findings, or conclusions in this material are those of the authors and do not necessarily reflect the views of the NSF or Colgate University.

6. REFERENCES

[1] Emulab. `http://www.emulab.net`, 2011.
[2] The ns-3 network simulator. `http://www.nsnam.org`, 2011.
[3] The Wisconsin Advanced Internet Laboratory. `http://www.wail.wisc.edu`, 2011.

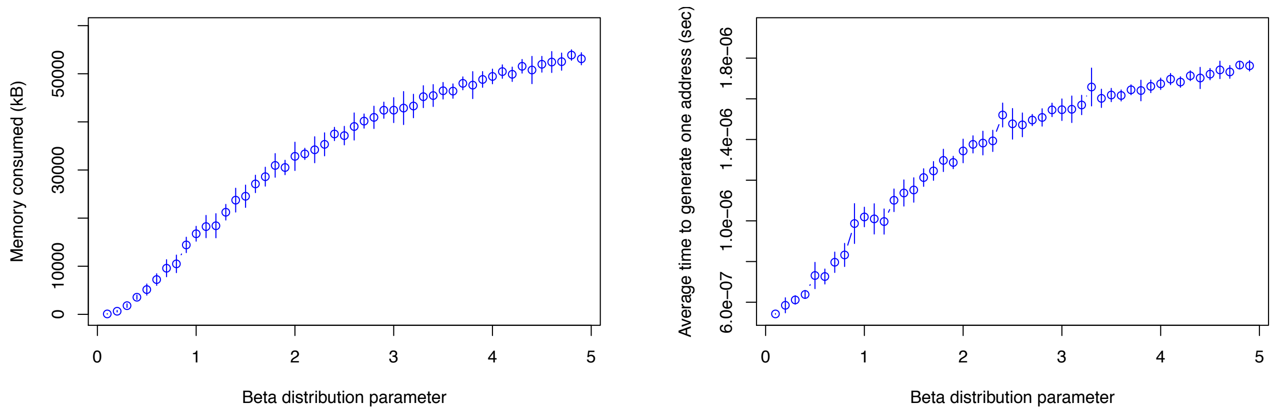


Figure 6: Performance data on memory consumption (top) and runtime (bottom) when varying the β parameter. 1 million addresses are generated from a /8 prefix for each value of β. Error bars show one standard deviation above and below the mean.

[4] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of ACM SIGMETRICS*, Madison, WI, June 1998.

[5] P. Barford, R. Nowak, R. Willett, and V. Yegneswaran. Toward a Model for Sources of Internet Background Radiation. In *Proceedings of the Passive and Active Measurement Conference (PAM '06)*, March 2006.

[6] J. Cao, W. Clevelan, Y. Gao, K. Jeffay, F. Smith, and M. Weigle. Stochastic Models for Generating Synthetic HTTP Source Traffic. In *Proceedings of IEEE INFOCOM '04*, Hong Kong, March 2004.

[7] X. Chang. Network simulations with OPNET. In *Proceedings of the Winter Simulation Conference*, volume 1, pages 307–316, 1999.

[8] Y. Cheng, U. Holzle, N. Cardwell, S. Savage, and G. Voelker. Monkey See, Monkey Do: A Tool for TCP Tracing and Replaying. In *Proceedings of USENIX Technical Conference*, Boston, MA, June 2004.

[9] J. Cowie, H. Liu, J. Liu, D. Nicol, and A. Ogielski. Towards Realistic Million-node Internet Simulations. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[10] E. Kohler, J. Li, V. Paxson, and S. Shenker. Observed Structure of Addresses in IP Traffic. *IEEE/ACM Transactions on Networking*, 14(6):1207–1218, 2006.

[11] S. McCanne, S. Floyd, K. Fall, K. Varadhan, et al. Network Simulator ns-2, 1997.

[12] J. Sommers and P. Barford. Self-Configuring Network Traffic Generation. In *Proceedings of ACM Internet Measurement Conference*, Taormina, Italy, October 2004.

[13] J. Sommers, R. Bowden, B. Eriksson, P. Barford, M. Roughan, and N. Duffield. Efficient Network-wide Flow Record Generation. In *Proceedings of IEEE INFOCOM '11*, Shanghai, China, April 2011.

[14] A. Tirumala, J. Ferguson, J. Dugan, F. Qin, and K. Gibbs. Iperf. `http://dast.nlanr.net/Projects/Iperf`, 2011.

[15] A. Turner. Tcpreplay. `http://tcpreplay.synfin.net`, 2011.

[16] G. Varghese. *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. Elsevier/Morgan Kaufmann, 2005.

[17] K. Vishwanath and A. Vahdat. Realistic and Responsive Network Traffic Generation. In *Proceedings of ACM SIGCOMM '06*, Pisa, Italy, September 2006.

[18] M.C. Weigle, P. Adurthi, F. Hernández-Campos, K. Jeffay, and F.D. Smith. Tmix: A Tool for Generating Realistic Application Workloads in ns-2. *ACM SIGCOMM Computer Communication Review (CCR)*, 36(3):67–76, July 2006.