Hashtray: Turning the tables on Scalable Client Classification

Nik Sultana∗, Pardis Pashakhanloo∗, Zihao Jin†, Achala Rao†, and Boon Thau Loo∗
∗University of Pennsylvania, †Tsinghua University

Abstract—Applications typically use a table to track their clients, which are classified according to their behaviour. But as the number of clients increases so does the amount of state required to remember this classification over time.

In this paper we explore the trade-off between data-structure accuracy and network size when needing to remember client state. We present Hashtray—a hash table library that consists of a generic API and instantiations of various kinds of maps—and a system to evaluate and compare different data structures.

We evaluate Hashtray in the context of Denial-of-Service mitigation using both a modelled network of $10^6$ machines, and a testbed experiment with over 200 hosts connecting to a version of Apache modified to use Hashtray. The system is open-sourced to enable others to extend or build on this work.

Index Terms—denial-of-service, randomized data structures, hashing

I. INTRODUCTION

Analytics applied to traffic patterns or application-specific metrics can scrutinise an online system’s networked clients to improve security or tune the overall system’s behaviour for performance and service quality [1]. Thus an online system such as a web service can classify clients into groups according to the clients’ behaviour using various techniques including signature matching and machine learning [2].

Once a classification is made, the classification must be remembered to avoid the expense of redoing the classification in the near future, or to change the system’s response to its clients. For example, a client that is behaving suspiciously might be quarantined for a period of time during which traffic from its network address is routed or inspected differently.

But as the number of clients increases so does the amount of state required to remember the clients’ classification over time. Forgetting clients’ classifications might harm the service’s quality or security. For example, clients who are judged to be participating in a Denial-of-Service [3] (DoS) campaign against an online service could be block-listed for a period of time to stymie the attack. Alternatively, or additionally, one could use load balancing to diffuse the memory pressure to several hosts, but this incurs the expense of maintaining redundant resources using which to scale out, and the difficulty of coordinating between those hosts to maintain the level of consistency required.

In this paper we explore the trade-off between data-structure accuracy and network size when needing to remember client state.

We present Hashtray: a hash table library that consists of a generic API and instantiations of various kinds of maps. These include various concurrency-friendly instances of our Cuckoo filter [4] implementation—a randomised and approximate data structure—and an example wrapper for third-party hash table implementations.

Hashtray is designed to fulfill two objectives: i) be straightforward to integrate with existing and new applications, which can make application-specific calculations of client features such as reputation, and offload this for storage in Hashtray; ii) provide an evaluation testbench for different kinds of data structures used through a common interface, to understand which data structures scale best when tracking client state.

We evaluate Hashtray in the context of mitigating Denial-of-Service attacks by measuring how well Hashtray can remember which hosts are likely to be participating in a DoS attack against a service.

Our evaluation consists of two approaches. The first approach uses a tool we implemented to model a large network:
Fig. 1 shows the result of modelling a network with $10^6$ clients, a varying percentage of which are hostile, and the effectiveness of different data structures in reducing the “stall” brought about by DoS. The second approach involves a testbed experiment with over 200 hosts connecting to a version of Apache modified to use Hashtray to remember clients’ classifications.

Contribution. This paper demonstrates the use of a common library to provide applications with different choices of storage for client state to evaluate which choice scales best. The two-pronged evaluation using both the model and the Apache extension demonstrates a methodology to use the same library to obtain complementary measurements on how the data structure deals with scale (when using the model) in a higher-fidelity evaluation (when using an actual application, such as Apache).

It is hoped that others can use or extend this work. Hashtray, including our network modelling tool, is released as open source software under a permissive license.1

II. BACKGROUND AND RELATED WORK

Probabilistic data structures such as Bloom filters [5] have been evaluated for host information management [6] and security [7] applications, including DoS mitigation for HTTP [8] and SIP [9].

But various such data structures exist, and it is not easy to evaluate them consistently from the same application. In this paper we based our evaluation on Cuckoo filters [4], which tout several benefits over Bloom filters, such as record deletion.

New variants of this filter are still being developed [10]. In the implementation evaluated in this paper, we settled on using a form of Cuckoo filter that behaves more like a hash table, based on Cuckoo hashing [11]—which has also been shown to perform well in high-throughput networking [12]—to allow us to associate information with a host’s hash (rather than only approximately checking whether that host belongs to a set).

Cuckoo hashing. Cuckoo hashing provides a probabilistic and approximate data structure that provides constant-time lookup and deletion, and amortised constant-time insertion. It behaves like a hash table where each key can be mapped to multiple addresses in the table, of which one is chosen at random. An address in a table references a block which contains a number of entries, each of which can store a key-value pair. Perhaps the most common configuration is a “2,4-table”: each item may map to 2 blocks, where blocks contain 4 entries. If a block’s entries are all full, then an entry may be displaced to its alternative block. An iterative “kicking out” process continues until some maximum value, displacing entries between blocks until a free entry is found.

III. HASHTRAY

Hashtray consists of an API and library that collects different hash table implementations. Currently it includes a wrapper for a third-party hash table and the hash-table variant of the Cuckoo filter [4] extended with the following features: concurrent access (thread safety), fine-grained locking (at the level of blocks), and eviction. We implemented three variants offering the same API: single-threaded (no locks), multithreaded, and multi-process. All features of the implementation are parameterizable: such as which hash function to use, the size of table, and the size (in bytes) of keys and values. We wrote this in C99, relying only on POSIX features—such as for inter-process communication—to ease—portability, and use the resulting header files and library in the Apache integration described below. We wrote a test rig that logs collisions and evictions, to ensure that the table behaves as intended. This is all the more helpful since, as a randomized structure, its behavior could vary from one run to the next; even if the random seed is kept constant, the thread interleaving may be varied by the OS’ scheduler.

IV. NETWORK MODEL

To evaluate DoS effects in large-scale networks we wrote a simulator to model the effects of using different instances of maps provided in Hashtray’s library. In our model, a multi-threaded server services hosts that are either “good” or “bad”. Servicing good hosts has no ill effect, but servicing bad hosts incurs stall. Stall is what results in a denial-of-service to bona fide users. When a system is stalled, it cannot serve clients; this is what makes bad hosts “bad”. Stall accumulates when bad hosts are serviced. We use a map in Hashtray to mitigate this stall, by recognising bad hosts (from past accesses) and refusing to service them. Without such a memory, we pay the cost of stall each time a bad host returns to our server. If the map being used is randomised then its behaviour varies. Moreover, since it is finite, it may lose information after a while. Thus this simulation serves to model the utility of a particular data structure (instantiated in Hashtray) to avoid stall.

In our model, we simulate having 10 serving threads, having shared access to a Cuckoo hash. Misclassification (i.e., confusing a “good” host with a “bad” one because of a hash collision) incurs a penalty. Our model uses a 100-block 2,4-table in a network of 1,000,000 hosts. Parameters of our model can be easily changed.

We vary two parameters in our simulation, PGH and PGC. PGH is the “percentage of good hosts”: how many hosts in our network are “good”. PGC is the “probability of a good connection”: how likely that each new connection arrives from a “good” host. We vary these parameters to model the changing ratio of “good” connections arriving at our system. This matters since when a system is under attack this ratio changes sharply since many connections are intended to induce stall to bring about the DoS.

V. APPLICATION MODEL

We also evaluate Hashtray by integrating it with an existing application, the Apache HTTP server. We developed a simple
A model for this integration, based on a structure to record client information:

```c
struct {
    unsigned class : 2;
    unsigned conns : 4;
    unsigned throttling : 1;
    unsigned last_classified : 16;
} data;
```

Here `class` encodes the connection’s classification (e.g., whether we consider the host to be “bad” based on past behaviour); `conns` the number of simultaneous connections the host has with us; `throttling` whether connection throttling is enabled (i.e., we can drop some connections from a client); and `last_classified` is a timestamp that we use to expire records. We take the lower 16-bits of the current time (at second-level precision), which means we can expire records after 18 hours at most.

We overlay this record into the entry stored in the Cuckoo table and implement a simple expiry policy—that we use for record eviction—based on the `last_classified` field.

### A. Integration with Apache

Starting with the pipelined Apache Worker MPM described in earlier work [13], we added logic that uses Hashtray as follows. We first gather information about the behaviour of clients, store it in a Hashtray instance, then use this information to change how Apache reacts to those clients in the future. We called our modified Worker MPM “Union” since all the worker threads are now sharing eventually-consistent state about how different clients are classified, based on the clients’ behaviour. Despite being a prototype, we sought to retain Apache’s portability, and we compiled and tested our Union MPM on both Linux and macOS.

**Global and cache tables.** We use the Cuckoo hash in two ways: i) a global table is shared by all threads (in all Apache processes), and ii) each thread keeps its own private table as a cache to reduce contention on the global table. The private copy does not need to use locks, therefore lookups happen faster. We try lookups on the cache first, and if there is a miss then we lookup the global table. Our expiry policy works consistently across both tables: a cache lookup on an expired record will result in a cache miss, thus we attempt to refresh the record from the global table.

**Logic.** When a connection arrives, we perform a lookup on the cache (and on the global table if the cache misses). If a (non-expired) record is retrieved, then the connection is treated according to its class. Otherwise, the connection is classified as an “under-observation” connection, on which data is gathered as the connection progresses through the application. Once a connection has been classified, it retains that class until the record expires.

1) **Parameters:** Our implementation accepts various parameters that influence its performance and tolerance to misbehaving clients, including the following:

- **Queue depth.** This is the size of the buffer in which queued connections are held until they are served by a worker thread. By default we set this to twice the number of threads Apache is using.
- **Connection threshold.** If a client exceeds this number of connections then it is classified as being malicious. By default, we set this to 10% of the queue depth.
- **Connection throttle threshold.** If a client exceeds this number of connections, then its classification is not changed, but new connections might be declined. By default, we set this to 5% of the queue depth.
- **Connection throttle ratio.** The ratio of new connections to decline if a client is being throttled. We set this to 2 by default: i.e., once a client is being throttled, every other connection is declined.
- **Good classification expiry.** How long (in seconds) does it take a good classification to expire. We set this to 5 by default.
- **Bad classification expiry.** How long it takes a bad classification to expire. This could be higher than “Good classification expiry” if there is confidence that classification is accurate, otherwise this expiry could be shorter. By default, we set this to 30 seconds.
- **Use cache.** As explained above, every thread keeps its own private table as a cache to relieve pressure from the global table. We can disable the cache and always query the global table. We enable this by default.
- **Measure connection duration.** If enabled, we take into account the duration of a connection when classifying it. This is a blunt instrument since the length of a connection is not necessarily indicative of malice, but this serves as an example that various considerations can be drawn on when classifying connections. We enable this by default.
- **Duration maximum.** If “Measure connection duration” is enabled, then this parameter sets the maximum duration of a connection. By default, we set it to 10ms.
- **Automated connection duration averaging.** Rather than having a constant “Duration maximum”, we also implemented a simple automatic averaging algorithm to calculate the maximum. We disable it by default because it often works too crudely, as described further below.

The problem with averaging and connection duration is that they currently take in too little information to be generally effective. There is a rather effective control process in TCP that influences the timeouts and sending rates it uses; ultimately suitable indicators would be needed in an application for connection duration to be a meaningful differentiator between clients. For example, if the client is carrying out a large download then its duration might be long indeed; the application’s indicators would have to reconcile duration with activities that can have a long duration. Even a small download might take long if the transport is operating with a small window and requires retransmissions. Analytics can be improved by drawing metrics from different parts of a system.

### VI. Evaluation

We evaluate the DoS-mitigating effectiveness of our Cuckoo filter instance in Hashtray in two ways:
• **Stall simulation** (§IV) we simulated a large network of hosts accessing a server to compare how much “stall” it would suffer from malicious hosts when compared to not using that filter. This gives us an approximate understanding of using that filter in a network that far exceeds our experimental resources.

• **Testbed experiment** (§V) we use a physical testbed to evaluate the DoS-mitigating ability of a modified version of Apache version 2.4.26.

A. Stall simulation

Our results are shown in Fig. 2. To understand the graphs, consider the case when connection filtering is not used (i.e., the continuous line). When PGH=51, then when 50% of connections are from good hosts (i.e., PGC=50), then the system stalls around 50% of the time (seen on the y-axis). But when PGH=21 and there’s a 50% chance of connections coming from good hosts (PGC=50), we have a harder time finding a good host, and roll the die every time we fail to find one. Thus there is a 50% chance that we will seek a bad host instead, and this is more likely to succeed quickly, and we get more connections from bad hosts as a result: i.e., the system is under attack. There we can see that the paucity of good hosts making connections to our server (relative to bad hosts making connections) lead to greater stall (over 75%). This means that 75% of the time the server is suffering the ill-affects of stall. In contrast, whenever PGC=100, then a bad host can never be picked, thus the system spends 0% of its time stalling.

**Limitations.** This is only a model. We assume that hosts are either good or bad, and stay so for the entirety of the experiment. Furthermore, we do not take variable network conditions into account. Factoring out those considerations, the model helps us understand the dynamics of using a chosen data structure for connection monitoring and filtering.

We run our simulator 5 times for each set of parameters. The unbroken line in the graphs shows the case when connection filtering is not used. The randomness used by our implementation results in an envelope of behaviour, which we average to give the dotted line. (We also ran the non-connection-filtered simulations 5 times, but the results were very deterministic and only varied slightly because of OS’ scheduling.) We ran the whole sweep of simulations several times, for different table and network sizes.

B. Testbed experiment

In this section we describe how we evaluate the performance of Apache extended with the use of Hashtray for connection monitoring and filtering, under normal and attack conditions.

**Experiment setup.** Our physical setup consists of 8 servers interconnected via 10GbE links to a switch, and each having an 8-core Intel Xeon E5-2630L 1.80GHz CPU and 64GB RAM on a Dell 07276D v.A01 motherboard. We use Ubuntu Linux 14.04.5 LTS running v4.4.0-31 of the kernel.

We run Apache on one of the servers. The other servers we use to run the measurement programs (we use httping, configured to make a single GET request each second, timing out after one second), and attack scripts.

Having only 7 machines does not put much pressure on the applications being evaluated (i.e., Apache) or the DoS mitigation. We need to increase address diversity to give the semblance of having a larger network. To this end we use DoSarray [14] to run 40 container instances on each machine, bridge their virtual network interfaces and give them all unique IP addresses. Thus we get a test network consisting of 240 nodes, any of which we can use to run measurement or attack scripts.

**Results.** Our results measure the availability of Apache as perceived by the measurement program run on each non-attack node.

We use the visualisation provided by DoSarray [14], which uses contour mapping to indicate the latency and responsiveness of the system-under-test. An example of this plot is provided at the base of Fig. 3a for a control experiment consisting of the unmodified Apache when not under attack, and in Fig. 3b showing the unmodified Apache when attacked by a two instances of SlowLoris [15].

We run the experiment for 120s, and set 3 attackers on the network (recall that the results in Fig. 3b used 2 attackers), and no loss of availability was observed. The results are shown in Fig. 4. In the case of Tor’s Hammer, latency appears to drop during the attack. This happens consistently each time we run the experiment. It might be an artefact of the virtualized network we use to simulate having a large network, or it might be that the larger quantity of network traffic triggers more frequent DMAs from the server NIC, leading to a perception of hastened service to the other clients. We get similar results when we used 4 attackers, but the mitigation broke when we used 6 attackers. Tuning the prototype and its parameters might yield better results.

**Overheads.** Using Union incurs very little overhead, and in this section we provide a break-down of the three kinds of overheads we consider: latency, memory, and CPU usage. Added latency is around 100µs as perceived and measured by the clients.

Two things contribute to memory overheads: the queue between threads, and the Cuckoo tables (both global and cache). We set the queue depth to be $2 \times \text{ThreadsPerChild}$ by default, where ThreadsPerChild is an Apache configuration parameter for “the constant number of worker threads in each server process”. We use Apache’s default value of 25. Each cell in the queue is 32 bytes wide, so the total overhead from the queue is $2 \times 25 \times 32 = 1.6\text{KB}$ per Apache server process, of which Apache starts 3 by default.

We use 32-bit keys and values in our tables. Recall that we use two kinds of tables: we use 100-block 2,4-table (so total of 3.2KB) shared among all threads and a 1000-block 2,4-table for cache (so a total of 32KB per thread). The maximum number of worker threads is set by Apache’s MaxRequestWorkers parameter, which has a default value of 400 threads, therefore the total overhead from the caches is 12.8MB. We use the same
number of threads as the standard Apache Worker MPM does: this is set in Apache’s configuration file.

We now turn to CPU overhead. We used the CPU’s timestamp counter to measure the time taken to perform lookups and insertions. We used a synthetic data set of 50,000 items and measured the lookups/insertions in a 1000-block 2,4-table.

The measurements were carried out on a laptop resourced with 2.9GHz Intel Core i5 and 16GB 2133MHz LPDDR3, and running macOS Sierra 10.12. We deliberately cooled the CPU’s caches by allocating and reading large blocks of memory to simulate interleaving with memory-intensive tasks. We found the maximum latency for lookup/insert operation to be around 100\(\mu\)s. Most of the time the latency is much less, on average in the order of 0.1\(\mu\)s even when the cache is cooled.

VII. CONCLUSION

Hashtray is designed to evaluate and tune in-application analytics, in particular the data structures that are used to track different kinds of client state. It complements other approaches such as DeDoS [16] which provides a platform for applications. Results from experiments that use Hashtray could be transferred to systems such as DeDoS.

In future work Hashtray can be used to understand the extent to which a single host’s client-tracking abilities can be optimised for the kind of state that an application needs to track. This is becoming more important since address-space diversity is increasing as the open Internet migrates to using IPv6 [17]. IPv4 addresses are not always good identifiers of single devices or users because operators often use multiple level of NAT. In one extreme case an entire country was banned from Wikipedia since the country NAT’d all its citizens, and Wikipedia’s banning applied at the level of IPv4 addresses [18]. Hashtray could help explore suitable data structures to serve more information-rich analytics.

ACKNOWLEDGMENT

We thank Markulf Kohlweiss for feedback and John Frommeyer for systems support. This material is based up work supported by the Defense Advanced Research Projects Agency (DARPA) under Contracts No. HR0011-17-C-0047 and HR0011-16-C-0056.
Figure 4: Apache running the Union MPM.