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*—Untrusted network clients can undergo a classification process before they are allowed to use more of a service's resources, and services typically rely on a table to remember the clients' classification. 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 tables—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, randomised data structures, hashing**

I. Introduction

Analytics applied to network traffic patterns or application-specific metrics can scrutinise the behaviour of an online system's networked clients to decide what limits to place on their access or resource use. If done adequately, this can improve security and help tune the overall system's performance and service quality [1].

Classification techniques include signature matching and machine learning [2]. Once a classification is made, it must be remembered in order to change the system's response to clients already classified, and to avoid the expense of redoing the classification in the near future. For example, a new client, or one that is behaving suspiciously, might be quarantined for a period of time during which traffic from its network address is routed or inspected differently. Following the quarantine, it might be subjected to deeper inspection or graduate into a "trusted" classification, thus freeing up resources to inspect other clients' connections.

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, if a service is being targetted by a Denial-of-Service [3] (DoS) campaign then classifying and remembering DoS-participating clients would allow the service to block those clients for a period of time, while allowing bona fide clients to continue using the service. One could use load balancing to diffuse the memory pressure from storing large tables on a single host, by sharding the table across several hosts, but that still leaves the question of what kind of data structure is best suited to remember the classification while using less resources. The contributions of this paper are complementary to distributed approaches to scaling systems.

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 tables. 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 (both existing and new) applications which can make application-specific calculations of client features that indicate a client's reputation, and offload this for storage in Hashtray; ii) provide an evaluation testbench for different kinds of data structures presented to applications 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 uses a tool we implemented to model a large network; and the second 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) and in a higher-fidelity evaluation (when using an actual application, such as Apache). 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 secu-

[1] http://www.seas.upenn.edu/~nsultana/hashtray

Listing 1: Key functions in Hashtray's API.

```
struct table* create_table(void);
void destroy_table(struct table* t);
enum outcome insert(struct table* t, data_t data, data_t metadata, int (*merge_fun)(data_t*
    stored, const data_t* new), int (*expiry_fun)(const data_t* metadata));
enum outcome delete(struct table* t, data_t data);
enum outcome lookup(struct table* t, data_t data, data_t* metadata, int (*apply_fun)(data_t*
    metadata));
```

rity [7] applications, including DoS mitigation for HTTP [8] and SIP [9].

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], 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 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 limit is reached, displacing entries between blocks until a free entry is found.

III. Hashtray

Hashtray consists of an API, shown in Listing 1, 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 parameterisable: 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 randomised structure, its behaviour 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.

How to use Hashtray. The evaluation of Hashtray in the following sections also serves as a demonstration of how to use it. Using Hashtray involves using its API (Listing 1) and specifying, at compile time, what kind of table the API is interfaced with. Demo code provided with Hashtray shows how to do this, and how to use different kinds of tables in the same application through the same API. By using the same workload to run evaluations using different kinds of tables (data structures) from the same application, one can obtain an idea of which kinds of data structure is best suited for that given workload. Note that Hashtray does not perform the logic to understand if a host is malicious; it only serves to remember that decision once it has been made. There is a rich literature on deciding whether a host is malicious but this is out of scope in this paper.

IV. Network model

To evaluate DoS effects in large-scale networks we wrote a simulator to model the effects of using different tables 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 table 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 table being used is randomised then its behaviour can vary across runs. Moreover, since the table is finite, it may lose information if it is filled. 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 PGC drops sharply because we are more likely to receive connections from a bad host. There might be many bad hosts on the network, but they might not be attacking us; this is modelled by a high PGH and high PGC. The model first rolls the die to decide whether to attempt to have a good or bad host establish a connection to the server. Then it rolls the die to pick a host at random, and if the picked host has the sought quality (good/bad) then the connection is made, otherwise we repeat the decision whether to have a good or bad host connect.

V. Application model

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

```
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.

Integration with Apache. Starting with the pipelined Apache Worker MPM described in earlier work [12], 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.

Logic. When a connection arrives, we perform a lookup on an thread-local cache table (and on a application-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 classed 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. Our implementation accepts various parameters that influence its performance and tolerance to misbehaving clients, such as the sizing of queues used to buffer connections, and the number of simultaneous connections a client is allowed to make.

VI. Evaluation

We evaluate the DoS-mitigating effectiveness of our Cuckoo filter instance in Hashtray in two ways:

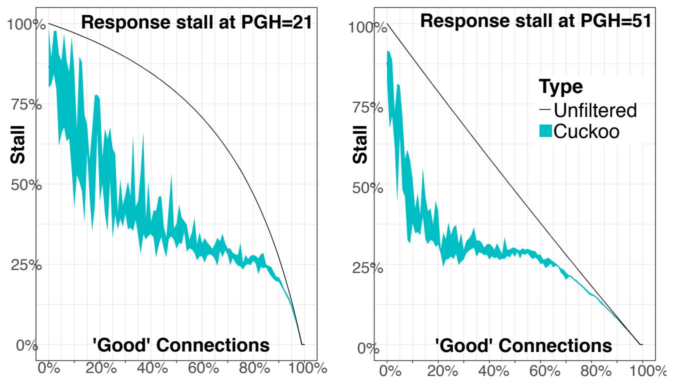


Figure 1: Simulated "stall" in a 10^6-host network, using a 10^2-size 2,4-table, when 21% and 51% of hosts on the network are "good hosts" (PGH). For each of these, we simulate the percentage of connections coming from "good hosts" (x-axis) to calculate the time we would spend stalling on connections from "bad hosts" (y-axis).

- **Stall simulation** 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** 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. 1. We run our simulator 5 times for each set of parameters. The randomness used by the Cuckoo filter results in a region of behaviour rather than a line. 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). When PGH=21 and there is a 50% chance of connections coming from good hosts (PGC=50) then because of how the model picks connections we are more likely to get more connections from bad hosts as a result: i.e., the system is under attack more of the time. There we can see that the paucity of good hosts making connections to our server (relative to bad hosts making connections) leads to greater stall (over 75%). This means that 75% of the time the server is suffering the ill-effects of stall. In contrast, whenever PGC=100, then a bad host can never be picked, thus the system spends 0% of its time stalling.

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.

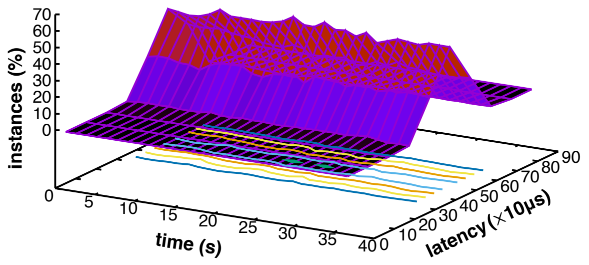
Our physical setup consists of eight 8-core Intel Xeon E5-2630L 1.80GHz CPU, 64GB RAM servers running Ubuntu

Linux 14.04 and interconnected via 10GbE links. 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.

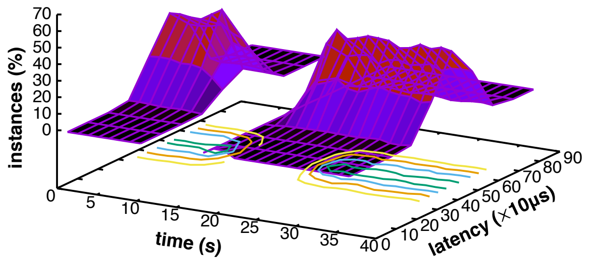
Having only 7 machines does not put much pressure on the applications being evaluated (i.e., Apache) or the DoS mitigation. We need to increase *address diversity* to give the semblance of having a larger network. To this end we use DoSarray [13] to run 40 container instances on each machine, bridge 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, and visualised by DoSarray. An example plot is provided in Fig. 2a for a control experiment consisting of the unmodified Apache when not under attack, and in Fig. 2b showing the unmodifed Apache when attacked by two instances of SlowLoris [14]. We run the experiment for 120s. We notice that, on average, latency increased by 100μs even when not under attack. We ran experiments in which Apache was attacked using SlowLoris, Tor's Hammer, and GoldenEye, popular HTTP-level DoS attacks for which scripts can be found on the Internet.

The modified Apache was able to resist up to 4 simultaneous attackers, for any type of attack we tried. In comparison, unmodified Apache can be rendered unresponsive by a single attacker. This suggests that Hashtray-backed in-application classification can be a viable mitigation for DoS, but further research is needed to establish general tools and techniques that can work against more attackers, more types of attacks, and for more kinds of applications.



(a) Normal activity in Apache (Worker MPM)



(b) SlowLoris on Apache (Worker MPM)

Figure 2: The gap in graph of (2b) indicates that the attack succeeds, since Apache stops responding. "Instances" indicates the concentration of hosts who responded to a specific probe at a given latency. Latency is measured in tenths of a millisecond. Probes occur at 1 second intervals.

Acknowledgment

We thank the anonymous reviewers, Henri Maxime Demoulin and Markulf Kohlweiss for feedback and John Frommeyer for systems support. This material is based on work supported by the Defense Advanced Research Projects Agency (DARPA) under Contracts No. HR0011-17-C-0047 and HR0011-16-C-0056.

References

[1] M. Kutare, G. Eisenhauer, C. Wang, K. Schwan, V. Talwar, and M. Wolf, "Monalytics: Online Monitoring and Analytics for Managing Large Scale Data Centers," in *Proceedings of the 7th International Conference on Autonomic Computing*, ser. ICAC '10. New York, NY, USA: ACM, 2010, pp. 141–150.

[2] S. Suthaharan, "Big Data Classification: Problems and Challenges in Network Intrusion Prediction with Machine Learning," *SIGMETRICS Perform. Eval. Rev.*, vol. 41, no. 4, pp. 70–73, Apr. 2014.

[3] M. Handley and E. Rescorla, "Internet Denial-of-Service Considerations," Internet Requests for Comments, RFC Editor, RFC 4732, December 2006.

[4] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo Filter: Practically Better Than Bloom," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14, 2014, pp. 75–88.

[5] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[6] M. Lee, N. Duffield, and R. R. Kompella, "MAPLE: A Scalable Architecture for Maintaining Packet Latency Measurements," in *Proceedings of the 2012 Internet Measurement Conference*, ser. IMC '12. ACM, 2012, pp. 101–114.

[7] S. Geravand and M. Ahmadi, "Bloom filter applications in network security: A state-of-the-art survey," *Computer Networks*, vol. 57, no. 18, pp. 4047 – 4064, 2013.

[8] S. Kandula, D. Katabi, M. Jacob, and A. Berger, "Botz-4-sale: Surviving Organized DDoS Attacks That Mimic Flash Crowds," in *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, ser. NSDI'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 287–300.

[9] D. Geneiatakis, N. Vrakas, and C. Lambrinoudakis, "Utilizing bloom filters for detecting flooding attacks against sip based services," *Computers & Security*, vol. 28, no. 7, pp. 578 – 591, 2009.

[10] N. L. Scouarnec, "Cuckoo++ Hash Tables: High-performance Hash Tables for Networking Applications," in *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '18. New York, NY, USA: ACM, 2018, pp. 41–54.

[11] R. Pagh and F. F. Rodler, "Cuckoo Hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004.

[12] N. Sultana, A. Rao, Z. Jin, P. Pashakhanloo, H. Zhu, K. Zhong, and B. T. Loo, "Making Break-ups Less Painful: Source-level Support for Transforming Legacy Software into a Network of Tasks," in *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*, ser. FEAST '18. New York, NY, USA: ACM, 2018, pp. 14–19.

[13] N. Sultana, S. Bose, and B. T. Loo, "An extensible evaluation system for DoS research," in *11th International Conference on Communication Systems & Networks*, Jan 2019, p. In press.

[14] "Secure your Apache server from DDoS, Slowloris, and DNS Injection attacks," https://www.techrepublic.com/blog/smb-technologist/secure-your-apache-server-from-ddos-slowloris-and-dns-injection-attacks/.