Evaluation and Design of Cache Replacement Policies under Flooding Attacks

Martin Zadnik
Brno University of Technology, Czech Republic
izadnik@fit.vutbr.cz

Marco Canini
EPFL, Switzerland
marco.canini@epfl.ch

***Abstract*—A flow cache is a fundamental building block for flow-based traffic processing. Its efficiency is critical for the overall performance of a number of networked devices and systems. However, if not properly managed, the flow cache can be easily filled up and rendered ineffective by traffic patterns such as flooding attacks and scanning activities which, unfortunately, commonly occur in the Internet.**

In this paper, we show that popular cache replacement policies such as LRU cause the flow caches to evict the so called *heavy-hitter* flows during flooding attacks. To address this shortcoming, we build upon our recent work [1] and construct a replacement policy that is more resilient to floods and yet performs similarly to other policies under common network traffic conditions.

I. Introduction

A variety of network services and applications depend on the ability to perform flow-based network traffic processing, that is, processing packets based on some state information associated to the flows to which the packets belong. For example, this form of stateful traffic processing is necessary to implement proxy firewalls, TCP offload engines, intrusion detection systems, traffic shaping, NAT, and collect statistics.

The fundamental element that enables flow-based processing is the flow table that is used to record flow information. Modern switches, routers and middle-boxes store the flow table either entirely in high-speed memory able to support operations at line rate or, due to the high costs of fast memory, in a memory hierarchy. The hierarchy is composed of a small, high-speed memory that, ideally, is responsible for processing the largest share of the traffic, and of a lower-speed but large memory that is used for processing the remaining share. We term *flow cache* the flow table that resides in the high-speed memory.

Clearly, the flow cache has a significant role in overall system performance and, therefore, it is important that the flow cache is continuously managed to consider the current traffic behavior. Even more so when the traffic exhibits an abrupt and very large increase of non-legitimate flows which typically consist of a few packets per flow caused by some form of Distributed Denial of Service (DDos) such as TCP SYN flood or by scanning activities. These attacks and activities can flood the flow cache with state information about short-lived flows that is of no benefit for the system performance as the hit rate of the cache reduces to zero and the response time noticeably deteriorates [2]. We argue that under these situations the replacement policy of a flow cache should strive to maintain the state of heavy-hitter flows because these flows are typically in a small number but are responsible for a prevalent portion of traffic. For example, consider an OpenFlow switch that upon a flow cache miss asks a remote controller to install a flow cache entry [3]. In this case, losing a flow entry of a heavy-hitter could cause significant disruption especially during a flood due to long response delay of the overloaded controller.

In this paper, we analyze the behavior of several popular replacement policies such, as LRU and Segmented LRU (SLRU) [4], under flooding attacks. These replacement policies often work close to optimum under normal traffic conditions [5] and are often implemented in commercial products due to their simplicity [2]. But they often fail upon specific workloads [2]. We also analyze two replacement policies that address the limitations of LRU: Low Inter-reference Recency Set (LIRS) [2] and Single-Step SLRU (S^3-LRU) [6]. Finally, we build upon our recent work [1] to construct, using Genetic Algorithms (GA), a replacement policy that preserves heavy-hitter flows during flooding periods. The evolved policy is simple enough to be implemented in hardware yet its results are comparable with more sophisticated policies.

II. Motivation and Background

It is well-known that many kinds of DoS attacks and scanning activities constantly happen over the Internet. Some of these are responsible for an abrupt increase in the number of concurrent, short-lived flows. The consequences of these flows on the end-to-end performance is potentially disrupting, especially if they result in resource exhaustion at the victim end. For example, a TCP SYN flood where a multitude of attackers send single-packet flows can easily exhaust the OS resources of a victim computer and/or cause severe packet drops at the network devices close to the victim. Other application-level attacks can be more subtle in that they can create seemingly valid application requests (e.g., web page requests) that can exhaust OS and network resources. Finally, the effects of these flows can be detrimental for flow-based traffic processing at the networking equipment on the paths to the attacked victims. This is because a flood of short-lived flows can fill up the flow cache of these devices with irrelevant flow state information.

However, a flow cache is ruled by a *replacement policy* (RP) that is responsible for deciding which state should stay

978-1-4244-9538-2/11/$26.00 © 2011 IEEE

in the cache and which one should be evicted any time the cache is full. Therefore, if the replacement policy is capable of evicting the irrelevant flows, a network device may be able to provide its service for legitimate flows, at least with an increased robustness than what is achieved by using standard replacement policies.

For instance, the Least Recently Used (LRU) replacement policy is widely used in networked systems, in particular, all systems evicting flows upon an inactivity timeout must inherently implement LRU. However, LRU caches are susceptible to the eviction of frequently used items during a burst of new items. Many efforts have been made to address its inability to cope with access patterns with weak locality. For example, Segmented LRU (SLRU) [4] seeks to combine both locality and frequency to achieve better hit rates. Low Inter-reference Recency Set (LIRS) [2] addresses the problem of working with sequential or cyclic (loop-like) pattern. The main idea behind the LIRS algorithm is to utilize the information from recency and recent Inter-Reference Recency (IRR, the number of other distinct items accessed between two consecutive references to an item).

Our previous work [6] introduces a replacement policy, called Single Step SLRU (S^3-LRU), that aims at tracking heavy-hitter flows. It is a simple variation on SLRU: specifically, instead of moving a flow state to the front of the list when it is accessed, S^3-LRU advances the state only of a single step toward the front of the list. Lastly, our recent work in [1] demonstrates how to apply Genetic Algorithms (GA) to "evolve" a replacement policy tailored at tracking the heavy-hitters starting from recorded traffic traces.

III. Definitions and Setup

A. Heavy-hitters

Depending on the application, the definition of a flow changes adequately. One that is commonly used identifies a flow based on the 5-tuple composed of its IP addresses, port numbers and protocol. In our work, we consider a flow to be a unidirectional stream of packets sharing the same 5-tuple, but our approach can be easily generalized to allow the flow identifier to be a function of the header field values. We use a 60 s timeout to determine the end of a flow unless we observe the TCP connection tear down.

We define a *heavy-hitter* (heavy-hitters) as a flow that utilizes more than a certain percentage of a link bandwidth during its whole lifetime. In order to avoid bias from short-lived flows which overall do not carry significant amount of traffic, we require a heavy-hitter to exist for at least five seconds otherwise it is penalized. Therefore, we compute a flow's link utilization as $\frac{totalbytes}{max(5,lifetime)}$. Throughout this paper, we group flows into three reference categories based on their utilization: very large flows ($>$ 0.1% of the link capacity), large flows (between 0.1% and 0.01%), medium flows (between 0.01% and 0.001%). We then presents results for each category.

	v. large	large	medium	Total
Flows	0.23%	0.93%	9.43%	4.0M
Packets	31.97%	18.92%	20.71%	53.0M
Bytes	68.35%	17.13%	9.22%	33.5G

TABLE I: Mawi dataset. (1 hour, 155 Mbps link, avg/min/max active flows: 67.3K/56.5K/250.1K)

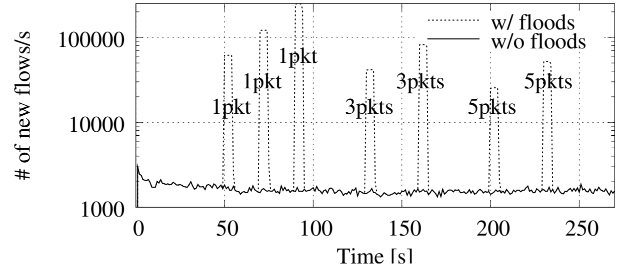

Fig. 1: Number of new flow/s in our dataset.

B. Dataset

We use a single trace of Internet backbone traffic: a 1-hour bidirectional trace from the Mawi archive collected at the 155 Mbps WIDE backbone link (samplepoint-F on March 20th 2008 at 14:00) [7]. Table I summarizes the working dimensions of our trace and shows a breakdown of the trace into the three flow categories. Throughout the rest of the paper we display only a short interval from this dataset to demonstrate the results closely.

On top of the traffic trace we inject traffic to simulate several periods of DDoS that flood a victim end host with a large number of new flows as shown in Fig. 1. Each period lasts for 5 s. First, we inject traffic consisting of single-packet flows in three periods with 60k, 120k and 240k flows/s, at time 49, 69 and 89 s, respectively. At 240k flows/s we reach the link capacity. Second, we simulate application-based DDoS attacks with flows of 3 packets (e.g., representing the TCP connection setup). The interval between subsequent packets of a flow is set to 20 ms. We generate two attacks at time 129 and 159 s with 40k and 80k flows/s, respectively, so as to not exceed the link capacity. Last, we inject attacks at time 199 and 229 s with 5-packet flows by generating 25k and 50k flows/s, respectively. We use a 20 ms interval between packets of the same flow.

C. Flow Cache Setup

We set the flow cache size to be 8K flow-states. The cache is divided into equally-sized buckets, each holding a list of 32 flow states. Each observed packet causes a lookup in one of the flow cache bucket based on an hash of the flow identifier. Each list is managed independently from others, although they share the same replacement policy. The last flow state of a list is the one to be evicted in case the list is full. Using LRU replacement policy, such a setup provides 95% hit rate on Mawi dataset, i.e., only 5% of lookups witness a cache miss. This is obtained without accounting the cache misses due to the first packet of each flow that are inevitable.

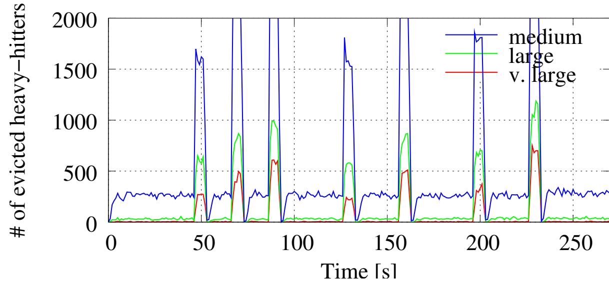

Fig. 2: Number of evicted heavy-hitters under LRU.

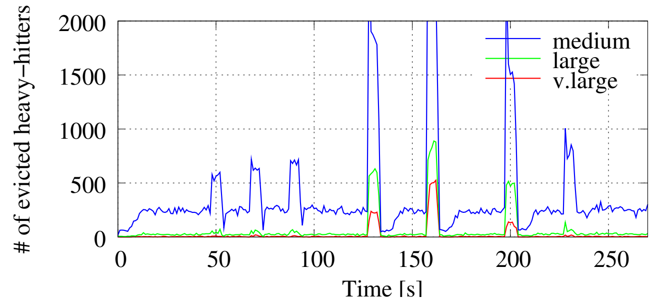

Fig. 3: Number of evicted heavy-hitters under S-LRU.

IV. Analysis

In this section, we present an analysis of the performance of several replacement policies in our dataset. We evaluate each replacement policy using the number of evicted heavy-hitter flows during the attack periods. The policies we consider are LRU, SLRU, S^3-LRU and LIRS.

We start with LRU. Fig. 2 shows that the LRU policy performs poorly: even under the weakest flood, LRU evicts many heavy-hitters in all three categories. Primarily, the cause of its poor performance is that the attack flows are inserted at the head of the list meaning they are given more importance than any flow already in the cache. During a flood, all flows whose packet rate is lower than the flow rate of the flood are evicted.

SLRU [4] extends LRU to take both recency and frequency into consideration. The list is divided in two segments: protected and probationary segment. The insert position of SLRU is at the head of the probationary segment. A flow only moves to the protected segment if it receives a lookup before being evicted from the probationary segment. After experimenting with several values, we set the insert position at item 21 in the list, making the size of the probationary segment about one third of the list. Flow states inside the protected segment are less exposed to floods whereas flow states in the probationary segment compete with the attack flows. Fig. 3 demonstrates that SLRU can cope with floods with single-packet flows. However, during a flood of multi-packet attack flows, SLRU evicts heavy-hitters in an amount comparable to LRU. This is due to the simple strategy for managing the protected segment. It is immediate to see that an attack flow with only two packets can easily be moved to the protected segment causing a heavy-hitter to be shifted back to the probationary segment.

Unlike SLRU, S^3-LRU does not order the items within each segment by their last access but at each lookup it advances the accessed flow of a single position towards the head of the list (protected segment) by swapping its position with that of the adjacent item. This makes it more likely that only heavy-hitters compete with each other in the protected segment. After experimenting with several values, we set the insert position at item 7 in the list, making the size of the probationary segment about three quarters of the list. Fig. 4 shows that overall the number of evicted heavy-hitters during attack periods is

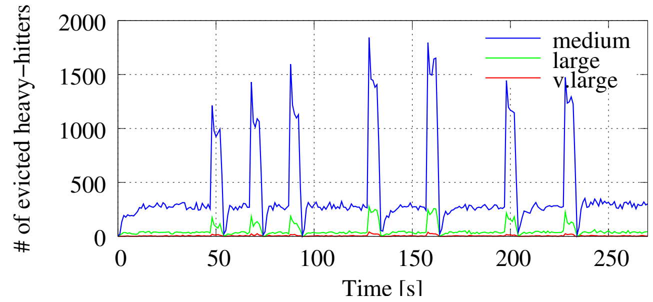

Fig. 4: Number of evicted heavy-hitters under S^3-LRU.

significantly lower than with the other policies. However, evicted large and medium heavy-hitters witness an increase.

Lastly, we analyze the LIRS [2] policy. This policy cannot utilize the structure of a flow cache described previously as it requires two stacks managed with LRU, one of the size of the flow cache and another smaller one with a variable size. The policy is executed upon all states in the flow cache. The stacks are used to compute recent Inter-reference Recency and to keep flow state in the memory as well as to keep the status of flows that are evicted from the memory. Based on IRR, the flows are divided into two sets, flows with low IRR (LIRS) and high IRR (HIRS). As suggested by the authors in [2], we allocate 99% of cache size for LIRS and 1% for HIRS. The simulation on the traffic trace shows that such a setup stabilizes the behavior of the cache to maintain heavy-hitters (see Fig. 5). Unfortunately, the number of evicted large- and medium-sized heavy-hitters is relatively large even during normal traffic conditions.

Therefore, we conduct an experiment with a different setup where we allocate only 90% of cache capacity to LIR items and the rest is available for HIR items. Such a setup allows to reduce the number of evicted heavy-hitters during normal periods but the flow cache is then more susceptible to floods, as depicted in Fig. 6. For instance, the flood at time 129 s causes the eviction of many heavy-hitters. In this case, the small intensity of the flood (40K flows/s) causes the packets of the flooding flows to be sufficiently close in terms of inter-reference recency. And so, they are deemed as worth caching by LIRS.

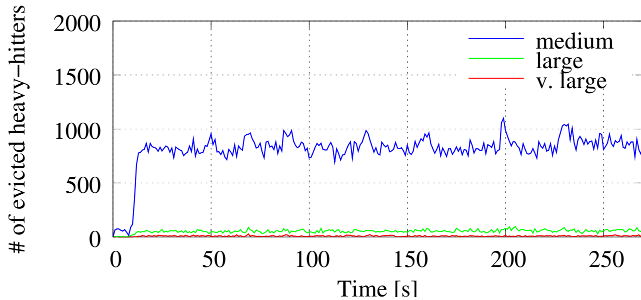

Fig. 5: Number of evicted heavy-hitters under LIRS (99% LIR items)

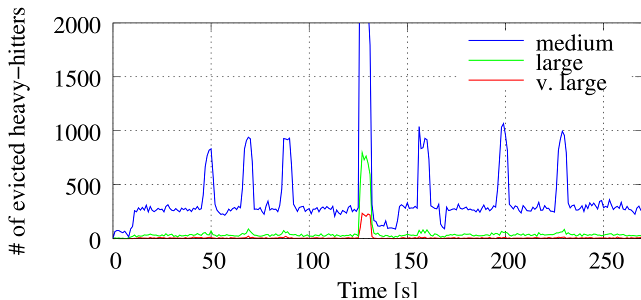

Fig. 6: Number of evicted heavy-hitters under LIRS (90% LIR items)

V. Design

Our goal is to design a replacement policy in the family of simple replacement policies. We define a simple replacement policy as an algorithm that works by ordering the list of flow states only. It requires no additional structures in comparison to others, more sophisticated algorithms such as LIRS.

Based on the analysis and especially on the results of S^3-LRU, several questions arise — Is it appropriate to advance a flow state receiving a hit just for a single step? Should the step be variable? When to trigger variation of the step? Where to insert new flow states in that case? To answer these questions, we decide to use Genetic Algorithms (GA) to explore the space of possible replacement policies to identify the most effective. Our approach leverages our previous experiences with the evolution of replacement policies [1] and focuses specifically on deriving a replacement policy that is more resilient to flooding attacks.

In this section, we provide a general definition of a replacement policy based purely on ordering the flow states in the list. This definition is subsequently adopted by GA to work upon.

A. *Replacement Policy Definition*

We regard the flow cache divided into equally-sized buckets of N flow states (or simply flows) F. Each bucket forms a list of flows ordered by the replacement policy. The role of a replacement policy (RP) is to reorder flows based on their access pattern. Each packet causes one cache access and one execution of the RP. If the current packet causes a cache miss (e.g., a new flow arrives) and the cache is full, the flow at the end of the list is evicted.

Formally, we can express a RP that is based solely on the access pattern as a pair $\langle s, U \rangle$ where s is a scalar representing the zero-based position for inserting new flow states and U is a vector $(u_1, u_2, \ldots, u_N)$ which defines how the flows are reordered. Specifically, when a flow F stored at position $pos_t(F)$ is accessed at time t, its new position is chosen as $pos_{t+1}(F) = u_{pos_t(F)}$, while all flows stored in between $pos_{t+1}(F)$ and $pos_t(F)$ see their position increased by one. For example, the LRU policy for a cache of size 4 is expressed with $LRU = \langle 0, (0, 0, 0, 0) \rangle$.

B. *Evolution of Replacement Policies*

The goal of GA is to find a RP that has the least number of evicted heavy-hitters or, using caching terminology, minimizes the miss rate for heavy-hitters. We use the number of heavy-hitters that witness a cache miss as a metric to capture the effectiveness of a RP—the objective is to reduce this number.

The evolution phase runs offline. We use a smaller traffic sample (5 minutes) from a previous day that was amended with just a single short flood with intensity 30k flows/s. If more floods are added, the GA would overfit the RP to floods while for common traffic pattern it would perform poorer than other policies.

The vector-based definition of a RP is a good fit to encode the candidate solution. It supports the standard genetic operators for mutation and crossover. Mutation modifies a particular value in the vector with given probability p_{mut} while crossover swaps parts of the vector between two solutions with probability p_{cross}.

We set the GA to evolve a population of six replacement policies (candidates), initially, generated at random. We use a relatively small population so the evolution process can converge quickly, potentially allowing the RP to be adapted to ongoing traffic. The evolution happens in cycles that progressively refine the solution. In each cycle, the candidates are evaluated by a fitness function. The fitness function is the sum of cache misses for the flows in the three reference groups weighted by the link utilization thresholds: 0.1% for the first group, 0.01% for the second and 0.001% for the third, respectively. This assigns higher importance to track larger heavy-hitters. Effectively, the fitness function simulates the cache behavior of a candidate RP. Evaluated candidates (offspring) replace the parent population If the parent population contains the best candidate of both population, the best candidate is preserved. Subsequently, tournament selection picks six candidates based on their fitness value (fitter RP have higher probability to be selected multiple times) to form a new population of stronger candidates. Crossover and mutation operators are applied to spread information and preserve variability among these candidates. Then, the new cycle begins with the evaluation of a new offspring population.

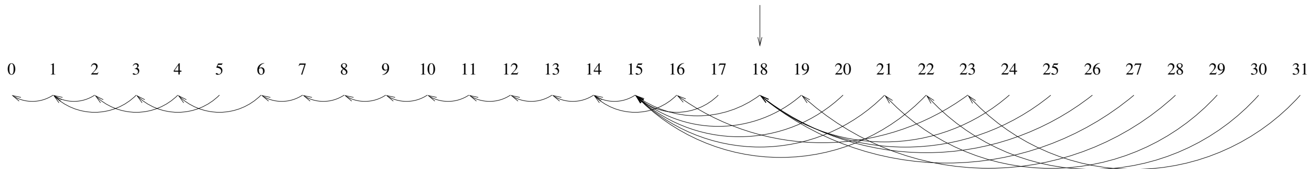

Fig. 7: An example of RP produced by GA using the Mawi dataset. The arrows represent where to move a flow when accessed. $RP = \langle 18, (0, 0, 1, 1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 14, 15, 15, 15, 15, 15, 15, 16, 18, 18, 18, 18, 19, 21, 22, 23)\rangle$.

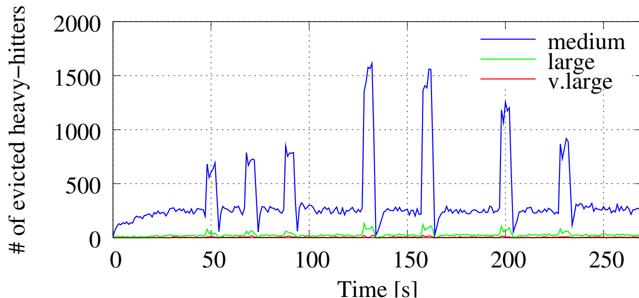

Fig. 8: Number of evicted heavy-hitters flows under GARP.

We split the run of GA to two consecutive phases each with a different setup of parameters which allows to improve the search time. We also apply heuristics to exclude bogus candidates prior to fitness function to exclude undesired solutions (e.g., those that do not utilize entire list due to unreachable positions). We refer reader to our previous work in [1] to learn about the precise GA setup. Fig. 7 presents an example of a GA-produced RP (GARP).

VI. Evaluation

In this section, we evaluate the genetically evolved replacement policy and compare its performance with the other replacement policies. The traffic trace as well as the flow cache setup is identical to the setup used during the analysis (Mawi traffic trace injected with floods and flow cache for 8192 flow states divided into buckets of size 32). We use the genetically evolved replacement policy (GARP) displayed on Fig. 7 to manage each bucket of the flow cache.

Fig. 8 shows that the impact of floods on the number of evicted heavy-hitters was further reduced due to GARP policy. Unlike SLRU, GARP implements more fine-grained structure to promote flows to the front of the list. This allows GARP to distinguish between multi-packet flooding flows and heavy-hitters. In comparison to S^3-LRU, GARP is quicker in promoting heavy-hitters to the front of the list and so achieves less evicted medium and large heavy-hitters during floods.

The theoretical limit that cannot be achieved by any real RP is an oracle replacement policy (ORC) based on a precomputed list of forthcoming heavy-hitters and their timeline. Therefore, it can always select the best victim available, preferably a flow not in the list or a heavy-hitter that is about to end soon. It also prioritizes heavy-hitters by the weight of each category.

f=40k flows/s, p=3, t=5 s			
RP	v.large	large	medium
LRU	243	581	1582
SLRU	39.0%	63.5%	104.9%
S^3-LRU	6.7%	25.0%	70.9%
LIRS90	38.5%	80.0%	124.9%
LIRS99	2.8%	6.7%	37.9%
GARP	3.3%	13.2%	62.0%
ORC	0.3%	0.9%	9.2%

f=25k flows/s, p=5, t=5 s			
RP	v.large	large	medium
LRU	750	1188	2729
SLRU	3.1%	13.7%	38.7%
S^3-LRU	3.9%	22.4%	56.7%
LIRS90	2.3%	8.5%	38.3%
LIRS99	2.5%	9.6%	42.3%
GARP	1.6%	8.2%	35.3%
ORC	0.2%	2.3%	11.0%

TABLE II: Reduction ratio of RP to LRU (with flow rate *f*, packet rate *p* and duration *t*, LIRS99 allocates 99% of cache capacity for LIR items whereas LIRS90 only 90%)

Table II quantifies the reduction of evicted heavy-hitters during various floods. The reduction is computed as a ratio of a maximum number of evicted heavy-hitters per second by the considered RP and the maximum number of evicted heavy-hitters by LRU (both maximum values are picked from within the interval of the flood). It is hard to compute the sum of all evicted heavy-hitters due to a certain flood as it is not clear what interval to consider. Each RP behaves differently and the effects of floods may be shifted to others in time. Nevertheless, we observe from Fig. 8 and Table II that GARP outperforms other simple replacement policies and performs comparably to the more sophisticated LIRS policy during attacks.

The presented policies slightly differ in their hit rate (not accounting for the misses due to the first packet of each flow) under normal traffic conditions in our setup. The best performing is SLRU with nearly 96% hit rate followed by LRU with 95%. GARP achieves a hit rate of 94% whereas LIRS99 and S^3-LRU achieve approximately 92%. The worsened hit rate of LIRS and S^3-LRU is due to focus on heavy-hitters – they tend to ignore the small flows which may be many in Internet traffic. On the other hand, during the flood at 129 s (that is 40k flows with 3 packets per flow), LRU and SLRU achieve a hit rate of around 77% for the legitimate traffic, S^3-LRU achieves 82%, GARP achieves 86% and the LIRS99 hit rate is still 92%.

VII. Conclusion

We evaluated several simple and one sophisticated cache replacement algorithms under flow flooding conditions which are common in the Internet. Further, we obtained a simple replacement policy optimized to keep the state of heavy-hitters in the flow cache during flood attacks. We envision that this simple mechanism may find its hardware and software implementation in various network devices that implement flow-based traffic processing such as OpenFlow switches, TCP offload engines and others.

Finally, we note that it may also be beneficial to cache low-rate flows carrying latency-sensitive traffic such as VoIP. We plan to consider this problem in our future work.

Acknowledgment: We thank the anonymous reviewers for their many helpful comments and suggestions. This work was partially supported by the BUT FIT grant FIT-10-S-1 and the research plan MSM0021630528.

References

[1] M. Zadnik and M. Canini, "Evolution of Cache Replacement Policies to Track Heavy-hitter Flows," in *Proceedings of the 12th International Conference on Passive and Active Network Measurement (PAM '11)*, 2011, pp. 21–31.

[2] S. Jiang and X. Zhang, "Making LRU Friendly to Weak Locality Workloads: A Novel Replacement Algorithm to Improve Buffer Cache Performance," *IEEE Trans. Comput.*, vol. 54, no. 8, pp. 939–952, 2005.

[3] M. Casado, M. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker, "Rethinking Enterprise Network Control," *IEEE/ACM Trans. Networking*, vol. 17, no. 4, pp. 1270 –1283, 2009.

[4] R. Karedla, J. S. Love, and B. G. Wherry, "Caching Strategies to Improve Disk System Performance," *Computer*, vol. 27, no. 3, pp. 38–46, 1994.

[5] C. Kim, M. Caesar, A. Gerber, and J. Rexford, "Revisiting Route Caching: The World Should Be Flat," in *Proceedings of the 10th International Conference on Passive and Active Network Measurement (PAM '09)*, 2009, pp. 3–12.

[6] M. Zadnik, M. Canini, A. Moore, D. Miller, and W. Li, "Tracking Elephant Flows in Internet Backbone Traffic with an FPGA-based Cache," in *19th International Conference on Field Programmable Logic and Applications (FPL '09)*, 2009, pp. 640–644.

[7] "Mawi working group traffic archive," http://mawi.wide.ad.jp/mawi.