Consolidated Review of

Scap: Stream-Oriented Network Traffic Capture and Analysis for High-Speed Networks

1. Strengths

The paper presents a well-designed and engineered system. The system embodies a well blended synthesize of various known techniques: for example migration of functionality into the kernel or the use of traditional memory to reduce the number of times data is copied. On a high level, the engineering within this paper essentially evolves packet capture techniques to account for evolution in processing (multi-core), and evolution in user requirements (application level versus packet). This evolution of basic monitoring tools will greatly benefit our community.

Significant performance improvements over existing tools

2. Weaknesses

Perhaps not so much novelty in terms of ideas (although exposing a higher-level API to monitoring apps is a nice systems insight).

The paper's greatest weakness is two fold: 1. The evaluation section and 2. The lack of novelty in the techniques applied. The evaluation section uses several unjustified parameters such as the inactivity timeout and the throughput of traffic being processed. The techniques presented in section 5: using kernel over user space, reducing the number of data transfers, and using multiple cores are all well known ways to speed up systems. Given, this it is not immediately clear what the technical contribution of the paper is.

Potentially an unfair comparison since Scap and Snort have different feature sets. If Scap were to support the same set of features, it is possible it would have to give up some performance.

Implementation tweaks mostly take advantage of new hardware features (NIC filters, multi-core) that older frameworks have not been updated to deal with. A fair comparison would be establishing that the old frameworks cannot fundamentally be updated to leverage new hardware features, or show that the time the authors spent in developing Scap from scratch is significantly lower than the time taken to update the old frameworks to leverage new hardware features.

Some details are straightforward and could be omitted. The main result is solid engineering, not a huge conceptual breakthrough.

3. Comments

This is a nice paper, with an impressive system that is carefully evaluated. Comparisons to other systems are nice, and the overall evaluation seems quite solid. The orchestration between NIC, kernel and user is a nice contribution. The paper has two drawbacks: sometimes it goes into more detail than is necessary. For example, Table 1 and the API is nice, but there's not much there that is surprising that isn't already in the discussion. (I did find heavy use of callbacks somewhat surprising, although it's a fine choice.) Another example with is the code in section 3.3.

I appreciate the fact that the authors have actually built a complete, useful system. I also like the idea of pushing stream reassembly into the kernel and exposing a higher-level API to the apps. Only one concern: is it worth designing for the "scarce resources" scenario, where apps cannot simply pull all the streams they are interested in? I like the cutoff and priority functions, and I see how they can make a difference when the monitoring platform is short on CPU cycles. But I am wondering whether, in practice, a network operator interested in high-speed, sophisticated monitoring would not simply buy more processing cores.

Other:

Why not use polling instead of interrupts to pull traffic into memory? I presume that Scap is targeted at platforms that are dedicated to network monitoring, so it seems reasonable to constantly poll for new traffic.

The "prioritized packet loss" feature is not really evaluated. I get the idea and the analysis, but I cannot put my finger on real applications that would benefit from it.

The evaluation does not consider Scap performance as a function of the number of TCP streams. This would be useful, since (I presume) a big chunk of Scap processing goes to TCP stream reassembly.

Is there adversarial traffic for Scap? I.e., if an attacker can send up to X bps and she wants to make stream reassembly as expensive as possible, what kind of traffic would she send?

It would be interesting to discuss the implications of a kernel level implementation especially since kernel changes relatively frequently when compared to the APIs exposed to the userland. The evaluation, section 6, is filled with many unjustified parameters. For example, it is not clear why the initial bandwidth limit or throughput cap for several of the initial experiments if 6GBit. Also, it is unclear why one of the experiments switches from using 6GB to using 4GB. Ideally, the experiments should be performed with traffic throughput of 10GB as this is the link capacity. Another example of an unjustified parameter is in section 6.1: the authors use a 10 second timeout which is contrary to the 60 second inactivity time out used in prior work.

Scap-based NIDS fundamentally can capture fewer classes of attacks than packet-filter-based NIDS. Consider classic TCP attacks like Ack-splitting, or Daytona attacks. In a packet-filter based system user-code can be written to detect the appropriate attack packets. In Scap-based NIDS, the Scap library hides the packets from user code. If the Scap stream reassembly code works like it should, Ack-splitting attacks would be invisible at the user-code level. Of course, the Scap library could hard-code detection of known attacks at/below the TCP level, but that would not help detect new attacks, and the Scap API would make it fundamentally impossible to write user-code to detect the same (unlike in a packet-based API). An argument can be made that TCP attacks are rare, and so the significantly improved performance of Scap outweighs the lack of TCP-level attack detection. That's a fair argument. But given that argument, the performance numbers become meaningless because it compares apples and oranges. An interesting question is how would Snort perform if Snort were updated to use NIC filters, and be better tuned for multi-core. If these fixes to Snort recover most or some of the performance difference, then there is a stronger argument to be made for sticking with the updated Snort and still operating on packets. The authors have not ruled out this possibility.

A few questions and suggestions:

- About details about callbacks: How are callbacks handled from kernel to user? Or is there a pre-allocated user thread for each kernel thread? How are multiple flows multiplexed across a single thread? How does flow affinity to threads work if a user thread is off doing arbitrary (long!) computation in a callback?
- Reading section 3 left me some questions that weren't answered until section 4:
    - In 3.2: are callbacks in kernel or user space?
    - Is chunk sharing in shared memory?
- In section 4.1, how are events triggered to the user? (Do user threads poll a socket, or something else?) In section 6.1, does replaying your trace while loading the next segment change traffic locality? Should you change the flow-ids each replay to prevent this? If you did, would that change your graphs?
- In your experiments, how long is each data point, or how many repetitions? How reproducible are the results if redone? (The graphs look smooth suggesting you don't have a lot of statistical variation, but it would be nice to get confirmation.)
- In figure 4: it looks like you stop at 6Gbps, which is exactly your saturation. Can you take figure 4a further?

4. Summary from PC Discussion

The paper was briefly discussed. The reviewers who gave low scores thought that the evaluation left room for improvement. However, even they agreed that the paper presented a nicely designed and useful system. In the end, this was considered a sufficient reason for accepting the paper.

5. Authors' Response

We thank the IMC reviewers for their insightful reviews and their valuable feedback for improving our paper. We identified four main points that the majority of the comments focus on: (i) experimental evaluation, (ii) attacks detected only by packet-based analysis, (iii) implementation details, and (iv) need for overload control and existence of adversarial traffic. We have tried to address these concerns in the final version of our paper:

1. Regarding the comments on evaluation, reviewers noted that we had not experimentally evaluated the prioritized packet loss with stream priorities and Scap with varying number of streams. To this end, we performed a new experiment comparing Scap with Snort and Libnids while varying the number of concurrent streams. The results in Section 6.4 and Figure 5 show that in contrast with existing systems, Scap scales very well with the number of concurrent streams. To evaluate prioritized packet loss, we ran a new experiment with the Scap-based pattern matching application by assigning a higher priority to streams with source or destination port 80. The results in Section 6.7 and Figure 9 show that although the application drops high percentages of packets in high traffic rates, the high-priority streams are not affected. Moreover, there were two questions about the 10 seconds inactivity timeout we used and the maximum rate of 6 Gbit/s. Although the large majority of TCP flows close explicitly with a TCP FIN or RST, we use an inactivity timeout to expire the small percentage of UDP flows or TCP flows that do not close normally. As we replay our trace at much higher rates than its captured rate, we believe that 10 seconds is a reasonable choice for the inactivity timeout. We have explained this choice at the end of Section 6.1. We acknowledge that replaying higher rates than 6 Gbit/s would be helpful. The reason we stopped at 6 Gbit/s is that it was the higher rate for which we could accurately replay real network traffic. We believe that sending real traffic is the best way to evaluate TCP stream reassembly implementations. Also, we found that existing single-threaded stream reassembly systems can handle up to 2.5 Gbit/s in our setup without any further processing, so 6 Gbit/s was enough to saturate them.
2. Another concern was the detection of TCP attacks like ACK splitting, which require packet-level processing. Scap already provides support for delivering the packets of each stream, if needed by the application. This is achieved by keeping the necessary metadata per each packet. To address this concern we have explained in more detail the packet delivery mechanism in Sections 3.2 and 5.7, noting that Scap can be used for packet-level processing and detection of TCP attacks as well. In order to evaluate the performance of Scap with packet delivery, we have added the case of pattern matching in the individual packets of each stream in Section 6.5. The results are presented in Figure 6 and discussed in Section 6.5.3, and they show that the performance of Scap remains the same with packet delivery.
3. Reviewers also asked for more implementation details regarding callback handling and event generation. Our system consists of one worker thread per each core that polls an event queue and executes a respective callback (registered by the application) for each event. The events are generated by a respective Scap kernel thread, running on the same core and processing the packets received at the respective hardware queue. We have added more detailed description of our system in Sections 2.4, 4, and 5. Two more questions were about the use of polling instead of interrupts and frequent kernel changes. Regarding interrupts versus polling, we used an existing NIC driver with interrupt coalescing. Scap can benefit from other existing NIC drivers that use polling as well. Regarding frequent kernel changes, we implemented the kernel part of Scap as a loadable kernel module, so it does not require any kernel patch and does not depend on kernel changes.
4. Two other concerns raised by the reviewers were about our choice to design for an overload scenario, and about possible adversarial traffic for Scap. Regarding the first concern, although a monitoring system can be provisioned to handle the worst case, e.g., by buying more cores, an attacker may still be able to overload this system, e.g., by sending adversarial traffic to exploit unknown algorithmic complexity attacks. Thus, we believe that overload control is necessary for the correct and secure operation of such monitoring systems that can be abused by attackers. We have discussed this issue in several parts of the paper. Adversarial traffic can be detected in Scap applications by the processing time of each stream, and can be handled by giving lower priority or discarding streams with adversarial traffic. We have explained this capability in Section 3.2. Moreover, in Section 6.4 we show that Scap can tolerate attacks trying to overwhelm the flow table up to the extent that there is available memory in the system.

We have also incorporated other suggestions from the reviews.