Real-time DDoS Attack Detection for Cisco IOS using NetFlow

Daniël van der Steeg, Rick Hofstede, Anna Sperotto and Aiko Pras
Design and Analysis of Communication Systems (DACS)
Centre for Telematics and Information Technology (CTIT)
University of Twente, Enschede, The Netherlands
E-mail: d.p.m.h.vandersteeg@student.utwente.nl, {r.j.hofstede, a.sperotto, a.pras}@utwente.nl

***Abstract*—Flow-based DDoS attack detection is typically performed by analysis applications that are installed on or close to a flow collector. Although this approach allows for easy deployment, it makes detection far from real-time and susceptible to DDoS attacks for the following reasons. First, the fact that the flow export process is timeout-based and that flow collectors typically provide data to analysis applications in chunks, can result in detection delays in the order of several minutes. Second, by the nature of flow export, attack traffic may be amplified by the flow export process if the original packets are small enough and are part of small flows.**

We have shown in a previous work how to perform DDoS attack detection on a flow exporter instead of a flow collector, i.e., close to the data source and in a real-time fashion, which however required access to a fully-extendible flow monitoring infrastructure. In this work, we investigate whether it is possible to operate the same detection system on a widely deployed networking platform: Cisco IOS. Since our ultimate goal is to identify besides the presence of an attack also attackers and targets, we rely on NetFlow. In this context, we present our DDoS attack detection prototype that has shown to generate a constant load on the underlying platform – even under attacks – underlining that DDoS attack detection can be performed on a Cisco Catalyst 6500 in production networks, if enough spare capacity is available.

I. Introduction

Distributed denial of service (DDoS) attacks are becoming a major technical and economical threat, overloading networks and servers with large amounts of network traffic. In early 2014, CloudFlare was hit by an amplified UDP flooding attack, reaching nearly 400 Gbps in bandwidth [1]. Although UDP flooding attacks typically aim at overloading targets with a vast number of bytes, there are also other attacks, such as TCP SYN flooding attacks, that result in a large number of connections. By the definition of a flow, "a set of packets passing an observation point in the network during a certain time interval, such that all packets belonging to a certain flow have a set of common properties" [2], this also results in a large number of flows. This makes it possible for flow-based technologies to detect such volume-based attacks [3]. Moreover, the use of flow export technologies, such as NetFlow and the recent IETF standardization effort IPFIX, are especially useful since they generate traffic aggregates. This approach reduces the amount of data to be analyzed significantly [4], as well as the necessary processing power for export and collection. Furthermore, these technologies are widely available on packet forwarding devices, making the flow data easily accessible and the technologies easy to deploy in existing networks.

Fig. 1. Typical flow monitoring architecture.

Flow-based intrusion detection in general – DDoS attack detection is no exception – is traditionally performed by *analysis applications* [5]–[7], as shown in Fig. 1. These applications operate on flow data exported by *flow exporters* and collected by *flow collectors*. Since the export of flow data is heavily based on timeouts and the collection is often designed to work in time intervals of several minutes, analysis applications are subject to various delays in the detection process [8]. Especially in the case of DDoS attack detection, where overload of network infrastructure can happen very quickly, this is something that must be avoided.

Recent work has shown that moving detection closer to the data source decreases detection delays significantly, from at least 165 seconds to 10 seconds [9]. The presented DDoS attack detection algorithm runs on a platform targeted at passive data export based on flows, namely INVEA-TECHs FlowMon platform. The goal of this paper is to investigate whether the detection algorithm presented in [9] can be deployed on a widely available networking platform. In this context, we target Cisco's IOS platform and in particular the Cisco Catalyst 6500, which is one of the most widely deployed packet forwarding devices [10]. We focus in particular on the operational experience of performing intrusion detection on packet forwarding devices in production networks.

The remainder of this paper is structured as follows. Section II introduces the terminology related to NetFlow and IPFIX that is used throughout this paper. An overview of the original detection algorithm from [9] is given in Section III. In Section IV, we explain how the required monitoring information, which serves as input to the detection algorithm, can be obtained from Cisco IOS. The implemented prototype is discussed Section V, which will be used for the validation presented in Section VI. In Section VII, we elaborate on further possibilities for DDoS attack detection and mitigation in Cisco IOS. Finally, we draw our conclusions in Section VIII.

II. Flow Metering & Export

In this section, we introduce the terminology related to flow metering and export that will be used throughout this paper. For a comprehensive overview of NetFlow and IPFIX, we refer to the tutorial in [4].

Flow metering and export are the two tasks performed by a *flow exporter* [4], as shown in Fig. 1. Packets in the network are aggregated into flows by the Metering Process. When a new flow is observed, an entry for this flow is created in the *flow cache*. This cache is a table that stores information on active flows in the network [4]. Aside from the key of the flow, i.e., the fields that identify a flow, some extra information is typically accounted, such as the number of packets and bytes in the flow. We refer to the event in which the cache is full and a flow cache entry cannot be created, which can happen during periods of high traffic if the flow cache is under-dimensioned, as a *flow learn failure* [11]. When a flow cache entry expires, for example when the flow has been active or idle for too long or because of resource constraints, a flow record is *exported*, i.e., it is inserted in a NetFlow or IPFIX message and sent to a collector for storage and pre-processing.

III. Detection Algorithm

We use an existing algorithm that has proven to satisfy the requirements of being lightweight, accurate and real-time in the context of DDoS attack detection, described in [9]. The algorithm[1] runs on a fixed time interval and measures the number of flow cache entry creations, as this metric was shown to be most usable of the four metrics presented in [9]. Based on this measurement, a forecast is made for the measurement value of the next interval. In case the number of flow cache entry creations is too high in comparison with the past measurement values, the measurement sample is considered anomalous. However, because Internet traffic shows diurnal patterns, such as strong increases and decreases in the number of flow cache entry creations during the start and end of a working day respectively, the algorithm also learns the normal behaviour of the network over a 24 hour period. The forecasted value is therefore defined as:

$$\hat{x}_{t+1} \quad = \quad b_t + s_t \ , \tag{1}$$

where $\hat{x}_{t+1}$ is the forecasted value for the next interval, b_t the base component, sometimes referred to as permanent component, which represents the trend of the Internet traffic, and s_t the seasonal component that represents diurnal patterns.

Several enhancements to this algorithm are discussed in [9]. First, in order to decrease memory usage, the values used for retaining seasonal patterns, s_t, are stored per hour and interpolated to estimate the value for a given time. Second, to prevent the algorithm from learning malicious traffic patterns, values such as s_t and b_t are discarded during an attack. Last, since traffic patterns during weekends usually differ from patterns during weekdays, a distinction is made between weekend and weekdays for season memory. This results in two training periods, one for weekdays and one for weekends.

[1] In [9] two algorithms are described. We use Algorithm 2, which showed the best results.

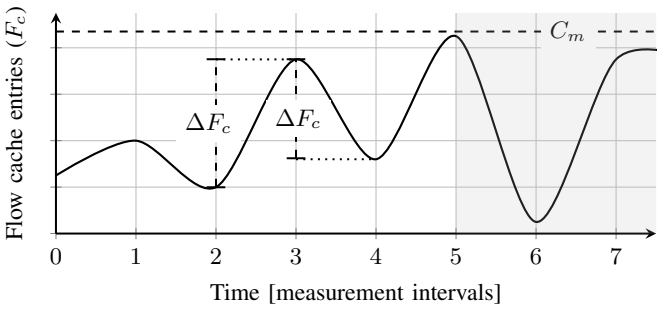

Fig. 2. Flow cache entry creations in Cisco IOS over time.

IV. Monitoring Information Available in IOS

The detection algorithm considered in this paper, which has been summarized in Section III, is heavily based on a single metric, namely the number of flow cache entry creations per time interval. This metric is easily accessible on the flow monitoring platform used in the original work of the prototype ([9]), INVEA-TECH's FlowMon. Since that platform has been designed with extendibility in mind, this information is directly available from the platform's API. However, the amount of information available in IOS strongly depends on the path the packet or flow has taken within the router or switch. More precisely, packets are switched either in hardware or in software, although most packets are hardware-switched. On the campus network of the University of Twente (UT), for example, 99.6% of the traffic is hardware-switched [11]. Situations that trigger a packet to be switched in software are fragmented packets, packets destined to the forwarding device itself, and packets that require ARP resolution [12], for example. For flows processed in hardware, information on the number of flow cache entry creations is not directly available. To approximate this metric, we use the following information available from the flow metering and exporting process:

- Number of *flow cache entries* (F_c).
- Number of *exported software-switched flow records* (F_e).
- Number of *flow learn failures* (F_f). This metric is expressed in terms of packets, rather than flows.

The number of flow cache entry creations since the last measurement can be approximated using the following definition:

$$F = \Delta F_c + \Delta F_e + \frac{\Delta F_f}{c_f} \tag{2}$$

When flow cache entries are exported, F_c will decrease which will cause the approximation to be less accurate if the measurement intervals are too long. For example, in Fig. 2, if the measurement were to cover two intervals, from $t = 2$ to $t = 4$, ΔF_c will not consider the peak at $t = 3$. By polling F_c more frequently, we can observe the changes more accurately, such that we observe the positive ΔF_c at $t = 3$ and the negative ΔF_c at $t = 4$, which is caused by exports. Then, if ΔF_c is

negative, we use an estimation of previous ΔF_c values instead. When the flow cache is nearing its capacity limit, the exporter issues an emergency expiration [4]. In Fig. 2 this is depicted in the shaded area. As F_c reaches C_m, the flow cache capacity, most flow cache entries are expired. If a measurement is made between $t = 6$ and $t = 7$, the algorithm may detect this as an attack for one measurement interval, due to the vast increase in the numer of cache entries compared to $t = 6$. To counteract this, the implementation waits for the next measurement if it suspects an attack, to validate whether it is an actual attack. This does however increase the detection delay.

Since the number of entries in the flow cache (F_c) only regards hardware-switched flows, we also add the number of exported software-switched flows (F_e), which can be obtained directly from IOS. Finally, adding F_f allows for regarding flows that should have been created but were not, which is especially the case during high-intensity DDoS attacks, for example. To compensate for the fact that F_f is expressed in packets while the other metrics are expressed in flows, we divide F_f by the average number of packets per flow, represented by c_f in Equation 2.

V. Implementation

The Embedded Event Manager (EEM) – part of Cisco's IOS that handles real-time network event detection – allows for the definition of *policies*, which can be used to execute an applet or script when events are triggered. For example, emails can be sent to network administrators when round-trip times reach a certain limit, or when network route changes occur. Another event type is based on time. This event can, among others, be scheduled at fixed time intervals. In this work, we use two time-based policies, implemented as TCL scripts:[2]

- **Measurement policy** – Determines the first component for our approximation of the flow-based metric: the number of flow cache entries (F_c), as described in Section IV.
- **Detection policy** – Retrieves the remaining components: the number of exported software flows (F_e) and the number of flow learn failures (F_f). Also, it implements the actual DDoS attack detection algorithm.

To obtain all three components, which are all made available using the SNMP protocol, we use a feature of the EEM environment that provides access to local SNMP objects. The reason for splitting the measurement policy from the detection policy is that we require a higher resolution for the former to detect changes more accurately, as described in Section IV.

Policy invocations are memoryless, and since we want to share data – both between policy runs and between policies – a method for sharing data needs to be implemented. Due to the fact that the filesystem is flash-based, we generally want to avoid excessive write actions that will shorten the memory's lifespan. The EEM environment therefore offers a *Context library* for this purpose; it allows for saving TCL variables to memory instead of writing them to disk. Besides for keeping track of our data between policy runs, we also use this feature to exchange information between the two policies, as the result of the measurement policy is needed by the detection policy.

The two policies discussed before are executed by the EEM at their respective intervals, which have been selected based on the runtime of the respective policies. When the switch is however under heavy load, its higher CPU utilization will cause the policies to take longer to execute. To avoid the policies from skipping an execution when the runtime of the policy exceeds the length of the interval, the prototype utilizes a feature from the EEM that can set a maximum policy runtime. If this runtime is exceeded, the policy terminates forcibly and data is lost. In the case of the detection policy, the algorithm has to start again from the learning phase as all state data is lost. If the measurement policy terminates prematurely, the measured number of created flow cache entries will be lower, as it missed a measurement, which will slightly impact the accuracy of the algorithm. To prevent the detection policy from being killed, a margin has been added to the interval which allows it to run longer if necessary, but never longer than the interval at which it is executed. The average runtime of the detection policy is 2–3 seconds under normal conditions, and has shown to reach 7–8 seconds under stress. Therefore, the final interval chosen for the detection policy is 10 seconds. For the measurement policy, measurements have shown that 2 seconds provides an optimal balance between detailed measurements and loss of data due to termination.

VI. Validation

In this section, we describe the validation of this work, starting by identifying the requirements in Section VI-A. Next, we give a description of the validation setup, as well as specifics regarding the deployment, in Section VI-B. Finally, we discuss the results in Section VI-C.

A. Requirements

Three requirements were defined for the original detection algorithm: 1) it should be lightweight in terms of CPU and memory utilization, 2) the accuracy should be high enough to ascertain a low number of false positives/negatives, and 3) the detection delay should be reduced to roughly 10% of conventional intrusion detection approaches [9]. However, since the the Cisco Catalyst 6500 is a high-speed packet forwarding device that has not been designed for performing intrusion detection tasks, special care must be taken to not overload the device and possibly interrupt forwarding activities. We therefore relax the real-time requirement to detection within 30 seconds, while the CPU and memory utilization must be 10% or lower. Since the accuracy of the algorithm has already been validated in [9] and because it is invariant to the underlying implementation platform, we discuss the accuracy requirement only briefly.

B. Setup & Deployment

The implementation described in Section V has been developed on a *Cisco Catalyst 6500* with *Supervisor Engine 720*,

[2]The open-source TCL scripts can be retrieved from https://github.com/ut-dacs/ios-ddos-detect/

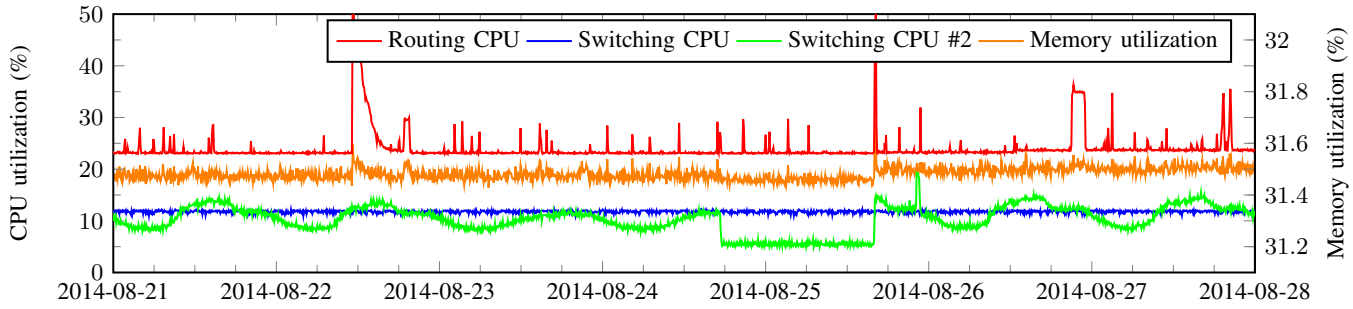

Fig. 3. Load of the Cisco Catalyst 6500 over time.

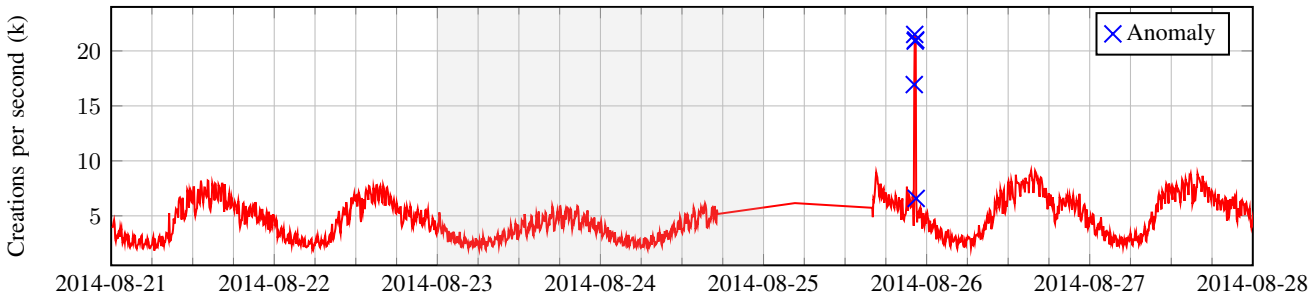

Fig. 4. Flow cache entry creations per second (averaged per 5 minutes), as processed by the detection algorithm over time.

running IOS 15.1(2)SY1. We have used this in combination with the *WS-X6708-10G-3C* line card for 10 Gbps Ethernet connectivity. The traffic used for validation is mirrored from the uplink of the UT campus network to the Dutch National Research and Education Network SURFnet and consists of both educational traffic, i.e., traffic generated by faculties and students, and traffic of campus residences. The link has a wire-speed of 10 Gbps with an average throughput of 1.8 Gbps during working hours. Furthermore, flow data is exported to a flow collector, such that attacks detected by the prototype can be validated manually.

The network traffic used in [9] differs from the network traffic used in this work, both from its nature (backbone traffic vs. campus traffic) and volume. It is therefore clear that we have to adjust the parameters of the detection algorithm to achieve similar accuracies as in [9]. As such, we have selected the optimal parameter values[3] for our observation point. For the parameter c_f, used for approximating the number of flow cache entry creations, as described in Section IV, we have measured $c_f = 59.8133$ packets per flow on average in our setup.

C. Results

The most important requirement to be validated in this work is that the implementation must be light-weight, such that the implementation does not interfere with the primary activities of the packet forwarding device, namely routing and switching. We measure the resource consumption both in terms of CPU and memory utilization. In Fig. 3, the CPU load of the device is shown together with the memory utilization, averaged over 150 seconds. Using SNMP, the load of the CPU is measured for three components, namely the *routing CPU*, which handles L3 traffic, and two *switching CPUs*, which process traffic at L2. Once a routing or switching decision has been made by the CPU, hardware handles subsequent packets if possible. Furthermore, the *routing CPU* also handles the network management (including the EEM), as most of this is done on L3. Consequently, our EEM policies also run on the *routing CPU*, and as such any load caused by our policies should account to the load of the *routing CPU*.

In Fig. 3, the policies are active during the entire measurement period, even in the period from August 24 18:00 to August 25 16:00 where the switch received no data. Because the CPU utilization of most individual processes is reported as 0–1% and only peaks are reported as more than 1%, we only consider the overal CPU usage. Consequently, the overhead of managing and executing only the policies cannot be observed. This overhead is caused by processes such as the *Chunk Manager*, which handles memory allocation, *EEM Server*, which manages all EEM policies and applets, and *SNMP ENGINE*, which handles all SNMP requests. Because the overhead of operating our policies is caused by multiple processes, which also run when our implemented policies are disabled, we have measured the difference in CPU and memory utilization between operation with and without our policies. To measure this, the switch has been rebooted to clear all memory and CPU utilization. During the measurements, we have observed a load on the *routing CPU* of 4%, combined

[3]The parameters used in this work are: $c_{threshold} = 4.0$, $M_{min} = 7000$, $c_{cusum} = 6.0$, $\alpha = \frac{2}{N+1}$, where $N = 540$, and $\gamma = 0.4$.

with a memory utilization of 31.3%. After enabling our policies we have observed an increase of 20% in CPU utilization, and an increase of 0.2% in memory utilization. This accounts for the average constant load added by our implementation.

During the period in which our detection algorithm was deployed, one attack passed our validation network on August 25. The attack lasted around 20 minutes and consisted of DNS reflection traffic and TCP traffic. During this attack, we only observe a minor increase of the load of the *switching CPU*, caused by the increased number of packets to be switched, and no increase in load for the *routing CPU*. As such, we conclude that the CPU load caused by our implementation during attacks does not peak and instead only consists of the constant load. The peaks in the load of the *routing CPU*, visible in Fig. 3, are likely the effect of other routing or management processes on the Catalyst 6500, as such processes are handled by the *routing CPU*. In terms of memory utilization, we clearly observe a stable pattern in Fig. 3. We do not observe any increase in memory utilization during the attacks, which makes us to conclude that the memory utilization does not create significant peaks.

Considering the above measurements, we conclude that the memory utilization does satisfy the requirement of using 10% of memory or less. However, the 20% CPU load caused by our implementation does not satisfy the requirement of 10% CPU utilization or less. As the Catalyst 6500 is a packet switching device and not meant to perform network attack detection, such other activities should not interfere with its main purpose of operation. As a load of 20% is probable to cause interference with the routing and switching tasks, we conclude that our implementation does not satisfy the requirement to be light-weight. The difference between the measured constant load and the lack of peaks in Fig. 3 can be explained by the fact that the amount of traffic does not change the number of computations performed by the policies, as only the calculated values are different. Furthermore, the short and frequent execution of the policies will be averaged out to a constant added CPU load. Especially the short intervals in which the measurement policy is executed (i.e., 2 seconds), increases the load. However, increasing this interval would decrease the measurement resolution, as described in Section IV.

The second requirement is the detection delay. This requirement, like the accuracy, has already been validated for the prototype in [9]. Our implementation uses an interval of 10 seconds between invocations of the algorithm, instead of 5 seconds as in the original work, due to the runtime of the algorithm, as described in Section V. This results in detection delays of multiples of 10 seconds, with a minimum of 10 seconds. The attack visible in Fig. 4 was detected within the third interval, resulting in a detection delay of 30 seconds.

The final requirement considered in this work is the accuracy of the DDoS attack detection. In Fig. 4, the number of flow cache entry creations per measurement interval is shown, averaged over 5 minute intervals. Weekends are shaded in light-gray. Diurnal patterns are clearly distinguishable and due to the nature of the traffic, we can also observe the difference between weekdays and weekends. The anomalous period around August 25 is caused by a lack of data as the switch did not receive any traffic during this period. The attack on August 25 is cleary distinguishable in Fig. 4. It resulted in around 200% more flow records than predicted by the algorithm, and lasted for roughly 20 minutes. Multiple detection marks are shown, as the attack spanned multiple 5 minute intervals.

VII. Discussion

The prototype presented in Section V retrieves information from the underlying platform using SNMP. We know that retrieving information using SNMP could be performed by any other system, even a Raspberry Pi, maximizing the available processing power of the forwarding device for routing and switching. However, since our ultimate goal is to perform attack mitigation that requires information on attackers and targets, we deliberately perform detection on the forwarding device itself (where NetFlow is available), which allows for rapid deployment in production environments at no additional cost.

Although the detection of attacks is a crucial first step, it merely serves the ultimate goal: attack mitigation. In [9], not only attack detection is discussed, but also mitigation. When the detection algorithm is run and a measurement sample is considered anomalous, mitigation is started by counting the number of exported flow records per source IP address; as soon as more than 200 flow records with three packets or less have been exported per second for a particular source IP address, the source IP address is blacklisted. Blacklisted IP addresses are added to a firewall to block traffic from the attacker. Furthermore, to prevent flow collectors from overloading, flow records with these IP addresses are not sent to the collector. When the algorithm detects the end of the attack, the created rules are removed from the firewall.

The information used to identify attackers in [9] is not available in IOS; only the total number of exported cache entries is available. An alternative approach for identifying attackers is to analyze the contents of the flow cache. However, the IP addresses of attackers will be overrepresented in the cache during a DDoS attack, since attackers generate large amounts of traffic, resulting in a large number of flow cache entries. However, the time needed to retrieve and process the entire flow cache under load – which consists of at least 128k entries, depending on the used hardware – can take up to tens of seconds, making timely mitigation hardly possible.

A different approach to implement mitigation is the use of an IOS feature that keeps track of the top $x \in (0, 200)$ flows featuring the highest volume, either in terms of packets or bytes, referred to as *NetFlow Top Talkers*. This feature cannot show the top talkers by the number of flows produced by a host, which would be very high for sources of DDoS attacks. Furthermore, it is likely that legitimate users will be in the top talkers list, as they can generate just as many packets and bytes. We therefore conclude that it is hard to identify the attackers and set aside mitigation in this work.

VIII. Conclusions

The goal of this research was to investigate the use of high-end packet forwarding devices for detecting, and ultimately mitigating, DDoS attacks in real-time. And yes, it is possible to detect DDoS attacks, which has been proven by the deployment of our prototype on a Cisco Catalyst 6500. Our results show that detection of flooding attacks is possible within tens of seconds, making real-time detection on a widely available switching platform possible. However, our prototype has also shown to cause a CPU load of 20%, which may cause interference with the routing and switching processes. According to various network operators we have stayed in touch with during this work, if the capacity of the packet forwarding device is available, it should be possible to run our DDoS attack detection in production environments. While it is possible to deploy our implementation with only 20–30% CPU capacity available, for example, it would require to be run with a lower priority, to not interfere with the routing and switching processes. As this may cause instability to our prototype, it is advised to have at least 40% CPU capacity available.

Several requirements were identified beforehand, the first being a small footprint of the implemented detection algorithm. Validation results have shown that there is no visible increase in CPU and memory utilization during attacks. However, when monitoring the overall increase in CPU and memory utilization, an increase of 20% CPU and 0.2% memory utilization can be observed when running the prototype. While the memory utilization satisfies the requirement of using 10% or less of the available resources, the CPU utilisation does not satisfy this requirement. Second, validation of our prototype in the UT campus network has shown that detection delays of 30 seconds are feasible for high intensity attacks, satisfying the requirement of real-time detection (within 30 seconds). This corresponds to three times our measurement interval of 10 seconds. Smaller measurement intervals may decrease detection delays, but will make it more likely that our detection runs overtime and is killed by a management process. The last requirement for our implementation is detection accuracy. Our validation results show that the number of false positives is low, while the detection rate is high, because of which we conclude that our prototype is accurate.

Mitigation is the next step, after detection. Our investigation has shown that while it is possible to obtain enough information to identify possible attackers, the command used to obtain this information can take tens of seconds when the switch is under heavy load, which occurs during flooding attacks. We therefore conclude that real-time mitigation is not possible on the hardware used in this work.

Future work includes investigating alternative implementations on different hardware. The successor of the *Supervisor Engine 720*, the *Supervisor Engine 2T*, contains more powerful hardware and provides additional functionality. This more powerful hardware is likely to influence the load caused by our implementation in a positive way, and potentially even allows for real-time mitigation. Furthermore, a brief investigation has shown that the *Supervisor Engine 2T* has the option of using events upon flow cache entry creations. This could replace our approximation of the number of flow cache entry creations, as described in Section IV, and make it more accurate and possibly faster.

Acknowledgments

Special thanks go to Vosko Networking B.V. and Cisco Systems International B.V. Amsterdam for providing the network devices used in this work, and Jeroen van Ingen Schenau, Roel Hoek, Niels Mejan and Jan Markslag (University of Twente) for their help in setting up the measurement infrastructure. This work was partly funded by FLAMINGO, a Network of Excellence project (ICT-318488), and SALUS, a STREP project (ICT-313296), both supported by the European Commission under its Seventh Framework Programme.

References

[1] CloudFlare, Inc., "Technical Details Behind a 400Gbps NTP Amplification DDoS Attack," 2014, accessed on 21 January 2015. [Online]. Available: http://blog.cloudflare.com/technical-details-behind-a-400gbps-ntp-amplification-ddos-attack

[2] B. Claise, B. Trammell, and P. Aitken, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information," RFC 7011 (Internet Standard), Internet Engineering Task Force, September 2013. [Online]. Available: http://www.ietf.org/rfc/rfc7011.txt

[3] A. Sperotto, G. Schaffrath, R. Sadre, C. Morariu, A. Pras, and B. Stiller, "An overview of IP flow-based intrusion detection," *IEEE Communications Surveys & Tutorials*, vol. 12, no. 3, pp. 343–356, 2010.

[4] R. Hofstede, P. Celeda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, and A. Pras, "Flow Monitoring Explained: From Packet Capture to Data Analysis with Netflow and IPFIX," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 2037–2064, 2014.

[5] A. A. Galtsev and A. M. Sukhov, "Network attack detection at flow level," in *Proceedings of the 11th international conference and 4th International Conference on Smart Spaces and Next Generation Wired/Wireless Networking*, 2011, pp. 326–334.

[6] H. A. Nguyen, T. Tam Van Nguyen, D. I. Kim, and D. Choi, "Network Traffic Anomalies Detection and Identification with Flow Monitoring," in *5th IFIP International Conference on Wireless and Optical Communications Networks, WOCN'08*, 2008, pp. 1–5.

[7] N. Muraleedharan, A. Parmar, and M. Kumar, "A Flow based Anomaly Detection System using Chi-square Technique," in *Proceedings of IEEE 2nd International Advance Computing Conference, IACC'10*, 2010, pp. 285–289.

[8] R. Hofstede and A. Pras, "Real-Time and Resilient Intrusion Detection: A Flow-Based Approach," in *Proceedings of the 6th IFIP WG 6.6 International Conference on Autonomous Infrastructure, Management, and Security, AIMS'12, Ph.D. Workshop*, ser. Lecture Notes in Computer Science, vol. 7279. Springer Berlin Heidelberg, 2012, pp. 109–112.

[9] R. Hofstede, V. Bartoš, A. Sperotto, and A. Pras, "Towards Real-Time Intrusion Detection for NetFlow/IPFIX," in *Proceedings of the 9th International Conference on Network and Service Management, CNSM'13*, 2013, pp. 227–234.

[10] J. Follett, "Cisco: Catalyst 6500 The Most Successful Switch Ever," 2006, accessed on 21 January 2015. [Online]. Available: http://www.crn.com/news/networking/189500982/cisco-catalyst-6500-the-most-successful-switch-ever.htm

[11] R. Hofstede, I. Drago, A. Sperotto, R. Sadre, and A. Pras, "Measurement Artifacts in NetFlow Data," in *Proceedings of the 14th International Conference on Passive and Active Measurement, PAM'13*, ser. Lecture Notes in Computer Science, vol. 7799. Springer Berlin Heidelberg, 2013, pp. 1–10.

[12] Cisco Systems, Inc., "Catalyst 6500/6000 Switch High CPU Utilization," 2012, accessed on 21 January 2015. [Online]. Available: http://www.cisco.com/c/en/us/support/docs/switches/catalyst-6500-series-switches/63992-6k-high-cpu.html