Tempesta: a Framework for HTTP DDoS Attacks Mitigation

Alexander Krizhanovsky
NatSys Lab.
ak@natsys-lab.com

Abstract

Modern application layer HTTP DDoS attacks employ complex techniques that make them difficult to detect. So modern DDoS defense approaches are also based on sophisticated methods to classify HTTP requests. Usually, these methods require exhaustive data on user activity from different layers of protocol stack, mainly from network, transport and application layers.

Moreover, a typical DDoS attack puts a huge load onto a target Web cluster. That demands a DDoS mitigation solution that delivers higher performance and is able to significantly reduce the load on defended back-end systems.

In this article we present Tempesta, a hybrid solution that combines a caching HTTP server and a firewall in one. It accelerates Web applications much efficiently than traditional Web accelerators and provides a high performance framework that offers easy access to data from all protocol layers. That facilitates the development and use of sophisticated DDoS classification and blocking algorithms.

1 Introduction

Modern DDoS attacks elude detection by employing techniques like making slow requests [17], sending random HTTP headers and URL values [16], requesting most expensive resources on the victim site at a low rate (e.g., heavy search in a database), putting periodic loads instead of a persistent load [18], as well as by using many other techniques. The most advanced attacks mimic flash crowds by sending different requests at a normal rate from hundreds of thousands of machines, and slowly activating their bots. Very sophisticated classification algorithms that analyze traffic in network, transport and application layers are required to properly filter bots out. Development of these classifiers is a challenging task, but it can be greatly simplified by the use of a framework that provides convenient access to all layers of network traffic. Neither traditional firewalls nor caching Web servers provide sufficient data for accurate traffic analyzing: Web servers have access to application layer only while firewalls can accurately operate on network layer only.

In this paper we address the problem of providing all necessary hooks for classification and filtering modules so that they can easily get access to all necessary information to detect malicious network clients and block them by using multi-layer filters (e.g., by particular IP address and some HTTP header value at the same time). At the same time, the system must be an active participant in TCP sessions. That is required for proper SSL termination, content caching, and accurate TCP stream assembling.

Proxy servers are active participants in TCP sessions which makes them more accurate in data assembly, but at the cost of a lower performance. At the same time, a DDoS attack puts a heavy load on the defense system itself, so the performance of a DDoS mitigation system is another big challenge. Moreover, a classification algorithm usually needs to analyze at least several packets from a client to understand whether it's a DDoS bot or not, so harmful traffic may pass through the system at least at the beginning of an attack. Also, the load caused by a DDoS attack can have sawtooth-like characteristics which may cause periodic passing of malicious traffic to a protected back-end sys-

tem. Hence, a defense system must be very powerful and able to cope with huge traffic spikes while its classifiers analyze network traffic. Robustness of Web acceleration under a huge load caused by a DDoS attack is the other problem addressed by the paper.

Moving HTTP server to the kernel space of an operating system is a very efficient way of improving its performance, but it was considered a bad practice [2]. However, we argue that there is a niche for specific type of applications that lies between firewalls that traditionally work in kernel space, and Web-accelerators that traditionally are user space applications. The niche is not limited to DDoS mitigation solutions, but also includes Web Application Firewalls [21].

This paper presents Tempesta, a high performance Linux kernel framework for classification of HTTP requests, efficient blocking of multiple IP addresses, and robust Web site acceleration in normal and DDoS conditions. We also introduce Synchronous Socket library for Linux kernel that offers better and more stable performance than traditional Berkeley Sockets, or even kernel sockets. Tempesta uses Synchronous Sockets in a very similar fashion as common user space servers use Berkeley Sockets to send and receive data over TCP/IP.

2 Motivation and Related Work

Many researchers proposed efficient methods for application layer DDoS attack detection [1] [3] [4] [12] [20]. However, such classifying systems need access to all network layers. Thus, to build a powerful DDoS prevention system that can analyze network traffic and block DDoS attacks using various heuristics, an appropriate platform is required. It needs to provide access to all protocol layers and have the ability to modify traffic in all layers.

Intrusion detection systems like Snort [25], based on Deep Packet Inspection (DPI) [37] [38], analyze traffic in all layers of protocol stack. However, the system does not participate in TCP connection handling such as connection establishing or termination, flow control, etc. Without that it doesn't have complete information about TCP control block (TCB) on each side of a connection. Thus it is prone to inaccurate TCP/IP packets assembly, especially in a hostile environment [24] [23]. On the opposite side are proxy servers. They are active participants in TCP sessions which makes them more accurate in data assembly.

Apache mod_evasive [11], Nginx HTTP request rate limit [9] and Nginx simultaneous connections limit [10] modules provide simple DDoS protection as part of a caching HTTP server. DDoS-Shield [3] is also integrated into Apache HTTP server, but implements a more advanced technique for classification of HTTP requests and sessions. In all these solutions the server must accept a client connection, then read and parse an HTTP request. Only then it might be able to determine that the client exceeded one of the limits, and respond with an error code. However HTTP servers are designed to provide application service rather than filter intensive traffic like firewalls. These solutions have very limited performance capabilities and are not suitable to filter DDoS attacks.

Sendfile(2) and *splice(2)* system calls make it possible for HTTP accelerators to avoid copying on data sending. However there is still no reliable zero copy mechanism for network input. Moreover, DDoS attacks in most cases are characterized by huge amount of small HTTP requests making zero-copy, which requires two system calls, quite inefficient.

All modern user space HTTP servers use Berkeley Socket API that has two performance drawbacks in context of an application layer DDoS attack. The first one is an overly detailed interface. For example, if we want to limit client connections at certain rate, then we need to *accept(2)* a connection, call *getpeername(2)* to get IP address of the client, look it up in our filter data structure, and finally *close(2)* the connection. I.e. there are 6 context switches to block one connection from one bot. Brecht et al. [5] demonstrated that careful optimization of strategy for accepting new connections leads to significant web server performance improvements. However, as we will see in section 5.1, such optimizations don't solve the issue of context switches completely. The second drawback is that reading from socket in user space is an operation that is asynchronous to arrival of a TCP segment, which aggravates the problem of context switches. We will discuss this case in the following section 3.

RouteBricks [32] (based on Click [39]) and NetSlices [33] use system optimizations that are common these days. Using a NUMA system with the number of cores that matches the number of net-

work adapter queues. Setting the IRQ affinity to pin each network adapter queue to a separate CPU core. Switching on GRO and GSO that are common in modern adapters and supported by Linux kernel. All of these are used in Tempesta as well.

Sandstorm [34], MegaPipe [35] and mTCP [36] replace Berkeley Socket API by their own optimized APIs. Sandstorm and mTCP implement tiny user-space TCP/IP stacks (without firewall, packet scheduling and several other features) that are faster than standard Linux TCP/IP stack, but still suffer from copying. MegaPipe batches system calls and replaces heavy Linux sockets with their own lighter implementation. These optimizations make user space TCP servers capable of handling much higher traffic volumes. However, user space solutions are still slow due to context switches, copying and uncontrolled preemption ([40] shows that RCU and read-write locks are much faster if preemption is disabled).

To address the problems of copying and context switches, a few in-kernel Web servers were developed: OpenKeta [8] for FreeBSD and TUX [6] and kHTTPd [7] for Linux. Joubert et al. proposed AFPA [2], which differs from other kernel based servers in that it moves a bunch of operations (like a firewall) to deferred interrupt context rather than processes requests in a separate kernel thread.

Our approach is similar to AFPA in that it also embeds an HTTP server into the operating system TCP/IP stack. However, AFPA is basically a common HTTP server which has been moved to kernel just to get more performance. Meantime, Tempesta emphasises advanced filtering abilities, what is traditionally done by kernel-based firewalls. Also the whole server logic is designed to efficiently handle extreme loads like DDoS attacks and drop malicious traffic as early as possible. For example, while AFPA uses slow filesystem operations to serve cached content, we use special lightweight embedded in-memory database, so all jobs are very fast and our server can work fully in softirq.

It is not sufficient to build just a fast HTTP accelerator. Sometimes front-end servers serve large static or cached content (e.g., photo-hosting) that does not fit into RAM, or execute lightweight dynamic content via FastCGI. Since application layer DDoS attacks frequently target the most computationally expensive resources, disk access or complex application logic may force the server to its knees under DDoS. Thus to be resistant to application-layer DDoS front-end server must be designed in special way. We should neither accept connections from DDoS bots nor process requests from them or send responses. It is better to quickly drop any traffic as soon as bots are detected and their addresses are determined. Voigt et al. [12] have shown the efficiency of a kernel based approach and an "early discard" principle of dropping invalid packets before they have consumed a lot of system resources.

One of the simplest ways of dealing with DDoS is using a log monitor [15] [14] to dynamically analyze Web server logs and add firewall rules to block clients by requests rate limit or by specific URL access. Filtering is cheap in this solution, but generating the filtering rules isn't. If DDoS botnet activates all its zombies at once, then victim host suffers from overloading, but it still needs to generate and upload thousands of filtering rules. In these conditions parsing logs and executing new processes to add firewall rules make the attack even worse.

Tight integration of Postfix mail server with Linux firewall and maintaining of the database of spam bots was proposed in LKML [13], so Postfix can dynamically block hundreds of thousands of spamming addresses without computationally expensive logs parsing.

Linux Iptables firewall provides *limit*, *hashlimit*, *connlimit*, *set* and *string* matching modules. They can efficiently block DDoS attacks at the network layer by limiting the number of parallel connections, specific signature or client packet rate. As opposed to Nginx's HTTP requests rate limit module, Iptables's rate limiting modules operate on IP packets instead of HTTP requests. That makes traffic analyzing less accurate since client HTTP requests may vary in size and contain various number of packets.

As a result, there are two obstacles to efficient DDoS filtering. The first is that low level tools like firewalls cannot reliably assemble transport and application layer messages and can operate only at the network layer. To be able to classify traffic at higher layers we need to recreate the operating system TCP and application protocols logic, which leads us to double processing of network traffic. Similarly, application servers don't have access to lower layers and can classify traffic only at the application layer. The second issue is that if an application server is responsible for traffic classification,

then it must issue thousands of blocking rules for the firewall, and that is costly.

3 Synchronous Sockets

As described in previous sections, traditional Berkeley socket API introduces two significant problems: data copying and context switches. To cope with the obstacles kernel sockets can be used. One of the examples, which can be found in Linux kernel tree, is Oracle Reliable Datagram Sockets (RDS) [26]. Like AFPA [2] RDS uses standard Linux kernel TCP socket callbacks to handle TCP connections. The hooks are called by TCP code when a socket changes its state, new data arrives to the socket, there is space in the write buffer, and in few other cases. RDS tries to do as much work as possible in bottom half, but it still uses standard Linux kernel *accept()* TCP callback that may sleep, so it must accept actual connections in a work queue (i.e. in a separate kernel thread). Another example of kernel sockets, Ceph [27], also reads data in a work queue.

Reading from a socket in a context other than deferred interrupt context (either in user space or in kernel thread) is asynchronous to arrival of TCP segments. Let's consider an example shown in Figure 1. An operating system receives three packets *p0*, *p1* and *p2* sequentially and places them in queues of three different sockets. When a packet is placed in a socket queue, a corresponding process is woken up (suppose that *epoll(7)* is used). Since the packet *p2* was received last, it is likely that it is in the CPU cache (assuming either Direct Cache Access is used or the OS had read the packet headers), but *p0* can be evicted by successive packets *p1* and *p2*. However, there is no guarantee which process (or thread) reads a packet first and which particular packet will be read first. Thus, at high packet rates application threads can read a packet that was already evicted from the local CPU cache, which leads to poor cache hit. Moreover, the user space application and softirq (deferred interrupt), which processed the packet, may work on different CPU cores, which also leads to unnecessary cache starvation. Linux offers Receive Flow Steering (RFS) to direct application steer packets to the correct CPU with application locality in mind. However, softirq and application or kernel threads still operate asynchronously, i.e. at different speeds, so a thread may read a packet that may already have been evicted by a packet read in the softirq. The problem is more critical in modern high speed networks.

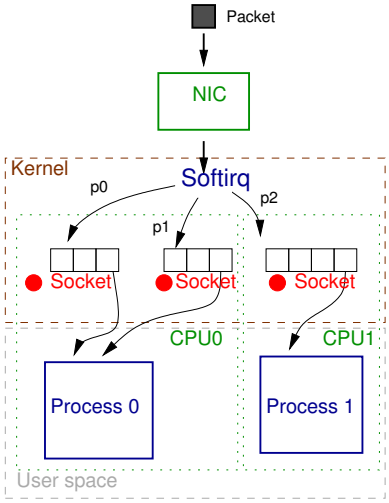

Figure 1: Reading from common sockets is asynchronous

To solve the issue we developed Linux kernel Synchronous Sockets library that is based on Linux TCP socket callbacks and provides an easy to use interface for socket reading, writing and multiplexing. Opposite to RDS TCP processing, it works completely in softirq context. Synchronous Sockets are part of Tempesta distribution, but they were developed as an independent Linux kernel loadable module that exported a set of calls. Any kernel-based application can use them for faster socket operations. The module uses a number of standard Linux routines (currently TCP only), so we patched the kernel to get access to some not-exported functions.

In kernel space we don't have to operate with sockets as with common file handles, so we implemented some standard Linux functions in more lightweight manner without operations with file descriptors. Also, the API was designed with filtering ability in mind, so we developed our own socket receiving function which does not call *__kfree_skb()*, as standard *tcp_read_sock()* does it, and allows user to manage *skb_buff*s (sk_buff, or socket buffer, is standard Linux kernel packet descriptor) himself.

Currently Synchronous Sockets support TCP protocol only. A user should define an application protocol handle inherited from *SsProto* struc-

ture and set data handling callback to it using *ss_proto_push_handler()*. The protocol handle is passed as an argument to socket callbacks. Synchronous Sockets allow registration of the following callbacks:

1. *connection_new()* - called when a new connection is accepted;
2. *connection_drop()* - called when TCP connection, associated with the socket, is dropped by the peer;
3. *connection_recv()* - process data received on the socket;
4. *put_skb_to_msg()* - add an sk_buff to the current connection message. We need this low-level sk_buff operation at connection (higher) layer to provide zero-copy data transfers with socket buffers reuse;
5. *postpone_skb()* - postpone an skb into internal protocol queue;

ss_hooks_register() registers the callbacks. *ss_close()* and *ss_send()* functions actively close and sends data through a socket, respectively. Also *ss_tcp_set_listen()* links protocol handle to the listening socket, so we can have many listening sockets simultaneously with each serving a different protocol.

The following pseudo-code example demonstrates the API usage (the real world example can be found in the library distribution, see section 6):

```
/* Inherited application logic class. */
struct {
    SsProto proto;
    /* some other members */
} my_proto;

struct socket *listen_sock;

int my_read(void *proto, char *data, int len)
{
    /* Read application level data. */
}

int my_conn_new(struct sock *sock)
{
    /* Handle a new TCP connection */
}

SsHooks ssocket_hooks = {
    .connection_new  = my_conn_new,
    .connection_recv = my_read,
};

int my_init(void)
{
    /* Register TCP (layer 4) callbacks. */
    ss_hooks_register(&ssocket_hooks);

    /* Set TCP handlers. */
    ss_tcp_set_listen(listen_sock->sk,
                      (SsProto *)&my_proto);
}
```

4 Tempesta Architecture

The main goals of Tempesta project are:

1. Providing a framework with full access to network, transport, session, presentation and application layers for building efficient and intelligent traffic classification, modification and filtering systems;
2. Working as part of Linux TCP/IP stack to efficiently handle short connections that are common in DDoS attacks;
3. Tight integration with native Linux security and netfilter subsystems to manage dynamic rules and efficiently classify and block extremely large botnets;
4. High performance HTTP sessions processing with caching reverse-proxy functionality to mitigate Web service overloading under huge loads (including DDoS attacks);
5. Providing normalized HTTP messages to classification algorithms and sending normalized client HTTP requests to back-end servers to avoid different HTTP interpretation by hosted analyzing algorithms and defended Web servers.

Basically, Tempesta is a hybrid of a caching HTTP server and a firewall with dynamic rule set. It is implemented as a set of Linux kernel loadable modules. Synchronous Sockets provide common interface to TCP and higher layer protocols. Also, small modifications of Linux kernel were done, so now it provides more control over TCP sockets to loadable kernel modules. All incoming packets are processed in deferred interrupt context which improves CPU cache hit on packet processing [2] and allows dropping of unwanted packets and connections as early as possible to minimize resource usage.

Tempesta has modular architecture for better flexibility. Following types of modules are supported:

- Traffic classification (analyzing) and statistics gathering;
- Detection and handling stress (overloading) of a local system or back-end Web servers (e.g., by integration with custom Nginx or Apache modules, described in more details in section 4.7);
- Filtering by using various techniques (silent traffic dropping, connection resetting, tarpit [22], sending HTTP error messages etc.)
- HTTP request distribution across back-end servers (round-robin, by Host or URL etc.);
- Generic modules for HTTP message processing;

Even though Web Application Firewalls [21] are out of the scope of this work, it is worth noting that this kind of security applications can be implemented in Tempesta environment since it provides filtering functionality in combination with access to HTTP request data.

4.1 Handling Packets

First, when a packet is retrieved, it goes into operating system IP receive routine, *ip_rcv()* or *ipv6_rcv()*, where it is verified by our filter hooks registered with Netfilter. If the packet passes the IP filter it goes into TCP input routines, *tcp_v4_rcv()* or *tcp_v6_rcv()*. TCP socket callbacks are called much deeper in the TCP code, but Linux provides security hooks for sockets (e.g., *security_sock_rcv_skb()*), that are called from the functions, so we use the hooks to register our TCP layer filtering and classification callbacks. Synchronous Sockets handle higher level TCP ingress and egress.

Tempesta network input and output subsystem works fully in deferred interrupt context without any helper threads. Incoming packets are parsed as soon as they arrive to network interface and while CPU caches are warm. HTTP cache resides in main memory only, so there's no disk IO (see subsection 4.5) or other sleeping operations, so all work on HTTP requests processing and sending responses can be done in the same softirq. Some of the packets could be cloned using *skb_clone()* call (subsection 4.6 motivates this) and passed through zero-copy interface to a user-mode daemon for slow classification logic.

HTTP message (either request or response) may consist of multiple packets. The packets are not sent to back-end server until the request is fully assembled and processed. Separate message segments are postponed in the session queue.

If an HTTP request can not be served from cache, then the request is adjusted, passed to the send queue of the socket connected to a back-end server and a TCP send is initiated on the socket. All of that occurs in the same softirq in which the last piece of the request was received.

4.2 Handling Connections

Two types of connections are handled - back-end connections with the farm of back-end Web servers, and front-end connections with HTTP clients. Sending and receiving packets as well as connection establishing and closing in both types of connections are handled in a similar manner through Synchronous Sockets API.

Tempesta maintains a pool of persistent TCP connections with back-end Web servers. Persistence is based on standard HTTP Keep-Alive option. If a connection fails due to a timeout or connection closing, then a Synchronous Sockets hook is called. The hook calls a Tempesta callback that establishes a new connection. Thus, the system guarantees that there is always an active connection to back-end servers, so admissible client requests won't have to wait for connection to the Web server.

Currently there is no HTTP request distribution by URL or Host HTTP header values. The requests are distributed across back-end servers in round-robin fashion. The logic is implemented in a loadable module, so the system can be extended by a module that implements sophisticated routing and load balancing algorithms.

The first time when a client connects to the server, the system allocates a new TCP socket as a normal Linux server, so SYN Cookies, TCP offloading and other TCP features are still applicable.

Tempesta provides connection establishing and closing hooks, so classification modules can easily

keep track of connections.

Classifier modules observe the client connection (see section 4.6) and decide if it is normal or malicious. All normal connections are handled in standard Linux *tcp_hashinfo* hash table, in which Linux stores sockets with full identity. Classifiers can initiate *tcp_hashinfo* shrinking (see section 4.6), so we patched the Linux kernel to be able to use functions operating on the hash from our loadable modules. If a classifier decides that a connection is malicious, the filter module is called to handle the connection (e.g., just drop it).

Currently a malicious connection is just evicted from the connections hash table and all data structures associated with it are silently freed (no TCP FIN or RST segments are sent to the client). In the end, a new filter rule is added for the client IP address, so all subsequent packets from the client are dropped by the filter module.

4.3 Generic Finite State Machine (GFSM)

Traditional HTTP servers like Apache HTTP Server or Nginx provide a set of hooks for different HTTP processing phases (e.g., reading a client request, resource access check, generating an HTTP response, etc.). If the server logic is considered a finite state machine (FSM), then the phases are simply just states of the machine.

The problem with this simplistic approach is that there is no way to interrupt an FSM in some state, do some work on the processed request, and get right back to the same, previously interrupted, state. The problem arises in the context of requests classification performed by an external server. One such example is ICAP protocol (RFC 3507). Let's consider a simple scenario for ICAP:

- client sends POST request with an attached file;
- HTTP server parses the request and passes it to an ICAP server for analyzing, meantime it continues processing other requests;
- ICAP server scans the file for viruses and responds to the HTTP server;
- HTTP server restores the request processing procedure (executes scripts, sends the request to backend server and so on) if the ICAP server has decided that the file is malware-free, or sends an error code to the client otherwise.

However, "It's not an easy task" [31] to implement such logic in modern HTTP servers.

HTTP state machine in Tempesta is implemented using Generic Finite State Machine (GFSM). Modules for ICAP, classification algorithms training (see section 4.6), and even FastCGI are also considered to be developed through GFSM interfaces.

GFSM operation can be illustrated by the following scenario:

1. HTTP is explicitly implemented as an FSM, i.e. there is a single entry function that has a switch statement over the HTTP states;
2. HTTP registers the entry function using a GFSM call, so now the GFSM knows that there is a FSM and there is an entry function for it;
3. HTTP code uses connection layer callback to say that connections for the specified listening socket must be linked with its FSM control block;
4. when a new connection is accepted, then the connection layer executes *gfsm_dispatch()* with the linked FSM control block as an argument. *gfsm_dispatch()* passes the control block pointer to the FSM entry point function;
5. FSM control blocks are organized in a stack, so all required data for the current FSM operation (its current state, temporal data associated with currently processed message, information about registered hooks and so on) is placed on top of the stack. GFSM code is responsible for the stack management and calling appropriate FSM handling functions;
6. when HTTP reads a request and is ready to go forward it calls *gfsm_move()* to move to the next state;
7. *gfsm_move()* checks the FSM control block to see if there is another FSM (say ICAP protocol) that should get control when current FSM reaches the state. If yes, then GFSM allocates a new control block for the FSM that must be executed, and places it on top of the stack.

Next it calls *gfsm_dispatch()*, so the new FSM starts executing;

8. ICAP FSM sends the request to ICAP server and then returns, leaving its control block on top of the stack;
9. *gfsm_move* returns POSTPONE code to HTTP, so HTTP handler just exits;
10. when a response from ICAP server is received, ICAP code calls *gfsm_move()* move;
11. if current FSM reaches its final state, then its control block is popped from the stack and an FSM for the underlying control block is called;
12. so HTTP FSM is called again for the next state to which it moved on step 6.

4.4 HTTP Processing

4.4.1 HTTP Hybrid State Machine

We have studied several HTTP servers and proxies (Nginx, Apache Traffic Server, Cherokee, node.js, Varnish and userver) and learned that all of them use switch and/or if-else driven state machines. If logging is switched off and all content is in cache, then HTTP parser becomes the hottest spot. Simplified output of *perf* for Nginx under simple DoS is shown below (Nginx's calls begin with *'ngx_'* prefix, *memcpy* and *recv* are standard LIBC calls):

```
   %        symbol name
1.5719   ngx_http_parse_header_line
1.0303   ngx_vslprintf
0.6401   memcpy
0.5807   recv
0.5156   ngx_linux_sendfile_chain
0.4990   ngx_http_limit_req_handler
```

The next hot spots are linked to complicated application logic (*ngx_vslprintf*) and I/O. I/O issue in Tempesta is addressed by Synchronous Sockets. The problem with HTTP parsing is that HTTP parsing code is comparable in size with L1 instruction cache and processes one character at a time with significant number of branches. Modern compilers optimize large switch statements to lookup tables that minimizes number of conditional jumps, but branch misprediction and instruction cache misses still hurt performance of the state machine. Moreover, Nginx parser processes some input data two and more times (e.g., *ngx_http_parse_request_line()* for *sw_method* state).

Typically HTTP deterministic finite automaton (DFA) has small number of branches and loops, but relatively big cardinality of the input alphabet (98 ASCII characters according to RFC 2616). Depending on the parser complexity the DFA can have few hundreds states. Thus, implementing the state machine as a classical DFA table that is just a matrix of size $Q * \Sigma$ where Q is number of states and Σ is alphabet cardinality, is impractical. Modern hardware has tons of memory, but the DFA has random access pattern which leads to poor data cache hit rate.

To get better data and instruction caches hit rate and lower code branching we use cache sensitive hybrid HTTP state machine which combines table-driven DFA with switch statements. Tempesta implements two different state machines for HTTP requests and responses and the bigger one (for requests) currently has 209 states (we keep the accelerator logic as simple as possible), so the state number can be encoded with single byte. Each automaton transition is described by 16 byte data structure:

```
#define CTL_STR             1
#define CTL_RANGE           2
#define CTL_SLOW_PATH       4
#define CTL_ACTION          8

struct {
    unsigned char     ctl;
    unsigned char     length;
    unsigned short    __padding;
    union {
        struct {
            unsigned int pattern[2];
            unsigned char next_state[2];
        } str;
        struct {
            struct {
                unsigned char r_begin;
                unsigned char sub;
            } r[4];
            unsigned char next_state[4];
        } range;
    } u;
} XTrans;
```

So each data cache line (64 bytes on modern x86-64 hardware) contains 4 DFA transitions.

ctl determines FSM processing logic for the transition:

- transition type: *CTL_STR* performs matches against up to 2 patterns with 4 characters in length, *CTL_RANGE* matches a character of input data against up to 4 character ranges;
- whether the transition has slow path: if cur-

rent state has more than 4 outgoing edges or 2 string patterns is not enough, that not all the edges can be encoded, then we have to fall to slow path encoded by switch statement;

- whether the automaton should perform some actions (e.g., write a pointer to name of processed header) when it leaves the state.

We use additional static table that maps ASCII codes to $[0...71]$ a sequence (it maps upper and lower case letters to the same codes). The table also keeps values in a special order, so that for example symbols allowed for an HTTP header ([A-Za-z_-]) are mapped to one continuous subsequence and can be described by a single range. Hopefully, due to HTTP DFA properties only the first range is observed in most cases, and slow paths are taken very rarely. Currently we have only 6 states that use slow path transitions for the whole requests processing state machine.

str union can decode up to 2 transitions matching to different string patterns. Figure 2 shows an example of an automaton that parses GET, PUT, PROPFIND and PROPPATCH methods in 3 transitions at most.

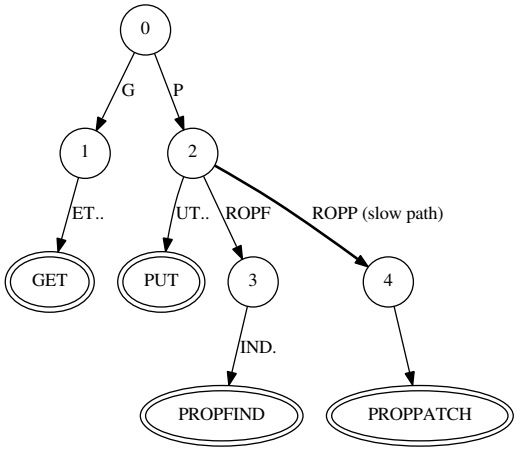

Figure 2: Automaton for parsing GET, PUT, PROPFIND and PROPPATCH HTTP methods

To match string patterns a 4-byte integer is used. If the pattern is shorter than 4 bytes, then the rest of the byte is filled by 0xff value that is treated as wildcard.

Note that state 3 has 3 outgoing transitions while we can encode only 2 of them. Since PROPPATCH method is relatively rare, then it can be parsed in slow path, switch-driven part of the state machine.

4.4.2 HTTP Normalization

There are many techniques to avoid HTTP intrusion detection systems [30] [29] [28]. In most cases the problem is that an IDS and a defended Web server interpret HTTP requests differently which leads to false negatives. It's a common practice for modern HTTP proxies to send client's HTTP request to back-end Web server unchanged. There is a possibility that in case a proxy changes HTTP requests, it can accidentally break Web application logic. So this kind of behavior is vulnerable, but it's safe in normal circumstances. This problem is mostly relevant to Web application firewalls rather than DDoS mitigation systems, so we skip details of HTTP requests normalization here and just mention that Tempesta removes some HTTP ambiguities and provides normalized HTTP requests to classification algorithms and transmits them to back-end servers in a normalized form. Like Snort [25] the normalization is performed depending on specified server personality, so a system administrator can specify which transformations are desired.

4.5 Caching

As it was mentioned previously, one of the main Tempesta goals is to provide robust foundation for DDoS prevention system. In some cases at least a few requests from a client are needed to classify its session (this is especially true for classifiers which works on application level). Since DDoS botnet can contain up to hundreds of thousands of zombies and activate them all at the same time, then up to million of requests should be serviced before the system goes to effective defense mode. If a reverse-proxy, which runs the classifier, uses on-disk cache to serve Web content significantly larger than available RAM, then under such circumstances it would go into thrashing mode, that causes the system performance collapse. So the Web accelerator itself becomes a victim of DDoS attack and cannot efficiently execute classification logic.

Traditional caching Web servers use filesystem to store their content. Filesystem operations are very slow due to disk operations, heavy synchronization mechanisms and dentry lookups suitable only for directories and files. Instead, Tempesta

uses our own lightweight embedded in-memory database with persistency to handle cache objects, static content, filter rules (section 4.8). The database also can be used to store resolver results, events and access logs or traffic dumps.

Thus, Tempesta Web cache and stored static content always fully fit into available main memory, so there is no disk I/O operations involved in client requests servicing. HTTP responses received from back-end server are saved in main memory after delivering to clients. To process further client requests quicker we store whole response messages (including HTTP headers) with a list of buffers, which are directly used in socket send queue.

All files of Web content cache are *mmap()*'ed and *mlock()*'ed to eliminate disk accesses. The cache persistency is provided by standard VMM mechanisms which synchronize dirty RAM pages with disk. A helping threads are introduced for eviction of old and/or rarely accessed cache entries and loading Web content from disk to the cache database. The second thread monitors on-disk Web content directory through *inotify(7)* interface for newly created or modified files. To load a Web content file into in-memory database the thread reads the file by pieces directly into the database *mmap()*'ed area. The pieces are indexed by the database and can be directly used by TCP sending code. Thus, the database cache contains preprocessed socket buffers instead of raw data. Meantime, it is possible to update Web server content directory in runtime.

4.6 Requests Classification

As it was depicted in section 2 that there are plenty of classification methods which require access to traffic data at different network layers. Tempesta provides following hooks to use in classification modules:

- *classify_ipv4()/classify_ipv6()* - called for each received IPv4/IPv6 client packet;
- *classify_tcp()* - called for each received TCP segment (not all IP packets contain TCP header, so we split IP and TCP callbacks);
- *classify_conn_estab()* - called when a new client connection is established (many TCP SYNs can precede an established connection, so it's more efficient to handle events for established and closed connections plus to events for SYN and FIN segments);
- *classify_conn_close()* - called when a client connection closed;
- *classify_tcp_timer_retrans()* - called on retransmits to client (e.g., SYN+ACK or data);
- *classify_tcp_timer_keepalive()* - called on sending TCP keep alive segments;
- *classify_tcp_window()* - called when we choose a our window size to report to client;
- *classify_tcp_zwp()* - called when peer reported zero window, so we cannot send data and must send TCP zero window probing segments;

Last four TCP specific hooks are quite useful to detect attacks like Slow HTTP [17] and/or implement client transmissions rate limiting and tarpit modules.

There is no specific classification hooks for HTTP, instead a traffic analyzing module should use GFSM hooks for particular HTTP processing phases.

The callbacks (except the last four TCP specific) return one of the following constants:

- PASS - current packet looks good and we can safely pass it;
- BLOCK - the packet looks malicious and we need to block all packets, including this, from the client;
- POSTPONE - we need more packets to make a decision, current packet must be stashed and will be sent to the destination (if is decided as innocent) with following packets at once.

If TCP event, incoming packet, request or whole client was decided dangerous, then the filter hook is called. So we block the first occurrence of malicious traffic and early detect intentional TCP connection slowdowns.

Currently Tempesta equipped with only one simple classification module which uses just a few of the callbacks. It analyzes HTTP requests rate, number of simultaneous connections and rate of new connections per one client. The first two limits work very similar to Nginx limiting modules [9] [10]. If a client exceeds one of the limit, then filter module is asked to block the client.

HTTP connection sockets are organized in ordered list using *tcp_hashinfo* hash table. Ordering in the list is determined by *weight* - specific value which is assigned by classification modules depending on how the connection aggressive is. If stress module detects the local or back-end system overloading, then it asks classification module to evict some connections, so the most aggressive connections are dropped and the clients are blocked.

In some cases classification logic can use machine learning algorithms. Some of the algorithms (e.g Hidden Markov Model) has two phases: learning (training) and actual detection work. Training is usually relatively expensive. Web application firewalls also can use very slow protection logic, which deeply parses HTML, XML and so on. So, there is no sense to run such heavy-weight logic in kernel, especially in deferred interrupt context. Thus, using GFSM Tempesta can offload heavy classification and/or training logic to external servers (e.g., through ICAP interface) or just clone a packet and deliver it it a local user space daemon.

4.7 Stress Accounting

To cope with DDoS attacks which are indiscernible from real flashcrowds, stress modules detect that local system (on which Tempesta is running) or back-end servers are overloaded. When a stress module decides that overloading occurred it calls generic classification logic, which shrinks current connections list. A classifier assigns weights to currently established connections depending on how much *stress* a connection causes to the system. The mostly aggressive connections are closed.

Currently only local stress module is implemented. It measures current memory consumption, internal queues size, latencies and few other metrics. All the metrics are configurable and if some of them are greater than specified limits, then the system stress is reported and Tempesta begins to evict connections. The weights are assigned to connections also depending on the values of the measures.

4.8 Filtering

Filter modules work on three network layers: Netfilter for network layer, Linux Security Modules (LSM) for TCP (transport layer) and application layer.

On application layer a filter module should be registered through GFSM interfaces, which are called by Web accelerator engine. Appropriate FSM control block contains connection socket, so the filter is able to terminate the connection, send HTTP error message or add a rule to block the client traffic on network layer.

Since we dynamically add new rules, we need to care about temporality of the rules. Since DDoS bots in most cases employ common user resources, it's unwished to block IP addresses forever. Rather we add blocking rules for some period of time after which they are obsolete. If the address is decided as attacking again, then a new blocking rule for it is produced. The rules are stored and managed using the same in-memory database as mentioned in section 4.5.

So like Kernel Blocking Firewall [13], the blocked addresses are also organized in LRU list, thus they can be unblocked by a timer expiration or be evicted from the list if there is no enough memory (the last is very rare case on modern machines with gigabytes of RAM). We move a blocked address to the beginning of LRU list each time when we get a packet from the address. LRU is most suitable cache eviction strategy for DDoS bots because it stores rules for the most active (and harmful) clients and evicts an address when it stops sending. The rules database is also persistent, so black list is loaded back after system reboot.

Tempesta also supports loading of external blocking rules. The rules can work on all network layers. For example, one can specify a rule which makes Tempesta to block all HTTP requests from IP 1.2.3.4 and which contain "Some browser" string in User-Agent HTTP header value.

5 Experiments

In all our experiments we used two servers connected back-to-back with a 10Gbps Ethernet link. One server, a 10-core Intel Xeon E7-4850 (2-2.4GHz) with 64GB RAM (One CPU with 10 cores), was used to run server applications such as an HTTP accelerator. It was running Linux 3.10.10. The other server, a 6-core Intel Xeon X5675 with 32GB RAM, was used to run a traffic generator on Linux 2.6.32. The servers were connected using 10Gbps Intel Ethernet adapters with 40 separate RX and TX queues.

5.1 Synchronous Sockets

To start with, we measured the performance of three socket implementations: Synchronous Sockets, plain kernel sockets (written similar to the code used in Ceph [27]), and user-mode sockets with optimizations in connection acceptance. In all tests we used a multi-threaded TCP client that established multiple connections with the server in each thread and sent several thousands 64-byte messages on each connection. The servers simply read messages and accept new connections in one thread, so only one CPU core is used for server.

First, we measured the time it takes to establish 20,000 connections while sending messages on each established connection (when a connection is established the client immediately starts sending messages). This is more complex, but realistic in context of DDoS, rather than just new connections establishing or sending messages over already established connections. Results are shown in Figure 3.

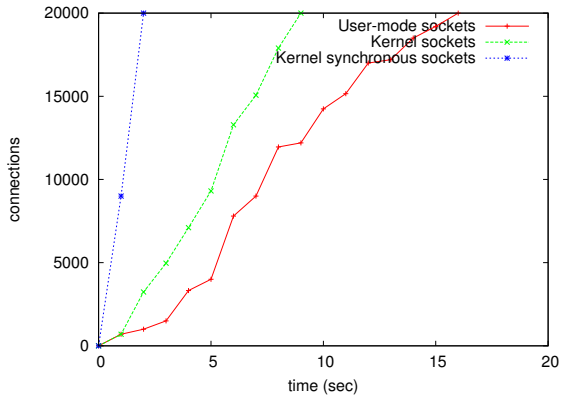

Figure 3: Performance of connection establishing

We made the following optimizations for the user-space sockets test. We increased the backlog parameter of *listen(2)* system call from 100 to 1000, and accordingly set */proc/sys/net/core/somaxconn* to 1000 as well (it is 128 by default). Also, we placed listening sockets in non-blocking mode and run *accept(2)* in a loop each time *epoll(7)* reported that the descriptor was ready.

Kernel sockets use work queues, so actual work is done in the kernel thread context rather than in the softirq context. They show better results and a smoother performance curve than user-mode sockets due to lighter context switches (between kernel threads and softirq only) and no copying. However, it takes over 9 seconds to establish 20,000 connections with kernel sockets, while with Synchronous Sockets it takes less than 2. The reason for such a big difference is not just the different execution context, but also the extra work that simple kernel sockets do (operating with work queues is relatively expensive).

In the second test of socket implementations we measured the relationship between the request rate (number of requests per second) and the number of established connections. Results are depicted in Figure 4 (connections axis is log-scaled).

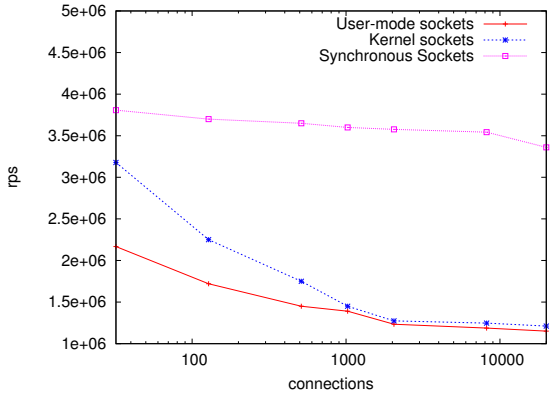

Figure 4: Requests per second for simple socket servers

The results show that Syncronous Sockets perform not only faster than other implementations, but also is more stable as the number of connections increases. At 20,000 concurrent connections they process more requests per second (3,36 million) than regular kernel-mode sockets at 32 connections (3,18 million).

5.2 HTTP Hybrid State Machine

Since Tempesta and other HTTP servers use very different HTTP state machines, it's difficult to directly compare the performance of a HTTP Hybrid State Machine and traditional switch-driven state machines. So we developed a microbenchmark that measured the performance of the two most critical HTTP parsing functions (*ngx_http_parse_header_line()* and *ngx_http_parse_request_line()*) of the popular user-

mode HTTP server Nginx version 1.5.6. The same state machine was implemented using our hybrid approach. In the test we executed the parsers one million times against a single HTTP request of 1381 bytes in length (note that all data was in L1 data cache). Results are shown in Figure 5.

	Hybrid SM	Nginx
Parse header line	4541ms	6101ms
Parse request method	2910ms	5378ms

Figure 5: Comparison of HTTP Hybrid State Machine and Nginx HTTP parser processing HTTP header lines and request methods

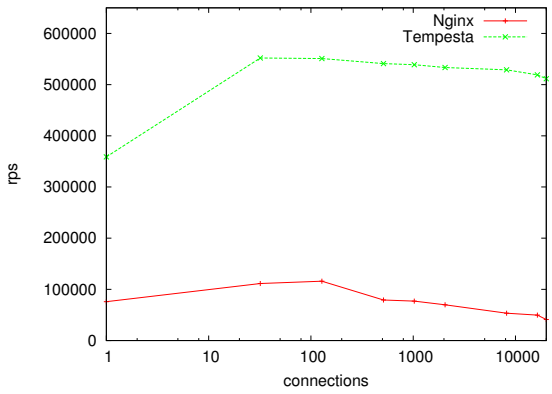

Figure 6: Requests per second for Tempesta and Nginx

5.3 Tempesta HTTP Accelerator

Next we measured the performance of our Tempesta prototype and Nginx in the most valuable metric - time to service a regular client request that is fulfilled from local cache, and time to block a client when HTTP request rate limit is exceeded. Nginx has been chosen as most widespread Web accelerator and platform for WAF.

Nginx was configured with 10 worker processes (one worker process per core). RFS was enabled on server system. NIC interrupt queues were binded to all CPU cores such that each core has 4 assigned TX and RX queues. Following configuration options were switched on: *multi_accept*, *sendfile*, *epoll*, *tcp_nopush* and *tcp_nodelay*. We also switched off any logging in Nginx configuration to eliminate unnecessary filesystem operations.

To measure the number of requests per second each server can handle depending on the number of concurrent connections, we sent 10 million HTTP requests using Siege [19]. We ran a warming test of one thousand requests just before the actual test to make sure that all software and hardware caches are warmed. In the test both servers were serving a 612-byte static index page. Tempesta had the page preloaded into its cache. The performance results are shown in Figure 6.

Results for a single connection are worse than in tests with more concurrent connections due to utilization of only one CPU core instead of 10 (we have switched off hyper threading). Tempesta achieves the best result, 552 thousands requests per second at 32 connections. Running on the same hardware, Nginx gets his performance peak, 116 thousands rps, at 128 concurrent clients. Nginx does more operations with memory and services static pages from filesystems, so concurrency slightly greater than available CPU cores (and worker processes in our case) is better for it. However, Nginx quickly loses points at 512 connections (only 79,300 rps) which stays almost the same at 1024 connections (76,980 rps). Further increase of the number of connections quickly degrade the server performance and leads to only 41 thousands request per second at 20 thousands connections.

Tempesta has smoother performance degradation than Nginx. Tempesta from 552 thousand rps at 32 connections to 511 thousand rps at 20 thousand connections versus from 116 thousands rps to 41 thousands rps correspondingly for Nginx.

Both servers have very different rate limit blocking strategies. Nginx sends 503 Service Temporarily Unavailable responses and gently closes the connection. Tempesta just frees all data structures linked with the connection and drops all packets from the client for period of time while the client still keeps the TCP connection open. This is roughly to normal users, but very efficient in battle with DDoS bots. So it's hard to perform blocking benchmarks and compare results for the servers.

However, we made latency microbenchmark to measure how fast the servers operate in normal conditions and block a user by rate limit. Nginx sends reply for normal and blocked requests, so we have measured time between receiving a request and sending an appropriate response. We did the

same to measure time required to process a request from local cache for Tempesta. Since Tempesta does not send any reply to blocked requests, but rather just silently drops it, we have added timing for request processing to our code.

	Block	Cache
Nginx	23us	22us
Tempesta	4us	6us

Figure 7: Average time to block a client or serve a normal request

Since a blocked client cannot send messages the values in the table were calculated as average values for service time for one thousand requests. In blocking tests Nginx responds with a 537-byte HTML file.

The time to block a request is slightly higher for Nginx than the time to service a normal request from local cache. That is because the limiting logic is implemented in an Nginx extension module, so Nginx must call the module in addition to sending static content. The rate limiting module also sends the HTTP response code 503 "Service Temporarily Unavailable" which is a static file. Substantial amount of system resources is used to block a client, therefore this approach is not suitable for filtering out a massive DDoS attack.

6 Availability

The Tempesta source code is published under GNU GPL version 2 and is freely available for download at `https://github.com/natsys/tempesta`.

References

[1] M. Srivatsa, A. Iyengar, J. Yin, L. Liu. *Mitigating application-level denial of service attacks on Web servers: A client-transparent approach.* TWEB, 2008.

[2] P. Joubert, R.B. King, R. Neves, M. Russinovich, J.M. Tracey. *High-Performance Memory-Based Web Servers: Kernel and User-Space Performance.* USENIX Annual Technical Conference, General Track, 2001, pages 175-187.

[3] S. Ranjan, R. Swaminathan, M. Uysal, E.W. Knightly, *DDoS-Resilient Scheduling to Counter Application Layer Attacks Under Imperfect Detection.* INFOCOM, 2006.

[4] S. Y. Nam, N. Nodir, T. Lee. *Defending HTTP Web Servers against DDoS Attacks through Admission Control and Attack Flow Detection.* Technical Report YU-CNS-12-01, 2012.

[5] T. Brecht, D. Pariag, L. Gammo. *accept()able Strategies for Improving Web Server Performance.* USENIX Annual Technical Conference, 2004, pages 227-240.

[6] I. Molnar. TUX patches. *http://people.redhat.com/~mingo/TUX-patches/.*

[7] A. van de Ven. kHTTPd Linux http accelerator. *http://www.fenrus.demon.nl*.

[8] P. Tan. OpenKeta. *http://openketa.sourceforge.net/.*

[9] Nginx HTTP requests limit module. *http://wiki.nginx.org/HttpLimitReqModule*.

[10] Nginx simultaneous connections limit module. *http://wiki.nginx.org/HttpLimitZoneModule*.

[11] Apache mod_evasive. *http://www.zdziarski.com/blog/?page_id=442*.

[12] T. Voigt, R. Tewari, D. Freimuth, A. Mehra. *Kernel Mechanisms for Service Differentiation in Overloaded Web Servers.* USENIX Annual Technical Conference, 2001, pages 189-202.

[13] Kernel Blocking Firewall. Linux Kernel Mailing List, 2009. *https://lkml.org/lkml/2009/1/8/467*.

[14] Fail2ban. *http://www.fail2ban.org*.

[15] B. Glass. *Log Monitors in BSD UNIX.* BSDCon, 2002, pages 131-142.

[16] Imperva. *Report 12 - Denial of Service Attacks: A Comprehensive Guide to Trends, Techniques and Technologies.* ADC Hacker Intelligence Initiative, 2012.

[17] Slowloris, the low bandwidth, yet greedy and poisonous HTTP client. *http://ha.ckers.org/slowloris/.*

[18] M. Guirguis, A. Bestavros, I. Matta, Y. Zhang. *Reduction of quality (RoQ) attacks on Internet end-systems.* INFOCOM, 2005, pages 1362-1372.

[19] Siege, http load testing and benchmarking utility. *http://www.joedog.org/siege-home/*.

[20] Y. Chen, K. Hwang. *TCP Flow Analysis for Defense against Shrew DDoS Attacks.* IEEE International Conference on Communications, 2007.

[21] I. Bar-Gad. Web application firewalls protect data. *http://www.networkworld.com/news/tech/2002/0603tech.html*. 2002.

[22] T. Liston. Tom Liston talks about LaBrea. *http://labrea.sourceforge.net/Intro-History.html*.

[23] G. van Rooij. *Real Stateful TCP Packet Filtering in IP Filter*. USENIX Security, 2001.

[24] T.H. Ptacek, T.N. Newshman. *Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection.* Secure Networks Inc, 1998.

[25] Snort, network intrusion detection system. *http://www.snort.org*.

[26] Oracle Reliable Datagram Sockets. *https://oss.oracle.com/projects/rds*.

[27] Ceph, distributed object storage. *http://ceph.com*.

[28] HTTP IDS Evasions Revisited. *www.snort.org/assets/164/sf_HTTP_IDS_evasions.pdf*. Sourcefire Inc, 2004.

[29] I. Ristic. Protocol-Level Evasion of Web Application Firewalls. *https://community.qualys.com/blogs/securitylabs/2012/07/25/protocol-level-evasion-of-web-application-firewalls*. 2012.

[30] L. Carettoni, S. di Paola. HTTP Parameter Pollution. *https://www.owasp.org/images/b/ba/AppsecEU09_CarettoniDiPaola_v0.8.pdf*. 2004.

[31] I. Sysoev. Help, I need write a nginx module. Nginx Mailing List, 2009. *http://forum.nginx.org/read.php?2,9136,9152*.

[32] M. Dobrescu, N. Egi, K.J. Argyraki, B. Chun, K.R. Fall, G. Iannaccone, A. Knies, M. Manesh, S. Ratnasamy. *RouteBricks: exploiting parallelism to scale software routers.* SOSP, 2009, pages 15-28.

[33] T. Marian, K.S.Lee, H. Weatherspoon. *NetSlices: Scalable Multi-core Packet Processing in User-space.* ANCS, 2012, pages 27-38.

[34] I. Marinos, R.N.M. Watson, M. Handley. *Network Stack Specialization for Performance.* HotNets, 2013, pages 9:1-9:7.

[35] S. Han, S. Marshall, B. Chun, S. Ratnasamy. *MegaPipe: A New Programming Interface for Scalable Network I/O*. OSDI, 2012, pages 135-148.

[36] E.Y. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, K.S. Park. *mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems.* NSDI, 2014, pages 489-502.

[37] T. AbuHmed, A. Mohaisen, D Nyang. *A Survey on Deep Packet Inspection for Intrusion Detection Systems.* CoRR, 2008.

[38] I. Dubrawsky. Firewall Evolution - Deep Packet Inspection. *http://www.symantec.com/connect/articles/firewall-evolution-deep-packet-inspection*.

[39] E. Kohler, R. Morris, B. Chen, J. Jannotti, M.F. Kaashoek. *The Click Modular Router.* ACM Trans. Comput. Syst., vol. 18, 2000, pages 263-297.

[40] P. E. McKenney What is RCU? Part 2: Usage. *http://lwn.net/Articles/263130/*. 2008.