

Sentinel: Hardware-Accelerated Mitigation of Bot-Based DDoS Attacks

Peter Djalaliev, Muhammad Jamshed, Nicholas Farnan and José Brustoloni

Department of Computer Science, University of Pittsburgh

Pittsburgh, PA 15260 – USA

Email: {peterdj,ajamshed,nlf4,jcb}@cs.pitt.edu

Abstract—Effective defenses against DDoS attacks that deplete resources at the network and transport layers have been deployed commercially. Therefore, DDoS attacks increasingly use normal-looking application-layer requests to waste server CPU or disk capacity. CAPTCHAs attempt to distinguish bots from human clients and are often used to avoid such attacks. However, CAPTCHAs themselves consume resources and frequently are defeated. Kill-Bots reduces CAPTCHA overhead by pushing client authentication into the kernel. However, Kill-Bots requires kernel modifications, which can be infeasible. We describe the design, implementation, and performance evaluation of Sentinel, a network device that overcomes several limitations in Kill-Bots. Sentinel can be easily deployed as a bridge in front of server farms, modularly accepts a variety of present and future authentication schemes, and can use network processors to accelerate authentication. Experiments show that Sentinel greatly reduces the impact of DDoS attacks on the response time experienced by legitimate clients.

I. INTRODUCTION

Distributed denial-of-service (DDoS) attacks are a major security threat to web servers. In such an attack, an attacker orders multiple hosts to send malicious traffic toward a server, so as to deplete network bandwidth or server CPU, disk, or memory capacity. DDoS activity can make commercial web servers unresponsive for days and cause damages of millions of dollars to the servers' owners.

The first DDoS attacks, such as smurf and SYN attacks, sought to deplete resources at the network or transport layers. Effective defenses against such attacks have been deployed commercially (e.g., [1], [2]). Therefore, attacks increasingly use compromised hosts (bots) to send just enough application-layer requests to deplete the servers' CPU or disk capacity. The latter attacks mimic flash crowds, i.e., natural peaks in client traffic. Existing network-based defenses are unable to distinguish between such attacks' and legitimate traffic.

A popular server-based defense against application-layer attacks is to use CAPTCHAs [3] to authenticate users. CAPTCHAs are perceptual challenges that are easy for humans and infeasible for computers (e.g., recognizing distorted text or speech). However, CAPTCHAs consume significant server CPU and other resources. Thus, the authentication process can itself be abused for denial of service. Moreover, authentication methods will need to evolve because of attackers' continued advances in breaking them. Kill-Bots [4] reduces, but does not eliminate, CAPTCHA overhead by pushing authentication into the server's operating system

kernel. However, this solution still burdens the server, and is infeasible if operating system code or personnel able to modify it are unavailable. Additionally, Kill-Bots does not support persistent connections and pipelining, and may blacklist IP addresses indefinitely.

This paper contributes the design and implementation of **Sentinel**, a network device for authenticating clients and mitigating bot-based DDoS attacks. Unlike Kill-Bots, Sentinel (1) can be deployed simply by installing it as a bridge in front of an unmodified server or server farm, (2) can use network processors to accelerate authentication and withstand large-scale attacks, (3) fully supports HTTP 1.1, (4) when warranted, rehabilitates previously blacklisted IP addresses, and (5) modularly accepts different client authentication methods.

Although Sentinel is easier to deploy and more scalable and flexible than are server-based solutions, its design also presents significant challenges. Because it is a network device, Sentinel needs to perform TCP connection splicing, stateful packet filtering, packet scrubbing, and deep packet inspection to avoid network- and transport-layer attacks. We describe how Sentinel avoids IP address spoofing, TCP SYN floods, fragmentation attacks, and session hijacking while filtering out bot traffic at Gbps transmission rates.

The rest of this paper is organized as follows. Section II discusses in greater detail botnets and existing defenses against their application-layer attacks. Sections III, IV, and V describe Sentinel's design, implementation and performance evaluation, respectively. Section VI compares Sentinel to Kill-Bots and other related work, and section VII concludes.

II. BACKGROUND

This section provides greater detail about botnets and methods for mitigating their application-layer DDoS attacks.

A. Botnets

Botnets are networks of compromised hosts, *bots*, controlled by a *botmaster* [5]. The hosts are usually infected by malware that spread like viruses (e.g., as e-mail attachments or downloadable files) or worms (e.g., by exploiting server applications with known vulnerabilities). Once it infects a host, the malware downloads the bot executable from a known URL. Bots receive commands from the botmaster through some command and control (C&C) infrastructure. Most bots use Internet Relay

Chat (IRC) or HTTP servers for C&C, but peer-to-peer (P2P) networks are increasingly being used.

Botnets are usually operated for profit and are available for hire. For instance, in 2006, the owner of an online sportswear store admitted having ordered a bot-based DDoS attack against competing online stores [6]. Botnets can contain tens of thousands of compromised hosts and generate up to tens of Gbps of traffic [7]. In 2006, the HoneyNet Alliance reported botnets with median size of 40,000 nodes and maximum size of approximately 350,000 nodes worldwide, 120,000 of which visible at any time of the day.

DDoS attacks previously used anomalous traffic (such as ICMP or SYN floods) with spoofed source IP addresses. More recent attacks use sufficiently large botnets for generating seemingly-normal HTTP requests that deplete the victim's CPU or disk capacity. In the network, it is often impossible to distinguish if sources of such HTTP requests are legitimate clients or bots.

B. CAPTCHAs

Researchers have proposed reverse Turing tests (RTTs) to distinguish human clients from bots. RTTs evolve from the original Turing test [8], but use a computer instead of a human entity to judge test responses. The tests must be solvable by a human, but infeasible for a computer.

Examples of reverse Turing tests are CAPTCHA graphical puzzles [3]. Web sites currently use CAPTCHAs to prevent automated spam-related activity, such as e-mail account creation and online forum posts. The images are generated using distortion techniques (Fig. 1) to resist automated object recognition techniques [9].

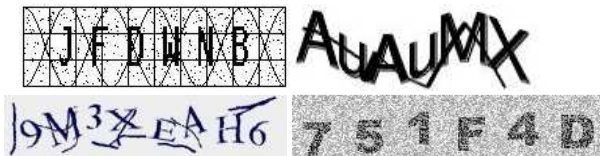


Fig. 1. CAPTCHA visual test designs

As artificial intelligence techniques evolve, existing CAPTCHA techniques can be expected to become vulnerable and need to evolve too [10]. In 2002, researchers reported a mechanism with 92% success rate of breaking Yahoo CAPTCHAs [9]. Yahoo subsequently improved its CAPTCHAs, but recent reports suggest successful attacks against Yahoo [11], Windows Live [12], and Gmail [13] CAPTCHAs. New, possibly radically different CAPTCHA or other approaches for client authentication can be expected in the future, e.g., using cognitive logic [14], pictures, or sound.

C. Kill-Bots

Kill-Bots' client authentication method is fixed: it uses CAPTCHAs to detect bot-based DDoS attacks that mimic flash crowds [4]. A web server running Kill-Bots operates in normal or attack mode, depending on the load it experiences. When under attack, the server processes only requests with

a valid HTTP cookie. If a request arrives without such a cookie, depending on the server's load, the server's kernel probabilistically drops the request or replies statelessly with a CAPTCHA puzzle. If a client request includes a valid CAPTCHA solution, the server's kernel replies with a cryptographic HTTP cookie that the client includes in subsequent requests to the server. Conversely, if a client includes invalid CAPTCHA solutions in too many requests to the server, the server blacklists the client and no longer considers any packets with that source IP address. Blacklisting reduces load, returning the server to normal mode. In normal mode, the server processes requests regardless of HTTP cookies, but not from blacklisted addresses. If legitimate clients later use those addresses, they could be locked out of service.

Kill-Bots runs on the server itself and creates additional load during an attack. The overhead can be greater during prolonged attacks if the server must also dynamically generate CAPTCHAs, which is a CPU-intensive task. Without CAPTCHA generation, Kill-Bots was reported to handle DDoS attacks of up to 6,000 requests per second without affecting response times [4].

III. DESIGN

This section describes Sentinel's goals and then each aspect of its design.

A. Goals

Sentinel's overall design goal was to overcome perceived shortcomings in existing methods for mitigating application-layer DDoS attacks against web servers. First, Sentinel should be transparently deployable without client, server, or network reconfiguration, including the case of server farms. Sentinel achieves this goal by operating as a network bridge that splices connections with clients and servers and performs stateful packet filtering. Second, Sentinel should completely offload attacks from servers and withstand large-scale attacks. Sentinel achieves this goal by enabling hardware acceleration with network processors. Third, Sentinel should fully support HTTP 1.1 features, such as persistent connections and pipelining. Sentinel achieves this goal by performing packet scrubbing and deep packet inspection. Fourth, Sentinel should not blacklist IP addresses indefinitely. Sentinel achieves this goal by estimating the load shed by the blacklist and clearing the blacklist when addition of the shed load would still place server load within normal range. Finally, Sentinel should enable diverse client authentication methods. For example, for a web server whose clients are securely registered out-of-band (e.g., bank), clients should be able to use for Sentinel credentials obtained during registration (e.g., passwords and tokens). Sentinel achieves this goal by providing an extensible framework for accommodating future CAPTCHAs, two-factor, or other authentication methods.

B. Modes of operation

Sentinel has two modes of operation, *normal* and *suspected_attack*. Sentinel switches between them based on

server load estimates and two configuration parameters, $L_{normal} < L_{attack}$. To obtain such estimates, Sentinel periodically sends an HTTP request and measures the response time of each server. Sentinel estimates the load on server i as an exponentially-weighted moving average L_i of i 's measured response times. When $\min_i L_i \geq L_{attack}$, Sentinel puts server i in *suspected_attack* mode. When $\max_i L_i \leq L_{normal}$, Sentinel places i again in *normal* mode.

C. Extensible in-network client authentication and collusion busting

In *suspected_attack* mode, Sentinel authenticates clients to filter out requests from bots. Sentinel gives to authenticated clients Sentinel cookies. If a request arrives without such a cookie and without a response to an authentication challenge, Sentinel replies with an authentication challenge. If a request arrives with a valid response to an authentication challenge, Sentinel replies with a cryptographic HTTP cookie that the client will include in future requests. Sentinel forwards to servers only requests with such a cookie. On the other hand, in *normal* mode, Sentinel does not challenge clients and forwards requests regardless of cookie.

The formats of the authentication challenge and response depend on the authentication method used. The challenge may contain a nonce to prevent replay. To facilitate use of future authentication methods, Sentinel separates in a specific module the functions for generating an authentication challenge and verifying an authentication response according to a particular method. The module also contains an initialization function that loads and initializes an authentication table.

Note that the client's IP address is insufficient to distinguish authenticated clients: Most clients reach web servers via a NAT router or web proxy [15] and consequently may share a same IP address with bots. Cookies permit distinguishing authenticated clients from other hosts that may exist behind the same middlebox.

A botmaster might have a human obtain a cookie from Sentinel and then distribute copies of the cookie to a large number of bots. To thwart such collusions, Sentinel maintains in an **HTTP cookie table (HCT)** the status of each cookie, including expiration time and number of outstanding requests. Sentinel can be configured to (a) allow use of a cookie only in requests from a particular IP address, and (b) drop a request containing a cookie if the request would cause the cookie's number of outstanding requests to exceed *cookie_req_lim*. Option (a) is undesirable if clients may use mobile IP, while option (b) requires Sentinel to analyze not only requests from clients but also responses from servers.

D. TCP SYN cookies, asynchronous authentication, and connection splicing

Sentinel combines three techniques to avoid committing to unauthenticated clients Sentinel's memory and any of the protected servers' resources.

First, when Sentinel receives from a client a request for a new connection with a protected server, if the client is not

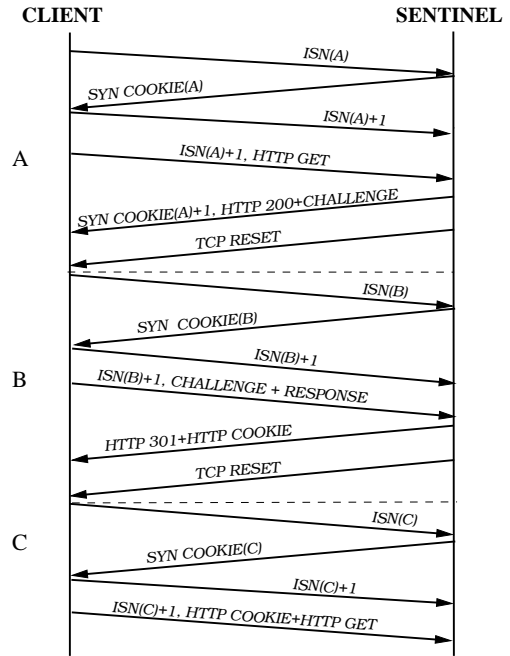


Fig. 2. SYN cookies and asynchronous client authentication

blacklisted, Sentinel replies on behalf of the server with a SYN cookie [2]. That is, Sentinel computes a cryptographic function of the client's and server's IP addresses and port numbers, the client's proposed maximum segment size, the current time, and a secret. Sentinel uses the result as the initial sequence number (ISN) it sends to the client. Sentinel does not store any values related to the client's connection request. When the client replies, its acknowledgment number should equal Sentinel's ISN plus one. Sentinel verifies this property cryptographically and recovers from the client's reply (instead of a table) the TCP state needed for the connection with the client. Sentinel may offload this function to the network interface card for hardware acceleration.

Second, if Sentinel needs to authenticate clients (*suspected_attack* mode), it does so asynchronously, as illustrated in Fig. 2. When Sentinel receives a request from an unauthenticated client, Sentinel uses the first connection A to convey Sentinel's authentication challenge to the client. To avoid holding TCP state, Sentinel forcibly closes this connection. Using a new connection B, the client presents its authentication response to Sentinel. If the response is valid, Sentinel redirects the client to the URL the client originally requested (HTTP 301 status code), provides the client an HTTP cookie, and again forcibly closes the connection (the client always starts a new connection after receiving status code 301). Finally, using another connection C, the client resends its original request, but now with the Sentinel cookie.

Third, when Sentinel receives a request in normal mode or with a valid Sentinel cookie on a connection C, Sentinel opens another connection C' with the protected server. Sentinel then splices together the client and server connections C and C', as shown in Fig. 3. The client's and Sentinel's ISNs in

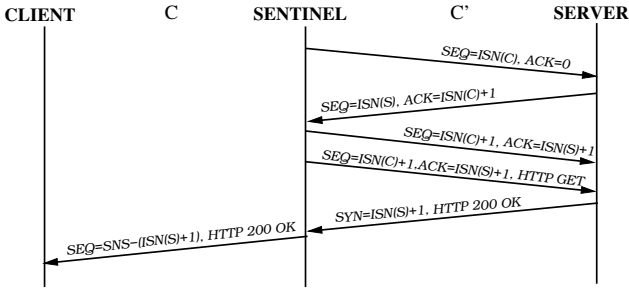


Fig. 3. TCP connection splicing

C are respectively ISN_c and $ISN_{sc} = SYN_cookie$. Sentinel's and the server's ISNs in C' are respectively $ISN_{ss} = ISN_c$ and ISN_s . To splice the two connections C and C' together, Sentinel computes C's *sequence number shift* (SNS): $SNS = SYN_cookie - ISN_s$. Sentinel allocates for the spliced connections an entry in Sentinel's **established connection table (ECT)**, and stores C's SNS in it.

ECT is a hash table indexed by the client's and server's IP addresses and port numbers. When Sentinel receives from a client a packet with acknowledgement matching an ECT entry, Sentinel adjusts the acknowledgement number by SNS and forwards the packet to the server, without checking for cookies. Conversely, when Sentinel receives a packet from the server, Sentinel adjusts the sequence number by SNS and forwards it to the client.

When the HTTP cookie used to create an ECT entry expires or Sentinel has not observed any valid packet in the respective connections for a period longer than a threshold, Sentinel forcibly closes the connections and destroys the ECT entry.

E. Blacklisting

Attackers might frivolously trigger authentications to exhaust Sentinel CPU cycles or network bandwidth, thus causing denial of service to legitimate users despite the defenses described in the previous subsection. To avoid such attacks, Sentinel keeps track of how many authentication challenges it has issued to each client IP address. Sentinel stores this information in a **Challenge Count Bloom Filter (CCBF)**. CCBF contains N cells, each with a b -bit counter num_chal_i . Each IP address corresponds to k of those cells, selected by k hash functions. Cell i is said to be *saturated* if $num_chal_i = S$, where $S = 2^b - 1$.

In *suspected_attack* mode, when Sentinel issues an authentication challenge to a client, Sentinel increments the corresponding CCBF counters by $\min(1, S - num_chal_i)$. Conversely, when Sentinel receives a valid response to an authentication challenge, Sentinel decrements the corresponding CCBF counters by $\min(\eta, num_chal_i)$. A value $\eta > 1$ is used to avoid long-term error accumulation.

If all CCBF counters corresponding to an IP address are saturated, Sentinel concludes that the IP address harbors a bot, because a human would have given up before making so many authentication errors. Sentinel *blacklists* the IP address and does not consider future packets from it. Sentinel may offload

this filtering rule to the network interface card for hardware acceleration.

During each time period T_{ave} , Sentinel also measures the number of requests that hit or miss the blacklist, respectively $B_{h,i}$ and $B_{m,i}$, and calculates the average server loads $L_{i,ave}$. By regression based on historical values, Sentinel estimates the *critical number* of blacklist misses $B_{c,i}$ that corresponds to response time $L_{i,ave} = L_{normal}$. Then, at the end of each time period, if $L_{i,ave} \leq L_{normal}$ and $B_{m,i} + B_{h,i} \leq B_{c,i}, \forall i$, Sentinel clears the blacklist, rehabilitating all IP addresses.

F. Stateful packet filtering, selective packet scrubbing, and deep packet inspection

Sentinel gleans from observed packets the state of each direction of a connection, including sequence and acknowledgment numbers and receive window size. Sentinel records this information in its ECT and uses it for stateful packet filtering [16]. If a packet has protocol field values that are inconsistent with the current state of the respective connection, Sentinel drops that packet.

TCP and IP allow packets, including client requests, to be split into multiple segments and fragments. Because reassembly occurs only at the destination, attackers may intentionally split packets to make it difficult for network-based defenses to identify patterns. For example, an attacker can split into two or more fragments a field containing a value that network intrusion detection systems (NIDSs) might use as a signature. Attackers can also intentionally make fragments overlap, such that even if a NIDS can detect patterns across fragments, the NIDS may miss a signature because the NIDS resolves the overlap differently from the destination. Alternatively, attackers can intentionally omit one or more fragments of each packet so as to fill reassembly memory, causing denial of service.

For efficiently dealing with such fragmentation attacks, Sentinel selectively performs packet scrubbing [17]. Sentinel drops fragments (but not segments) smaller than a certain threshold. Until Sentinel has all segments and fragments in a request's application-layer header, Sentinel timestamps and enqueues them by sequence number and offset. If contiguous segments or fragments overlap, Sentinel trims the older one before enqueueing the new one. If some segment or fragment of an application-layer header is still missing T_{out} time after Sentinel has received the first segment or fragment of that header, Sentinel drops all enqueued segments and fragments of that connection.

When a request's entire application-layer header is available, Sentinel analyzes it in place (deep packet inspection without copying). Sentinel uses the Knuth-Morris-Pratt (KMP) algorithm [18] to search for keywords and patterns in the header. The algorithm runs in $O(http_header_length + pattern_length)$ time, enabling high throughput. Analysis also determines the sequence number of the next application-layer header.

After analysis of an application-layer header, Sentinel forwards any segments or fragments with ending sequence number lower than the beginning of the next application-layer

header (i.e., Sentinel does not scrub the application-layer payload). If analysis does not reveal the sequence number of the next application-layer header then, after analysis of the current header, Sentinel can be configured to forward packets without scrubbing or further analysis until the connection is closed. The latter case corresponds to legacy HTTP 1.0 without persistent connections or pipelining [19], i.e., with only one request per connection.

IV. IMPLEMENTATION

This section discusses Sentinel’s implementation, including conventional and hardware-accelerated versions.

A. Conventional version

We implemented Sentinel as a Linux (v2.6.19.2) kernel module using the ebttables link-level extension of the netfilter framework [20]. We added Sentinel as a netfilter target. Users can enable Sentinel at compile time using the kernel configuration tools (e.g. make menuconfig) and load the module during runtime. Sentinel uses a softirq thread to examine each incoming packet in an `sk_buff` data structure. The thread communicates directly with network interfaces (e.g., Ethernet drivers) to transmit frames.

The netfilter link-level extension is configured using the ebttables userspace library. This library loads the authentication table into a memory-mapped segment shared with the kernel. The organization of the authentication table depends on the authentication module configured for Sentinel. Therefore, other Sentinel components treat this memory segment as opaque data.

B. Hardware-accelerated version

Although the conventional version described above filters packets in the kernel, close to the network interfaces, in softirq context, it still uses significant system resources (e.g., CPU cycles and buffer space). Thus, even blacklisted attackers can cause denial of service to legitimate clients, because Sentinel spends significant resources to discard their packets. On the contrary, in the hardware-accelerated version, packets from blacklisted clients are discarded by network interface hardware. Thus, Sentinel can withstand attacks of much larger scale. The following paragraphs discuss the hardware we used for acceleration.

1) *Netronome NFE-I8000*: The Netronome Flow Engine (NFE) I8000 platform is a network acceleration card that contains an Intel IXP2855 network processor unit (NPU) and on-card memory (Fig. 4). The NPU comprises an XScale microprocessor and sixteen programmable microengines for on-card packet classification. The NFE platform provides up to 768 MB of RDRAM, 40 MB of QDR SRAM and Ternary Content Addressable Memory (TCAM) space. The card has 4 SFP network ports and is connected to the host through a PCI Express bus with 4 active lanes providing 2GB/s aggregate capacity.

The NPU processes Ethernet frames arriving at the NFE ports by matching them consecutively against the NFE’s flow

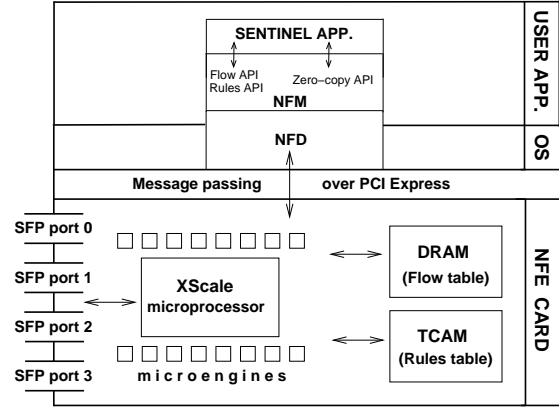


Fig. 4. Structure of the NFE Platform

and rules tables. The flow table contains actions for existing connections. If a packet does not belong to an existing flow, it is matched against the rules table, which specifies actions for packets from future flows. If the NPU finds a match, it copies the rule back to the flow table (to process further packets from the same flow) and executes the specified action. The NPU can execute one of the following actions on a frame: drop, reject (drop and send TCP reset message), pass (forward to all NFE ports) or forward up to the host for further analysis.

2) *Software APIs*: The NFE platform comes with two software packages. The **Network Flow Driver (NFD)** is a set of Linux kernel modules and userspace libraries providing an API for low-level communication with the NFE over the PCI Express bus (see Fig. 4). The **Network Flow Manager (NFM)** uses NFD to provide high-level APIs for application-level packet classification:

- the **zero-copy packet access API** allows NFM applications to access packets that the NFE NPU forwards to the host. The API provides high performance by using zero-copy DMA mapping to make frame buffers available to userspace applications.
- the **flow and rules APIs** allow NFM applications to make decisions about packet classification on the application level and pass them to the NFE platform to enforce. The flow API provides interaction with the NFE flow table (existing flows) and the rules API is used to populate the NFE rules table (future flows).

Sentinel’s NFM implementation uses the zero-copy packet access API to allocate and deallocate packet buffers and to send and receive frames through the NFE card. In suspected attack mode, when a client is blacklisted, Sentinel uses the NFM rule and flow APIs to instruct the NFE NPU to drop packets from the client’s IP address in existing and future flows. Dropping malicious packets in the NFE card significantly reduces the number of packets reaching the host and the resources used per bot connection.

C. Authentication modules

To test Sentinel, we implemented two authentication modules using different authentication methods.

The first module uses CAPTCHAs to distinguish bots from human clients. CAPTCHA images with distorted text containing each approximately 2,000 bytes are stored in the authentication table. Each table entry also contains fields with the respective solution, the time when the puzzle was sent to a client, and whether a solution has been received. The module prepares the challenge as two back-to-back packets. The module rejects solutions received too long after the challenge was sent or received after another solution for the same challenge has been received.

The second module uses HTTP digest authentication (HTTP 401 status code) [21] to authenticate registered users. The authentication table contains usernames, passwords, time of last challenge, and number of successive failures. It is organized as a hash table with username as the key. The challenge preparation function creates a single packet about 200 bytes long. The challenge includes a nonce and a message authentication code (MAC) that is a cryptographic function of the concatenation of the nonce, current time, client address, and a secret. The response includes the username and both challenge and response. If the user’s last challenge was more than *failure_reset* time before, the response verification function resets the user’s number of successive failures. If the user’s number of successive failures is greater than *max_failures*, the function returns an error. Otherwise, the function verifies the challenge and response and updates the user’s time of last challenge and number of successive failures. Finally, the function returns.

V. PERFORMANCE

We measured two metrics to evaluate Sentinel’s benefits during attack: (a) response time for legitimate client requests and (b) bridge CPU utilization. Ideally, Sentinel would preserve response time for legitimate clients even when bot attack rates increase. Moreover, Sentinel would ideally preserve low CPU utilization even as bot attack rates increase. Low utilization suggests the ability to handle attacks of even larger scale. We report only results obtained with the HTTP digest authentication method. The results with the CAPTCHA method were similar and are omitted due to page limitations.

A. Experimental Setup

Fig. 5 shows our experimental setup. We connected Sentinel, a web server and a gigabit Ethernet switch using 1 Gbps links. We also connected to the switch a legitimate client and seven attacker hosts using 100 Mbps links. Sentinel’s NFE card was configured to intercept TCP traffic and forward it to an application running on the host for classification and inspection.

Sentinel was implemented on a Dell Dimension 9200 configured with Pentium Core Duo at 1.6GHz and 2GB RAM, NFE-I8000 card, PCI Express bus, and Fedora Core 5 operating system. We turned off one of Sentinel’s CPU cores to facilitate performance measurement. The web server was Apache 2.2.0 running on an IBM ThinkCentre with Pentium 4 CPU at 3 GHz and 512MB RAM. The two attackers were,

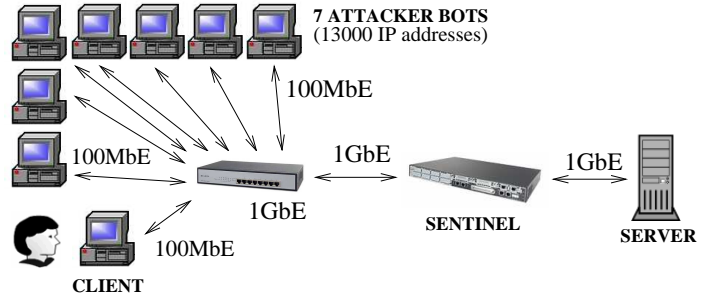


Fig. 5. Experimental Setup for Flooding Attacks

respectively, one IBM ThinkCentre with Pentium 4 at 3GHz (512 MB RAM, Fedora Core 3) and six Dell Dimension 4550 machines with Pentium 4 at 2.4 GHz (256 MB RAM, Fedora Core 5). The legitimate client was a Dell Dimension 4550 with Pentium 4 at 2.4 GHz, 256 RAM and Fedora Core 5. The gigabit Ethernet switch was an HP ProCurve 1800-8G.

We developed for the experiments a flooding utility, *asyncflood*, that ran on the attacker hosts. It uses 13,000 aliased IP addresses and creates asynchronous TCP connections to the web server at high rates (up to 10,000 connections per second on a single attacker). When a connection is established, the utility sends an HTTP GET request for the web server root index page. The utility ignores HTTP replies and TCP reset messages.

B. Results

1) *Response time for legitimate client requests:* We measured response times in three scenarios: (a) direct DDoS attack on the web server without Sentinel (baseline case), (b) web server protected by Sentinel’s conventional implementation, and (c) web server protected by Sentinel’s hardware-accelerated version. Blacklisting allows Sentinel to drop bot-originated packets as early as possible and can affect performance under attack. Therefore, we measured response times in (b) and (c) both before Sentinel starts blacklisting and after it blacklists all attackers.

In the baseline case (without Sentinel), we measured response times up to 100 s (average 15-20 s) for DDoS attack rates between 5,000 and 8,000 requests per second. In the cases of the conventional and hardware-accelerated implementations before blacklisting, we observed response times up to 54 s (average 20-30 s) for attack rates between 24,000 and 120,000 requests per second.

Fig. 6 shows response times for legitimate client requests measured for the Sentinel conventional and hardware-accelerated implementations after blacklisting. The figure shows measurements for attack rates between 24,000 and 120,000 requests per second and includes 90% confidence intervals.

Fig. 6 shows that Sentinel’s conventional implementation, combined with blacklisting, significantly mitigates the effect of DDoS attacks on the response time experienced by legitimate clients. Their response time was unaffected for DDoS attack

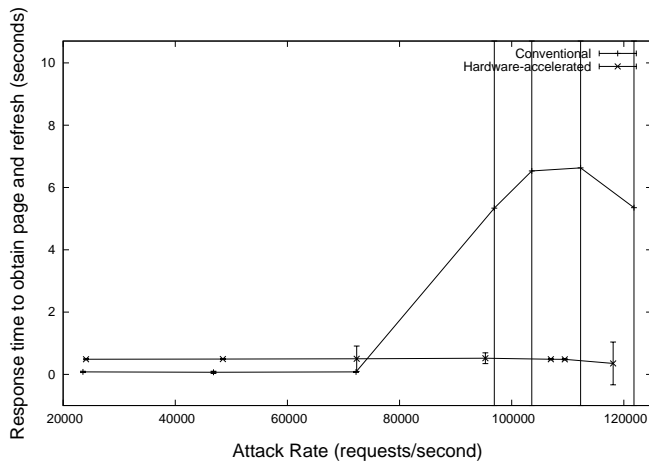


Fig. 6. Response time for legitimate client requests as DDoS attack rates increase (after blacklisting)

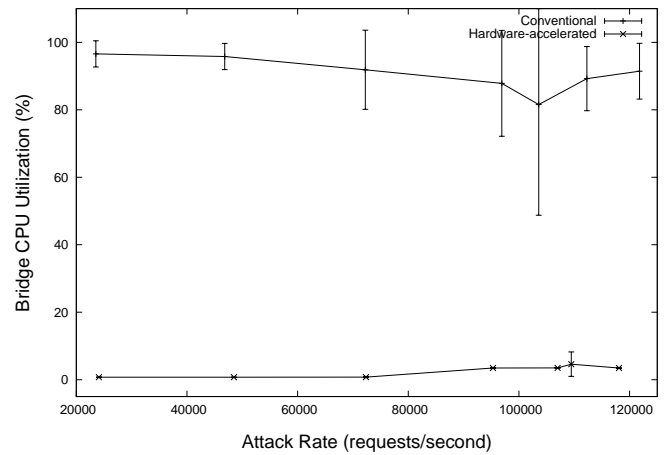


Fig. 8. Bridge CPU utilization as DDoS attack rates increase (after blacklisting)

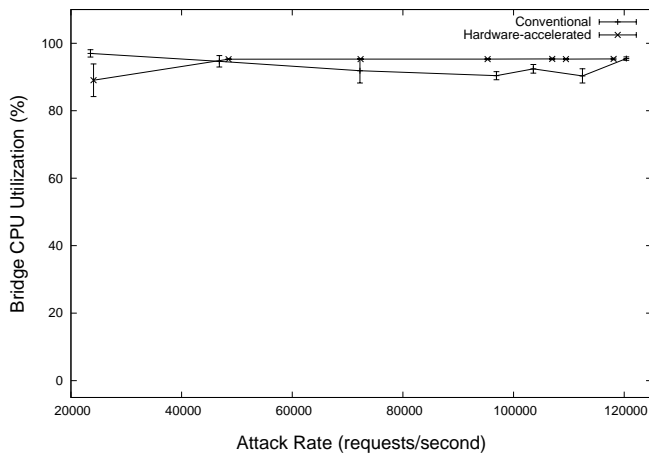


Fig. 7. Bridge CPU utilization as DDoS attack rates increase (before blacklisting)

rates up to 72,000 requests per second. However, larger DDoS attack rates caused marked increase in response time.

The figure also shows that Sentinel’s hardware-accelerated version provides even stronger benefits. The response time experienced by legitimate clients was largely unaffected over the entire range tested (up to 120,000 requests per second).

2) *Bridge CPU utilization*: Fig. 7 shows the bridge CPU utilization of Sentinel’s two versions before blacklisting (90% confidence intervals). CPU utilization exceeds 90%, explaining the high response times experienced by legitimate clients: Sentinel’s CPU is unable to keep up with packet arrival, and queuing delays and timeouts increase.

Fig. 8 compares the bridge CPU utilization of the two implementations after blacklisting (90% confidence intervals). The hardware-accelerated implementation barely had any CPU utilization over the entire range tested. This is explained by the filtering of the attackers’ packets by the network processor card, without consuming any resources of the host. In comparison, Sentinel’s conventional implementation processes all

blacklisted packets in the softirq handler before discarding them, thus consuming host CPU cycles for every malicious packet.

VI. RELATED WORK

Sentinel overcomes several limitations and refines algorithms and data structures used in Kill-Bots [4]. Kill-Bots implicitly assumes that each HTTP request uses a separate TCP connection. This is true only for legacy HTTP 1.0. On the contrary, Sentinel works correctly with HTTP 1.1, and in particular persistent connections and pipelining. The latter require Sentinel to perform deep packet inspection and parsing HTTP headers, which Kill-Bots doesn’t do. Kill-Bots also implicitly assumes that attackers do not spoof fields other than the IP address or maliciously fragment packets. On the contrary, Sentinel performs stateful packet filtering and packet scrubbing to avoid such attacks. Unlike Sentinel’s CAPTCHA module, Kill-Bots does not detect if a solution for a puzzle has already been received; consequently, bots can guess multiple times. Also, unlike Sentinel, Kill-Bots drops requests from blacklisted IP addresses even after malicious clients stopped using them.

Blacklisting based on IP addresses, as done by Kill-Bots and Sentinel, can cause false positives. The source address in a packet received by a server can be that of a middlebox, such as a network address translator (NAT) or proxy. A middlebox can serve both bots and legitimate clients; blacklisting blocks access by both. Moreover, many legitimate clients have dynamic IP addresses and could be assigned a blacklisted address. In an extensive study, Casado and Freedman [15] found that, from the point of view of a server, client address modifications are very slow, taking on the order of days. They also found that about 60% of the clients used NATs and 15% used proxies. However, the number of clients using a particular NAT is, in overwhelming majority, very small (one or two computers) and located in the same place. On the contrary, proxies tend to have more clients (ten or more) and those tend to be more

distributed. Thus, collateral damage from IP-based blacklists would tend to concentrate on the latter, which are only 15% of all clients. Moreover, clients whose proxy is blacklisted may be able to use another proxy (there are many free available).

There are several proposals involving solution of puzzles by clients. Juels and Brainard [22] proposed computationally expensive cryptographic puzzles. However, such a defense may be ineffective against botnets, because the latter can effectively have unlimited computational power. CAPTCHA alternatives using speech [23] or facial features [24] instead of text have also been proposed.

WebSOS [25] is another proposal for protecting web servers from DDoS attacks, using CAPTCHA puzzles and network overlays. Compared to Sentinel, WebSOS is harder to implement because it requires changes in network router configurations, and the Web server must be aware of the WebSOS architecture.

Speak-Up [26] proposes that a server instruct its legitimate clients to generate additional traffic to crowd out traffic from automated bots. Such an approach, however, could cause “bandwidth wars” in which botnets could have an advantage because they have a larger pool of nodes to operate from.

A complementary approach consists in trying to detect and eliminate bots, rather than mitigating their effects. Proposals along these lines include identifying botnets or bots by considering their anomalous IRC behavior ([27],[28]), DNS traffic patterns [29], DNS blacklist (DNSBL) queries [30], or general network traffic patterns ([31],[32]). However, isolating and removing botmasters, C&C servers, and bots is often difficult because botnets typically span multiple administrative and political boundaries.

VII. CONCLUSION

Effective defenses against DDoS attacks that deplete resources at the network or transport layers have been deployed commercially. Therefore, DDoS attacks increasingly use normal-looking application-layer requests to waste server CPU or disk capacity. CAPTCHAs attempt to distinguish bots from human clients and are often used to avoid such attacks. However, CAPTCHAs themselves consume resources and frequently are defeated. Kill-Bots reduces CAPTCHA overhead by pushing client authentication into the kernel. However, Kill-Bots requires kernel modifications, which can be infeasible. We described the design, implementation, and performance evaluation of Sentinel, a network device that overcomes several limitations in Kill-Bots. Sentinel can be easily deployed as a bridge in front of server farms, modularly accepts a variety of present and future authentication schemes, and can use network processors to accelerate authentication. Our experiments demonstrated that Sentinel greatly reduces the impact of DDoS attacks on the response time experienced by legitimate clients. Benefits are especially large when Sentinel uses hardware acceleration, because Sentinel’s CPU utilization remains low even as attack rates increase. We tested Sentinel with CAPTCHAs and HTTP digest authentication.

We are currently evaluating a replacement module with more sophisticated client authentication methods.

Acknowledgements

This project was funded in part by The Technology Collaborative through a grant from the Commonwealth of Pennsylvania, Department of Community and Economic Development.

REFERENCES

- [1] “Arbor Networks Peakflow SP,” <http://www.arbornetworks.com/en/peakflow-sp.html>.
- [2] D. Bernstein, “SYN cookies,” <http://cr.yo.to/syncookies.html>.
- [3] L. von Ahn, M. Blum, N. Hopper, and J. Langford, “CAPTCHA: Using hard AI problems for security,” in *Proc. Eurocrypt*, 2003, pp. 294–311.
- [4] S. Kandula, D. Katabi, M. Jacob, and A. Berger, “Botz-4-Sale: Surviving organized DDoS attacks that mimic flash crowds,” in *Proc. 2nd Symposium on Network Systems Design and Implementation (NSDI)*, Boston, MA, 2005.
- [5] M. A. Rajab, J. Zarfoss, F. Monrose, and A. Terzis, “A multifaceted approach to understanding the botnet phenomenon,” in *Proc. 6th ACM SIGCOMM Conference on Internet Measurement*, Rio de Janeiro, Brazil, 2006, pp. 41–52.
- [6] M. Drewniak, “Michigan man gets 30 months for conspiracy to order destructive computer attacks on business competitors,” <http://www.usdoj.gov/criminal/cybercrime/araboSent.htm>, August 2006.
- [7] “The botnet trackers,” <http://www.washingtonpost.com/wp-dyn/content/article/2006/02/16/AR2006021601388.html>, February 2006.
- [8] A. Turing, “Computing machinery and intelligence,” *Mind*, vol. 59, no. 236, pp. 433–460, 1950.
- [9] G. Mori and J. Malik, “Breaking a visual CAPTCHA,” <http://www.cs.sfu.ca/~mori/research/gimpy/>.
- [10] “Breaking CAPTCHA without using ocr - a new technique,” http://www.puremango.co.uk/cm_breaking_captcha_115.php, December 2007.
- [11] T. Claburn, “Yahoo’s CAPTCHA security reportedly broken,” <http://www.informationweek.com/news/showArticle.jhtml?articleID=205900620>, January 2008.
- [12] G. Keizer, “Spammers’ bot cracks Microsoft’s CAPTCHA,” <http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9061558>, February 2008.
- [13] J. Leyden, “Spammers crack Gmail CAPTCHA,” http://www.theregister.co.uk/2008/02/25/gmail_captcha_crack/, February 2008.
- [14] M. May, “Inaccessibility of CAPTCHA,” <http://www.w3.org/TR/turingtest/>, World Wide Web Consortium (W3C), Tech. Rep., November 2005.
- [15] M. Casado and M. Freedman, “Peering through the shroud: The effect of edge opacity on IP-based client identification,” in *Proc. 4th Symposium on Network Systems Design and Implementation (NSDI)*, Cambridge, MA, 2007.
- [16] G. van Rooij, “Real stateful TCP packet filtering in IP Filter,” Invited talk in the 10th USENIX Security Symposium, August 2001.
- [17] M. Handley, V. Paxson, and C. Kreibich, “Network intrusion detection: Evasion, traffic normalization and end-to-end protocol semantics,” in *Proc. 10th USENIX Security Symposium*, 2001, p. 9.
- [18] D. Knuth, J. H. Morris, and V. Pratt, “Fast pattern matching in strings,” *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1973.
- [19] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, “Hypertext Transfer Protocol – HTTP/1.1,” <http://www.w3.org/Protocols/rfc2616/rfc2616.html>, June 1999.
- [20] “Ebttables,” <http://ebtables.sourceforge.net/>.
- [21] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart, “HTTP authentication: Basic and digest access authentication,” <http://www.ietf.org/rfc/rfc2617>, June 1999.
- [22] A. Juels and J. Brainard, “Client puzzles: A cryptographic countermeasure against connection depletion attacks,” in *Proc. Network and Distributed Security Systems (NDSS)*, 1999, pp. 151–165.
- [23] G. Kochanski, D. Lopresti, and C. Shih, “A reverse Turing test using speech,” in *Proc. 7th International Conference on Spoken Language Processing*, Denver, CO, September 2002, pp. 1357–1360.
- [24] Y. Rui and Z. Liu, “ARTIFACIAL: Automated reverse turing test using facial features,” in *Multimedia*, November 2003.

- [25] W. Morein, A. Stavrou, D. Cook, A. Keromytis, V. Misra, and D. Rubenstein, "Using graphic Turing tests to counter automated DDoS attacks against web servers," in *Proc. 10th ACM International Conference on Computer and Communications Security (CSS)*, October 2003, pp. 8–19.
- [26] M. Walfish, M. Vutukuru, H. Balakrishnan, V. Karger, and S. Shenker, "DDoS defence by offense," in *Proc. ACM Special Interest Group in Data Communications (SIGCOMM)*, Pisa, Italy, September 2006.
- [27] J. Binkley and S. Singh, "An algorithm for anomaly-based botnet detection," in *Proc. 2nd Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI)*, 2006.
- [28] J. Goebel and T. Holz, "Rishi: Identify bot contaminated hosts by IRC nickname evaluation," in *Proc. 1st Workshop on Hot Topics in Understanding Botnets*, Cambridge, MA, April 2007.
- [29] H. Choi, H. Lee, H. Lee, and H. Kim, "Botnet detection by monitoring group activities in DNS traffic," in *Proc. 7th IEEE International Conference on Computer and Information Technology*, 2007, pp. 715–720.
- [30] A. Ramachandran, N. Fearnster, and D. Dagon, "Revealing botnet membership using DNSBL counter-intelligence," in *Proc. 2nd Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI)*, 2006, pp. 49–54.
- [31] A. Karasaridis, B. Rexroad, and D. Hoefflin, "Wide-scale botnet detection and characterization," in *Proc. 1st Workshop on Hot Topics in Understanding Botnets*, 2007.
- [32] C. Livadas, R. Walsh, D. Lapsley, and W. T. Strayer, "Using machine learning techniques to identify botnet traffic," in *Proc. 2nd IEEE LCN Workshop on Network Security (WoNS)*, November 2006.