A Two-Stage Defense Against HTTP Flood Attacks: From Botnet Filtering to Browser-Emulation Mitigation

Karel Panchártek

Department of Computer Science, Faculty of Science, Palacký University Olomouc, Czech Republic

Abstract

This paper proposes a defense against HTTP flood attacks built on top of a packet filter and a lightweight auxiliary HTTP server. The work is at an early stage, and our aim here is to present the core idea rather than an exhaustive solution. Unlike prior SDN-based defenses, the proposed system acts as a single entity behind one public IP address and can be scaled across machines on a private network. The defense is organized into two stages according to the sophistication of the attack. The first stage, which we consider the core layer, targets requests originating from IoT botnets and relies on a token-based HTTP redirect that such botnets are unable to follow. The second stage targets more sophisticated attacks that use browser emulation by issuing a scalable computational (and, optionally, human verification) task. In preliminary experiments with the Bonesi attack simulator, the system reduced the average response time of a server under attack from roughly 4 seconds to about 200 milliseconds for an unverified client and to about 50 milliseconds for an already verified one. We further outline directions, such as HTTPS support and distributed deployment, that we intend to develop in an extended version of this work.

Keywords

DDoS, HTTP flood, botnet, HTTP redirect, packet filtering, network intrusion mitigation

1. Introduction

Web services are among the most exposed assets on the Internet, and distributed denial-of-service (DDoS) attacks against them keep growing in both frequency and volume. Recent industry reports document a sharp rise: application-layer (HTTP) DDoS attacks increased by more than 100% year over year, with large botnets responsible for the majority of HTTP flood incidents [1]. Because such attacks operate at the application layer and mimic legitimate traffic, they are notoriously hard to detect and filter [2]. Keeping a web server available under such conditions, ideally with modest resources, is therefore a practically important problem.

An HTTP flood floods a web server with requests that resemble legitimate ones, degrading its quality of service for real users – from slower responses, through occasional outages, to complete unavailability (denial of service, DoS). Such attacks are most often distributed (DDoS) and carried out by botnets: groups of poorly protected IoT devices controlled by an attacker. More customized attacks that emulate a web browser are also possible.

We propose a system that mitigates HTTP floods using a lightweight *auxiliary web server* (AX) together with packet-filtering (firewall) rules. The defense works in two stages: botnets are first neutralized by an HTTP redirect they cannot follow, and devices that do follow it (including browser emulations) are then given a scalable computational task and a CAPTCHA. Clients are placed on an allowlist or a denylist by source IP address. Crucially, the design also makes a successful attack progressively harder and more expensive to mount, which lowers the attacker's determination to continue. This paper presents the core idea and a first single-machine evaluation with the Bonesi simulator [3]; a thorough design-space study and a distributed evaluation are left to an extended version.

ITAT 2026: Information Technologies – Applications and Theory, 2026

karel.panchartek@upol.cz (K. Panchártek)

0009-0000-7883-2550 (K. Panchártek)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-09

CEUR-WS.org/Vol-4323/paper41.pdf

1.1. HTTP flood attacks

Within the network model, HTTP flood attacks target the application layer. A prerequisite for the attack is an established TCP connection, which means that packets must be routed back to the requesting device. Spoofing the source IP address is therefore difficult, although in real-world attacks spoofing is often unnecessary.

There are several variants of HTTP flood attacks that bypass different levels of defense. The simplest one repeatedly sends identical requests from a single machine; this is essentially a basic DoS attack whose intensity is limited by the connection speed and the computational power of the attacking machine. Even so, since a server usually performs significantly more work per request (for example, querying a database) than the attacker, such an attack can still be effective. When requests come from many devices, the attack becomes a distributed (DDoS) one of higher intensity and greater chance of success.

A more advanced level makes flood requests harder to distinguish from legitimate ones by varying the parameters used to identify attack traffic, such as IP, TCP, or HTTP headers and request parameters. Variability also defeats caches that would otherwise serve repeated identical requests; for example, the URL of a request may carry random parameters, as in `example.com?cahd=dhaskdjf`. An even more advanced level uses web-browser emulation, which makes a request sender harder to tell apart from a genuine client. Such emulations often execute JavaScript and mimic real browser behavior, submit forms, or emulate mouse movements [4], so defenses based on simple JavaScript tasks fail.

The HTTP protocol defines several request methods, such as GET, POST, OPTIONS, and DELETE [5]; the attacks above can use any of them, although GET and POST are the most common. HTTPS variants are also possible, as are attacks that target the TLS protocol directly or exploit vulnerabilities in particular server implementations.

Let us additionally mention the so-called Slowloris attack. It targets the limited number of TCP connections a server can handle: the attacker slowly opens connections and sends an incomplete HTTP request in each in order to keep it open as long as possible. Once the number of open connections exceeds the server's capacity, the server can no longer respond to legitimate clients. Network traffic does not increase significantly, so detection based on traffic growth fails [6]. We refer to this family of attacks as Slowloris. Whether to classify it as an HTTP flood attack is debatable, since it exploits an implementation limitation rather than overloading the server's computational resources.

1.2. Execution of an HTTP flood attack

To send HTTP requests, attackers exploit so-called zombie devices, whose owners are unaware that the devices run malicious software sending requests on the attacker's command. A particularly attractive subset are botnets: groups of IoT devices that, in general, have weak protection and can be easily enrolled. Their advantage for an attacker is their number and the fact that they operate under different IP addresses, which removes the need for IP spoofing and reverse routing: botnets establish TCP connections directly with their own addresses [7]. Their disadvantage is limited programmability and computational power.

According to [8], botnets are not able to follow an HTTP redirect, and [7] assumes that botnets do not implement the whole HTTP protocol. The first stage of our defense is therefore built on the inability to follow an HTTP redirect. This assumption is further supported by the fact that it is not desirable for an attacker, whose goal is to send as many requests as possible, to spend the botnet's scarce resources on parsing responses; the controlling malware must also remain simple because of limited RAM [9]. If an attacking device does follow the redirect, we are already dealing with a more advanced attack, which the second stage of the defense addresses. With the same computational budget, such an attack necessarily has lower intensity.

1.3. Focus of this paper

From the perspective of the request method, we focus on GET-request attacks over unencrypted HTTP. Solutions for other methods and for HTTPS are discussed but were not yet implemented. The proposed defense uses the packet-filtering (firewall) capabilities of the Linux system [10], but the concept applies to other platforms as well. In the experiments, we evaluate the impact on the availability of a server under attack by botnets that send a large number of HTTP GET requests from a large number of IP addresses.

2. Related work

Defending against application-layer DDoS attacks is an active research area; broad overviews are given by several surveys [2, 11, 12, 13]. Their common conclusion is that, because HTTP floods operate at the application layer with requests that look legitimate, volume-based filtering is largely ineffective and detection is hard.

One line of work tries to tell attackers from clients by analysing traffic behavior. Xie and Yu [14] model normal browsing behavior of popular websites and flag deviations, while later methods apply machine learning, including in software-defined networks [15]. A fundamental limitation, however, is that modern attacks deliberately vary the parameters used as detection features (Section 1.1), which degrades feature-based classifiers. Slow-rate attacks such as Slowloris are a separate, well-studied family that exploits implementation limits rather than sheer volume [16].

A different line, closest to ours, separates clients from bots with an active challenge rather than passive classification. Lim et al. [8] propose an SDN system that relies on botnets being unable to follow an HTTP redirect; an attacker that reveals the redirect target is countered with a further address and a CAPTCHA based on hard AI problems, and an attacking address is flagged after repeatedly failing to follow the redirect (a threshold of three failures). Gonçalves et al. [7] build on this idea: a component called the *Redirector* sends a redirect that changes the server's IP address to one of 10 chosen addresses, recognizing legitimate clients when they follow it (allowlist) and attackers when the redirect times out (denylist). Their defense is built in an SDN and its load can be distributed across autonomous systems besides the main one (their Local System); using six such systems over VPN, they report a drop of a real client's response time during an attack from 6 s to 0.4 s, with the attack simulated by Bonesi at a configured 100 requests per second.

Our work follows this redirect-based line but addresses several of its weaknesses (use of multiple public addresses, a single non-replicated Redirector, and reliance on the redirect alone), as detailed in Section 3.

3. Proposed system

Our system builds on the redirect-based defenses of [7] and [8] but is designed to remove some of their weaknesses. We first state the design rationale, then describe the architecture, the two-stage client verification, a concrete configuration, and finally optional refinements.

3.1. Design rationale

Our system acts as a single entity in front of the protected server and operates under one public IP address even when distributed across several machines, so internal traffic runs in a private network separate from the public one. By contrast, the redirection addresses in [7] no longer correspond to a DNS record, which raises doubts about the server's identity (aggravated under HTTPS) and, given the scarcity of IPv4 addresses, may be costly. A small set of redirection addresses can also be traced by an attacker and overcome with a probability set by their number: with 10 addresses, every tenth bot could randomly hit the correct one, lowering the attack intensity only to 10%. In our system every redirect instead carries a unique token from a large space, so the chance of guessing it is negligible.

In [7] the Redirector exists in a single copy in the main system. Our experiments show this component is heavily loaded, so a single copy limits scalability; in our design it can be replicated as needed.

Beyond the redirect, our system can issue additional client tasks, so it does not rely on the redirect alone but anticipates more sophisticated attacks and a need to mitigate them. Overall, it can dramatically reduce, or even eliminate, the impact of simpler bot-driven attacks while also raising the threshold for a successful, more sophisticated one. This second feature restricts service to web-browser users, which we consider both the most common case and the main target of such attacks. A further advantage shared with any redirect-based defense is immunity to attacks that vary header and request parameters – the very variability that complicates learning-based defenses.

3.2. Architecture and components

The goal of the defense is to keep the server available to legitimate clients, under the assumption that botnet devices cannot follow an HTTP redirect. It is realized by three parts: an *attack-detection program*, a *firewall* (packet filtering and modification), and an *auxiliary web server*. The firewall operates on the public-network interface; the other parts may run on the same or on separate machines (separate machines are preferable in a real deployment, but the particular architecture is not important for conveying the idea).

The detection program can manipulate the firewall: when it detects an attack, it installs a rule that redirects traffic from unverified clients to the AX. The particular detection method is not essential – a simple rise in packets per time unit can serve as a trigger, as can more sophisticated, e.g. learning-based, methods. The AX verifies client legitimacy and can also manipulate firewall rules. A client judged legitimate is added to the *allowlist* and its requests are forwarded to the protected server; a client judged an attacker is added to the *denylist*, its connection is terminated, and further traffic from that address is dropped. The denylist is not strictly necessary, but during an attack it gradually reduces the load on the system.

3.3. Two-stage client verification

The first stage tests whether the client can correctly follow an HTTP redirect. The client is redirected to a URL containing a unique token (the response carries a `3xx` header as in [5]); the token is bound to the client's source IP address and is later checked. Botnet devices that cannot follow the redirect are filtered out here: if the redirect is not followed after repeated requests, or the returned token is invalid, the address is classified as an attacker. If the only goal is to filter out botnets – often sufficient – a client that follows the redirect with a valid token can already be considered legitimate.

The second stage defends against more advanced attacks. The client must complete a task contained in a web page returned by the AX: a human verification test (e.g. a CAPTCHA), a computational task, or any other task that typical botnet devices cannot fulfill. A correct answer marks the client as legitimate (allowlist), an incorrect one as an attacker (denylist). Verification can stop at any time when the detection program reports the end of the attack or when the attack is judged repelled, e.g. based on the machine's load or on the rate at which addresses are added to the denylist.

Example configuration. The web server, AX, and detection program run on one machine; the web server listens on port 80/tcp and the AX on, e.g., 8000/tcp. During normal operation clients are served directly. After an attack is detected, the firewall redirects requests for the web server to the AX, which first replies with a redirect to a URL pointing back to itself and carrying a unique token. After validating the token, the AX either rejects the client or assigns the second task: a page with a CAPTCHA and a JavaScript problem – completing a given string with a few unknown characters so that the whole string hashes to a given value. A correct answer marks the address as legitimate, otherwise as an attacker.

3.4. Refinements and extensions

The design admits many refinements; we sketch the most important ones and leave a full treatment to the extended version. Both tasks can be parameterized and changed in real time, making the defense harder to adapt to – for instance, the location of the token URL within the redirect header can be varied, forcing the attacker into at least partial header parsing. A CAPTCHA alone can authenticate a human but may annoy users and can fail (as also noted in [8]), so a scalable computational task is a useful complement, optionally combined with further signals such as timing, mouse movements, or browser-only interfaces; a good example is the delayed-button measure of [9]. The computational task should be scalable (so its completion time can be estimated) and have many possible solutions (so answers cannot be precomputed into a dictionary), trading off attacker burden against client annoyance. On the AX side, keeping tasks and precomputed answers in memory enables fast responses even under intense attacks.

4. Experiments and results

The goal of the experiments was to verify that the proposed defense increases the resilience of the attacked server and preserves its availability for legitimate clients under attack. We deliberately keep the evaluation simple and run it on a single machine, so that the measured improvement reflects the defense itself rather than additional hardware; scaling across machines, as discussed in Section 3.1 and 3.2, can only raise the capacity further. This represents a conservative experimental setting, as a larger number of virtual machines within the VirtualBox environment could compete for the same processing resources. Deploying the components in parallel, either within the limitations of the VirtualBox environment or on separate physical hardware, is therefore expected to maintain or improve the observed performance. Below we describe the setup, the tools used to simulate attacks, the defense configuration, and the measurement methodology, and then report and discuss the results.

4.1. Experimental setup

Experiments were performed on two virtual machines, each allocated 8 GB of RAM and running Xubuntu 24.04, hosted by VirtualBox [17] on an Intel Core i5-2500 3.3 GHz CPU with 32 GB of RAM (also running Xubuntu 24.04). One machine generated both legitimate requests and the simulated attacks; the other ran the attacked web server, the AX, the detection program, and the firewall.

The web server (Apache2 [18] in its default configuration) served a PHP web application [19] backed by a MySQL database [20]. The AX was implemented with the Python HTTP server [21]; packet filtering and rule checking used the integrated Linux firewall [10]; and the detection program was a simple Python packet counter that flags an attack above 300 packets per half-second window.

Running the web server and its defense on a single machine was a deliberate choice: the purpose was to determine the resilience of the server machine under a fixed amount of processing power. Dedicating separate machines to the defense, as preferred in a real deployment, simply adds processing power and thus raises resilience.

4.2. Attack simulation

Several tools were used to simulate attacks. Wireshark [22] was used to observe the transmitted data. A very basic attack of repeated requests from a single machine was generated with `wget`; it was so weak that even without any defense it caused no noticeable delay. GoldenEye [23] caused delays but did not easily produce attacks from many source IP addresses (it opened multiple connections from one address on multiple ports, varying both the browser header and the GET parameters). The need for many source addresses was met by Bonesi [3], which can set different source IP addresses (from a pool of 50,000 in our case) and reach sufficient intensity (2,500 requests per second configured). At full intensity the response delay grew to 20 seconds or more. Bonesi sends requests to a single URL only,

which is a disadvantage mainly for machine-learning experiments. We set the working attack intensity to 2,500 IP packets per second, above which the system could no longer withstand the attack, to ensure a repeatable and reliable simulation.

Slowloris [24] and Tor's Hammer [25] were used to simulate Slowloris attacks. The firewall is a sufficient defense against this type: an attacking address is easy to recognize from the number of packets and the amount of data transferred (we used a threshold of 7 packets carrying fewer than 7,000 bytes).

4.3. Defense configuration

For the computational task we used the problem of completing a given eight-character string with three unknown characters (from the lowercase English alphabet) so that the SHA-512 hash of the complete string matches a given value. With these parameters the task is reasonably computable by brute force and took a client about 1 second on average. The task was implemented and validated separately; the botnet-mitigation experiments reported below do not include the more sophisticated attacker that would reach this stage.

The firewall configuration was as follows. First, requests whose source address is on the denylist are dropped. If no attack is detected, requests to both the web server and the AX are accepted. If an attack is detected, a request for the web server whose source address is not on the allowlist is redirected to the AX, otherwise it is accepted. The AX added an address to the denylist after three repeated requests without a token, or upon an invalid token (the latter did not occur in our experiments), and to the allowlist upon a correctly followed redirect with a valid token.

4.4. Measurement methodology

To evaluate availability and responsiveness, the `wget` program sent a request every 5 seconds from an address distinct from all attacking addresses, and we measured the time between sending the request and receiving the response (the *request duration*). To distinguish verified from unverified clients when the defense was active, every odd request required verification by the AX (unverified client), while every even request was served directly by the web server (already-verified client). The response time was measured in four situations: (I) no attack, defense off; (II) no attack, defense on; (III) attack, defense off; and (IV) attack, defense on. In addition, during the defense the number of client verifications processed by the AX and the number of denylisted addresses were monitored in 10-second intervals.

4.5. Results

With *no attack and the defense off* (situation I), the average request duration was 31 ms and the slowest request took 81 ms. With *no attack and the defense on* (situation II), the average duration was 59 ms for a not yet verified client and 34 ms for an already verified one (slowest 166 ms and 87 ms, respectively). Compared with situation I, verification roughly doubled the average duration (an increase of 28 ms, i.e. 90%), but responses remained very fast; the increase for an already verified client (about 10%) is negligible.

The decisive comparison is under attack. With the *attack and the defense off* (situation III), the average time to complete a request was 3.944 seconds: of 130 requests, 129 took longer than 1 second and 86 longer than 3 seconds, with the slowest taking 9.444 seconds.[1] This is almost 130 times the duration in situation I – two orders of magnitude slower. Figure 1 plots the request durations over time.

With the *attack and the defense on* (situation IV), the average duration fell to 196 ms for a not yet verified client and to 44 ms for an already verified one. Of 130 requests from an unverified client, only 10 took longer than 1 second and only 2 longer than 3 seconds (slowest 3.213 seconds); for a verified client the slowest was 130 ms. This is a dramatic improvement over situation III: the average duration dropped to less than 5% of the no-defense value for an unverified client and to slightly over 1% for a verified

[1]The attack intensity was 2,500 requests per second. The previously mentioned delay of 20 seconds or more was observed under the full intensity Bonesi was capable of.

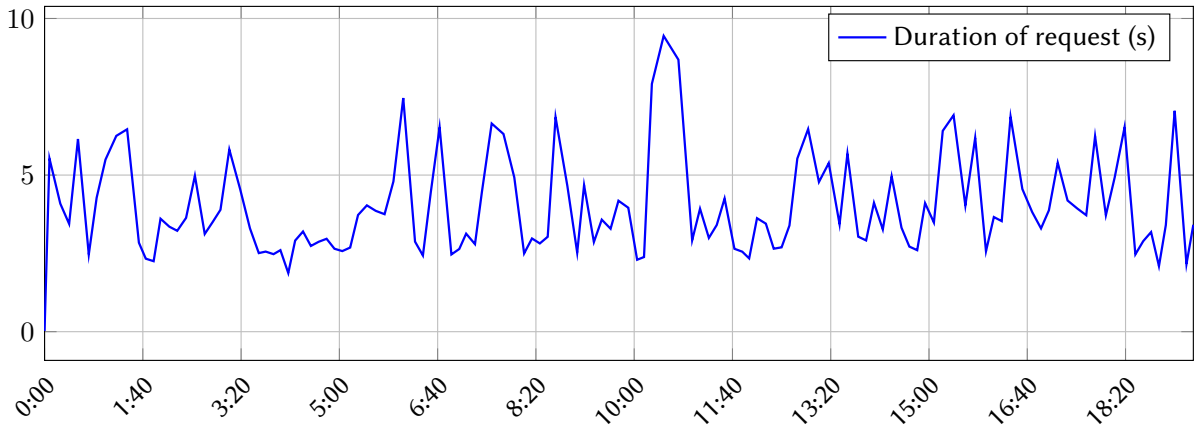


Figure 1: Request duration (seconds) over time during an attack with the defense off (situation III).

one (speed-ups of more than 20× and nearly 90×, respectively). In other words, the duration returns to about 330% (unverified) and 130% (verified) of the corresponding no-attack value in situation II. From the point of view of an unverified client the server is about three times slower than without an attack (196 ms vs. 59 ms, still good), while from the point of view of a verified client it is almost indistinguishable (44 ms vs. 34 ms). Figure 2 plots these durations.

Even some of the requests requiring verification before the attack is repelled are never slower than the average request without the defense. The number of requests taking longer than 3 seconds fell from 86 to 2. During the mitigation phase (1:33–5:31) the average duration for unverified clients was 706 ms – less than 18% of the no-defense value (a speed-up over 5.5×). The growth of the denylist over time and its per-second increment are shown in Figure 3.

Figure 3(a) shows the total number of denylisted addresses during the attack. Approximately half of the attacking addresses were denylisted within the first five and a half minutes – specifically, 22,502 out of 50,000 by 5:28. Most of the growth occurs within the first ten minutes, after which the total increases more slowly and gradually approaches a stable level.

Figure 3(b) shows the per-second increment, i.e., the rate at which new attacking addresses are discovered and added to the denylist. It confirms the trend observed in Figure 3(a) while providing a more detailed view of the discovery rate. The rate rises rapidly at the beginning of the attack, reaching a peak of approximately 120 newly denylisted addresses per second, and then gradually decreases as the attack progresses and fewer previously unseen attacking addresses remain. The short delay between the onset of the attack and the peak corresponds to the three request attempts required from each attacking address before it is denylisted.

4.6. Discussion

The results confirm that a redirect-based first stage is highly effective against botnet HTTP floods: it restores a verified client's responsiveness almost to its no-attack level and keeps even an unverified client well within usable limits, despite a deliberately under-powered single-machine setup. The denylist is central to this behavior. As attacking addresses are recognized and dropped by the firewall, the load on the AX falls, so the cost of verification is highest at the onset of the attack and decreases as mitigation proceeds (Figure 3); the defense is thus self-limiting rather than requiring resources proportional to the full attack volume.

The reported attack strength should be read with care. Although Bonesi was configured for 2,500 requests per second, packet inspection showed it did not reach this ceiling; the maximum rate actually observed at the AX was 668 requests per second. The configured strengths therefore overstate the true load, which also makes a direct comparison with the 100 requests per second reported in [7] unreliable, the more so given the different hardware. What the experiments do show is that, on a single modest

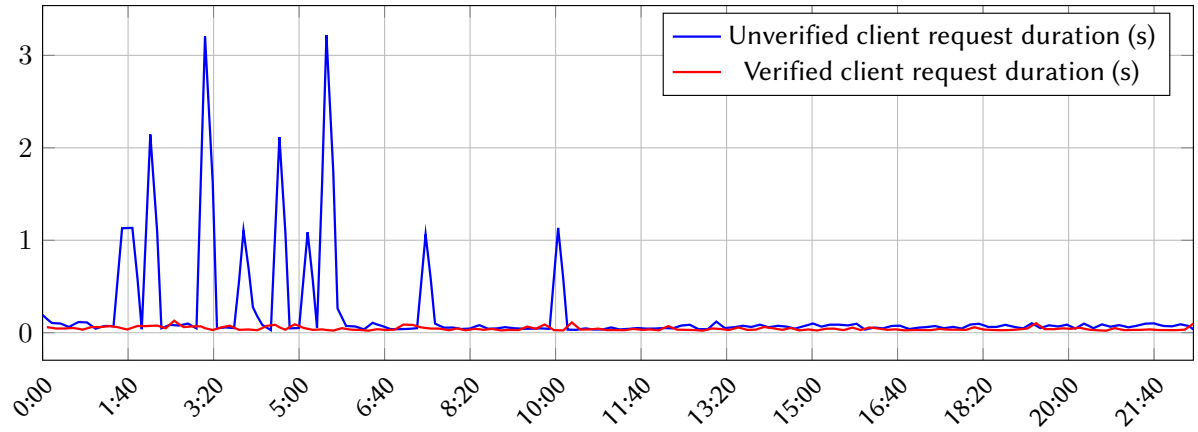
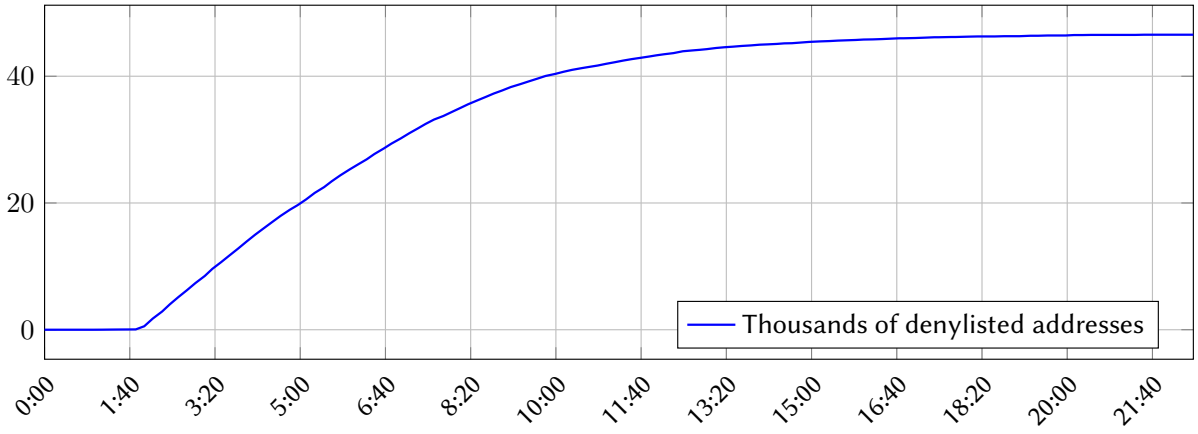
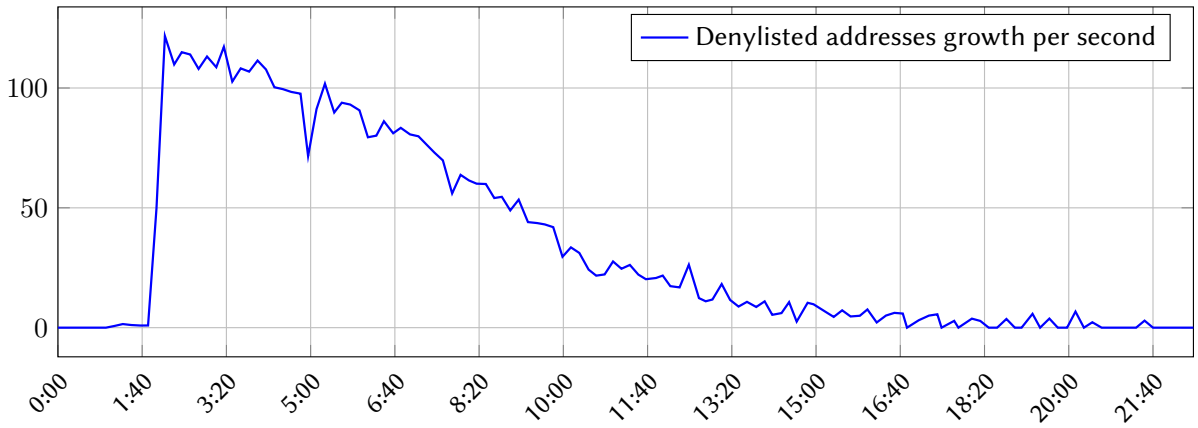


Figure 2: Request duration (seconds) over time during an attack with the defense on (situation IV), for an unverified and a verified client.



(a) Total number of denylisted addresses (thousands).



(b) Per-second growth of the denylist.

Figure 3: Build-up of the denylist over time during the attack.

machine, the defense copes with an attack from 50,000 distinct addresses and processes hundreds of verifications per second while keeping the protected server responsive.

In practical terms, the botnet-mitigation stage alone is cheap and, with the scaling discussed in Section 3.1, suitable even for larger servers; the second stage targets the rarer, more sophisticated attacks at the cost of restricting service to genuine browsers. A more rigorous, multi-machine and multi-tool evaluation is left to future work.

5. Conclusion and future work

We presented the core idea of a two-stage system that mitigates HTTP flood attacks by combining a token-based HTTP redirect with an optional scalable client task, all behind a single public IP address. The redirect cheaply filters out botnets that cannot follow it, while the second stage raises the cost of more sophisticated, browser-emulating attacks. Compared with prior redirect-based defenses [7, 8], the design avoids using multiple public addresses, allows the redirect component to be replicated, and explicitly anticipates attacks beyond simple botnets.

Even in a deliberately constrained single-machine setup, the results are encouraging. Under an attack of configured strength 2,500 packets per second, the average response time of the attacked server dropped from about 4 seconds without any defense to about 200 ms for a previously unknown client and about 50 ms for an already known one, and the defense processed up to roughly 700 requests per second. The solution suits small to medium servers with a not-too-wide traffic spectrum, whose administrators can tailor the second stage to more sophisticated attacks; the botnet-mitigation stage alone, with appropriate scaling, is suitable for larger servers as well.

We also note the present limitations, which set the agenda for future work. First, the evaluation uses a single machine and a single attack tool (Bonesi); because the attacker and the defense run on virtual machines hosted by one physical CPU, the measurements capture their simultaneous operation rather than true hardware parallelism. There is a need to build the distributed variant of the defense to characterize the system's true capacity in such assembly. Second, the second-stage task was implemented and validated only in isolation, so an end-to-end evaluation against an actual browser-emulation attacker remains open. Third, because clients are identified by IP address, clients sharing a NAT (a single public address) cannot yet be told apart and need a different treatment.

Beyond addressing these limitations, several directions are of practical interest. HTTPS is now far more common than plain HTTP, so the defense must be adapted to HTTPS servers and to attacks on the TLS handshake, ideally rejecting a client before the costly handshake completes. The design space of the second stage – task difficulty, DOM-tied and obfuscated problems, allowlist/denylist expiration, per-country filtering, and connection limiting toward the AX – deserves a systematic study. Finally, the defense is independent of the IP protocol version; as IPv6 adoption grows [26], confirming this experimentally and identifying IPv6-specific challenges is worthwhile. We intend to pursue these questions in an extended version of this work.

Acknowledgments

This work was supported by Grant No. IGA_PrF_2026_26 of the Internal Grant Agency of Palacký University Olomouc. I would like to express my gratitude to Martin Trnecka for his valuable assistance with the preparation and structuring of this manuscript. Furthermore, I would like to thank Jan Outrata for his insightful ideas regarding the interpretation of the results, as well as his contributions to the editing and phrasing of the text.

Declaration on Generative AI

During the preparation of this work, the author(s) used ChatGPT, DeepL and Grammarly in order to: grammar and spelling check. After using these tools, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] Cloudflare, DDoS threat report for 2025 Q1, https://blog.cloudflare.com/ddos-threat-report-for-2025-q1/, 2025. Accessed: 2026-06-29.

[2] K. Singh, P. Singh, K. Kumar, Application layer HTTP-GET flood DDoS attacks: Research landscape and challenges, Computers & Security 65 (2017) 344–372. doi:10.1016/j.cose.2016.10.005.
[3] M. Goldstein, BoNeSi – the DDoS botnet simulator, https://github.com/Markus-Go/bonesi, 2024. Accessed: 2024-06-21.
[4] MazeBolt, HTTP/s flood – with browser emulation, https://kb.mazebolt.com/knowledgebase/https-flood-with-browser-emulation/, 2024. Accessed: 2024-04-13.
[5] R. T. Fielding, M. Nottingham, J. Reschke, HTTP Semantics, Technical Report RFC 9110, RFC Editor, 2022. URL: https://www.rfc-editor.org/info/rfc9110. doi:10.17487/RFC9110.
[6] MazeBolt, Tor's hammer attack, https://kb.mazebolt.com/knowledgebase/tors-hammer-attack/, 2024. Accessed: 2024-04-13.
[7] D. S. M. Gonçalves, R. S. Couto, M. G. Rubinstein, A protection system against HTTP flood attacks using software defined networking, Journal of Network and Systems Management 31 (2023) 16. doi:10.1007/s10922-022-09704-1.
[8] S. Lim, J. Ha, H. Kim, Y. Kim, S. Yang, A SDN-oriented DDoS blocking scheme for botnet-based attacks, in: Proceedings of the Sixth International Conference on Ubiquitous and Future Networks (ICUFN), 2014, pp. 63–68. doi:10.1109/ICUFN.2014.6876752.
[9] H. Mirvaziri, A new method to reduce the effects of HTTP-Get flood attack, Future Computing and Informatics Journal 2 (2017) 87–93. doi:10.1016/j.fcij.2017.07.003.
[10] The netfilter.org project, The netfilter.org "nftables" project, https://netfilter.org/projects/nftables/index.html, 2024. Accessed: 2024-04-13.
[11] S. T. Zargar, J. Joshi, D. Tipper, A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks, IEEE Communications Surveys & Tutorials 15 (2013) 2046–2069. doi:10.1109/SURV.2013.031413.00127.
[12] A. Praseed, P. S. Thilagam, DDoS attacks at the application layer: Challenges and research perspectives for safeguarding web applications, IEEE Communications Surveys & Tutorials 21 (2019) 661–685. doi:10.1109/COMST.2018.2870658.
[13] N. Tripathi, N. Hubballi, Application layer denial-of-service attacks and defense mechanisms: A survey, ACM Computing Surveys 54 (2021) 1–33. doi:10.1145/3448291.
[14] Y. Xie, S.-Z. Yu, Monitoring the application-layer DDoS attacks for popular websites, IEEE/ACM Transactions on Networking 17 (2009) 15–25. doi:10.1109/TNET.2008.925628.
[15] R. Mohammadi, C. Lal, M. Conti, HTTPScout: A machine learning based countermeasure for HTTP flood attacks in SDN, International Journal of Information Security 22 (2023) 367–379. doi:10.1007/s10207-022-00641-3.
[16] N. Tripathi, N. Hubballi, Slow rate denial of service attacks against HTTP/2 and detection, Computers & Security 72 (2018) 255–272. doi:10.1016/j.cose.2017.09.009.
[17] Oracle, Oracle VM VirtualBox, https://www.virtualbox.org, 2024. Accessed: 2024-06-21.
[18] The Apache Software Foundation, Apache HTTP Server, https://httpd.apache.org, 2024. Accessed: 2024-06-21.
[19] K. Panchártek, Department library system, Diploma thesis, Palacký University Olomouc, 2021.
[20] Oracle, MySQL, https://www.mysql.com, 2024. Accessed: 2024-06-21.
[21] Python Software Foundation, `http.server` – HTTP servers, https://docs.python.org/3/library/http.server.html, 2024. Accessed: 2024-06-21.
[22] The Wireshark Foundation, Wireshark, https://www.wireshark.org, 2024. Accessed: 2024-06-21.
[23] J. Seidl, GoldenEye http DoS test tool, https://github.com/jseidl/GoldenEye, 2024. Accessed: 2024-06-21.
[24] G. Yaltirakli, Slowloris – low bandwidth DoS tool, https://github.com/gkbrk/slowloris, 2024. Accessed: 2024-06-21.
[25] dotfighter, Tor's hammer – slow POST DoS testing tool, https://sourceforge.net/projects/torshammer/, 2024. Accessed: 2024-06-21.
[26] Google, IPv6 adoption statistics, https://www.google.com/intl/en/ipv6/statistics.html, 2026. Accessed: 2026-06-15.