Defending Web Servers Against Flash Crowd Attacks

Rajat Tandon(✉), Abhinav Palia, Jaydeep Ramani, Brandon Paulsen, Genevieve Bartlett, and Jelena Mirkovic

University of Southern California Information Sciences Institute, Marina del Rey, CA, USA
{rajattan,palia,jramani,bpaulsen}@usc.edu, {bartlett,mirkovic}@isi.edu
https://steel.isi.edu/Projects/frade/

Abstract. A flash crowd attack (FCA) floods a service, such as a Web server, with well-formed requests, generated by numerous bots. FCA traffic is difficult to filter, since individual attack and legitimate service requests look identical. We propose robust and reliable models of human interaction with server, which can identify and block a wide variety of bots. We implement the models in a system called FRADE, and evaluate them on three Web servers with different server applications and content. Our results show that FRADE detects both naive and sophisticated bots within seconds, and successfully filters out attack traffic. FRADE significantly raises the bar for a successful attack, by forcing attackers to deploy at least three orders of magnitude larger botnets than today.

1 Introduction

Application layer DDoS attacks or flash-crowd attacks (FCAs) are on the rise [23, 25, 30]. The attacker floods a popular service with legitimate-like requests, using many bots. This usually has a severe impact on the server, impairing its ability to serve legitimate users. The attack resembles a "flash crowd", where many legitimate clients access popular content. Distinguishing between a flash-crowd and a FCA is hard, as the attack uses requests whose content is identical to a legitimate user's content, and each bot may send at a low rate [24, 27, 44]. Thus, typical defenses against volumetric attacks, such as looking for malformed requests or rate-limiting clients, do not help against FCAs.

We propose FRADE, a server-based FCA defense, which aims to identify and block malicious clients, based on a wholistic assessment of their interaction with the server. FRADE views the problem of distinguishing between legitimate and attack clients, as distinguishing between humans and bots. Thus, FRADE is well-suited to protect applications where legitimate service requests are issued by humans, such as Web servers.

FRADE leverages three key differences between humans and bots. First, humans browse in a bursty manner, while bots try to maximize their request rate and send traffic continuously. FRADE learns the dynamics of human interaction with a given server over several time scales, and builds its *dynamics* models.

Second, humans follow popular content across pages, while bots cannot identify popular content. FRADE learns patterns of human browsing over time, and builds its *semantics* model. Third, humans only click on visible hyperlinks, while bots cannot discriminate between hyperlinks based on their visibility. FRADE's *deception* module embeds invisible hyperlinks into the server's replies. When the load on the server is high, FRADE labels as "bot", and blocks clients whose behavior mismatches its dynamics or semantics model, or clients that access deception hyperlinks.

FRADE does not make FCAs impossible, but it successfully mitigates a large range of attack strategies. Our evaluation with real traffic, servers and attacks, shows that FRADE identifies and blocks naive bots after 3–5 requests, and stealthy bots after 15–19 requests, thus significantly raising the bar for attackers. To perform a successful, sustained attack, an attacker must employ more sophisticated bots, and deploy them in waves, retiring old ones as they are blocked by FRADE and enlisting new ones. The attacker needs at least *three orders of magnitude* more bots than used in today's attacks.

Our prior work by Oikonomou and Mirkovic [39] proposed the high-level ideas of differentiating humans from bots using dynamics and semantics models, and decoy hyperlinks. We refer to this work as OM. We build upon the basic ideas in OM, but significantly modify and improve them, to make the system robust against sophisticated adversaries, and practical to implement. Our contributions are (also summarized in Table 1):

Sophisticated Attack Handling: OM cannot handle attacks by an attacker familiar with the defense, while FRADE can (Sect. 3.3).

Stealthier Decoy Hyperlinks: FRADE uses *stealthier deception hyperlinks* than OM (Sect. 2.6), which cannot be detected via automated Web page analysis.

Improved Models: FRADE has *simpler and more robust dynamics and semantics models* (Sect. 2.4 and 2.5), which only require legitimate clients' data to train. OM also required attack data for training, which is hard to obtain and may impair detection of new attacks. FRADE is *much more accurate* than OM in differentiating bots from humans (Sect. 3.5).

Implementation and Evaluation: FRADE is *implemented as a complete system* and *evaluated with real traffic and server content*, while OM was evaluated in *simulation only*. FRADE's implementation-based evaluation helped us *discover and solve major real-time processing issues*, such as enabling the defense to receive and analyze requests during FCAs, and dealing with missed and reordered client requests (Sect. 2.7). FRADE as a complete system, mitigates FCAs about *ten times faster* than OM.

Section 2.8 provides a detailed explanation of the novelties and improvements that FRADE offers over OM. Our code and data are accessible at the link [47].

2 FRADE

We next give an overview of FRADE's goals and operation.

Attacker Model. In our work we consider two attacker models. A *naive attacker* launches FCAs that are observed today and is not familiar with FRADE. A *sophisticated attacker* is familiar with FRADE and actively tries to bypass it.

Design Goals. We aim to design an FCA defense that mitigates both *naive* and *sophisticated* attacks. Our design rests on two premises. First, FRADE's models are based on features that are difficult, albeit not impossible, for an attacker to learn, because they are only observable at the server. Second, if an attacker does successfully learn and mimic our models, it drastically lowers the usefulness of each bot and forces the attacker to employ many more bots to achieve a sustained attack. In our evaluation, FRADE raises the bar for a successful FCA from just a single bot to 8,000 bots. Extrapolating from the botnet sizes observed in contemporary FCAs, FRADE would raise the bar from 3–6 K to 24–48 M bots—far above the size of botnets available today.

Anomaly detection methods regularly learn feature thresholds from training data, and apply them in production. Our contribution lies in (a) selecting which features to learn, to be effective against both naive and sophisticated attacks, (b) implementing and evaluating our approach in three different Web servers, with different content.

2.1 Feature Selection

FRADE aims to differentiate human users from bots during FCAs, and to do so transparently to the human users. Differentiating humans from bots is challenging in an FCA, since legitimate and attack requests can be identical. Our key insight is that *while individual requests are identical, the behavior of traffic sources (humans and bots), observed over sequences of requests differs with regard to dynamics and semantics of interaction with the server, and how they identify content of interest.*

Dynamics: Human users browse server content following their interest, and occasionally pause to read content or attend to other, unrelated tasks (e.g., lunch). Their rate is therefore bursty – it may be high in a small time window, but not sustained over time. Bots are incentivized to generate requests more aggressively, generating a sustained rate of requests over long time. To capture these differences we develop models that encode the dynamics of human user interaction with the server over *multiple time windows.*

The main challenge lies in how to properly model various types of requests to make it hard for bots to avoid detection. Because requests may be generated in different ways and may consume different resources at the server, we develop three dynamics models: (a) *main-page requests* are generated through human action, such as clicking on a hyperlink or scrolling to the bottom of a page – we model their rate directly over multiple time windows, (b) requests for *embedded* *content*, such as images, are automatically generated by a Web browser, and their rate will vary depending on the browser configuration and the number of embedded objects per page – instead of modeling request *rate* for embedded objects, we associate each object with its parent page, and allow only those objects to load that belong to a recently loaded parent page, (c) requests for dynamic pages can consume many *server resources*, even at low rate – we model the demand for server resources over different time windows.

Semantics: Since humans follow their interests and understand content, they tend to click on popular content more often than not. Bots, on the other hand, must either hard-code a sequence of pages to visit, fabricate requests for non-existing pages, or choose at random from hyperlinks available on the pages, which they previously visited. *The main challenge lies in building a model that properly leverages popularity measures to detect random, fake or hard-coded sequences of bot requests, while being able to handle user sequences that were not seen in training.*

FRADE models sequences of human user's requests, and learns the probabilities of these sequences over time. Clients whose request sequences have low probabilities according to the model will be classified as bots. FRADE has a special *fall-back* mechanism to handle sequences not seen in training.

Deception: We expect human users to visit only those hyperlinks that they can see and that are interesting to them, in the rendered content. *The main challenge in leveraging this difference lies in developing ways to automatically insert decoy hyperlinks in pages, which humans will not visit, and to make it hard for bots to identify them via page source parsing.* FRADE dynamically inserts *decoy* hyperlinks [46], into Web pages, which are linked to anchors invisible to human eye (hidden, small or transparent). FRADE leverages page analysis and CSS files to make these anchors hard to identify by automated analysis. Clients that click on decoy anchors are identified as bots.

We discuss novelty in Sect. 2.8 and demonstrate effectiveness in Sect. 3.

Table 1. Comparison between OM [39] and FRADE.

Feature	OM [39]	FRADE	Section
Web req. FCA	Yes	Yes	Sect. 3.2
Embd. obj. FCA	No	Yes, DYN_e mod	Sect. 3.3
Costly req. FCA	No	Yes, DYN_c mod	Sect. 3.3
Accuracy	$\text{fp} \geq 0$, $\text{fn} \geq 0$	$\text{fp} = 0$, $\text{fn} = 0$	Sect. 3.5
Models	DYN_h & sem. mod. improved		Sect. 2
Honeytokens	Simple	Sophisticated	Sect. 2.6
Training	Leg. & attack data	Leg. data	Sect. 2
Evaluation	Simulation	Real traffic/servers	Sect. 3

2.2 Overview

FRADE runs in parallel with the server and not inline. It includes an attack detection module and three bot identification modules—*dynamics*, *semantics* and *deception*. It interfaces with a firewall (e.g., `iptables`) to implement attack filtering. FRADE learns how human users interact with the Web server that it protects. It builds the semantics and dynamics models by monitoring *Web server access logs* (WAL) in absence of FCAs. Deception objects, invisible to humans in rendered content, are also automatically inserted into each Web page on the server. When a potential FCA is detected, FRADE enters the classification mode. FRADE loads its learned models into memory, and begins tracking each user's behavior. When a user's behavior deviates from one of the learned models, the user is put on the filter list and all their requests are dropped. When attack stops, the detection module deactivates classification. A filtering rule is removed when the traffic matching it declines.

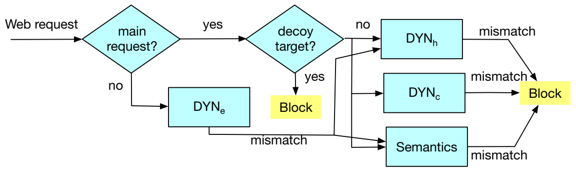


Fig. 1. Overview of FRADE's processing of a Web request.

FRADE uses some customizable parameters for its operation. The parameters and values we used in evaluation are shown in Table 2 and explained below. We perform sensitivity analysis over these parameters in Sect. 3.6.

2.3 Attack Detection

The attack detection module runs separately from the rest of FRADE, and activates and deactivates other modules by starting and stopping processes. Our detection module is intentionally simple, since our focus was on bot identification. We focus on detecting increase in incoming requests, regardless of whether they are due to legitimate flash-crowd event or due to FCA. We then rely on our, very accurate, identification of bots to handle the event. A deploying network can replace our detection module with other mechanisms, such as the Bro Network Security Monitor [40].

Learning. FRADE's attack detection module monitors incoming service requests rate, and learns its smoothed historical mean. If the current incoming rate of requests exceeds the historical mean multiplied by the parameter *attack-High*, this module raises the alert. Otherwise, the module updates the mean. The update interval, *intDet*, and the parameter *attackHigh*, are configurable (we use $intDet = 1$ s and $attackHigh = 10$).

Table 2. FRADE's parameters and the values we used.

Parameter	Meaning	Value
intDet	Monitoring int	1 s
attackHigh	Incoming req high thresh	10 * avg
attackLow	Incoming req low thresh	2 * avg
windows	Time int. for dyn. models	1 s, 10 s, 60 s, 300 s, 600 s
ρ	Ratio of dec. to vis. obj	1
ThreshPerc	High perc. of a modeled quantity	100

Classification. During an FCA, the detection module continues to collect and evaluate the incoming request rate, but does not update its historical mean. When the current rate falls below the pre-attack historical mean, multiplied by a configurable parameter *attackLow* (we use *attackLow* = 2), FRADE signals the end of the FCA and turns off bot classification modules. Figure 1 shows FRADE's processing of a Web request during attack. In the rest of this section we describe each processing step.

2.4 Request Dynamics

The *dynamics* module models the rate of a user's interaction with a server within a given time interval, and consists of three sub-modules. $\boldsymbol{DYN}_h$ models the rate of main-page requests, such as clicking on a hyperlink or scrolling to the bottom of a page. $\boldsymbol{DYN}_e$ models embedded-object requests, such as loading an image. $\boldsymbol{DYN}_c$ models the rate of a *user's demand for server resources*, where the demand is represented as the total time it took to serve the given user's requests in a given time period.

Learning. DYN_h and DYN_c learn the expected range of the quantity they model (e.g., request rate, processing time, etc.) over all users, by analyzing WAL. We group requests by their source IP address, and assume that each IP address represents one user or a group of users. FRADE classifies each request as either a main-page or embedded. Section 2.10 describes how to detect these two types of requests. DYN_h and DYN_c model the main-page requests and use a high percentile of the range (controlled by *ThreshPerc*, e.g., 99%) as their learned *threshold* for the quantity they model. In our evaluation we use *ThreshPerc* = 100%. The number and sizes of windows are configurable parameters. As humans browse in a bursty manner, having *multiple windows* allows monitoring at different time scales, and drastically raises the bar for a successful FCA. It enables us to correctly classify legitimate bursts and distinguish them from sustained attack floods, even when their peak request rates are equal. We use windows of 1, 10, 60, 300 and 600 s.

DYN_c models the processing time spent to serve a user's requests. This time depends both on the complexity of the user's request, and the current server load. DYN_c models the time to serve a user's request on *lightly loaded server* to capture only that cost to the server that the user can control – the "principal cost". During an attack, we use this principal cost, rather than the actual processing time (inflated cost), to calculate a user's demand on server's resources. This allows us to avoid false positives, where legitimate users hitting a heavily loaded server, experience a large inflated cost through no fault of their own. During learning, each request and its processing time are recorded in a hashmap, called the *ProcessMap*. DYN$_c$ looks up the principal cost for each request in the ProcessMap, and adds it to the running total for the given user. It then learns the *ThreshPerc* value over these totals and for each window.

DYN$_e$ learns which embedded objects exist on each Web page and records this in a hashmap, called the *ObjectMap*.

Classification. During classification, DYN$_h$ and DYN$_c$ collect the same measures of user interaction, per each user, as they did during learning. These measures are continuously updated as new requests arrive. After each update, the module compares the updated measure against its corresponding threshold. If the measure exceeds a threshold, the client's IP is communicated to the filtering module. Whenever a client issues a main-page request, DYN$_e$ loads all the embedded objects related to this request from *ObjectMap* into this user's ApprovedObjectList (AOL). DYN$_e$ checks for the presence of the embedded object requests made by the same user in his AOL. If found, the object is deleted from the AOL. If not found, DYN$_e$ treats this request as a main page request, and forwards it to DYN$_h$ and semantic modules. We do this because a user may bookmark an embedded object, e.g., an image, and request it separately at a future time. Our design allows such requests to be served, while preventing FCAs that create floods of embedded requests.

2.5 Request Semantics

The *semantics* module models the probability of a sequence of requests generated by human users.

Learning. We consider only requests classified as main-page requests. In the learning phase, we compute transition probabilities between each pair of pages (e.g., A to B) on the server using Eq. (1), where $N_{A\rightarrow B}$ is the number of transitions from page A to page B, and $N_{A\rightarrow *}$ is the number of transitions from page A to any page. We learn $N_{A\rightarrow B}$ and $N_{A\rightarrow *}$ from WAL. We define the probability of sequence S = $\{u_1, .. , u_n\}$ as compound probability of dependent events, which are page transitions, using Eq. (2).

$$P_t(A \rightarrow B) = \frac{N_{A\rightarrow B}}{N_{A\rightarrow *}}, \quad (1); \qquad P(S) = \prod_{i=1}^{n-1} P_t(u_i \rightarrow u_{i+1}), \quad (2)$$

During learning, the semantics model calculates sequence probabilities for each user. Since sequence probability declines with length, we learn the probability for a given range of sequence length (e.g., 5–10 transitions), grouped into a bin. We also ensure that bins are of balanced size. When learning the threshold for each bin, we sort probabilities of all sequences in our training set that fall into that bin and take a low percentile (1-*ThreshPerc*) to be the threshold.

In practice, if a server has very dynamic content, the semantics module may not see all the transitions during learning, leading to false positives in classification. To handle incomplete training data, semantics module has a *fall-back mechanism*. It views Web pages as organized into groups of related content. During learning, it learns transitions from pages to groups, groups to pages, and groups to other groups. We define a group as all the pages that cover the same topic. On some Web sites, the page's topic can be inferred from its file path, while others require analyzing each Web page content to determine the topic (Sect. 2.9). The probability of transition from a page/group to a group, is calculated as the average probability of transition to any file within the group:

$$P_t(A \rightarrow group(b)) = \frac{\sum_{f \in group(b)} P_t(A \rightarrow f)}{N_{A \rightarrow group(b)}}, \tag{3}$$

Classification. FRADE processes the request sequence for each client in the active session list (ASL). When a new request arrives, the module updates the client's sequence probability, just like it did during learning. If a transition from page A $\rightarrow$ B is not found, FRADE falls back to using groups instead of pages. It attempts to find transitions A $\rightarrow$ group(B), group(A) $\rightarrow$ B and group(A) $\rightarrow$ group(B) in that order. When the first transition is found, its probability is used to multiply the current sequence probability, according to Eq. 2. If no transitions are found, FRADE multiplies the current sequence probability with a constant called *noFileProb* $\ll 1$. After each update, it compares the current sequence's probability against the corresponding threshold for the sequence's length. Values lower than the threshold lead to blocking of the client.

2.6 Deception

The *deception* module follows the key idea of honeytokens [46], special objects meant to be accessed only by attackers. The module embeds decoy objects, such as overlapping/small images, into Web pages. In websites with mainly textual content, like Wikipedia, we insert hyperlinks around random pieces of text, but do not highlight them. This makes the hyperlink invisible to humans. In websites with mainly media content, like Imgur, we embed hyperlinks around small images, or small-font text. We insert these decoy objects away from existing hyperlinks, to minimize the chance that they are accidentally visited by humans.

We automatically insert decoy objects into a page's source code so that they do not stand out among other embedded objects in that page. The number of decoy objects to be inserted is guided by the parameter ρ – the ratio of the decoy to original objects on the same page. We make decoy hyperlinks hard to identify from the page's source code by creating separate styles for them in the site's CSS file. We automatically craft the names of the pages, pointed to by decoy hyperlinks, similar to the names of other, non-decoy pages on the server. We introduce some randomness into the deception object's placement, to make it harder to identify them programmatically.

2.7 Using a Proxy to Speed up Servers

FRADE mines data about user payload from WAL, to classify users as humans/ bots, as shown in Fig. 2(a). A server may be so overwhelmed under FCA, that it cannot accept new connections, slowing down logging and delaying FRADE's action.

We explored two approaches to boost the number of requests a server is able to receive and log during FCAs. Our first approach, **transparent proxy (Trans)**, shown in Fig. 2(b), uses a lightweight proxy between clients and the server. It completes the 3-way handshake with the client, receives and logs Web page requests. It then recreates the connection with the backend server. This can speed up logging, but ultimately the target server may overload before we block all bots, and this will back up the Trans server as well. We use http-proxy-middleware [14] as our transparent proxy. It lets us log requests as soon as they arrive, and forward them to the backend server.

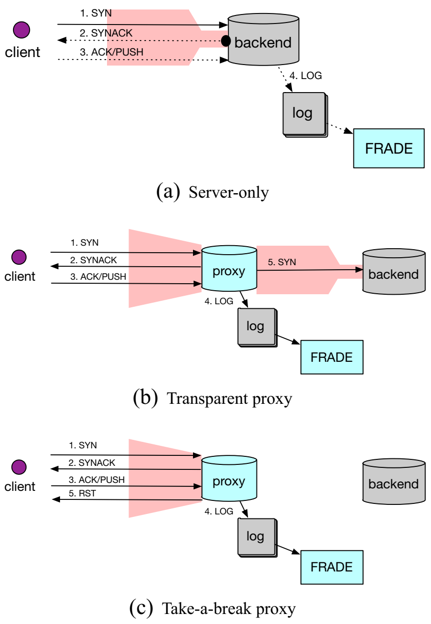


Fig. 2. Illustration of high-rate attack handling, (a) by the server itself, (b) by Trans approach and (c) by TAB approach

Our second approach, **take-a-break proxy (TAB)**, shown in Fig. 2(c), uses a dropping proxy between clients and the backend server. FRADE runs on the dropping proxy, which logs and drops all the requests, until our blocking manages to reduce the request rate. Logging requests and dropping them immediately allows for faster blocking, as immediate closure of a connection frees the port and socket on the proxy for reuse. Dropping all requests hurts legitimate clients, but it ensures the fastest bot identification, helping us serve users well for the remaining (possibly lengthy) duration of the FCA. We implement the proxy in http-proxy-middleware [14].

To improve the speed of bot detection, we further stop building the ApprovedObjectList (AOL) once TAB proxy is active. Since no replies are returned to users while the TAB proxy is active, a human user will not issue embedded object requests, while a bot may. This helps us identify bots faster.

2.8 Improvements over OM

We now detail improvements of FRADE over OM – these improvements enable FRADE to be robust against sophisticated attacks, while OM only handles naive attacks.

Stealthier Decoy Hyperlinks: FRADE uses stealthier decoy targets and anchors, and makes the placement of decoy anchors more robust against false positives than OM. FRADE learns the page naming structure from Web server logs, and automatically crafts the names of the target pages for decoy hyperlinks. OM creates target pages with random names, which can be detected by bots.

FRADE inserts the decoy anchors away from the existing, visible anchors to reduce the chance that they are accidentally visited by humans. OM does not address such concerns, and is prone to false positives.

FRADE makes decoy anchors invisible by adding new styles to the site's CSS file, while OM manipulates the anchors in the Web page source, making them small or changing their color or z-index. OM's anchors can be detected more easily by bots.

Improved Dynamics Model: OM models the request dynamics only for main-page requests, while FRADE models it for main-page and embedded requests, and also models each request's principal cost. This helps FRADE handle a variety of sophisticated attacks (see Sect. 3.3) that OM cannot handle.

OM uses decision trees to capture request dynamics, grouping requests into sessions and using four features per session. This makes OM's model more complex than FRADE's, which uses just one feature – the *threshold rate of requests per time window*. OM further requires both legitimate and attack data for training. Attack data is hard to obtain and overfitting can impair detection of new bot variants. FRADE only requires legitimate data for training.

Improved Semantics Model: Both OM and FRADE build the request graph to encode transition probabilities from one Web page to another. But OM focuses only on pages, while FRADE also models transitions between page groups. This fall-back mechanism enables FRADE to handle transitions in production that were not seen in training. Further, OM computes the sequence probability as the average of probabilities on the request graph, while FRADE computes it as a product (compound probability of dependent events), which ensures fast decline with sequence size.

Implementation and Evaluation: FRADE is implemented as a complete system and evaluated in a realistic setting, while OM was evaluated in simulation.

2.9 Deployment Considerations

Customization. To use FRADE, the Web site administrators must (1) categorize their Web pages into groups for the semantics module, and (2) insert decoy hyperlinks into Web pages. This may in some cases require minor human effort, depending on the server's content. Table 3 shows how we classified pages into groups. For Wikipedia, we leveraged its existing categorization of pages into topics. Imgur and Reddit have a folder-based Web site structure, with related files grouped into the same folder. In absence of both, a Web site could use a topic identification tool, such as [2]. We have automated decoy hyperlinks insertion (around 100 lines of code), which can be customized for a new Web site.

User Identification. FRADE currently blocks IP addresses, but this can lead to collateral damage when clients share a NAT. FRADE could use cookies and block users at the application level, but when a server is under FCA, it is too overloaded to process each request and mine its cookie. We thus view IP-based filtering as necessary to relieve the load at the server.

Training Data. FRADE requires training data of legitimate clients and needs to be trained per server. Each server needs to tune the frequency of their training and decoy hyperlink insertion to match the frequency of their content updates. Attackers may introduce adversarial data before the attack to dilute the learned models. One could address this issue by: (1) sampling training data over multiple days, (2) excluding outliers by adopting lower values for *ThreshPerc* parameter, (3) using techniques such as machine unlearning [13].

Dynamic Content and Misclassification. If a server does not update its models on new content, FRADE may miss some transitions in the semantics model, or embedded objects in the AOL. Our fall-back mechanisms for the semantics model and treating embedded requests not found on AOL as main requests, help minimize this effect. We used the data from Internet Archive [34] to measure the daily updates on some frequently-updated Web sites, CNN, NY Times, Imgur and Amazon. On the average, a small percentage of the Web site's content (0.17–0.31%) is added daily, around 6 K–54 K objects and pages. FRADE's models can be incrementally updated this often, without full re-training.

Load Balancers. Larger sites deploy load balancers in front of server farms; we would have to periodically gather web access logs to a central location and run FRADE there to learn models and classify bots. FRADE could then block bot IPs by inserting filtering rules into the load balancer.

2.10 Implementation

FRADE's core engine is written in C++, and runs on the Web server/proxy. Filtering is achieved by interfacing with a host-specific mechanism. We use `iptables` with `ipset` extension, which scale well with large filter lists. We classify each request as either main-page or embedded in the following way. We crawl the full Web site using the Selenium-based [43] crawler. This helps us identify both static and dynamically generated HTML content. We extract main requests by finding elements with tag "a" and attribute "href". We label other requests as embedded. These steps are fully automated.

3 Evaluation

Ideally, we would evaluate FRADE with operational servers, real logs, human users and real FCAs. Unfortunately there are many obstacles to such evaluation: (1) there are no publicly available WAL from modern servers, (2) paying real users to interact with a server during evaluation can get costly and prevent repeatable experiments, (3) there are no publicly-available logs of real FCAs. We test FRADE in emulated experiments on the Emulab testbed [50], using replayed human user traffic and real FCAs. We try to make our experiments as realistic and representative as possible, given the obstacles listed above.

3.1 Emulation Evaluation Setup

We mirror *dynamic* content for three popular Web sites: Imgur, Reddit and Wikipedia. All content is generated dynamically by pulling page information from the server's database, using the original site's scripts. This content is copyright-free and server configuration files were publicly available. We download each full site, modify it by automatically inserting decoy hyperlinks, and deploy the site's original configuration and scripts on our server within Emulab testbed. While we wanted to replicate more servers in our tests, this was impossible because their implementation was either private (e.g., Facebook, YouTube, etc.) or their content was not copyright-free (e.g., major news sites).

We engage human users to browse our Web sites and gather data to train and test FRADE's models. We replay human user data in a controlled environment and launch FCAs, with real traffic, targeting our servers from an emulated botnet. We launch repeated FCAs with various botnet sizes and bot behavior, and measure the time it takes to identify and block bots.

Our chosen Web sites had server software diversity. Imgur runs on Apache, Reddit runs on haproxy, and we deployed Wikipedia on nginx.

Human User Data. We obtained human user data using Amazon Mechanical Turk workers. This study was reviewed and approved by our IRB. In the study we presented an information sheet to each worker, paired the worker with a server at random, and asked the worker to browse naturally. We intentionally did not create specific tasks for workers, as we wanted them to follow their interests and produce realistic data for our semantic models. We also asked each worker to browse at least 20 pages so that we would have sufficient data for training and testing. To keep engagement high, and discourage workers who just click through as fast as possible, we asked each worker to rate each page's loading speed on a 1–5 scale. These ratings were not used in our study. Human behavior may become more aggressive during FCAs (e.g., more attempts to refresh content), which may lead to misclassification. However, QoS studies show that users tend to click less and not more when the server's replies are slow [6]. Our dataset does not capture any adaptation of users to speed of server replies. Each server had 243 unique users for training and 107 users for testing in our dataset.

Legitimate Traffic Generator. During each experiment, we replay user traffic from testing logs. We wrote a custom traffic generator, which extracts timing and URL sequences from logs, and then chooses when to start each sequence based on the desired number of active users. The generator uses many different source IPs. Our replay maintains timing between requests in a sequence, and traffic is replayed at the application level. When a user sequence completes, another sequence is selected and another IP becomes active. If we run out of sequences to replay, we reuse the old ones.

Attack Traffic Generator. Our attack traffic generator is a modified *httperf* tool [37]. We added the ability to choose source IPs from a pool, and to select requests for each IP from a given sequence, in order. Before building our own attack tool, we have investigated popular attack tools, such as HULK [1], LOIC [22] and HOIC [8]. These tools do not allow us to use multiple IP addresses when running on the same physical machine. This feature is important as one can mimic large botnets using few machines. Our tool can generate all attacks generated by HULK, LOIC and HOIC tools, and more.

Table 3. Group assignment for our three Web sites.

Server	Groups
Wikipedia	Topic-based categories
Imgur	Folder-based groups
Reddit	Folder-based groups

Table 4. Time to block all bots

Windows	Time to block all bots		
	Botnet size		
	8 bots	800 bots	8,000 bots
Non-unif-5 (current)	3 s	8 s	16 s
Uniform-5	4 s	15 s	47 s
Uniform-10	3 s	10 s	38 s
Uniform-20	3 s	7 s	23 s

Experiment Topology and Scenarios. Our experiment topology is shown in Fig. 3. It has 8 physical attack nodes (each emulating 1–1,000 virtual attackers each), 1 node emulating 100 legitimate clients, and 3 nodes for mirrored servers. All nodes are of type d430 on Emulab, with 32 cores and the Ubuntu 14.04 OS. We fine-tuned the nodes to maximize the request rate that each client could generate, and to maximize the request rate that our servers could handle. While having a larger topology would have helped us perform larger-scale tests, Emulab is a shared resource and we were limited in how many nodes we could request. Our tests suffice to illustrate trends in FRADE's effectiveness as botnet size increases.

To identify an effective attack rate, we measured the request rate required to slow down each server's processing below 1 request per second. For all the Web sites, this rate was around 1,000 rps. We chose to generate 8 times this rate during an attack – 8,000 rps. We test one server per run. Legitimate clients start sending traffic to the server following the timing and sequences from the testing logs. We maintain 100 active, parallel virtual clients throughout the run, each with a separate IP address. After a minute, our virtual attackers (1–1,000 per physical machine) start sending requests to the server at the aggregate rate of 8,000 rps. After 10 min we stop the attackers, and a minute later we stop the clients.

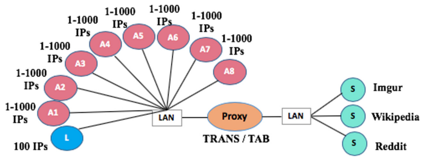


Fig. 3. Attackers: A1–A8 (up to 8,000 virtual bots), Legitimate: L (100 clients), the proxy and 3 Web servers.

We illustrate FRADE's handling of an FCA in Fig. 4, which shows legitimate and FCA traffic (sent to the server, and allowed by FRADE), and the blocked bots. Legitimate traffic declines at first, until FRADE manages to identify most bots. After 20 s, FRADE blocks all bots and legitimate traffic returns to its pre-attack levels. At that point, although the attacker keeps sending the attack traffic (the *actual attack* area in Fig. 4), the attack requests cannot reach the server, as the bot IP addresses are blocked at the proxy.

3.2 Today's (Naive) Attacks

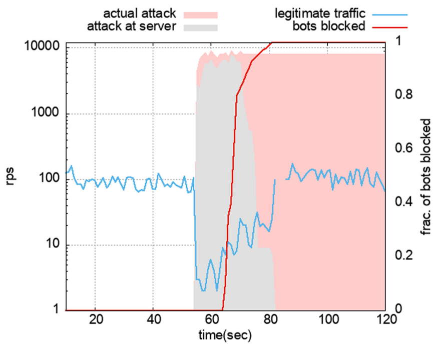


Fig. 4. FRADE's handling of an FCA.

First, we test FCAs, that resemble today's attacks as noted by [16]. Our attackers repeatedly request: (t1) **non-existing URLs**, or (t2) the **base URL**. In the case (t1), we tailor the URL's syntax for it to be identified as main requests. Figures 5(a) and 5(b) show the time that FRADE took to block all the bots, in these FCAs, for each server, and for 8 and 800 bots. Both attacks show similar trends, with the smaller botnet being blocked sooner (around 4 s instead of 8–10 s). All bot classification is done by the DYN_h module.

3.3 Sophisticated Attacks

An attacker familiar with FRADE could attempt to launch a sophisticated FCA, where bots mimic humans to evade detection. To evade DYN_h, bots would send at a lower rate, necessitating a larger botnet. Bots could also attempt to generate requests mimicking a human's semantics, i.e., trying to guess or learning popular sequences. Finally, bots could leverage knowledge of FRADE's different processing pipelines to engage in embedded or costly request floods.

We first explore *fully automated* FCAs. An attacker has previously engaged a crawler to learn about the target server's *Web site graph*, i.e., which pages point to which other pages and to match pages to embedded objects. The attacker knows that lower request rates per bot mean longer detection delays, but does not know each page's popularity and which hyperlinks are decoy links. We only show results for Imgur. FCAs on other servers show a similar trend.

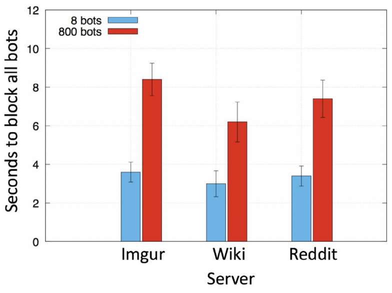
Fully-Automated: Larger Botnet and Smarter Sequences: These FCAs include a larger botnet—8,000 bots. The first two FCAs are using the same (s1) **non-existing** and (s2) **base** URLs as described in Sect. 3.2, with a larger botnet to evade DYN_h detection. The third FCA performs a (s3) **random walk** on the Web site graph, making only main page requests (we investigate FCAs that use embedded links in Sect. 3.3). It cannot differentiate between decoy and non-decoy links.

Figure 6(a) illustrates the time it takes to block all 8,000 bots in s1–s3 attacks, using the TAB proxy approach. The non-existing URL attack (s1) is fully handled within 16 s, with each bot blocked after ≈5.8 requests, by the semantics module. The random walk (s3) is handled within 16 s, with each bot blocked after 3.8 requests on average by the deception module. For the base URL attack (s2) it takes 36 s to block 8,000 bots, with each bot blocked after ≈15 requests by the DYN$_h$ module.

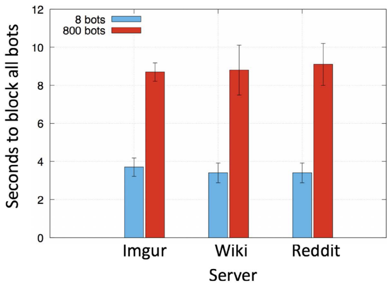
Fully-Automated: Embedded and Costly Request Floods: Attackers could attempt to flood with embedded or costly requests. The **non-existing-object** attack (s4) requests made-up URLs, which end up treated as main page requests by FRADE. Figure 6(b) shows the time to block all 8,000 bots in this FCA. Within a few seconds the FCA is fully handled. Each bot is blocked within 2–3 requests. The semantics module blocks all bots. The **costly** attack (s5) sends the most expensive main page request repeatedly to the server. All bots are blocked by the DYN$_c$ module, within a few seconds.

An attacker could collaborate with some human users to learn popular pages on a server, and decoy objects, and then launch *semi-automated attacks*. The attacker then leverages what they learned to craft sequences of requests, which may evade detection by FRADE's semantics and deception modules. The requests are sent automatically by bots at predetermined timing.

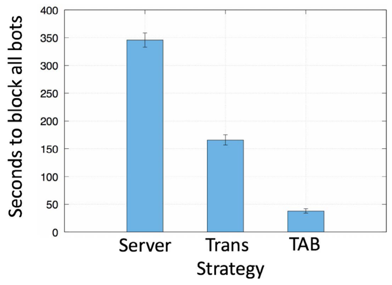
Semi-Automated: Floods that Avoid Deception. The **smart walk** attack (s6) performs a random walk on the Web site graph avoiding decoy links. The **smart-walk-object** (s7) performs a smart walk among all embedded objects on the site, and **smart-walk-site** (s8) performs a smart walk on the site, and requests all non-decoy embedded objects for each



(a) The time FRADE takes to block 8 and 800 bots for *Non-existing URLs Attacks*



(b) The time FRADE takes to block 8 and 800 bots for *Base URL Attacks*

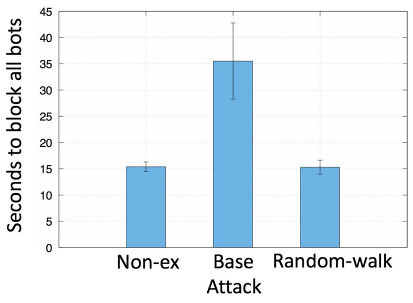


(c) TAB has superior performance for sophisticated attacks

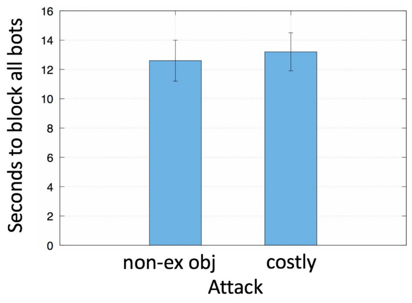
Fig. 5. Today's (Naive) attacks and performance comparison for sophisticated attacks.

main page request. A replay attack [52], where the attacker records and replays legitimate users' requests, is a special example of the smart-walk-site attack. Figure 6(c) shows the time it takes to block all 8,000 bots in these FCAs. In a smart-walk attack (s6), FRADE takes 38 s to block all 8,000 bots. Each bot is blocked after 19 requests on the average, by the DYN$_h$ module. Figure 5(c) illustrates benefits of using a proxy. Without a proxy, it would take around 6 min to block all the bots. With Trans, it takes under 3 min, and with TAB it takes 38 s—almost 10-fold speed-up compared to the server-only approach!

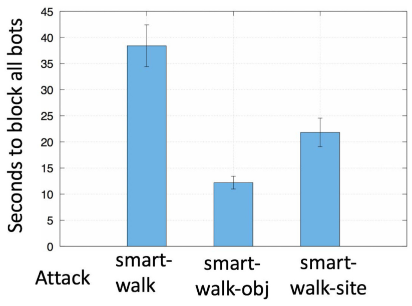
In the smart-walk-object attack (s7), all bots are blocked within a few seconds. Each bot is blocked within 2–3 requests, as it requests embedded objects that are not on the AOL during FCA. All bots are blocked by DYN$_e$ module. The smart-walk-site attack interleaves main page and their corresponding embedded requests, and it avoids decoy links. It thus manages to slip under the radar of DYN$_h$ (main page requests come at a low rate), DYN$_c$ (requests are not costly) and deception (asking for non-decoy links only) modules. All 8,000 bots are blocked within 22 s. Each bot is blocked on the average after 6 requests. The complete blocking is done by the semantics module. Since no replies are returned to users while the TAB proxy is active, a human user will not issue embedded object requests. Hence, FRADE does not keep embedded objects on the AOL while TAB is active. Instead, embedded object requests are treated as main-page requests, and forwarded to DYN$_h$ and semantic modules, which model only main-page requests. The semantics module blocks all the bots, due to the random walk created, leading to low-probability sequences.



(a) Larger botnets and random-walk



(b) Embedded and costly requests



(c) Semi-automated attacks

Fig. 6. The time to block 8,000 bots in sophisticated attacks.

Semi-Automated: Floods that Use Popular Sequences. An attacker may learn which sequences are popular among humans and generate main page requests for them. They need to distribute the rate among many bots to evade detection by DYN$_h$. We evaluate this FCA analytically, using the WAL of a large public network testbed, that serves thousands of users. The logs covered three months of data and around 5 K users. Few users were obvious outliers, making thousands of requests. If we prune the most aggressive 5% of the users and analyze the rest of the user sequences, 95% were shorter than 17 requests. To evade FRADE, the attacker would need to retire each bot after 17 requests. For a 10-min, 1,000 rps FCA, the attacker would need to recruit 35 K bots *to attack this specific server*. Today, a single server can be brought down by a single, aggressive bot. FRADE thus raises the bar for this specific server's FCA 35,000 times.

3.4 Evasion Attacks

It may still be possible to evade FRADE and launch a successful FCA. This would require: (1) Recruiting very large botnets, so each bot is used intermittently. As per our evaluation, FRADE raises the bar from 1 bot to more than 8,000 bots, so at least three orders of magnitude. (2) Leveraging humans instead of bots and instruct users to click on visible, popular content, following their interests. Then, FRADE would not be able to identify malicious (human) clients, but the attacker would need thousands of humans for a sustained FCA. The attacker could combine these two approaches, learning popular sequences from human collaborators, then encoding them in stealthy, low-rate bots. This attack would not be detected by FRADE, but it would require at least 3 orders of magnitude more bots than are in use today (see discussion above of floods that use popular sequences).

3.5 FRADE Outperforms OM

We experimentally compare the accuracy of FRADE versus OM for DYN$_h$ and semantics models. These models exist in both solutions and FRADE improves on OM's design. We use the same legitimate traffic as in Sect. 3.2, interleaved with synthetically generated FCA bot traffic, exploring a range of request rates as suggested in [39]. For OM, we train decision trees using Weka on the training data and test on the testing data. When testing DYN$_h$ we run base-URL FCA, and use 8–8,000 bots. When testing semantics models we run the smart-walk FCA, and also use 8–8,000 bots. A false positive means that the defense classified a human user as a bot. A false negative means that the defense failed to identify a bot. For space reasons we summarize our findings. While **FRADE had no false positives or false negatives** in our tests, **OM had many false positives (7–76%) for the DYN$_h$ model, for Wikipedia and Reddit**, due to high dimensionality [51] of its models and overfitting. OM also had some **false negatives (5–13%) for the semantics model** and the 8,000-bot FCAs, because OM cannot handle transitions not seen in training data, while FRADE can using its *fallback mechanism*. FRADE's models thus outperform those of OM.

In addition to this comparison on attacks they both handle, FRADE also outperforms OM by handling a wider range of attacks (embedded and costly request floods).

3.6 Sensitivity

FRADE uses multiple parameters in its operation, as shown in Table 2. We focus here on analyzing sensitivity of parameters that influence classification accuracy. DYN$_h$ and DYN$_c$ currently use 5 window sizes as time intervals, during which they learn thresholds for their models. These window sizes follow a non-uniform, exponential-like pattern, with increasing gaps between windows. We also tested 3 different uniform distributions: *uniform-5*, *uniform-10* and *uniform-20* with 5, 10 and 20 windows in the 0–600 s range, respectively. We tested non-existing URL FCAs on Imgur with these alternative windowing approaches, and compared the speed of FRADE's response. Results are shown in Table 4. Non-uniform window sizes perform better than uniform sizes, especially for bots that send at a low rate.

Both dynamics and semantics modules use *ThreshPerc* to find the percentage of the quantities they model. In our evaluation, we use 100% as *ThreshPerc*. We chose this value to achieve zero false positives since we had small training data. In reality, a large server would have logs of millions of clients, some of which could be outliers. We have evaluated values of 99%, 95% and 90% for *ThreshPerc* with non-existing URL FCAs on Imgur. For DYN_h model false positives were 3%, 5% and 9% with *ThreshPerc* values of 99%, 95% and 90% respectively. This is mainly because our training data is small and does not have outliers, so removing some percentage of aggressive behaviors from training will lead to the similar amount of misclassifications on test data. Semantic model did not generate any false positives with tested *ThreshPerc* values. Another parameter is the decoy object density ρ—the ratio of decoy objects to visible objects on the same page. In our experiments we use $\rho = 1$. The higher the ρ, the faster a bot's identification, but the higher chances that a human user could accidentally access a decoy object and visible distortion to the original page's layout. In our MTurk experiments no humans have clicked on our decoy objects. We also observed no visible distortion. Around $\rho = 1.5$ we observe distortion in Imgur's Web pages, and around $\rho = 5$ distortion becomes severe.

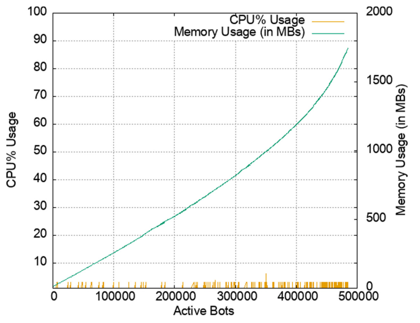


Fig. 7. Memory and CPU cost vs # bots.

3.7 Operational Cost and Scalability

We tested FRADE with attacks of up to 0.5 M bots to evaluate its scalability. FRADE's operational cost is modest. The CPU load never exceeded 5% and the memory grew linearly to around 1.5 GB for 0.5 M bots (Fig. 7), or around 3 KB per bot or client. Extremely active Web sites like Amazon can see about 4 M active clients per hour [3,4], and would need 12 GBs of memory, which is feasiable today. It takes on the average 0.05 ms to process a Web log request in FRADE. Thus, FRADE could easily process around 20,000 rps on a single core. Since FRADE does not operate in line, it does not add any user-visible delay to request processing.

Table 5. Page serve time in ms

Number of IPs	0	100	1 K	10 K	100 K	1 M
iptables	4.5	4.5	4.5	4.7	4.9	N/A
ipset	4.8	4.8	4.9	4.9	4.9	5.3

We evaluate scalability of FRADE's filtering using `iptables` and `ipset`. We artificially insert a diverse set of IP-rules and send packets matching these rules at a high rate. This emulates the situation when a server is under FCA by numerous bots. We issue Web page requests and measure the time it takes to receive the reply. Table 5 shows the averages over ten runs. `iptables`'s processing time grows modestly until 100 K IPs, but then explodes. We were not able to complete the tests with 1 M IPs. However, `ipset` imposes only a small delay of 8% as the rules table grows from 100 K to 1 M, and no measurable delay for fewer than 100 K rules. Thus, FRADE can block a million IPs using `ipset`.

4 Related Work

Clouds are a common solution for DDoS. They may offer "attack scrubbing" services, but the details of such services are proprietary. Clouds handle volumetric attacks well, but FCAs may fly under their radar. They also use Javascript-based cookies [17,41], to detect if a client is running a browser. These challenges are transparent to humans, and good for detecting automated bots. However, attackers can use the Selenium engine to generate requests. Since Selenium interprets Javascript, it would pass the cookie challenge. FRADE can complement cloud defenses, enabling server-based solutions for FCAs.

CAPTCHAs [7,29] are another popular defense against FCAs. Users, who correctly solve a graphical puzzle have their IPs placed on "allow" list. While a deterrent, CAPTCHAs have some issues. Multiple on-line services offer bulk CAPTCHA solving, using

Table 6. Rel. work comparison, showing the absence or presence of human Web server interaction features, even if present at the very basic level.

Detection mech.	Dyn	Sem	Dec
Jung et al. [28]	✓	✗	✗
Ranjan et al. [42]	✓	✗	✗
Liao et al. [33]	✓	✗	✗
Wang et al. [48]	✗	✓	✗
Xie and Yu [54]	✗	✓	✗
Beitollahi et al. [9]	✓	✓	✗
FRADE	✓	✓	✓

automated and semi-automated methods (e.g. [32]). CAPTCHAs also place a burden on human users, while FRADE does not. Google's reCAPTCHAs [20] and similar approaches for human user detection are transparent to humans, but can still be defeated using deep learning approaches [5,11,45]. These approaches are complementary to FRADE, as they model complementary human user features.

Jan et al. [26] propose a stream-based bot detection model [49] and augment it with a data synthesis method, using Generative Adversarial Networks [36], to synthesize unseen bot behavior distributions. While we lack the data they have, and cannot compare our systems directly, we can comment on their expected relative performance based on their design. Jan et al. system focuses on *eventually detecting advanced bots*, and is well-suited for click bot or chat bot detection. Authors show that it can adapt to new bot behaviors with small re-training, and that it is robust to adversarial attacks. FRADE focuses on *quickly detecting bots involved in an FCA*. Such bots are likely to exhibit specific, aggressive behaviors, since they seek to maximize request rate at the server. When FRADE misses a bot, such bot has a low yield to the attacker, necessitating a large botnet for a sustained attack. Thus FRADE could miss some bots that Jan et al. approach detects, but these bots would not be very useful for flash-crowd attacks.

Comparing reported performance, Jan et al. require long request sequences (30+ requests in a month) to classify a user as benign or bot. This means that new bots will not be detected for at least 30 requests. FRADE can identify and block most bots within 3–6 requests, and sophisticated bots with less than 20 requests. FRADE also achieves higher accuracy – it identifies all bots in our tests and does not misidentify any benign users as bots. Finally, Jan et al. use a small fraction of bot data in training, while FRADE uses only benign user data.

Rampart [35] and COGO [18] build models of resource consumption over time to detect and handle resource exhaustion states. Such defense mechanisms could handle FCAs that employ costly requests, but not other FCA variants.

Like FRADE's *dynamics* model, several efforts use timing requests to detect FCAs [33,42]. Ranjan et al. [42] use the inter-arrival of sessions, requests and the cost profile of a session to assign a suspicion value and prioritize requests. Liao et al. [33] look at the inter-arrival of requests within a window. They use custom classification based on sparse vector decomposition and rely heavily on thresholds derived from their dataset. These works have limited evaluation compared to ours and rely only on modeling human requests, while we also deal with embedded and costly requests, we build semantic models of request sequences and use decoys to bait bots. Yatagai et al. [55] look for repetitive sequences of resources, and clients which spend shorter than normal periods of time between requests. Bharathi et al. [10] use fixed sized windows to examine which, and how many, resources a client accesses and to detect repetitive patterns. Najafabadi et al. [38] use PCA and fixed windows to examine which resources a client requests. Beitollahi et al. [9] propose ConnectionScore, where connections are scored based on history and statistical analysis done during the normal conditions. Models engaged in connection scoring are coarser (e.g., 1 rps vs our rate per several time intervals) than FRADE models, and thus we believe that FRADE would outperform this. Jung et al. [28] learn existing clients of a Web server, and perform network aware clustering [31]. When the server is overloaded, they drop aggressive clients that do not fit in the existing clusters. In comparison to these works, we evaluate timing dynamics at a much finer granularity, and evaluate the strict order of requests, allowing us to detect stealthier FCAs.

Multiple works are related to FRADE's *semantics* model. Wang et al. [48] examine requests over 30-min windows (sessions) and use a click-ratio (page popularity) model and Markov process to model clients. Their detection is highly accurate for bot identification, but has a high false-positive rate, while we have zero false positives. Similar to [48], Xie et al. [54] capture the transition probabilities between requests in a session through a hidden semi-Markov model. Our approach to training and modeling is simpler, while still very accurate.

Our deception model uses honeytokens [46], similar to [12,19,21]. We build on ideas from these prior works (use of decoy links), but we use a variety of decoy objects, configurable object density and automate object insertion code for each site. To our knowledge, our work is the first to combine dynamics, semantics of user requests, and the decoy objects in a single defense, and evaluate its effectiveness using realistic traffic and real servers (Table 6). Our results show that different modules are effective against different FCAs. Thus, a combination is needed to fully handle FCAs. Software and datasets for these prior works are not publicly available, and thus we could not directly compare FRADE to them.

Biometrics solutions (e.g., [15] or [53]) can distinguish bots from humans by capturing mouse movements and keystrokes. These approaches are orthogonal to FRADE, and may suffer from privacy issues.

5 Conclusions

FCAs are challenging to handle. We have presented a solution, FRADE, which models how human users interact with servers and detects bots as they deviate from this expected behavior. Our tests show that FRADE stops naive bots within 3–5 requests and sophisticated bots within 15–19 requests. A bot could modify its behavior to bypass FRADE's detection, but this forces the attacker to use botnets at least three order of magnitude larger than today, to achieve sustained attack. FRADE thus successfully fortifies Web servers against today's FCAs.

Acknowledgment. This material is based upon work supported by the National Science Foundation under grant number 1319215.

References

1. Hulk DDoS tool, May 2018. https://tinyurl.com/y49tze6w. Accessed 31 Mar 2021
2. Classification tools, May 2019. https://tinyurl.com/y6cdav26. Accessed 31 Mar 2021
3. Combined desktop and mobile visits to amazon.com from February 2018 to April 2019 (in millions), May 2019. https://tinyurl.com/y25d8ln8. Accessed 31 Mar 2021

4. Most popular retail websites in the United States as of December 2019, ranked by visitors (in millions), September 2020. https://www.statista.com/statistics/271450/monthly-unique-visitors-to-us-retail-websites/. Accessed 31 Mar 2021
5. Akrout, I., Feriani, A., Akrout, M.: Hacking google reCAPTCHA v3 using Reinforcement Learning (2019)
6. Arapakis, I., Bai, X., Cambazoglu, B.B.: Impact of response latency on user behavior in web search. In: Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval, pp. 103–112. Association for Computing Machinery, New York (2014)
7. Barna, C., Shtern, M., Smit, M., Tzerpos, V., Litoiu, M.: Model-based adaptive DoS attack mitigation. In: Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2012, pp. 119–128. IEEE Press, Piscataway (2012)
8. Barnett, R.: HOIC, January 2012. https://tinyurl.com/y6en34r3. Accessed 31 Mar 2021
9. Beitollahi, H., Deconinck, G.: Tackling application-layer DDoS attacks. Procedia Comput. Sci. **10**, 432–441 (2012)
10. Bharathi, R., Sukanesh, R., Xiang, Y., Hu, J.: A PCA based framework for detection of application layer DDoS attacks. WSEAS Trans. Inf. Sci. Appl. **9**(12), 389–398 (2012)
11. Bock, K., Patel, D., Hughey, G., Levin, D.: unCAPTCHA: a low-resource defeat of reCAPTCHA's audio challenge. In: 11th {USENIX} Workshop on Offensive Technologies ({WOOT} 2017) (2017)
12. Brewer, D., Li, K., Ramaswamy, L., Pu, C.: A link obfuscation service to detect webbots. In: 2010 IEEE International Conference on Services Computing, pp. 433–440, July 2010
13. Cao, Y., Yang, J.: Towards making systems forget with machine unlearning. In: 2015 IEEE Symposium on Security and Privacy, pp. 463–480. IEEE (2015)
14. Chim, S.: Http proxy middleware, July 2016. https://tinyurl.com/y6td93p4
15. Chu, Z., Gianvecchio, S., Koehl, A., Wang, H., Jajodia, S.: Blog or block: detecting blog bots through behavioral biometrics. Comput. Netw. **57**(3), 634–646 (2013)
16. Cid, D.: Analyzing popular layer 7 application DDoS attacks. Sucuri blog. https://tinyurl.com/y3p7mokb. Accessed 6 Dec 2020
17. Cloudflare. How can an HTTP flood be mitigated?, March 2020. https://www.cloudflare.com/learning/ddos/http-flood-ddos-attack/. Accessed 6 Dec 2020
18. Elsabagh, M., Fleck, D., Stavrou, A., Kaplan, M., Bowen, T.: Practical and accurate runtime application protection against DoS attacks. In: Dacier, M., Bailey, M., Polychronakis, M., Antonakakis, M. (eds.) RAID 2017. LNCS, vol. 10453, pp. 450–471. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66332-6_20
19. Gavrilis, D., Chatzis, I., Dermatas, E.: Flash crowd detection using decoy hyperlinks. In: 2007 IEEE International Conference on Networking, Sensing and Control, pp. 466–470, April 2007
20. Google. reCAPTCHA v3. https://www.google.com/recaptcha/intro/v3.html. Accessed 31 Mar 2021
21. Han, X., Kheir, N., Balzarotti, D.: Evaluation of deception-based web attacks detection. In: Proceedings of the 2017 Workshop on Moving Target Defense, MTD 2017, pp. 65–73. ACM, New York (2017)
22. Imperva. Low orbit ion cannon. https://tinyurl.com/y3wy32fo. Accessed 31 Mar 2021
23. Imperva. 2020 cyberthreat defense report (2020). https://tinyurl.com/y5jmjuzv. Accessed 31 Mar 2021

24. Imperva Incapsula's. Q1 2017 global DDoS threat landscape report, May 2017. www.incapsula.com. Accessed 6 Dec 2020
25. INDUSFACE (2019). https://tinyurl.com/y4c3ywry. Accessed 6 Dec 2020
26. Jan, S.T., et al.: Throwing darts in the dark? Detecting bots with limited data using neural data augmentation. In: 2020 IEEE Symposium on Security and Privacy (SP), pp. 1190–1206. IEEE (2020)
27. Jonker, M., King, A., Krupp, J., Rossow, C., Sperotto, A., Dainotti, A.: Millions of targets under attack: a macroscopic characterization of the DoS ecosystem. In: Internet Measurement Conference (IMC), November 2017
28. Jung, J., Krishnamurthy, B., Rabinovich, M.: Flash crowds and denial of service attacks: characterization and implications for CDNs and web sites. In: Proceedings of the 11th International Conference on World Wide Web, WWW 2002, pp. 293–304. ACM, New York (2002)
29. Kandula, S., Katabi, D., Jacob, M., Berger, A.: Botz-4-sale: surviving organized DDoS attacks that mimic flash crowds. In: Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation, NSDI 2005, vol. 2, pp. 287–300. USENIX Association, Berkeley (2005)
30. Kaspersky. Report finds 18% rise in DDoS attacks in Q2 2019 (2019). https://tinyurl.com/y258rnpm. Accessed 31 Mar 2021
31. Krishnamurthy, B., Wang, J.: On network-aware clustering of web clients. ACM SIGCOMM Comput. Commun. Rev. **30**(4), 97–110 (2000)
32. Leyden, J.: Russian serfs paid three dollars a day to break CAPTCHAs, March 2008. https://tinyurl.com/y2czs7xd. Accessed 6 Dec 2020
33. Liao, Q., Li, H., Kang, S., Liu, C.: Application layer DDoS attack detection using cluster with label based on sparse vector decomposition and rhythm matching. Secur. Commun. Netw. **8**(17), 3111–3120 (2015)
34. Wayback Machine. Internet archive (1996). https://archive.org/web. Accessed 31 Mar 2021
35. Meng, W., et al.: Rampart: protecting web applications from CPU-exhaustion denial-of-service attacks. In: 27th USENIX Security Symposium (USENIX Security 2018) (2018)
36. Mirza, M., Osindero, S.: Conditional generative adversarial nets. arXiv preprint arXiv:1411.1784 (2014)
37. Mosberger, D., Jin, T.: Httperf: a tool for measuring web server performance. SIGMETRICS Perform. Eval. Rev. **26**(3), 31–37 (1998)
38. Najafabadi, M., Khoshgoftaar, T., Calvert, C., Kemp, C.: User behavior anomaly detection for application layer DDoS attacks. In: 2017 IEEE International Conference on Information Reuse and Integration (IRI), pp. 154–161, August 2017
39. Oikonomou, G., Mirkovic, J.: Modeling human behavior for defense against flash-crowd attacks. In: 2009 IEEE International Conference on Communications, pp. 1–6. IEEE (2009)
40. Paxson, V.: Bro: a system for detecting network intruders in real-time. In: Proceedings of the 7th Conference on USENIX Security Symposium, SSYM 1998, vol. 7, p. 3. USENIX Association, Berkeley (1998)
41. Radware. JS cookie challenges, March 2020. https://tinyurl.com/y2bqmtac. Accessed 6 Dec 2020
42. Ranjan, S., Swaminathan, R., Uysal, M., Knightly, E.: DDoS-resilient scheduling to counter application layer attacks under imperfect detection. In: Proceedings IEEE INFOCOM 2006, pp. 1–13 (2006)
43. Selenium. Selenium webdriver (2012). https://tinyurl.com/y6a4czhe. Accessed 6 Dec 2020

44. V. S. Services. Verisign DDoS trends report q2 2016, June 2016. https://verisign.com/. Accessed 6 Dec 2020
45. Sivakorn, S., Polakis, I., Keromytis, A.D.: I am robot:(deep) learning to break semantic image CAPTCHAs. In: 2016 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 388–403. IEEE (2016)
46. Spitzner, L.: Honeytokens, July 2003. https://tinyurl.com/y4gzbjqz
47. STEEL Lab. Frade: Flash crowd attack defense (2021). https://steel.isi.edu/Projects/frade/
48. Wang, J., Yang, X., Long, K.: Web DDoS detection schemes based on measuring user's access behavior with large deviation. In: 2011 IEEE Global Telecommunications Conference - GLOBECOM 2011, pp. 1–5, December 2011
49. Wang, S., Liu, C., Gao, X., Qu, H., Xu, W.: Session-based fraud detection in online e-commerce transactions using recurrent neural networks. In: Altun, Y., et al. (eds.) ECML PKDD 2017. LNCS (LNAI), vol. 10536, pp. 241–252. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-71273-4_20
50. White, B., et al.: An integrated experimental environment for distributed systems and networks. In: Proceedings of the Fifth Symposium on Operating Systems Design and Implementation, Boston, MA. USENIX Association, December 2002
51. Wikipedia. Curse of dimensionality. https://en.wikipedia.org/wiki/Curse_of_dimensionality/. Accessed 6 Dec 2020
52. Wikipedia. Replay attack. https://en.wikipedia.org/wiki/Replay_attack. Accessed 31 Mar 2021
53. Winslow, E.: Bot detection via mouse mapping, September 2009. https://tinyurl.com/y3kbgwuw
54. Xie, Y., Yu, S.Z.: Monitoring the application-layer DDoS attacks for popular websites. IEEE/ACM Trans. Netw. **17**(1), 15–25 (2009)
55. Yatagai, T., Isohara, T., Sasase, I.: Detection of http-get flood attack based on analysis of page access behavior. In: 2007 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, pp. 232–235, August 2007