Don't Tread on Me: Moderating Access to OSN Data with SpikeStrip

Christo Wilson, Alessandra Sala, Joseph Bonneau[†], Robert Zablit and Ben Y. Zhao
Department of Computer Science, U. C. Santa Barbara, Santa Barbara, USA
[†]Computer Laboratory, University of Cambridge, Cambridge, UK
{*bowlin, alessandra, rzablit, ravenben*}*@cs.ucsb.edu, jcb82@cl.cam.ac.uk*

Abstract

Online social networks rely on their valuable data stores to attract users and produce income. Their survival depends on the ability to protect users' profiles and disseminate it to other users through controlled channels. Given the sparse user adoption of privacy policies, however, there is increasing incentive and opportunity for malicious parties to extract these datasets for profit using automated "crawlers" and "screen-scrapers." With the arrival of distributed botnets and low-cost hosted VMs, attackers can perform fast, distributed crawls that evade traditional detectors and rate limiters. We propose SpikeStrip, a server add-on that uses light-weight link encryption to isolate and rate limit crawlers. We experiment with real OSN data, and show that SpikeStrip successfully curtails sophisticated, distributed crawlers while imposing minimal server throughput overhead and inconvenience to end-users.

1 Introduction

The wealth of information hosted by online social networking (OSN) sites make them high-value targets for spammers looking to harvest e-mail addresses and personal details for use in phishing and malware campaigns [1, 17, 26, 28]. This presents a problem for OSN operators, who are torn between opposing desires for openness and security. On one hand, openly accessible content is necessary for services like search and targeted advertising that drive new users, traffic, and ultimately revenue to OSNs. On the other hand, open access policies endanger the privacy of user's personal information, which can then undermine confidence in OSNs and threaten their income. Researchers have already demonstrated the extent of this problem by downloading over 15% of the Facebook user-base in 2008 [29].

OSN users are unwilling or unable to solve this issue on their own. While most OSNs offer privacy settings to help users secure their personal information, studies have shown that the majority of users do not use these features [19, 29]. This issue is compounded by recent trends in OSN privacy policies that encourage users to set their default settings to "publicly viewable" [2]. Even though less than 50% of OSN users bother to change their privacy settings from the (permissive) defaults, users still have expectations of privacy that are violated when crawlers gather large amounts of data from OSNs. Thus, it falls to OSN operators to implement measures to prevent the large scale crawling and scraping of data from their websites.

While standards have been enacted that attempt to regulate the behavior of web crawlers [25], following these guidelines is a voluntary measure. In the past, these guidelines were sufficient since the effects of rogue crawlers were limited by bandwidth and computational resource restrictions. However, ubiquitous broadband, cheap clouds, and botnets have increased the capabilities of crawlers dramatically. These technologies have lowered the barrier of entry to the point where anyone can set up a highly parallel, distributed crawler that is capable of traversing even the largest websites in a matter of hours. In fact, recent startups have made mass crawling a cheap commodity [23].

Given the problems rogue crawlers can cause, coupled with their expanding range of capabilities, it is imperative that OSNs equip themselves with countermeasures against them. However, stopping crawlers is a challenging proposition. Identifying crawlers by their IP address, as some websites have done [21], is easily sidestepped through the use of proxies, distributed botnets, or virtual machines hosted in the cloud. Attempting to use HTTP functionality such as per-account session keys for user tracking fares better, but is still insufficient. For example, Facebook bans user accounts that are suspected to be crawling on a daily basis. However, banning a single account does not invalidate the crawler's *frontier*, *i.e.* queue of uncrawled URLs [16]. A dedicated attacker can simply create new accounts and continue unhindered.

We believe these weaknesses are not fundamental to the web architecture. Using the right mechanisms, greater content control can be given to OSNs while preserving open access for legitimate users. Our solution relies on "link encryption," a server-side primitive that encrypts hyperlinks within served HTML pages using a combination of session keys and server-side secret keys. Link encryption binds links to the session key of the active browser. Thus, links observed within one session cannot be traversed by other sessions, creating a unique "view" of the website for each client. Users cannot switch sessions while browsing, because links observed in the original view will no longer be valid within the new view. This tight binding between browsers and their session keys allows the server to maintain per-session traffic counters and reject requests above a rate limit chosen by the administrator.

For normal users, link encryption integrates unobtrusively into their OSN browsing experience. However, crawlers attempting to traverse the site using a single session key can be trivially identified and throttled by the rate limiter. If the crawler attempts to circumvent the rate limit by switching sessions the links in its queue will no longer be valid, since they are bound to its original session key. This forces the crawler to restart its traversal from scratch, effectively defeating it.

We implement a full prototype of our "link encryption" technique in SpikeStrip, an Apache module that works out of the box with existing Apache setups. SpikeStrip is transparent to site administrators and end-users: other than installing the Apache module, no changes are necessary to the web server, the site's content, or client's web browsers. SpikeStrip gives site admins fine grained control over which individual pages should be protected, as well as allowing for the IP addresses of known, friendly crawlers to be whitelisted.

To evaluate SpikeStrip, we perform microbenchmarks, and repeatedly crawl a SpikeStrip-protected Apache server hosting anonymized data from Facebook. Our measurements show that SpikeStrip successfully bounds crawler traffic within specified rate limits, and has minimal impact on aggregate server throughput. Finally, SpikeStrip is available for download and immediate use.

2 Crawler Defenses for OSNs

In this section, we examine existing mechanisms used to control access to online data, and discuss why they provide insufficient defenses against rogue crawlers. Our discussion is organized by increasing complexity of mechanism: we begin with simple, passive measures, then move on to active defenses that rely on identifying individual users. We end by discussing authentication-based strategies used by existing OSNs.

2.1 Passive Defenses

The most basic, passive crawler control mechanisms rely on crawlers to identify themselves up-front and obey rules posted by websites. The primary example of this is the Robot Exclusion Protocol (a.k.a. robots.txt) [25]. This protocol allows webmasters to set up lists of prohibited URLs on their sites that should not be traversed by crawlers. Since compliance with this protocol is voluntary, it has no deterrent effect on rogue crawlers.

Another example of passive defense are Apache modules like mod_robots that perform website access control based on HTTP "User-Agent" and "Referer" headers. Well-meaning commercial crawlers and standard web browsers reliably report information about themselves using these headers. However, these headers can be arbitrarily modified by attackers in order to bypass access control mechanisms.

2.2 Active Defenses & Client Identification

Active defenses against crawlers attempt to use network- and transport-layer information to uniquely identify individual clients. This enables clients' browsing behaviors to be tracked. Armed with this information, servers can identify crawlers as clients sending abnormally large numbers of requests within a short time frame. Once crawlers have been identified, retaliatory actions can be taken against them.

Client tracking seems like a straightforward process. Some websites track by IP address [21]; other possible identifying tokens include TCP port numbers and SSL session IDs. In practice, however, tracking users by these low-level identifiers is fundamentally flawed. Attackers have access to a sufficiently large pool of possible identifiers that they can perform a Sybil attack [11] against a website's tracking system, effectively circumventing it.

As an example, consider an IP-based client tracker. The assumption underlying this mechanism is that each IP address represents a unique client, and that clients are not colluding to circumvent tracking. However, there is not a 1-to-1 correlation between IPs and clients. A distributed crawler is a single, large "client" that uses multiple machines/IPs to collude towards a common goal. Tracking any one IP does not reveal the crawler's overall behavior. Dedicated attackers can easily gain access to large blocks of IP space by virtualizing in a cloud, forwarding requests through open proxies such as Tor[10], or leveraging a botnet, in order to implement this attack.

Tracking TCP ports and SSL session IDs fails for similar reasons. Rogue crawlers can restart connections at any point in time in order to switch ports/sessions, thus appearing to the server as a different client.

Combining identifiers from different layers does not solve this issue. For example, servers can derive session keys by hashing clients' IPs. This prevents clients from switching their identities by purging their session, since their key is deterministic. However, crawlers can still switch IPs and thus gain new credentials.

2.3 Authentication Based Defenses

OSNs generally require users to create accounts and authenticate themselves using valid e-mail addresses, CAPTCHAS, and passwords in order to access private content. These access controls are implemented using HTTP "Cookie"-based session keys. This approach to client identification is superior to previously discussed techniques because the pool of valid session keys is completely under the server's control. Unlike network-layer tokens, clients can not easily acquire huge numbers of user accounts due to the authentication measures that secure the account creation process. Similarly, session keys assigned to clients after they log in cannot be modified or erased. In either case, the server will notice the invalid/missing credentials and reject the client's requests.

While authenticated user accounts help resolve the client identification/tracking issue, they still fail to enable strong countermeasures against rogue crawlers. The reason for this is that web clients access a particular page using a common URL accessible to all clients, *i.e.* URLs are *session-independent*. Distributed crawlers exploit this fact: an outgoing link in a page parsed by crawler t_1 using session key k_1 can be given to crawler t_2 and retrieved using session key k_2. Even if crawlers get their user accounts banned and are forced to switch to new ones, all the URLs they have enqueued in their frontier are still valid, allowing the crawl to continue unabated.

Facebook's current crawler defenses suffer from this flaw. Facebook monitors the number of requests attributed to each user per twelve-hour period, and bans accounts that breach some threshold. However, because the crawler's frontier is unaffected by banning accounts it can simply switch to new accounts and continue, as researchers have demonstrated [29]. Shortening the measurement interval between bans does not solve the problem: if the interval is long (on the order of hours) an attacker can create enough accounts to counteract attrition. Conversely, if the interval is short (seconds, minutes) the OSN risks accidentally banning normal users.

Some OSNs, notably Twitter, enforce strict quotas on the number of page hits user accounts can generate per-hour. This approach is more restrictive than Facebook's periodic bans, however, it still does not solve the fundamental problem. As soon as a crawler drains the quota of one account, it can switch to another account and pick up right where it left off. These accounts can be recycled each hour after the quotas reset, meaning that a crawler does not need a huge number of accounts in order to continuously crawl at high throughput.

2.4 Summary

In this section we have outlined the key shortcomings of current access control mechanisms used to stop rogue crawlers. In summary:

- Passive measures like robots.txt that rely on crawlers to voluntarily identify themselves and obey posted rules can be ignored by attackers.
- Identifying clients using network/transport-layer information (*e.g.* IP address, TCP port, SSL session) is insufficient, as attackers have sufficient resources to mount Sybil attacks against these systems.
- Authenticated user accounts are good client identifiers, but fail to couple browsers to the content they view, thus rendering punishment of misbehaving crawlers ineffectual.

In this work, our aim is to develop a new mechanism that OSNs can leverage to overcome these problems. This technology should facilitate stronger access controls over content, while preserving open access for legitimate users and friendly search-engine crawlers.

3 SpikeStrip Design

In the previous section we detailed why existing technologies do not offer sufficient content access control to OSN operators. Our aim is to create a new system that overcomes these issues and works with existing OSN authentication mechanisms to improve security against rogue crawlers. In particular, we seek to create a system that exhibits the following properties:

- Browsers are identified by individual sessions tied to authenticated user accounts, allowing administrators to apply strict per-session rate limits.
- Administrators can selectively apply our technique to subsets of pages on a site.
- Administrators can whitelist the IP address ranges and/or domains of legitimate crawlers so they can continue to index content normally.
- Our technique is completely transparent to existing web browsers. Users accessing the site observe no visible changes in semantics and negligible drop in server performance.

We propose a server-side primitive called "link encryption" to combat rogue crawlers. We utilize link encryption as a core component in SpikeStrip, a web server add-on that allows admins to moderate data access and prevent crawling by securely identifying and rate limiting individual sessions. In this section we describe the high level design of SpikeStrip, with particular emphasis on using link encryption for session identification and crawler mitigation. We also discuss how to scalably track

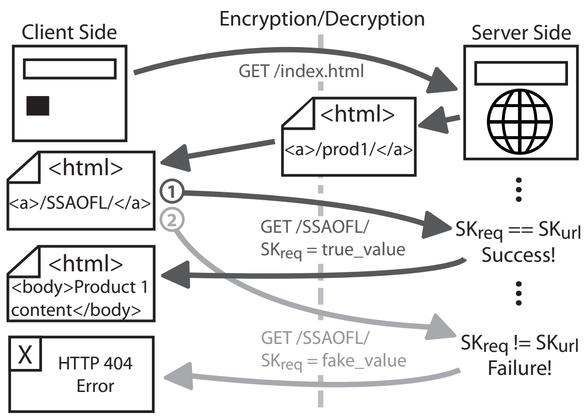

Figure 1: SpikeStrip link encryption. After requesting a site's homepage, a client may browse as long as their session key SK_{req} remains at its true value (1). If the client modifies SK_{req} it will no longer match SK_{url}, the session key embedded in the encrypted URL, resulting in request failure (2).

per-session traffic load while imposing minimal overhead on the web server.

3.1 Link Encryption

As discussed in Section 2.3, URLs on today's websites are *session independent*, meaning they are the same for all browsers regardless of user account/session key. Existing information at the TCP/IP and HTTP levels cannot uniquely bind browser instances to the content they are viewing, which makes it difficult to combat crawlers. To solve this, SpikeStrip uses link encryption to create unique, per-session "views" of the protected website that forcibly tie each client to their session key.

Client Views. SpikeStrip introduces link encryption as a technique to couple a browser to the content it is visiting. SpikeStrip appends each user's session key SK to served URLs and then encrypts the result using a server-side, secret symmetric key PK. SpikeStrip uses a random initialization vector $salt$ to make each URL unique. $salt$ is appended to the URL after encryption, so that it can be recovered and used for decryption when the encrypted link is requested by a client. Link encryption can be summarized using the formula: $\mathsf{new\_url} = \left\langle \langle \mathsf{url}, \mathsf{SK} \rangle_{PK}^{salt}, \mathsf{salt} \right\rangle$ where $\langle \cdot \rangle_{*}^{iv}$ denotes encryption using the subscripted key with the superscripted initialization vector.

Link encryption prevents crawlers from tampering with their session keys, *e.g.* to evade session-based traffic counters. Figure 1 shows an example of how link encryption enforces session key integrity. A client visits *www.example.com*, and receives a new session key and a copy of the main page whose links are all encrypted. The server decrypts each requested link and compares the embedded session key SK_{url} to the client-reported session key SK_{req}. If SK_{req} is modified the comparison will fail, and the client must restart its traversal of the website from an unencrypted "entry point", *i.e. /index.html*. Browsers attempting to access protected content directly via an unencrypted link also get redirected by SpikeStrip to a safe entry point, thus ensuring that crawlers cannot circumvent SpikeStrip.

Implications of Link Encryption. Link encryption allows web servers to reliably track clients by session key which poses serious problems for crawlers. If a crawler traverses pages using a single session, it will be trivially identified. However, if the crawler attempts to obfuscate its behavior by distributing across many sessions, its frontier becomes partitioned, since each queued URL is tied to one of the crawler's many sessions.

Figure 2 illustrates the ramifications of link encryption on a crawler's frontier. Initially, the crawler starts at the site's homepage. As time goes on, the crawler progressively covers more pages and expands its frontier. At some point, the crawler is forced to change its session key, *e.g.* because its session expired or was banned for malicious activity. Normally, switching sessions would not be a problem: all URLs in the frontier would be immediately accessible under the new session. However, SpikeStrip couples all URLs to the browser's session key; when the crawler changes sessions all URLs in it's frontier are invalidated. Hence, it must restart its traversal at *index.html*, and duplicate work by re-traversing pages to return to its previous position.

Given that OSNs already ban crawling accounts on a coarse grained schedule, this means that multi-session crawlers will periodically have large portions of their frontier invalidated. Recovering from these losses is non-trivial: returning to an arbitrary node in the web graph requires storing the full traversal path from the entry point node to the target node. Keeping this amount of state for each URL of a large, power-law social graph is technically infeasible, thus effectively defeating crawlers.

An added benefit of tracking clients by session keys is that it overcomes problems associated with traditionally IP-based tracking. For example, multiple clients hidden behind a single proxy or NAT can be successfully disambiguated by session. Similarly, crawlers attempting to evade detection by leveraging a botnet can no longer use their large pool of IP addresses to their advantage, since tracking is done by session and not IP.

Link Opacity. Besides the primary implications of link encryption discussed above, there is a secondary effect that warrants discussion. Unlike standard URLs, it is not immediately apparent where encrypted links are pointing to. SpikeStrip's link encryption makes links "opaque." Encrypted links are randomly salted, meaning

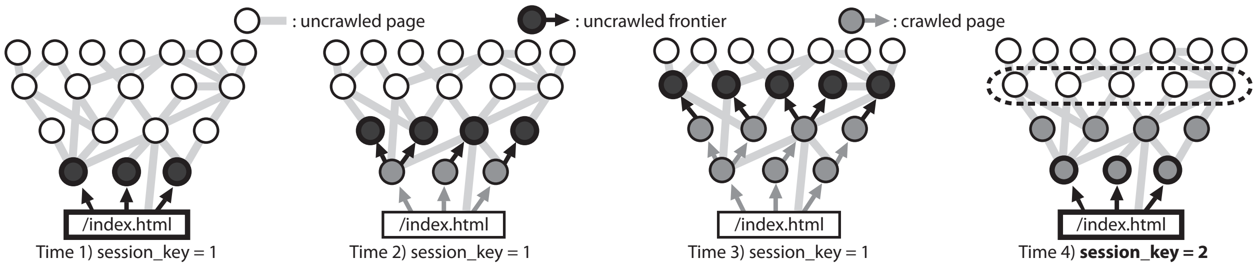

Figure 2: Implications of link encryption. Starting at time 1, a crawler begins traversing a website protected by SpikeStrip. By time 3, its frontier is deep within the target website. At time 4, the crawler is forced to change session key, thus invalidating all URLs in its frontier. It must restart at the site's root page, and re-traverse already covered pages to return to its former position.

that even direct comparison will not reveal if two links point to the same target.

While link opacity makes it more challenging for crawlers to traverse a site, SpikeStrip does *not* rely on this property for any of its security guarantees. The reason is that links usually have unencrypted meta-data associated with them that an attacker can use to disambiguate encrypted links. For example, links to Facebook profiles are accompanied by that persons name, which a crawler can use to infer the links destination.

3.2 Rate Tracking and Limiting

We have demonstrated how SpikeStrip reliably identifies web clients through session keys. The next step is to differentiate crawlers from normal users and apply scalable rate limits. SpikeStrip does this by performing rate tracking on client sessions, and dropping requests that exceed a predefined rate limit. The rate limit is greater than the maximum request rate for normal users, and thus SpikeStrip does not hinder normal browsing behavior.

The key challenge here is how to perform rate tracking for high volume sites with millions of daily visitors. To efficiently track sessions with minimal storage overhead, SpikeStrip uses Counting Bloom Filters (CBF) [14], a probabilistic data structure whose variants have often been used in networking applications [6, 9, 27]. SpikeStrip uses the "d-left" CBF variant, shown in measurements to be the fastest and most accurate of the bunch [4, 30]. dlCBF allows for extremely fast set element counting and prevents SpikeStrip from limiting HTTP throughput at the server. As we show in Section 5, dlCBF is extremely space and time efficient, and is scalable enough to support even the largest OSN site.

3.3 Balancing Security vs. Openness

Server admins can specify which pages should remain open and unprotected by SpikeStrip using a *URL whitelist*. Whitelisting allows site administrators to manage tradeoffs between content security and openness on the Web. For example, encrypting all links on a website prevents search engines from being able to properly index that site, and also prevents users from bookmarking and/or sharing links over email and IM. Whitelisting enables site admins to provide "perma-links" to content, a concept that is already used on Facebook for providing users with public, permanent links to photos.

Additionally, OSNs want crawlers from major search engines to traverse them quickly and often, so that the search index reflects the latest updates to their sites. Admins can use a *crawler whitelist* to mark the IP ranges of known crawlers. Whitelisted crawlers may bypass SpikeStrip link encryption and rate limiting.

4 A SpikeStrip Prototype

In this section, we describe mod_spikestrip, our SpikeStrip implementation for Apache 2.x. mod_spikestrip is written in C and is portable to most platforms supported by Apache [1]. mod_spikestrip works seamlessly with all standard Apache modules and content generators (static HTML, PHP, CGI, etc).

Figure 3 depicts the flow of HTTP requests through Apache when mod_spikestrip is installed. After each HTTP request is received and its headers are parsed, the first two SpikeStrip handlers execute. This gives mod_spikestrip the opportunity to perform rate limiting immediately, before time is potentially wasted processing a dead request. mod_spikestrip extracts the session key from the request's "Cookie" header and uses it as the key to test and increment a counter in a global d-left CBF. If the counter value exceeds an administratively defined limit, an HTTP 503 "Resource Unavailable" message is returned to the client. If the request is below the rate limit, or it doesn't have a session key, control passes to the next handler. A background thread periodically clears the dlCBF in order to restart request counting on a configurable time schedule.

If the rate limiter accepts the request then the link decryption handler is run. Encrypted URLs are decrypted

[1]GPL mod_spikestrip source code is available for download at `http://www.cs.ucsb.edu/~bowlin/projects.html`.

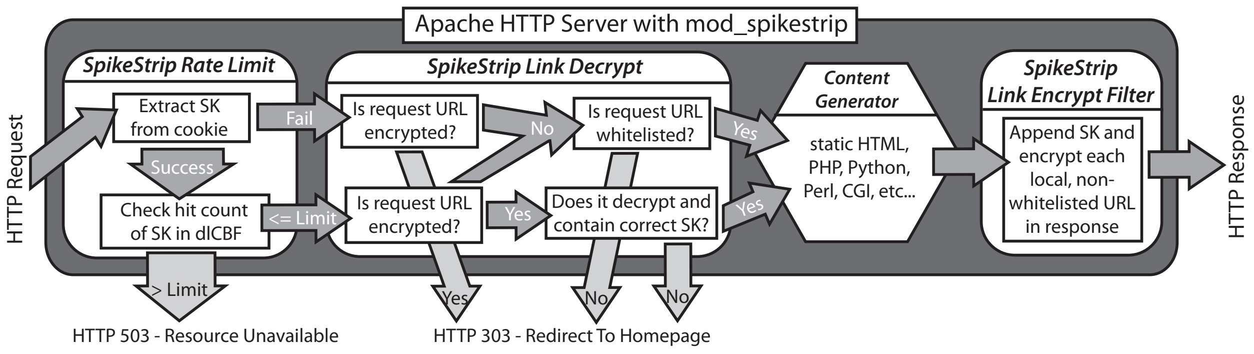
Apache HTTP Server with mod_spikestrip
SpikeStrip Rate Limit
Extract SK from cookie
Success
Check hit count of SK in dlCBF
> Limit
HTTP 503 - Resource Unavailable
HTTP Request
Fail
<= Limit
SpikeStrip Link Decrypt
Is request URL encrypted?
No
Is request URL whitelisted?
Is request URL encrypted?
Yes
Does it decrypt and contain correct SK?
Yes
No
No
HTTP 303 - Redirect To Homepage
Yes
Yes
Content Generator
static HTML, PHP, Python, Perl, CGI, etc...
SpikeStrip Link Encrypt Filter
Append SK and encrypt each local, non-whitelisted URL in response
HTTP Response

Figure 3: Flow chart of Apache with mod_spikestrip enabled. SK = session key.

using the server-side key and the embedded session key is compared to the client's key. If the two keys match the request is processed normally by Apache. Requests that do not have session keys, or are not encrypted, are only allowed to URLs that have been explicitly whitelisted by the administrator. Requests failing to meet any of these conditions are immediately dropped, and an HTTP 303 "Redirect" is returned to the client. This redirect points to an implicitly whitelisted page specified by the administrator (most likely the site's homepage), that will assign the client a valid session key if they do not have one.

mod_spikestrip's third and final handler executes after the content generator has created a response to the HTTP request. The link encryption handler parses output HTML and replaces plaintext URLs with encrypted versions. Only non-whitelisted, local links get encrypted, since it does not make sense to encrypt links pointing to external websites.

Encryption/Decryption Details mod_spikestrip uses 256-bit AES in CBC-mode with a random 20 byte initialization vector to encrypt URLs. The initialization vector is appended to the encrypted URL and the whole string is base-64 encoded so it is safe to transmit over HTTP. Decryption is performed in the reverse order.

d-left CBF Details mod_spikestrip's dlCBF is tuned for maximum performance and low probability of bucket overflow. We use the optimal parameter settings derived in [4]: $\mathsf{number\ of\ tables\ d} = 4$, $\mathsf{cells\ per\ bucket} = 8$, and $\mathsf{target\ load} = 75\%$ (6 items per bucket). The other important parameters for the dlCBF (acceptable false-positive error rate and maximum capacity) are configurable by the admin. For speed we use fixed size 1-byte counters and 1 or 2-byte fingerprints, depending on the target error rate. In the worst case with 2-byte fingerprints, the size of our dlCBF is $max\_capacity * 4$ bytes. Practically, this overhead is quite reasonable: a site expecting to serve 10 million unique users every second (an absurdly high estimate) would only need a dlCBF ~ 40 megabytes in size.

Our dlCBF implementation uses interprocess shared memory to store its hash tables. This is necessary because Apache uses a multi-process/-threaded architecture, across which request counting must be consistent. Normally, access to shared data structures must be mediated by locks. However, because the probability of hash collisions in the dlCBF is negligible, locking it during insertions is not necessary. Additionally, because the dlCBF is frequently cleared (to reset the rate limit counters after each counting interval), errors due to concurrency are transient and can be safely ignored.

5 Benchmarks and Evaluation

In this section we evaluate our SpikeStrip prototype to ascertain its computational performance and its effect on crawlers. We begin with server benchmarks of a stock Apache instance compared to Apache with SpikeStrip. Our results show that SpikeStrip imposes only a modest 7% performance penalty on Apache. Next, we examine the effect of SpikeStrip on crawlers attempting to mine an artificial OSN website hosted by our lab. Results indicate that SpikeStrip successfully rate limits crawlers to only a small fraction of our web server's total bandwidth, increasing crawl times by an order of magnitude.

5.1 Evaluation Setup

To perform our experiments, we set up a mock OSN website and initiated crawler attacks against it. Crawlers and web servers run on Dell PowerEdge 1750 servers. Databases run on Dell PowerEdge 1950 servers. All machines are directly connected over gig-Ethernet. We used a typical LAMP setup, including Apache 2.2, .Python 2.6, Django 1.1, and MySQL 5.1 [2]. All machines run CentOS, kernel version 2.6.

Our test OSN website, called *Fakebook*, is populated with anonymized data gathered from the London re-

[2]Apache, MySQL, and Django config files for our benchmark servers are available for download at `http://www.cs.ucsb.edu/~bowlin/projects.html`.

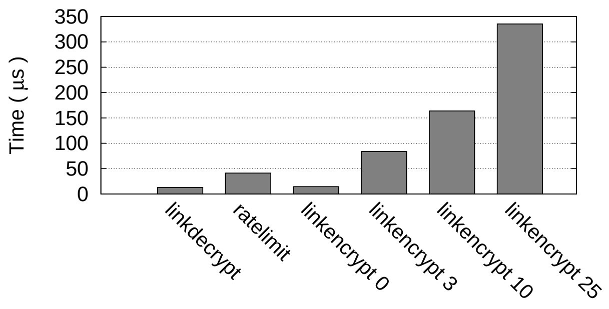

Figure 4: Benchmark times of SpikeStrip routines, averaged over 1,000 HTTP requests.

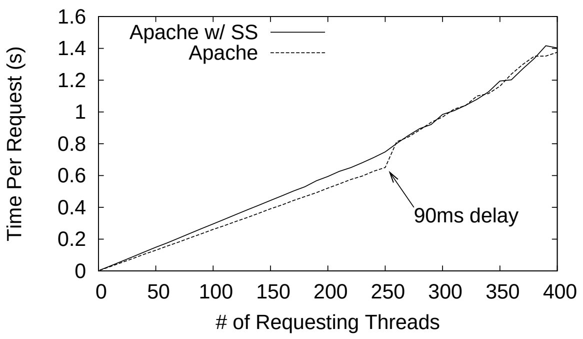

Figure 5: Average time for Apache to respond to HTTP requests w/ and w/o SpikeStrip.

gional network on Facebook [29]. It includes data from ∼1.2 million users, resulting in a website with 3,523,620 unique pages. The site has three page templates: a homepage that displays ten random links to user profiles, a user profile display, and a friend list display (paginated with twenty friends per page).

5.2 SpikeStrip Benchmarks

Our first set of tests assess the performance of mod_spikestrip by performing microbenchmarks of Apache with and without SpikeStrip. For this test we chose four pages from Fakebook (containing 0, 3, 10, and 25 links, respectively) and made 1,000 request for each page using ApacheBench. For these experiments only, we instrumented mod_spikestrip to record the execution times of its key procedures to the Apache log file.

Figure 4 charts the average execution times for SpikeStrip's key procedures. Decrypting links from incoming requests and checking the dlCBF for rate limit violations execute extremely fast. Link encryption, however, takes more time. SpikeStrip exhibits a baseline ∼10 microsecond overhead due to HTML parsing for each outgoing page, as shown by the results for the page with zero links. As the number of links grows, so does the execution time, since each encryption necessitates a round of AES, as well as a memory move to make room in the output stream for the encrypted link.

Figure 5 shows the practical implications of SpikeStrip overhead on a typical Apache setup. We measure Apache's throughput by using Apachebench to request a single user profile 1,000 times as the number of concurrent requesting threads increases. On average, SpikeStrip adds only ~ 30 milliseconds of delay to responses, or a 7% reduction in performance. Even in the worst case, Spikestrip only adds ~ 90 milliseconds of delay, which is small enough to be imperceptible to most users.

5.3 Crawler Efficiency Tests

In this section we evaluate the deterrent effects of SpikeStrip on rogue crawlers. Our distributed crawler is written using Crawl-e [3], the same framework used to download 10 million Facebook pages in a previous study [29]. The crawler was executed on three machines with 200 crawling threads each. Crawls began with a single seed: the homepage of Fakebook. For this test, we set up a load-balanced cluster of 10 Apache instances, in order to simulate a reasonably well provisioned website.

SpikeStrip Configuration. According to recent figures, the average Facebook user spends ∼14 minutes online each day, and views 26 pages per visit [12, 13]. In order to realistically accommodate users who spend more time on Facebook than the average, for our experiments we assume super-users who visit 1,000 pages every hour. We use this is as the maximum rate limit setting for mod_spikestrip: 1,000 requests per hour, resulting in an overall rate of ∼0.25 requests per second (RPS).

Setting the rate limit time frame by hour permits bursty traffic from web users while also preserving the overall, low target rate. For example, if the rate limit were 1 request per 4 seconds (also an effective rate of 0.25 RPS), a user who quickly opens multiple pages in different browser tabs would trigger the rate limit, which is undesirable. Set to 1,000 requests per hour, bursty requests from users are not hindered, but crawlers quickly exhaust their allotted requests and are forced to sit idle for the remainder of the hour.

Since we use a cluster of 10 Apache instances with round-robin load balancing, the per-server rate limit is set to 100 requests per hour ($100*10 = 1000$). SpikeStrips's dlCBF capacity is set to 8 million, with a target error rate of 0.0004%. These values were chosen to minimize hash collisions by over-provisioning the dlCBF.

Results. Figure 6 depicts the results of crawlers traversing Fakebook. Baseline results were gathered

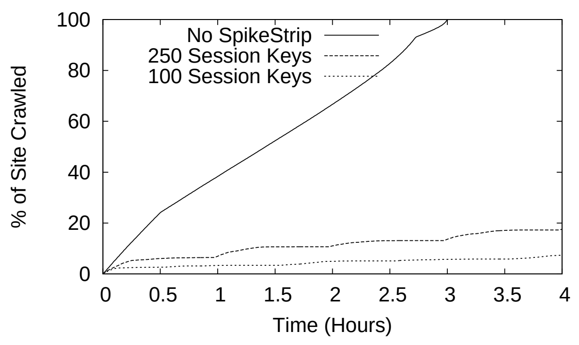

Figure 6: A crawler's rate of progress traversing Fakebook is proportional to how many session keys it uses.

by running a complete crawl of Fakebook without SpikeStrip. For the rest of the experiments SpikeStrip was active, and thus the per-session rate limit prevents the crawlers from traversing the site in a timely fashion. Each rate limited crawler achieves an RPS rate equal to the number of session keys it holds, multiplied by the configured rate limit (0.25 RPS). The effect of the rate limiter is most visible for the 250 key-crawler: at the beginning of each hour the rate limit counters reset to zero, and the crawler immediately downloads pages until its request allotment is depleted. Its progress then stalls until the next hour marker passes. Ultimately, even with 250 session keys the crawler is far from achieving equal performance with the unprotected baseline standard.

There is a subtle limitation of SpikeStrip that can be observed in Figure 6. As the number of session keys held by the crawler rises, so does its aggregate throughput. Theoretically, if an attacker acquired a massive number of keys it could circumvent SpikeStrip. The success of this attack hinges on the crawler's ability to quickly create thousands of user accounts, which seems unlikely since crawlers would first need to break the authentication measures surrounding account sign-up.

6 Deployment Considerations

As with any piece of software that must work with existing systems, there are additional points to consider when deploying SpikeStrip. In this section, we will identify potential challenges in deploying SpikeStrip and detail workarounds when possible.

Multiple Data Centers. Protecting web servers at geographically diverse data centers with SpikeStrip necessitates two considerations. First, if each data center uses a different secret key, then URLs will not be portable between data centers. This only causes problems for the rare instances where mobile clients migrate between data centers in the middle of a single session.

The second consideration concerns the rate limiters. A crawler could attain a d times speedup by directing requests to all d data centers. The best mitigation strategy against this is to reduce the acceptable rate by some fraction of d. Realistically, the significance of an O(d) time speedup is small enough to be negligible.

Caching. SpikeStrip is compatible with server-side caches such as Memcached. Since Memcached operates behind the web server-tier, unencrypted URLs are visible to it, and it can operate normally. However, SpikeStrip is incompatible with client-side caches such as Akamai, because link encryption prevents them from identifying cache hits. This issue can be mitigated by having customers share their server-side SpikeStrip keys with Akamai so they can decrypt links. Given the close relationships between Akamai and its customers, this stipulation seems reasonable.

Secret Key Compromise. While server-side secrets are common for websites (usually database login credentials), the compromise of this information is still a concern. Changing SpikeStrip's secret key after a compromise causes all active sessions to become invalidated, and thus browsers must restart their sessions. Advanced versions of SpikeStrip can minimize disruption to users by maintaining support for an outgoing key while using a new key for new sessions. As long as key refresh periods are significantly longer than most user sessions, servers can hide the impact of key refreshes.

Javascript Generated Links. Links that are dynamically generated client-side by Javascript cannot be encrypted by SpikeStrip since they do not originate from the server. Care should be taken when generating links client-side, as these plain-text URLs must be whitelisted with SpikeStrip. Alternatively, client-side link generation can be replaced with calls to server-side AJAX methods that serve the same purpose. Links returned via AJAX can be safely encrypted by the server.

API Crawling. Many OSNs offer APIs that allow developers to interface with the information hosted by the OSN. Even if screen-scraping is prevented, these APIs offer crawlers a secondary channel to retrieve information about users. Fortunately, access to most OSN API platforms requires authenticated developer credentials, much like normal users need accounts to login. Since access to OSN APIs is mediated by authenticated sessions, SpikeStrip can be leveraged to hinder crawlers attempting to mine these channels.

7 Related Work

Crawlers. The first public search engine relied on data compiled by WebCrawler [24]. Commercial search engines soon followed with their own crawlers [5, 16]. Crawling research has focused on algorithmic methods to improve crawl speed and effectiveness. Page *importance* metrics [5] can be used to augment crawlers with page *selection* strategies that prioritize the uncrawled frontier to improve expected returns [7, 8, 10]. Unlike rogue crawlers, academic and commercial crawlers are all "well-behaved:" they obey robots.txt rules and do not flood individual web sites with requests.

Apache Security A number of security related Apache modules exist. mod_cband, mod_evasive, mod_bandwidth, and mod_limitipconn implement HTTP request rate limiting based on IP address and target URL. mod_security implements an application-level firewall, but does rate limit requests or address rogue crawlers.

DDoS Defenses. DDoS mitigations offer complementary protection to that provided by SpikeStrip. Router level solutions [20, 22] can stop traffic floods before they cripple the target web server. However, DDoS flood detectors that rely on disparities in ingress and egress traffic [15] are not sufficient to catch crawlers disguised as legitimate browsers. DDoS defense via dynamic injection of puzzles into the browsing stream [18] can curb crawlers, but may also potentially annoy users.

8 Conclusion

Today's social network users are increasingly careless about protecting their information online. Therefore, it falls to OSN providers to prevent malicious crawlers from harvesting user information for use in spam, phishing and malware campaigns. Through the application of our novel link encryption primitive, our SpikeStrip prototype demonstrates that access to content can be successfully moderated without limiting open access to legitimate users or imposing significant performance overheads on web servers. SpikeStrip effectively throttles rogue crawlers by performing per session rate limiting, all while allowing benevolent crawlers uncompromised access to data. By incorporating page-level whitelists, SpikeStrip provides site administrators with access control on the granularity of individual pages. Finally, SpikeStrip does not require changes to user's browsers, existing web servers, or website contents, and it is freely available now for immediate use.

References

[1] ACOHIDO, B. Phishing attack spreads through facebook. USA Today Blog, May 2009.

[2] BANKSTON, K. Facebook's new privacy changes: The good, the bad, and the ugly. EFF, December 2009.

[3] BOE, B., AND WILSON, C. Crawl-e: Highly distributed web crawling framework written in python. Google Code.

[4] BONOMI, F., ET AL. An improved construction for counting bloom filters. In *Proc. of ESA* (2006).

[5] BRIN, S., AND PAGE, L. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst. 30*, 1-7 (1998), 107–117.

[6] BRODER, A., AND MITZENMACHER, M. Network applications of bloom filters: A survey. In *Proc. of Allerton* (2002).

[7] CHAKRABARTI, S., BERG, M. V. D., AND DOM, B. Focused crawling: a new approach to topic-specific web resource discovery. In *Computer Networks* (1999).

[8] CHO, J., GARCIA-MOLINA, H., AND PAGE, L. Efficient crawling through url ordering. *Comput. Netw. ISDN Syst. 30*, 1-7 (1998), 161–172.

[9] CZERWINSKI, S. E., ZHAO, B. Y., HODES, T., JOSEPH, A., AND KATZ, R. An architecture for a secure service discovery service. In *Proc. of MobiCom* (1999).

[10] DILIGENTI, M., COETZEE, F., LAWRENCE, S., GILES, C. L., AND GORI, M. Focused crawling using context graphs. In *Proc. of VLDB* (2000).

[11] DOUCEUR, J. R. The Sybil attack. In *Proc. of IPTPS* (March 2002).

[12] And the most engaging social network is. Pingdom, January 2010.

[13] Top u.s. web brands and site usage: December 2009. Nielson Wire, January 2010.

[14] FAN, L., CAO, P., ALMEIDA, J., AND BRODER, A. Z. Summary cache: A scalable wide-area web cache sharing protocol. *ACM/IEEE ToN 8*, 3 (1998), 254–265.

[15] GIL, T. M., AND POLETTO, M. Multops: a data-structure for bandwidth attack detection. In *Proc. of SSYM* (2001).

[16] HEYDON, A., AND NAJORK, M. Mercator: A scalable, extensible web crawler. *World Wide Web 2* (1999), 219–229.

[17] JONES, K. C. Facebook expands security tools while combating phishing attack. InformationWeek, May 2009.

[18] KANDULA, S., KATABI, D., JACOB, M., AND BERGER, A. Botz-4-sale: Surviving organized ddos attacks that mimic flash crowds. In *Proc. of NSDI* (May 2005).

[19] KRISHNAMURTHY, B., AND WILLS, C. E. Characterizing privacy in online social networks. In *Proc. of WOSN* (2008).

[20] MAHAJAN, R., ET AL. Controlling high bandwidth aggregates in the network. *ACM SIGCOMM Computer Communication Review 32* (2002), 62–73.

[21] MAY, K. Ryanair asks microsoft to help block 'scrapers'. Travolution, August 2008.

[22] MIRKOVIC, J., AND REIHER, P. D-ward: A source-end defense against flooding denial-of-service attacks. *IEEE Trans. Dependable Secur. Comput. 2*, 3 (2005), 216–232.

[23] NAONE, E. A web spider for everyone. MIT Technology Review Online, September 2009.

[24] PINKERTON, B. Finding what people want: Experiences with the webcrawler. In *Proc. of WWW* (1994).

[25] A standard for robot exclusion, June 1994. `http://www.robotstxt.org/wc/robots.html`.

[26] SELTZER, L. Koobface smacks facebook users. PC Magazine Blog, December 2008.

[27] SINGH, S., ESTAN, C., VARGHESE, G., AND SAVAGE, S. Automated worm fingerprinting. In *Proc. of OSDI* (December 2004).

[28] STONE-GROSS, B., ET AL. Your botnet is my botnet: Analysis of a botnet takeover. In *Proc. of CCS* (2009).

[29] WILSON, C., BOE, B., SALA, A., PUTTASWAMY, K. P. N., AND ZHAO, B. Y. User interactions in social networks and their implications. In *Proc. of EuroSys* (April 2009).

[30] ZHANG, J., WU, J., LAN, J., AND LIU, J. Performance evaluation and comparison of three counting bloom filter schemes. *J. of Electronics (China) 26*, 3 (2009), 332–340.