Measuring the Web for Security and Privacy

William Li
williamli@berkeley.edu

Richard Shin
ricshin@berkeley.edu

Abstract

As researchers and web developers, we have have uncovered many ways the web could compromise users' security or reveal private information about them, as well as ways to defend against these concerns. However, it is unclear how the large number of real-world web sites actually remain vulnerable to threats against users' security and privacy. In this paper, we seek to answer these types of questions with empirical data directly collected from a large number of web sites. We build a robust, scalable system for crawling the web, specialized towards extracting properties relevant to security and privacy from a large number of pages and summarizing the results. We measure metrics such as the prevalence of outdated software, third-party widgets, cookies, and the use of secure log-in forms. We ran our system on the top 1,000 and 100,000 web sites as collected by Alexa, and among several of our discoveries, we found that about 90% of web sites used an outdated version of PHP with a well-known vulnerability.

1 Introduction

As the web has grown tremendously over the last few years, so have concerns regarding its security and privacy. Researchers and web developers have uncovered many possible ways that technologies used on the web could compromise users' security or reveal private information about them, and the security community as a whole has gained a comprehensive understanding of the threats users face and how we might defend against them. In particular, many vulnerabilities have been discovered over the years in software commonly used to power web sites, such as web servers, server-side runtimes, and off-the-shelf applications like forums and content management systems built upon them; it is critical that web site operators update their software timely to avoid exposure to widely-known vulnerabilities. However, given the low barrier of entry to operating a web site due to the open and democratic nature of the web, it is unclear how well the over 130 million real-world web sites have kept up with updating their software and following other best practices to ensure security [1].

Not only has the web grown in size, the individual sites that constitute it have also become significantly more complex. Compared to just only a few years ago, it is now much more typical for a web page to contain many 'widgets' from third parties, which have functions such as measuring analytics, integrating with social networks, showing advertisements, and embedding videos. They vary in their implementation, ranging from JavaScript code inserted directly into the target page to a simple image, but given that a few prominent companies such as Google or Facebook control these widgets, their prevalence may pose a significant privacy threat. In particular, browsers send the embedding page's URL as the referrer when loading these widgets, allowing their third-party hosts to learn about the user's browsing habit. Again, though, it is difficult to know precisely how widespread these widgets may be on real-world web sites.

In this paper, we seek to answer these types of questions with data collected directly from a large number of web sites. We want to uncover empirical facts about real-world web sites that confirm or challenge the intuitions or conventional wisdom we have as security and privacy researchers. To that end, we develop a robust, scalable system for crawling the web, specialized for extracting properties relevant to security and privacy from a large number of pages and summarizing the results. In particular, we focus on three specific properties—accuracy, flexibility, and scalability:

- We want the analyses that we make on each page to be accurate. Given the complexity of modern browsers, many parts of a web page that can interact in complicated ways in order to produce the final result shown to the user. Nave methods that directly analyze the source code of the pages may produce inaccurate results, as they do not sufficiently model the behavior of real browsers.

- Given the fast-paced nature of web security, we want to be able to easily and quickly measure new types of metrics as needs for them arise. We must ensure that our system can adapt to unforeseen types of measurements that we might want to conduct in the future.

- We want to inexpensively survey as many as possible of the billions of pages on the web. In order for such analyses to finish in a reasonable amount of time, we must remove bottlenecks from our system and ensure that it can scale as arbitrarily as needed.

To address these challenges, we base our system for extracting metrics on a real web browser, instrumented to use a set of independently-written modules based on familiar APIs for the actual data collection. We parallelize the time-consuming process of visiting each web page using Amazon EC2, allowing a throughput of over 10 pages processed per second. To compile the summaries of the data collected, we use the map/reduce paradigm, which also allows effective parallelization.

Demonstrating the ease with which our system allows for measuring various types of metrics, we selected and implemented a comprehensive set of metrics that we felt provides useful information about privacy and security on the web, as well as exercise our system's capabilities. In addition to the issues of outdated software and third-party widgets as discussed above, we also address the prevalence of cookies and the use of secure log-in forms.

We ran our system on the top 1,000 and 100,000 web sites, as collected by Alexa. As an example of our findings, we discovered that about 90% of web sites used an outdated version of PHP with a well-known vulnerability, an alarmingly high number. On the other hand, we found Facebook's social widgets on only about 16% of the pages analyzed, a relatively small fraction considering the popularity of the service. Comparing the results for the two sets gives us illuminating data as well; we found that about 62% of log-in forms for the top 1000 web sites were securely submitted, while the portion was only 3% for the top 100,000 web sites.

We organize the remainder of the paper as follows. The next section provides an overview of selected similar work in the area. We then describe our implementation of the measurement system in detail, including how we designed it to provide the desired properties, and how we addressed various challenges we encountered. We describe the specific measurements we implemented and conducted.

2   Related Work

Internet-scale measurements.   Many others have performed internet-scale measurements for various purposes. For example, Netcraft tracks millions of web hosts for server statistics [2], and Akamai publishes quarterly "State of the Internet" reports on the latest attack traffic and worldwide bandwith/connection statistics [3]. There have even been single-purpose internet-scale measurements such as the HTTP Header Survey [4], which looked at what security and server version headers were sent by the top 10,000 web sites according to Alexa. However, these all involve relatively simple measurements per server or page, at the HTTP protocol level rather than the behavior of the site when loaded in a browser.

Privacy on the web.   Privacy reports include examinations of a number of web sites for browser cookies and other tracking mechanisms. The Wall Street Journal reviewed 50 web sites [5]; KnowPrivacy reviewed 100 web sites [6]; and a joint WPI/AT&T study automated a Firefox browser to look at 1,000 sites [7]. Finally, for the end user, browser extensions such as Ghostery, NoScript, and Adblock Plus help notify the user of tracking cookies and scripts, or just block them outright [8, 9, 10]. As far as we could determine, none of the privacy reports were using automated, scalable systems that utilize real browsers to look at a large number of websites. The closest is the WPI/AT&T study that used a batch script in Firefox to automatically load webpages. The rest of the studies had no automation, so we wanted to tackle this problem. Also, most of the studies listed above only look at the home page of each domain, which will miss data on secondary pages. In contract, this study analyzes multiple pages per domain, up to 100,000 unique domains—an order of magnitude higher than similar privacy studies.

3   Metrics

In this section, we describe the concrete measurements that we conducted for each of the pages that we crawled using our system.

3.1   Tracking mechanisms

One of the main goals of this paper is to provide a system to record and analyze the many web-based tracking mechanisms employed in production today. Through these methods, the site operator or third-party advertisers can track your browsing habits and history to a very fine-grained level across many sites.

Iframes

Iframes are a popular mechanism for embedding content from another web server, such as videos from YouTube [11], showing PDF files, or Facebook applications [12]. However, they can also leak a lot of information about the current user, including the referring page, cookies, and the IP address. Arbitrary scripts can execute within the iframe as well.

Cookies

Advertisers and analytics providers use cookies to track users during their browsing sessions. A tracking cookie is usually set with a very distant expiration date, and then a third party can track your movement across the website even if your browser or computer is restarted (with the exception of users who clear their cookies frequently).

Images

Another tracking mechanism is the so-called "web bug" [13] or "tracking pixel"—typically a 1x1 pixel transparent GIF that is loaded on the page. The URL of the picture can be appended with additional information about the current browsing session. In addition, in the process of making the request for the 1x1 pixel, the referrer (the URL of the page which includes the tracking pixel) and other headers of the user's browser are sent to the tracking server.

Scripts

Scripts tie all of the above tracking together—for example, they can add tracking pixels to track client-side properties (such as screen resolution or Flash support). Also, third-party widgets often come as scripts to be included into a page, which (again) sends referrer information to a third-party server, and furthermore executes whatever code that the third-party provided with the page's privileges.

3.2   Security issues

We also measured some metrics more relevant to web security: the use of HTTPS for sending passwords and the distribution of server software versions. We also look at the prevalence of jQuery, a popular JavaScript library, and how each of its versions are in use.

HTTPS forms

When sent over unencrypted Wi-Fi networks, passwords in login forms can easily be passively sniffed by nearby adversaries if the form submission is not sent over a secure HTTPS channel. While tools like Firesheep [14] have recently increased awareness of this issue, and spurred large web sites such as Google, Facebook, and Twitter to further adopt HTTPS [15, 16, 17], we wish to see how less prominent web sites have taken care to implement secure, HTTPS-based password forms.

Server software versions

One of the HTTP headers that are typically sent back by web servers is an identification string, which sometimes includes the version of the server and popular plugins. This data is useful because it can show how many web sites are running outdated versions of software that may have known security vulnerabilities.

jQuery

JQuery is a popular JavaScript library used by many web sites [18]. Since the system is actually executing the scripts present on the web page, we are able to detect the jQuery version by looking at the contents of the `jQuery.fn.jquery` variable. We can do this regardless of how jQuery was included on the page, or if it has been compressed and minified. While jQuery in itself poses little security risks, it serves as a convenient proxy for other libraries which may introduce vulnerabilities, and allows us to see how up-to-date sites are in such matters.

4   System Design

Our system has three main components: the page queue, the measurement worker, and the results store. The queue tracks which pages need to be measured, first initialized with a list of web sites that we want to crawl, and later supplemented

with links found on the pages that we have already crawled. The measurement worker receives URLs from the page queue and actually collects the metrics for those pages. Along with collecting the metrics that we want, the worker also gets a list of links from the page so that we can crawl more pages on the same site, which goes to the page queue. The other metrics are sent to the results store, which later processes the data collected per page in order to produce summarized results over the entire crawl.

We do not explore all the links on a page for efficiency reasons. Instead, we select a fixed number of links from each page, randomly sampled according to its level of visibility. We compute the amount of space taken up by each link, and follow links that use more space with proportionally higher probability.

Since each web page can take many seconds to load completely, we run many workers concurrently to collect the metrics for many web pages in parallel. The page queue and the results store must handle the demands placed by the many concurrent workers, while remaining consistent to avoid overwriting existing results and crawling the same page multiple times.

5 Implementation

Since we wanted high accuracy from our measurements, we wanted to examine a real browser's notion of a web page's structure and contents (i.e., the DOM) rather than a surface examination of its source code. Conveniently, we found a tool called PhantomJS [19], a simple non-graphical wrapper around QtWebKit that lets us load and examine arbitrary pages using JavaScript. By running our JavaScript on many web pages, we can already do things like extract the list of iframes, scripts, and images by probing the DOM. This provides an interface familiar to anyone who has used JavaScript in web development, and allowed us to quickly extract different pieces of information from a page based on the browser's notion of the page's structure.

However, as our JavaScript measurement code has no more access into the browser's internals than any other script on the page that we are examining, we could not measure all of the metrics that we set out to, such as getting the entire list of cookies set by visiting a page, or recording the bandwidth consumed by each resource on the page. To solve these problems, we had to modify PhantomJS to expose the additional information through JavaScript. We also implemented an HTTP proxy which intercepts all the requests made by PhantomJS and allows us to record the times and bytes used by each one.

Since network latency and bandwidth limitations seemed to be the biggest bottleneck in crawling hundreds of thousands of web pages, we ran many copies of our PhantomJS-based measurement worker on Amazon EC2 to parallelize the loading of web pages [20]. Amazon EC2 allows us to launch as many workers as needed on a moments notice with an image that was configured once with the correct operating environment and all of the necessary software installed. In addition, the image automatically updates the measurement code to the latest version and runs the code when it starts, or when ordered to by the page queue. This allows our system to be as up to date as possible with very little manual intervention, so that we can quickly and easily make changes to what is measured by the workers.

While parallelizing the workers was relatively simple with EC2 in place, we need a reliable and consistent queuing and data storage mechanism to support the workers' operation. While we evaluated several different solutions, in the end, we used two popular software packages—RabbitMQ and CouchDB [21, 22]—to, respectively, queue pages and to store the results of each crawl. We found that this combination delivered high performance and reliability, as well as allowing us to easily perform analysis on the resulting data. For simplicity, we ran one instance of each on a single command-and-control server, directing all the workers on Amazon EC2.

RabbitMQ is a message queue system that is designed to handle thousands of messages per second without the overhead of an indexed database. This fit our use case very well and completely removed all the bottlenecks we had experienced with previous iterations of the page queue based on relational databases. Each item in the queue contains the URL to analyze, the number of links from that page to add to the queue, and the remaining depth of the crawl. Even with 100 clients, our RabbitMQ setup was processing the approximately 10 messages per second produced at peak with no problems.

We use CouchDB, a schema-less document database, to store the workers' results. As it does not maintain any structured columns or foreign keys, it provides great write performance, and enables us to store arbitrarily structured data for every page that we crawl. In order to perform analyses on the data, we use CouchDB's built-in MapReduce-based view functionality [23], which allows us to flexibly operate over any of the fields in all the documents in the database. Querying for results require building indexes on the fields—a process which can take some time, but it can happen entirely after the crawl of web pages. This allows the server to handle all incoming data without slowing down, and then doing a post-processing indexing step when all the data has been uploaded.

We review how our implementation meets the characteristics that we set out to provide:

Accuracy Using PhantomJS, which is based on the popular WebKit engine, we see and examine web pages just as a real web browser would, taking into account the many complex interactions between the HTML, CSS, and JavaScript that make up a page.

Type	Top 1,000 sites	Top 100,000 sites
AdSense	12.0%	22.5%
DoubleClick	15.5%	22.7%
Google Analytics	46.9%	54.9%
All Google-Affiliated	52.7%	60.7%
Facebook	16.0%	17.1%
Twitter	4.5%	4.44%

Table 1: Percentage of pages analyzed that embedded the above third-party software.

Scalability We use Amazon EC2 and high-throughput databases and queues to ensure that we can quickly crawl a very large number of pages (up to 10 pages per second when running our experiments). While we did not need to run CouchDB or RabbitMQ on more than one machine, their design allows us to easily scale them as well to support higher throughput if needed.

Flexibility We write most of our analyses in JavaScript which provides great flexibility and familiarity to web developers. We can quickly update the code that runs on the workers, and CouchDB's schema-free nature allows us to make these changes with minimal hassle. Its MapReduce functionality also allows us to conduct relatively complicated queries with ease.

6 Results and Analysis

We conducted two sets of measurements, first for the top 1,000 sites and then for the top 100,000 sites, as compiled by Alexa [24]. We present some of the results gained from summarizing the raw data collected, compare the results gained from each of the crawls, and provide analysis of the findings.

6.1 Tracking

One of the key privacy metrics that the system set out to track is the percentage of pages that include third-party plugins or widgets, such as AdSense, Google Analytics, DoubleClick, Facebook social plugins, and Twitter buttons, which can pose a privacy risk by revealing the user's browsing habits. Each of these percentages were calculated by looking for scripts or frames included from these websites or services. We show the results in Table 1; the most prevalent is Google Analytics, included as a script on 46.9% of pages in the top 1,000 sites and 54.9% of pages in the top 100,000 sites. Combining AdSense, DoubleClick, and Google Analytics together, Google can theoretically track visitors to about 52.7% of pages crawled for the top 1000 websites and 60.7% of the pages in the top 100k websites.

Between the two datasets, the Facebook and Twitter numbers are very similar. The largest jumps are in AdSense and DoubleClick, possibly because smaller sites rely on the revenue streams from do-it-yourself online ads more than the larger ones. In addition, Google Analytics also saw an increase in market share for the top 100,000 sites when compared to the top 1000 sites, possibly because smaller sites do not have the budget to pay for an analytics solution.

6.2 Secure Forms

For all the password fields that we found, we measured whether the target of the form enclosing it was a secure (HTTPS) or insecure page. As we show in Figure 1a, of the 2018 forms tracked across the top 1000 websites, 1254 (62.1%) forms were submitted to an HTTPS page, while the remaining 764 (37.9%) forms were sending to an unencrypted page or had an undetermined method of submitting the form.

For the larger dataset of 100,000 sites (in Figure 1b), the system detected 393,855 forms with a password field. Of these, only a very small 3% of forms submitted to an HTTPS page. This is in stark contrast to the top 1,000 sites (38% secure forms). The most likely explanation is that smaller sites do not have the budget, or do not have the expertise, to set up SSL certificates. While smaller sites do not receive as much attention from automated tools such as Firesheep, they still remain very vulnerable to attack when used over an insecure wireless network.
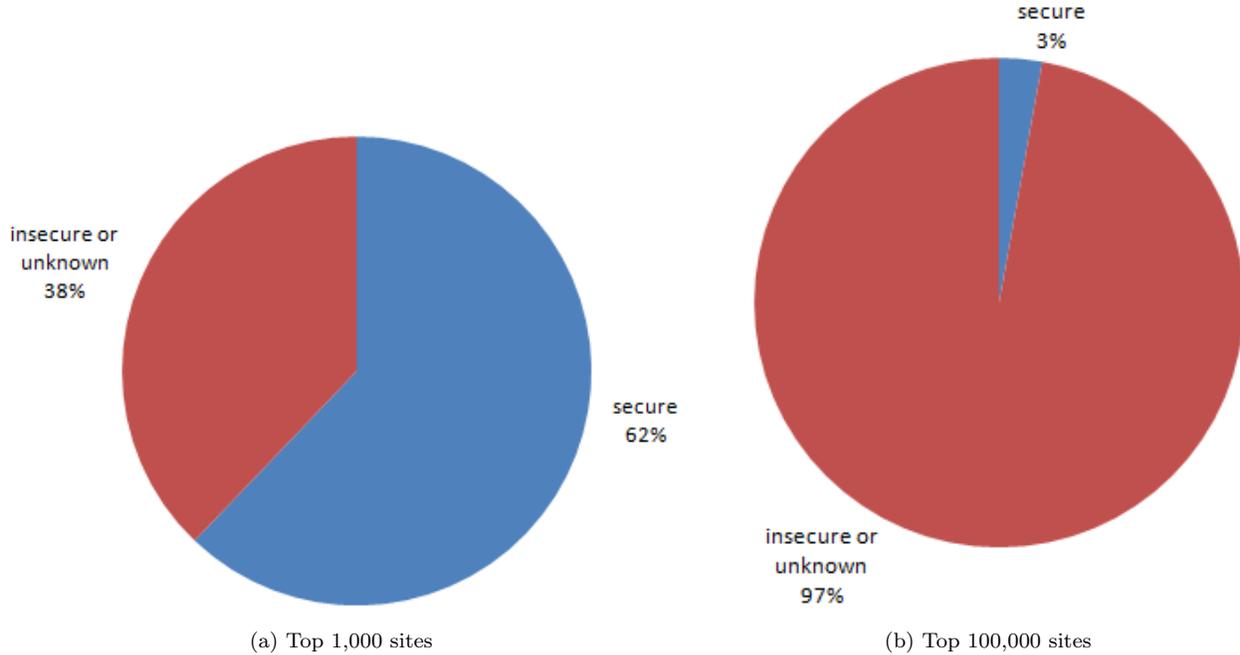
(a) Top 1,000 sites

(b) Top 100,000 sites

Figure 1: HTTPS usage in forms that contain a password field.

Media type	Mean	Median	Max
Cookies	11.47	7	141
Frames	1.83	0	66
Images	32.43	14	2394
Scripts	10.58	7	104
Stylesheets	2.81	2	40

(a) Top 1,000 sites

Media type	Mean	Median	Max
Cookies	11.47	7	141
Frames	1.83	0	66
Images	32.43	14	2394
Scripts	10.58	7	104
Stylesheets	2.81	2	40

(b) Top 100,000 sites

Table 2: Statistics for number of media items included per page.

6.3 Included Media

All of the cookies, frames, images, scripts, and stylesheets that were included on the pages were also tracked and counted. We show the results in Table 2. While the vast majority of pages were unremarkable, a few outliers seem to exemplify bad coding practices; for example, there were a number of pages with thousands of images per page or hundreds of iframes per page. In addition to heavily slowing down page loading time and making the user experience really slow, these pages show what happens when little concern is given to optimization and security. We also found many pages that set several hundred cookies, worrying from the perspective of privacy.

Comparing the top 1,000 as shown in Table 2a, to the top 100,000 sites as shown in Table 2b, we can see that the averages do not change much. The outliers certainly increase even more as we crawl more pages, but that is to be expected.

6.4 jQuery

In Figure 2, we show the prevalence and version distribution of jQuery, a popular JavaScript library. An astonishing 47% of pages embedded some version of jQuery, which is impressive given that the library was first released in 2006 [25]; however, Figure 2 shows a great deal of fragmentation in the version used. Fortunately, this does not lead to any security vulnerabilities by itself, since it is a client-side library running in a sandbox, but it does show that if an issue were to be discovered in a more sensitive, different and widely-used library, an attacker could potentially target a large number of popular websites.
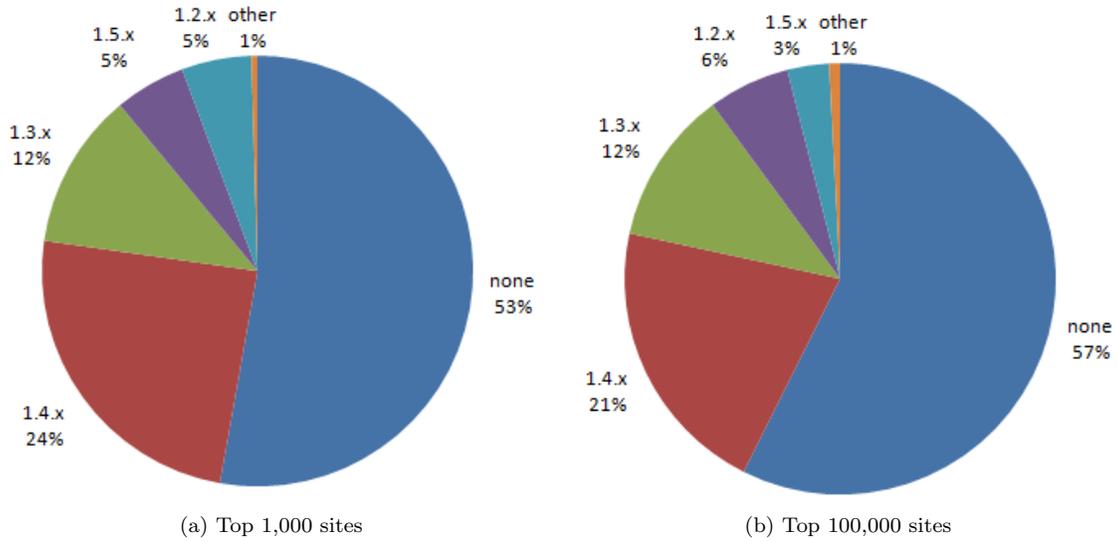
(a) Top 1,000 sites   (b) Top 100,000 sites

Figure 2: Distribution of jQuery versions.

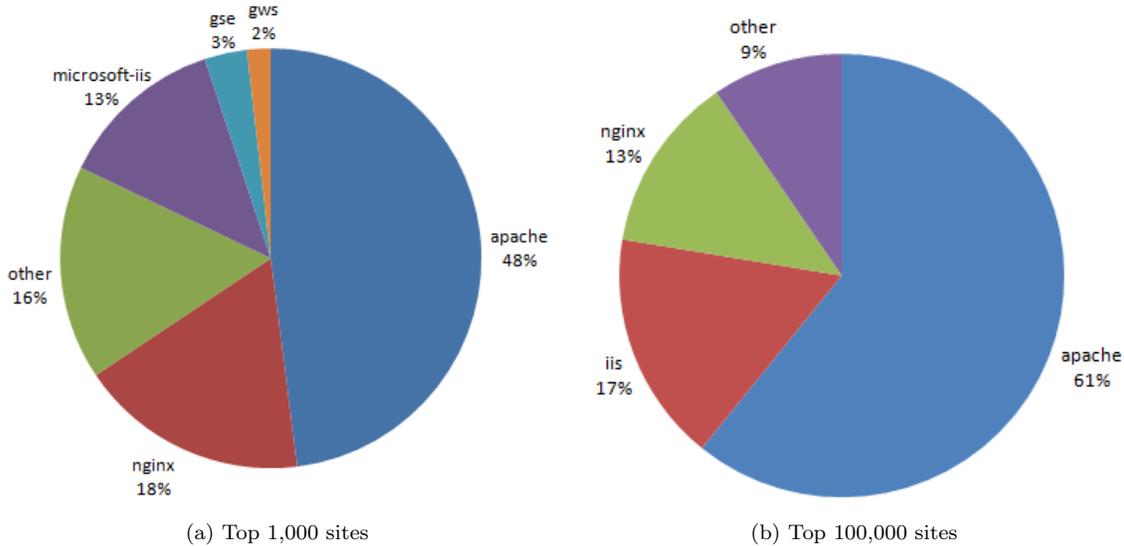(a) Top 1,000 sites   (b) Top 100,000 sites

Figure 3: Distribution of web server software.

6.5 Server software

Figure 3 shows the distribution of web server software used to serve the pages that we analyzed. Of the top 1,000 sites, Google's web servers account for 5% of all served pages. Apache is by far the most popular web server, running 48% of web sites; nginx and IIS are distantly behind. However, once we move to the top 100,000 sites, we see that Google's share declines to 1.2% of all pages, while Apache increases to 61% of all pages.
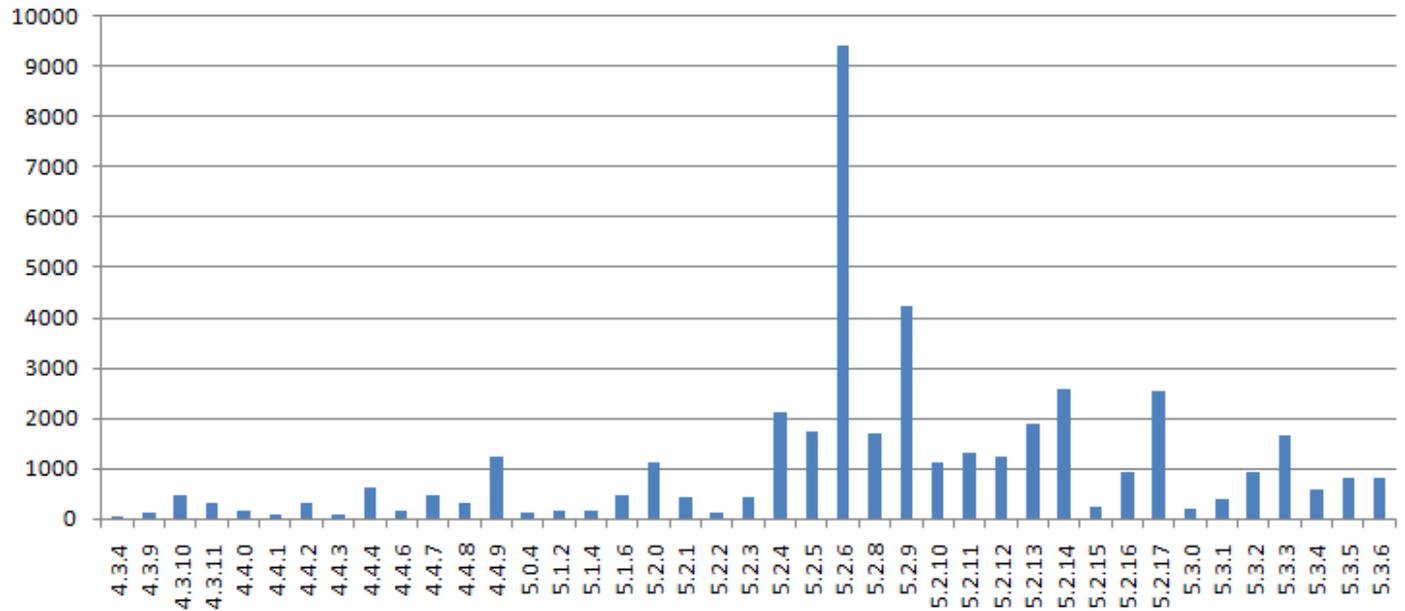
6.6 PHP versions

In addition to the server version string itself, Apache on Linux sends back the PHP version by default, making for some interesting data mining. Running older versions of PHP is a significant problem since security holes are discovered quite often and the version number is effectively public for many servers.

    For example, in January 2011, security researcher Rick Regan published a report about certain floating point numbers
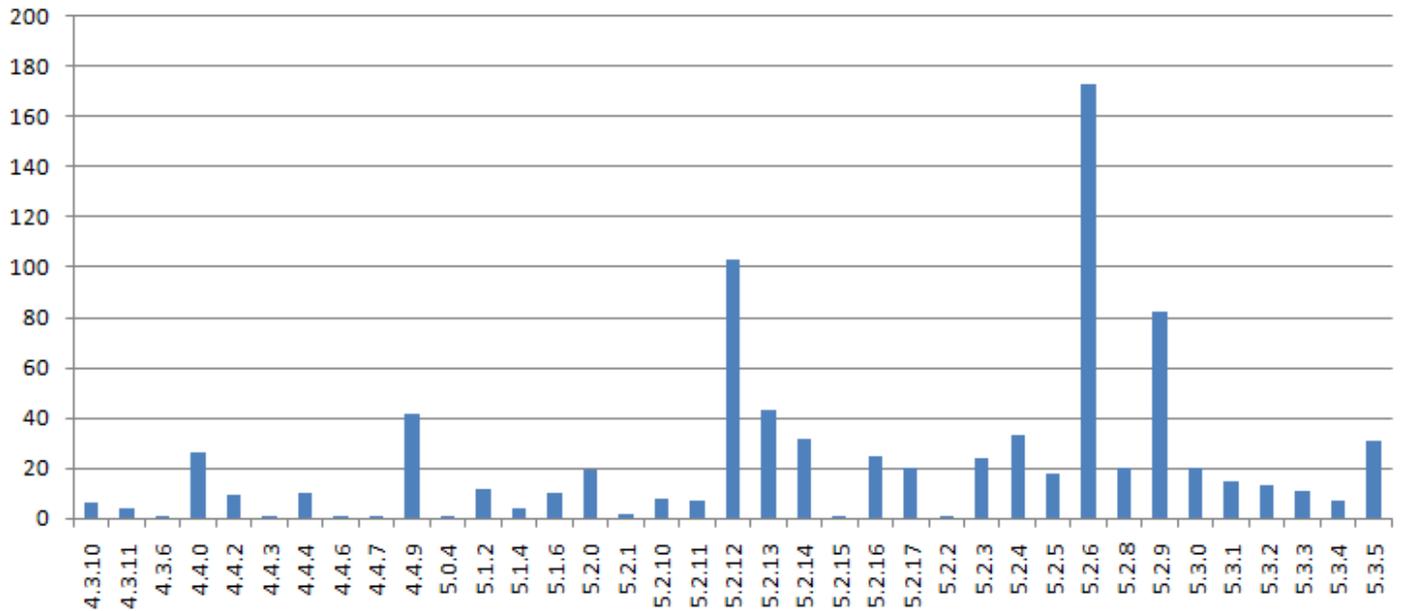
(a) Top 1,000 sites

(b) Top 100,000 sites

Figure 4: Distribution of PHP versions.

crashing servers running certain versions of PHP running on 32-bit architectures [26]. The bug was quickly fixed in versions 5.3.5 and 5.2.17 [27], but a number of servers are still vulnerable to this attack. While its impossible to determine whether a server is 32-bit or 64-bit from the outside, there are still 591 out of 611 pages (96.7%) in the 5.2.x branch and 66 out of 97 pages (68.0%) in the 5.3.x branch running on servers that could be vulnerable in the top 1000 sites. Thats not even counting the 101 pages (0.5% of all crawled pages) confirmed to be running an obsolete PHP 4 binary.

As for the top 100,000 sites, there are 30,587 pages out of 33110 (92.4%) in the 5.2.x branch and 3769 pages out of 5385 (70.0%) in the 5.3.x branch that are still vulnerable to this specific bug. The rates between the two runs are similar, which suggests that webmasters are running outdated versions of software regardless of available resources/budgets. There are even 11 pages running on PHP3, which has been obsolete since the year 2000 [28].

6.7 Performance

The first run consisting of the top 1,000 web sites, with a limit of 25 links crawled per page, and a max crawl depth of 2, took approximately 6 hours to run using 37 Amazon EC2 workers. The process crawled 19292 valid pages (0.6 GB of saved data), out of a theoretical maximum of 31000 pages. We can attribute the attrition mainly to broken links and inaccessible or slow pages, possibly many of which are foreign websites.

The second run consisting of the top 100,000 web sites, with a max fanout of 5 links per page, and a max crawl depth of 1. This run took approximately 17 hours with 86 clients, and ended up crawling 485617 pages (19.9 GB of saved data), out of a theoretical maximum of 600,000 pages.

We experimentally verified the scalability of the components of this system. RabbitMQ and CouchDB both scale nearly linearly with the number of servers. Each message and result is independent from any other result except in the final analysis queries. As for the workers themselves, they are the most scalable of all components, since all that is required is a larger machine count from Amazon EC2.

7 Limitations

The methods used in the analysis assumes that each site is the same size and has the same browsing patterns, since it looks at a fixed number of pages per site. For example, Wikipedia has over 23 million unique pages [29], but a personal blog may only have 10 pages. If Wikipedia used Google Analytics, then the real number of pages tracked by Google Analytics should be much higher than the numbers presented in the results. In addition, the links that are followed do not always stay on the same domain, so it is possible that some sites had very little crawled urls.

A number of calls to PhantomJS ended up timing out due to either bugs in PhantomJS or an overload of data on the page. While some of these are unavoidable due to the inconsistent nature of web pages, the results are nonetheless slightly skewed. PhantomJS also returns soon after the onload event is fired in the web browser. As a result, some http requests are not tracked if they occur a significant time after the onload event is fired. One such example is the Google Analytics tracking pixel, which is not loaded inline with the page, but is instead loaded after the onload event is fired.

Finally, one of the longest steps was analyzing the data after the crawl had been run. For the larger database of the top 100k sites, it took over 8 hours to generate the indexes to query for data. The reason is because CouchDB does not index data as it gets written to avoid unnecessary inefficiencies [?]. Instead, the indexes get updated every time the views get accessed. In the future, a cron job that accesses the view in a periodic manner (say, every 15 minutes) would result in a much faster analysis.

8 Future Work

Right now, the system only relies on one command-and-control server running a single instance of RabbitMQ and a single instance of CouchDB. However, this should be improved since this single CouchDB instance most likely now serves as the bottleneck. We expect it to be quite simple to modify the client script to send the results to a sharded database running on multiple computers to distribute the load.

We had implemented some additional metrics for tracking Flash cookies and HTML5 local storage. The client worked on local development machines for tracking both these technologies, but the Amazon EC2 image was not fully compatible. Fixing this and running across many sites would provide some insight into these lesser-studied tracking mechanisms.

Finally, having this as a constantly-running service or a searchable database would be extremely useful as a further analysis tool or simply as an awareness website. Creating a browser extension to display this information using the online database would be very useful for the privacy-conscious user.

9 Conclusion

Given the previous work and the lack of a large-scale privacy measurement study, we thought it would be important to conduct this study and look at two orders of magnitude more websites than previous studies. We built a system that is easily scalable, easily modified, and uses a real web browsing engine that can run rich media like JavaScript or Flash.

The results aren't necessarily ground-breaking news, but we feel they are fairly interesting statistics regardless. The overall results suggest that webmasters do not really keep up to date with the latest patches, and they run older, bugged versions of software very frequently. In addition, the best practice of submitting password fields to an HTTPS page is not as widespread as it should be, especially in the long tail of websites past the top 1000. Widgets from third parties are increasingly popular for social media purposes (as opposed to traditional banner advertising), but privacy concerns with these networks has not caught up in the public eye. With future work, we hope to bring to expand this project to allow for more refined data and analysis, keeping users and webmasters alike informed about security and privacy concerns for as much of the web as possible.

Bibliography

[1]   DomainTools. http://www.domaintools.com/internet-statistics/.

[2]   NetCraft. http://news.netcraft.com/.

[3]   Akamai. State of the Internet. http://www.akamai.com/stateoftheinternet/.

[4]   Shodan Research. http://www.shodanhq.com/research/infodisc/report.

[5]   Julia Angwin. "The Webs New Gold Mine: Your Secrets" (July 30, 2010). http://online.wsj.com/article/SB10001424052748703940904575395073512989404.html.

[6]   Joshua Gomez, Travis Pinnick, and Ashkan Soltani. KnowPrivacy. June 1, 2009.

[7]   Balachander Krishnamurth, Delna Malandrino, and Craig E. Wills. Measuring Privacy Loss and the Impact of Privacy Protection in Web Browsing (2007).

[8]   Ghostery. http://www.ghostery.com/.

[9]   NoScript. http://noscript.net/.

[10]   AdBlock Plus. http://adblockplus.org/en/.

[11]   YouTube API Blog. "A New Way to Embed YouTube Videos" (July 22, 2010). http://apiblog.youtube.com/2010/07/new-way-to-embed-youtube-videos.html.

[12]   Apps on Facebook.com. https://developers.facebook.com/docs/guides/canvas/.

[13]   Richard M. Smith. The Web Bug FAQ. http://w2.eff.org/Privacy/Marketing/web_bug.html.

[14]   Eric Butler. Firesheep. http://codebutler.com/firesheep.

[15]   Brian Prince. "Google Gmail Switches HTTPS to Always On by Default". http://www.eweek.com/c/a/Security/Google-Gmail-Switches-HTTPS-to-Always-on-by-Default-656394/.

[16]   Audrey Watters. "Zuckerbergs Page Hacked, Now Facebook to Offer 'Always On' HTTPS". http://www.readwriteweb.com/archives/zuckerbergs_facebook_page_hacked_and_now_facebook.php.

[17]   Twitter. "Making Twitter more secure: HTTPS" http://blog.twitter.com/2011/03/making-twitter-more-secure-https.html.

[18]   jQuery. http://jquery.com/.

[19]   PhantomJS. http://www.phantomjs.org/.

[20]   Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[21]   RabbitMQ. http://www.rabbitmq.com/.

[22]   CouchDB. http://couchdb.apache.org/.

[23]   CouchDB Wiki. Introduction to CouchDB Views. http://wiki.apache.org/couchdb/Introduction_to_CouchDB_views.

[24]   Alexa. Top Sites. http://www.alexa.com/topsites.

[25]   jQuery project. History. http://jquery.org/history/.

[26]   Rick Regan. "PHP Hangs On Numeric Value 2.2250738585072011e-308". http://www.exploringbinary.com/php-hangs-on-numeric-value-2-2250738585072011e-308/.

[27]   Paul Krill. "PHP floating point bug fixed". http://www.infoworld.com/d/security-central/php-floating-point-bug-fixed-887.

[28]   PHP. "History of PHP". http://www.php.net/manual/en/history.php.php.

[29]   Wikipedia. "Statistics". http://en.wikipedia.org/wiki/Special:Statistics.

[30]   CouchDB Wiki. "Regenerating views on update". http://wiki.apache.org/couchdb/Regenerating_views_on_update.