Vulnerability Factors in New Web Applications: Audit Tools, Developer Selection & Languages

Jason Bau, Frank Wang, Elie Bursztein, Patrick Mutchler, John C. Mitchell
Stanford University

Abstract—**We develop a web application vulnerability metric based on the combined reports of 4 leading commercial black box vulnerability scanners and evaluate this metric using historical benchmarks and our new sample of applications. We then use this metric to examine the impact of three factors on web application security: provenance (developed by startup company or freelancers), developer security knowledge, and programming language. Our study evaluates 27 web applications developed by programmers from 19 Silicon Valley startups and 8 outsourcing freelancers using 5 programming languages. We correlate the expected vulnerability rate of a Web application with whether it is developed by startup company or freelancers, the extent of developer security knowledge (assessed by a simple quiz), and the programming language used. We provide statistical confidence measures and find several results with statistical significance. For example, applications written in PHP are more prone to severe vulnerabilities, especially injection, and applications developed by freelancers tend to have more injection vulnerabilities. Our summary results provide guidance to developers that we believe may improve the security of future web applications.**

Keywords-**security, web applications, developers, languages, vulnerability scanners**

I. Introduction

Web application vulnerabilities are widely tracked [1] and widely recognized as a serious problem [2]. Data breaches resulting from vulnerabilities, for example, have been a recurring problem in past years [3] and will likely continue to be a major problem in the future [4]. In light of their importance, we ask: "What factors associated with the development process correlate with vulnerability rates?". Focusing on initial development decisions, we look at two types of development teams, Silicon Valley startups and outsourced freelancers, and measure the impact of additional factors such as developer security knowledge and programming language selection.

In order to estimate vulnerability rates, we develop a vulnerability metric based on black-box vulnerability scanners. Although black-box scanners are far from perfect at detecting actual vulnerabilities, past studies have systematically measured their performance [5], [6], using both test sites constructed with known vulnerabilities and back-dated open source sites in which actual vulnerabilities have been found over time. Building on the previous study, we measure improvements in site coverage (crawling) and vulnerability detection. In addition, by comparing the behavior of different commercial scanners on the same new sample sites constructed in our study, we evaluate the collective behavior of 4 leading commercial scanners, manually audited for false positives, and suggest that the resulting metric provides a meaningful way of comparing web applications developed by various teams.

Using our vulnerability metric, we scan 27 early stage web applications built by 19 startups and 8 teams of freelance developers hired for this purpose. Developers used 5 different programming languages, and were given a simple quiz to determine their familiarity with relevant security concepts. Because of confidentiality reasons, we could not access the source code of the startups, so we interviewed them before the scan to find more information about the company and codebase (e.g. framework, lines of code, etc.). For verification, we also interviewed them after the scan to discuss the severe vulnerabilities and confirm their existence in the codebase, allowing us to determine whether a report-item is a false positive.

We collected various kinds of data and present anecdotal evidence for various observed phenomena. As part of the analysis, we correlate the expected vulnerability rate of a Web application with the nature of the developer team, the extent of the developers' security knowledge, and the programming language used. We provide statistical confidence measures for each category and find several results with statistical significance. For example, applications written in PHP are more prone to severe vulnerabilities, especially injection, and applications developed by freelancers tend to have more injection vulnerabilities.

Our summary results, collected in Section VII, provide guidance to developers that we believe may improve the security of future web applications. Our main findings include the following:

1) Security scanners are not perfect and are tuned to current trends in web application development, but a combination of results from scanners provide a reasonable comparative measure of code security compared to other quantitative methods.
2) Variability in scanner performance from one application to another implies tool comparisons performed on singular testbeds are *not* predictive in a statistically meaningful way.
3) Freelancers are more prone to introducing injection vulnerabilities when compared to startup developers, in a statistically meaningful way.
4) PHP applications have statistically significant higher

rates of injection vulnerabilities than non-PHP applications, and PHP applications tend not to use frameworks.

5) Startup developers are more knowledgeable about cryptographic storage and same-origin policy compared to freelancers, again with statistical significance.

To our knowledge, this is the first study that evaluates security tools, web developers, and applications to understand and expose root causes of web vulnerabilities at this scale. From these findings, we offer a few recommendations to avoid security vulnerabilities in Web applications. For more comprehensive detection of security vulnerabilities, we believe that using multiple scanners is helpful, and developers should test scanners on their own environments rather than relying on narrow benchmarks. Freelancers tend to produce more insecure code and tend to be more unreliable, so we recommend carefully auditing their code and managing them tightly. Moreover, we advise caution when using PHP as a programming language, as our study and various related sources report higher rates of vulnerabilities in PHP web applications, and especially caution against "raw" PHP development without a supporting application framework. Finally, we recommend against hiring any freelancers that use PHP as this combination has a very high vulnerability rate.

Section II reviews web programming languages and vulnerabilities. In Section III, we describe the details of our experimental setup and overall process. Section IV establishes the effectiveness of scanners in terms of crawling and vulnerability detection. In Section V, we evaluate and compare the security knowledge of freelancers and developers and examine vulnerability rates for each group. We also describe our experiences interacting with each group. Section VI looks at vulnerability rates of different programming languages. In Section VII, we provide a summary of our findings and provide recommendations for future development. In Section VIII, we discuss related studies, and Section IX describes future research resulting from this work.

II. The Internet Today

This section provides background information on prominent web development languages and web application vulnerabilities.

A. Distribution of Languages

As documented by BuiltWith Trends [7], PHP is the most popular language on the web, with more than 37 percent of the top one million websites built using PHP. ASP is the second most popular language and although Java was not strongly represented in the top one million sites, it appears to be used by 8 percent of the top 10,000 sites [8] and rank third after PHP and ASP. Because we wanted to survey applications written in popular web programming languages, we chose PHP, ASP, and Java as the 3 languages for web application development for the freelancers. Interestingly, Ruby on Rails is used in less than 1 percent of the top one million sites but about 3 percent in the top 10,000 site. Although it is a successful language, we chose not to ask freelancers to develop sites in Ruby because of our limited resources - we estimated that it would be more meaningful to obtain a larger number of data points on a smaller number of languages.

TABLE I
Distribution of CVEs according to NVD [1]

Language	Percent of CVEs
PHP	38.6
ASP	5.8
Java	3.2
Python	0.3
Ruby	0.2

B. Distribution of Vulnerabilities

As most web security experts likely expect, XSS and injection are the most pressing and severe vulnerabilities, as shown by the Open Source Vulnerability Database (OSVDB) [9] and The Open Web Application Security Project (OWASP) [2]. From OSVDB, we found that XSS and injection were the two most prevalent vulnerabilities in the web over the last six years. Some vulnerabilities, like file inclusion, were prevalent for small periods of time but quickly disappeared. In contrast, XSS and injection vulnerabilities still seem to be a persistent problem on the web and have not improved over time [9]. XSS and injection also occupy the top two spots on the OWASP Top 10 list. While evaluating the audit tools, developers, and languages, we therefore isolate and analyze these two vulnerability types separately.

We use the National Vulnerability Database (NVD) [1] to give us a first-order approximation of the distribution of vulnerabilities across languages. This is a very rough count of common vulnerabilities and exposures (CVEs) reported by security experts, and it is across all sites and not normalized or classified by type. We want to emphasize that we use this data as a baseline for our study to show the distribution of web vulnerabilities currently reported in the wild. In contrast, the purpose of our study is to look at early stage applications in a controlled manner to see if we can understand how such vulnerabilities are created. Our data is normalized and aimed at finding the root cause of vulnerabilities.

Doing a quick search in NVD of the 5 languages that appear in our study, we find the number of displayed results and divide it by the total number CVEs in NVD (51,730 at the time of search) to get the percentage of CVEs for a given language. As seen in Table I, we found PHP to by far have the most vulnerabilities with ASP and Java a distinct second and third respectively.

III. Study Setup

Our study included 27 total Web applications recruited from two different types of sources: 19 startups from established companies such as Inkling to student-built applications in the seed stage, and 8 freelancers hired from two well-known freelancer sites, Elance.com and Freelancer.com. We display some basic statistics regarding the study participants. Figure 1 shows a distribution of languages of the study participants, and Table II shows the average lines of code for each language.

We spent numerous weeks scanning the applications developed by the 27 participants with 4 commercial black-box Web application vulnerability scanners, Acunetix WVS, HP WebInspect, IBM AppScan, and Qualys Web Application Scanner and auditing the generated reports to validate critical detected vulnerabilities, a process about which we will provide more details in the next two sections.

A. Startups

To recruit the 19 startups for our study, we sent an email to various lists known to reach Silicon Valley startups. We offered compensation in exchange for scanning their sites with the 4 scanners and for their time to do a background interview, a security quiz, and a post-scan interview. For responsible disclosure, we provided all startups with copies of each scanner report and offered help with remediation advice if desired.

In the background interview, we asked details about their development and educational background as well as more information about the application, such as backend language and framework, lines of code, man-hours spent on development, man-hours spent on security, etc.

After we scanned their applications, we again interviewed the developers, asking them to evaluate the accuracy of the report and give their impression of the scanners. During this interview, we went over the serious vulnerabilities with them and asked them to confirm the vulnerability exists in the codebase, allowing us to also determine if a report-item was a false positive or not. We also offered help with implementing remediation measures for any vulnerabilities during this follow-up process, which roughly a third to a half of the startups requested.

B. Freelancers

To survey a diverse group of freelancers, we hired 3 sets of freelancers grouped by programming language (PHP, Java/JSP, and ASP) from 3 different price ranges ($<$ \$1000, \$1000-\$2500, and $>$ \$2500) for a total of 9 freelancers. (One ASP freelancer could not deliver a usable application at the time of this writing, 40 days past the promised delivery period of 1 month, hence only 8 freelancers, from 6 different countries, are included in our study.)

Since we could not scan or obtain pre-existing applications from freelancers, we ask them to create an application given a detailed set of specifications. To encourage uniformity, we gave the same specification to all the freelancers. They were asked to design a youth sports photo-sharing site with logins and different permission levels for coaches, parents, and administrators. The site allowed users to upload and download photos and documents, leave comments on photos, update and view contact information, create and sign consent forms, and send email to all coaches, parents, or admins via a Web form.

To remind the freelancers about security, we mentioned that our site was mandated by “legal regulations” to be “secure”, due to hosting photos of minors as well as storing sensitive contact information. We explicitly mentioned only those logged in as a member of a team (as admin, coach, or parent) should be able to access any of the sensitive photos, files, or contact information. We asked the freelancers to enforce these policies but did not offer any further guidance on implementation. We also did not provide any explicit security specifications outside of those mentioned above.

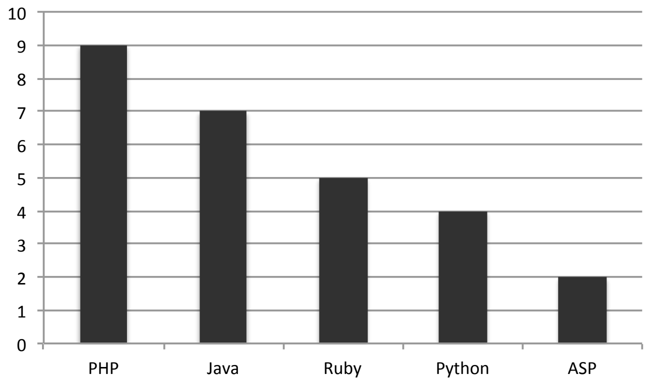


Fig. 1. Distribution of Languages for Study Participants

TABLE II
AVERAGE LINES OF CODE FOR EACH LANGUAGE

Language	Average Lines of Code
ASP	24,320
Java	14,630
PHP	17,020
Python	23,125
Ruby	7660

Though our specification is simple, a properly functioning implementation would need to correctly defend against most of the OWASP Top 10 Web application vulnerabilities [2] by correctly sanitizing user input (A1-XSS and A2-Injection), properly handling logins and permission levels (A3-Authentication), keeping non-logged-in users from directly accessing sensitive photos by typing in URLs (A4-Insecure Direct Object Reference), preventing CSRF on the forms (A5), correctly storing account passwords (A7), keeping ordinary users from using admin pages URLs (A8), properly securing the network connection (A9), and preventing arbitrary redirects after the login flow (A10). Thus, our specification amounts to a good test for the security awareness and practices of its developer.

After each freelancer delivered an application that functioned according to our specification, we scanned the application with the same 4 scanners, producing a vulnerability report. We then audited each report to validate critical vulnerabilities, including all XSS and Injection vulnerabilities, either by manually trying attack vectors or by examining source code.

C. Security Quiz

We asked both startup and freelancer developers to answer basic questions about Web security, in the form of a short-answer quiz. Table III presents more details regarding the quiz. These questions were meant to cover concepts learned in an introductory security class with specific emphasis on the more

TABLE III
QUIZ CATEGORIES AND QUESTION SUMMARY

Q	Category Covered	Summary
1	SSL Configuration	Why CA PKI is needed
2	Cryptography	How to securely store passwords
3	Phishing	Why SiteKeys images are used
4	SQL Injection	Using prepared statements
5	SSL Configuration/XSS	Meaning of "secure" cookies
6	XSS	Meaning of "httponly" cookies
7	XSS/CSRF/Phishing	Risks of following emailed link
8	Injection	PHP local/remote file-include
9	XSS	Passive DOM-content intro. methods
10	Information Disclosure	Risks of auto-backup (˜) files
11	XSS/Same-origin Policy	Consequence of error in Applet SOP
12	Phishing/Clickjacking	Risks of being iframed

severe vulnerabilities in OWASP Top 10 [2]. The full security quiz is reproduced in Appendix A, and the questions were selected from various homework exercises and exam questions formerly given at an undergraduate computer security class at a top U.S. university.

For the freelancers, we administered the quiz well before or after the development cycle, and without providing the correct answers, to minimize the influence the quiz would have on the freelancer implementations.

We will give more details regarding the performance of both groups on the security quiz in Sec. V-A.

IV. AUDIT TOOLS

In this section, we compare the performance of four commercial scanners listed in Table IV first against a previously-published [5] synthetic testbed, for calibration and longitudinal study, and then against the broad testbed comprising all 27 participating Web applications. We test the scanners to find how they have changed from the previous study and to determine their coverage and other characteristics. We conclude that although they are imperfect, the combined results of several scanners provide a reasonable comparative measure of code security, at least in comparison with other quantitative measures such as vulnerabilities course from established databases.

A. Synthetic Testbed Study

We checked scanner vulnerability detection capabilities on a synthetic testbed where the vulnerabilities are known in advance. The motivation for this testbed calibration are twofold: First, we re-use a previously published scanner testbed [5], allowing us to perform a longitudinal study on scanner development. Second, this testbed provides us concrete data on false negatives resulting from scanners missing existent vulnerabilities, data that is difficult to obtain from scanning real applications. (We account for false positives by manual auditing, detailed later in Sec. IV-B1).

The detailed version numbers of four scanners used in our study are listed in Table IV. These four scanners were all part of the study in [5], affording us an opportunity to measure their progress (or regress) in crawling and detection performance since 2010. We obtained the testbed and raw data from the authors of [5] and ran the current versions of the tools against the testbed.

TABLE IV
BLACK BOX SCANNERS

Tool	Version in [5]	Our Version
Acunetix WVS	6.5	7.0.20111005
HP WebInspect	8.0	9.20.247.0
IBM AppScan	7.9	8.5.0.1
Qualys	PCI 2009	Enterprise WAS 2.0

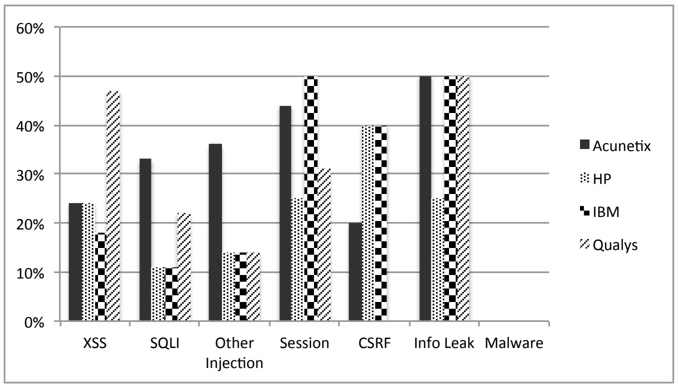
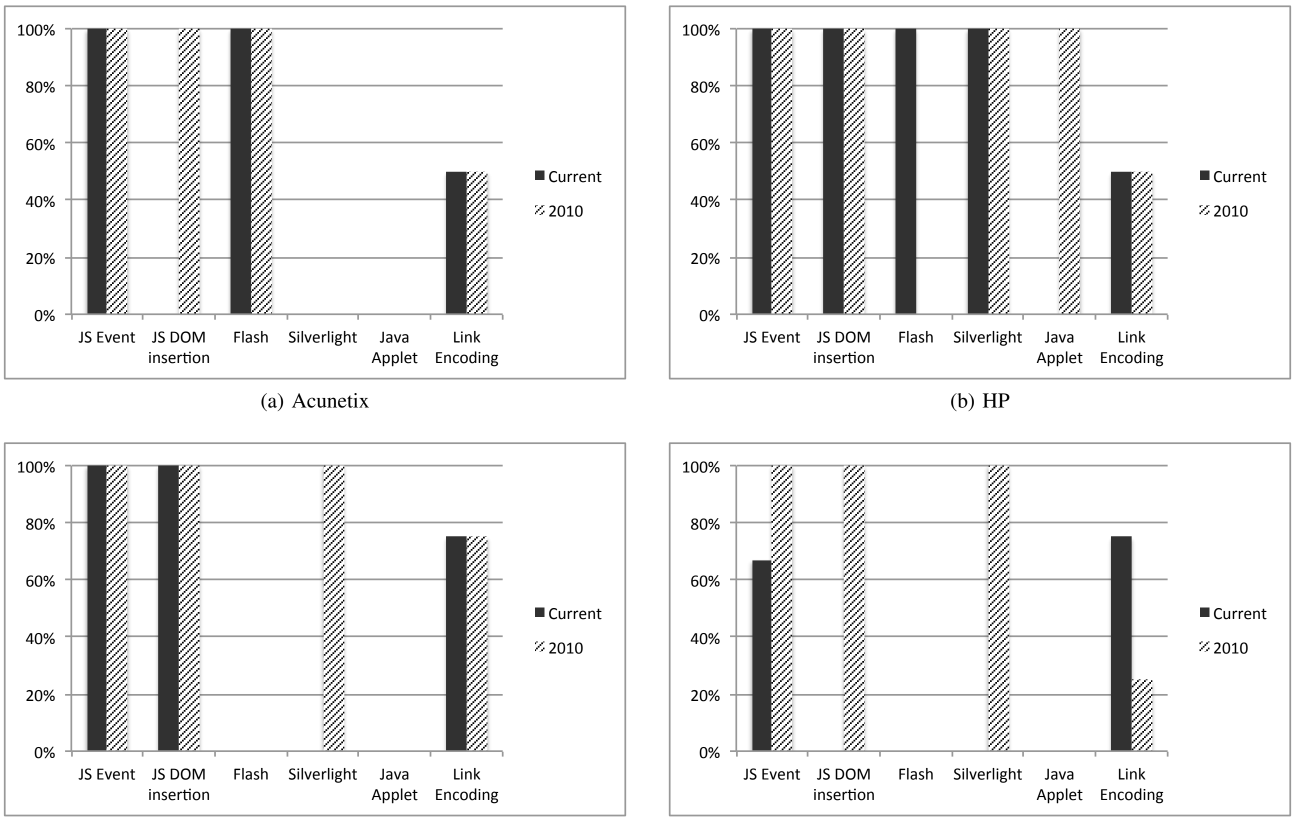


Fig. 3. Comparsion of Scanner Vulnerability Detection

1) Crawling: In the crawling test suite, URL links are written in different encodings (such as decimal) and technologies (such as Flash or DOM manipulation, in the manner of a drop down menu), with a unique landing page for each test that records every page request. Figure 2 shows all tests whose results have changed for any scanner since 2010, solid bar representing the current percentage whereas the striped bar represents the percentage from 2010. It is interesting to note which categories have experienced change. All of the Javascript-related categories have seen their crawling results change, we believe because scanner vendors are attempting, successfully or otherwise, to adapt their crawling engines to the increasing number of sites designed with Web 2.0-style Javascript-based interfaces [10]. With their continued prevalence, Flash links are now correctly crawled by the HP scanner, and the Qualys scanner has gained support for URL-encoded and octal-encoded links. On the other hand, with the apparent decrease in popularity of Silverlight [11] and Java applets [12], support has waned for both technologies, with no vendor now following links encoded in Java applets and both IBM and Qualys dropping support for Silverlight links.

In addition to these categories with changes, all scanners both now and back in 2010 were able to follow PHP redirects, meta-refresh tags, pop-ups, iframe links, and links that appear based on POSTing certain values from a `<select>` menu. Also, both now and in 2010, all scanners except Acunetix were able to follow a link encoded in VBscript.

2) Vulnerability Detection: We will briefly describe the testbed here, as further testbed detail can be found in [5]. The testbed is a PHP application with intentional coding vulnerabilities in 10 categories. It contains 3 types of XSS vul-



(a) Acunetix (b) HP (c) IBM (d) Qualys

Fig. 2. Scanner crawling results on synthetic testbed on categories that saw change from 2010 results

TABLE V
SCANNER VULNERABILITY DETECTION RESULTS ON SYNTHETIC TESTBED

Category	Acunetix		HP		IBM		Qualys	
	Current	Δ from ’10	Current	Δ from ’10	Current	Δ from ’10	Current	Δ from ’10
XSS Type 1	50%	0%	50%	0%	**100%**	+50%	**100%**	0%
XSS Type 2	0%	-20%	0%	-20%	0%	0%	**40%**	0%
XSS Advanced	30%	-20%	30%	+30%	10%	+10%	**40%**	+20%
XSS Total	24%	-18%	24%	+12%	18%	+12%	**47%**	+12%
SQLI Type 1	**43%**	0%	14%	0%	14%	0%	29%	0%
SQLI Type 2	0%	0%	0%	0%	0%	0%	0%	0%
SQLI Total	**33%**	0%	11%	0%	11%	0%	22%	0%
Other Injection	**36%**	-18%	14%	-5%	14%	-23%	14%	+14%
Session	44%	+25%	25%	0%	**50%**	+13%	31%	+6%
CSRF	20%	0%	**40%**	+40%	**40%**	0%	0%	0%
Information Leak	**50%**	0%	25%	0%	**50%**	0%	**50%**	+25%
Malware	0%	0%	0%	0%	0%	0%	0%	0%

nerabilities: “textbook” reflected (Type 1), “textbook” stored (Type 2), and more novel forms of reflected XSS such as using encoding or the `<prompt>` tag (Advanced), and 2 types of SQL injection vulnerabilities: ”non-stored” where user input is directly used in a query (Type 1), and “stored” where user input first enters the database and then the input is used in a query (Type 2). It also contains other forms of vulnerabilities to server code injection e.g. file includes and header injection, session management vulnerabilities such as fixation and insecure login, CSRF vulnerabilities, information leakage such as directory listings and the presence of predictably named backup files, and finally the presence of a few malware files. We did not replicate or test the Server Configuration vulnerabilities from [5] since the focus of our study is on vulnerabilities introduced in the development process, rather than during deployment operations.

Table V presents a summary of how well each scanner detected the existent vulnerabilities in the synthetic testbed, again with the gray column reporting current results and the white reporting change from 2010. Cells in bold indicate the scanner with the best performance in the given category (row).

The data reports a baseline for the false-negative performance of each scanner. Roughly, the scanners detected between 21% (HP) and 32% (Acunetix) of all vulnerabilities in the testbed. The scanners as a group performed best in the XSS Type 1, Session, and Information Leak categories, with detection performance for the categories equaling 75%, 38%, and 43% respectively when averaged across scanners. Average scanner performance for XSS Advanced, SQLI Type 1, Other forms of injection, and CSRF were all between 20% and 25%. Finally, all scanners performed poorly on the "stored" forms of XSS and SQLI, with average detection rate below 10% for XSS and at 0% for SQLI, and no scanner was able to detect any malware, both now or back in 2010, likely because such detection is outside the scope of the products we used.

To compare scanners on the testbed by category, as seen in Figure 3, the Qualys scanner was superior in XSS detection, as it found more Advanced types and was the only scanner to find any stored vulnerabilities. Acunetix leads in detection of injection vulnerabilities, whether SQL or other forms, while IBM detects the most session vulnerabilities. No single scanner was superior for the CSRF or information leak categories.

It is also interesting to observe how scanner performance has (or has not) changed in the past 3 years, and how it may map to the evolving perception of risk in the security industry. For instance, we note that SQLI detection performance has not changed for *any* scanner, possibly indicating a lack of developmental interest from the industry. We speculate this may be due both to the perception of SQLI as a "solved problem" due to the existence of a known solution (prepared statements) and the increasing popularity of Web application backend storage that avoids SQL altogether. Further, scanner performance in detecting "other forms of injection" in the testbed, in particular PHP local/remote file includes and CRLF/header injection, has decreased since 2010, possibly because the industry is optimizing its scanning vectors (limited to maintain scanning performance in time/traffic) towards vulnerabilities considered more prevalent or important.

On the other hand, all of the XSS categories have seen significant change, with 3 scanners improving by 12% for XSS overall while one detected 18% less vulnerabilities. We interpret these deltas as indication that the industry is quite attuned to the need for XSS mitigation and thus as been actively at work in adapting their products to security needs. Given how the relative number of vulnerabilities found by each scanner varies by tested application (as shown in the next section), we believe these scanner changes are capable of causing regression in detection performance as well as progression, and thus we do not necessarily believe decreased XSS detection performance to be an indictment on overall scanner quality.

Finally, we mention that the Qualys and IBM scanners have started flagging UI-spoofing based vulnerabilities, which no scanner did back in 2010. Qualys reports a legitimate lack of "clickjacking" protection in our testbed, while IBM reports vulnerability to "phishing through frames".

B. New Application Testbed Study

We next ran all four scanners on the 27 applications developed by the startup and freelancer participants and screened their report outputs for false positives. In the end, we chose to manually verify all XSS, SQLI, and other injection vulnerabilities while disregarding the other categories for three reasons. First, these vulnerabilities are prevalent and cause "severe" code integrity breaches–they currently occupy the top two spots in the OWASP Top 10 [2]. Second, vulnerabilities in these categories are directly attributable to *developer* error, which is the focus of the second half of this study, as opposed to errors in system configuration which can be attributable to system administrators. Further, we tested many startups using their development environments, which usually did not have SSL, thus making counting certain session vulnerabilities somewhat unfair. Third, we found far fewer false positives in the scanner output for these categories than any other, especially information leak, which we will now briefly discuss.

1) False Positives and Vulnerability Count: In terms of raw, unchecked output, Acunetix reported a total of 4411 vulnerabilities, HP 28,026, IBM 4895, Qualys 1061, for a grand total of 38,393. Of these, 24,931 (65%) of the report items were "information leaks", and HP by itself reported 20,051 such items. After some examination, we quickly determined a vast majority of info leak items to be false positives. This was because a scanner would try fuzzing for auto-backup files on known URL filenames with many variations (e.g. `#profile.php#`, `profile.php~`) in order to test for source code disclosure. However, the scanner then has trouble determining whether an information leak has occurred based on the HTTP response. We suspect it relies heavily on the response code, thus causing false positives in site which, say, return 200 instead of 404 for "resource not found" messages. Since scanners contain many vectors for fuzzing, each of these becomes a false positive, thus rapidly skewing the results. For this reason we eliminated info leaks from our study.

On the other hand, we found very few false positives for XSS, SQLI, or other injection items reported by scanners. Anecdotally, during our post-scan interviews, only two startups out of 19 found false XSS items, and none found false SQLI or injection items. We found a similar level of accuracy for scanner performance on the freelancer sites, where all XSS and SQLI findings were deemed correct. This result also accords fairly well with the previous findings on the synthetic testbed in [5], where only 2 false positive incidents involving XSS or SQLI were mentioned, involving only 2-3 scanners out of 8.

By manually auditing for false positives, our dataset represents a baseline measure of the XSS and injection vulnerabilities existent in each app. Other forms of audit, such as manual review, may reveal more vulnerabilities, but we believe they will not lower our reported vulnerability counts.

Finally, we count vulnerabilities as distinct if they differ in either URL or input parameter, as these could all be caused by different server-side code paths. While we acknowledge there are other ways of vulnerability accounting that are more

TABLE VI
SUMMARY OF SEVERE VULNERABILITIES FOUND BY SCANNERS

	Acunetix	HP	IBM	Qualys
Total XSS and Injection Found	177	102	254	248
# of applications where scanner found most vulns	5	6	8	4

closely tied to server-side code, our accounting method is proportional to exposed attack surface, which we believe to be appropriate for reporting black-box testing results.

2) Results By Application: Figure 4 depicts the number of severe (XSS, SQLI, or other injection) vulnerabilities found by each scanner on each application. We present data in this fashion to illustrate the large variability in the relative scanner "rankings" from application to application. In total, as shown in Table VI Acunetix found the most vulnerabilities on 5 applications, HP on 6, IBM on 8, and Qualys on 4. Furthermore, there were 11 applications (41%) on which one or more scanners found more than 10 serious vulnerabilities, and on each of these applications the worst-performing scanner found at most 2 vulnerabilities, and always at least 10 less than the best-performing scanner.

Given this amount of variability in detection performance across applications, we think that it is likely imprudent to assign relative values to scanners based on results from a single or even a few narrowly-focused benchmarks, however carefully designed they may be. We recommend potential customers of these products to make thorough comparative evaluations of the scanner in their own environments and possibly incorporate more than one audit tool in their security process. In some sense, this variability reflects the reality for applications facing attackers: it only takes one correct vector for an application that was previously "secure" to become completely vulnerable. However, such variability seems to confirm that black-box scanners rely on a set of vendor-developed detection *heuristics* that behave brittlely when faced with broad application coding practices and deployment conditions. For researchers, this may represent a challenge and opportunity to apply systematic methods to yield consistency in vulnerability detection.

3) Statistical Study: With the variability by application mentioned in the previous section, we perform some statistical analysis on the set of vulnerability data to see if some humanly identifiable scanner detection rate trends are statistically significant. Our analysis views each application as a sample in a population of 27 and each scanner as an instrument which measures vulnerabilities per (1000) line of code as a normalized detection metric for each sample. We then consider detection of all "severe" vulnerabilities (XSS + Injection) and also XSS and Injection vulnerabilities separately as three experimental runs over the same population. For each of the three experimental runs, we compute the average over 27 samples of the metric measured by each scanner. These are reported in Table VII.

At first glance, our data shows that, in accordance with the results from the synthetic testbed, the Qualys scanner performs superior to the others in XSS detection while the Acunetix scanner (tied with IBM) is best at detecting injection vulnerabilities. The data also seems to indicate that the Qualys scanner identifies injection vulnerabilities at a lower rate than the others.

However, to quantify the statistical significance in comparing the average normalized detection rate between scanners, we conduct both an 1-way ANOVA test between all scanner means and also paired student's t-tests with Bonferroni correction between all pairs of scanners' detection results on the 27 samples. These tests produce a p-value which indicates the degree of statistic confidence in the comparison of means. (All p-values are two-tailed in this paper.) ANOVA results for the three categories are (F=1.161 p=0.328) for XSS, (F=1.367 p=0.257) for injection, and (F=0.977 p=0.407) for XSS+injection, indicating no statistically significant differences in mean detection performance in any category between the 4 groups, at the common 95% confidence level.

For the student's t-test, since we apply the pairwise test 6 times, a stricter p value of 0.0083 using the Bonferroni correction is required for 95% confidence. This level was also not met by any comparison, since the lowest p-values for pairwise comparison are 0.044 for Acunetix-Qualys in injection and 0.069 for IBM-Qualys in injection. We thus conclude that, in rough accordance with our manual observation of variability in the previous section, there are no statistically significant comparisons between the average detection performance of scanners in any category, using our testbed of 27 sample applications.

Since we cannot even compare average performance meaningfully despite having 27 real-life samples, we strongly believe this statistical conclusion argues that any tool comparisons performed on a narrower (or singular) testbed will not be generalizable, and therefore, not be predictive. Furthermore, for the remainder of this study, we will combine the audited vulnerability reports from all four scanners into a single metric when evaluating application vulnerabilities, in an effort to reduce susceptibility to variation in a single tool and create a more reliable metric.

4) Scanner Handling of Javascript: Finally, we wish to relay two anecdotes that occurred during our weeks of testing which highlight some scanner limitations and successes when it comes to dealing with client-side Javascript.

First, two of the startups were coded in a very Javascript-heavy fashion on the client side. One rich-content creation application generated most of its interface with DOM manipulation performed by Javascript organized by backbone.js [13]. While the application used Javascript `pushstate()` [14] to update the browser URL bar (so that copy-and-pasted links would bring users to the appropriate application state), there were no links coded in HTML at all–the served index.html only consisted of the `<head>` element plus several `<script>` tags linking in Javascript libraries and setting the user session state needed by those libraries. Despite the application containing a large number of distinct URLs and pages

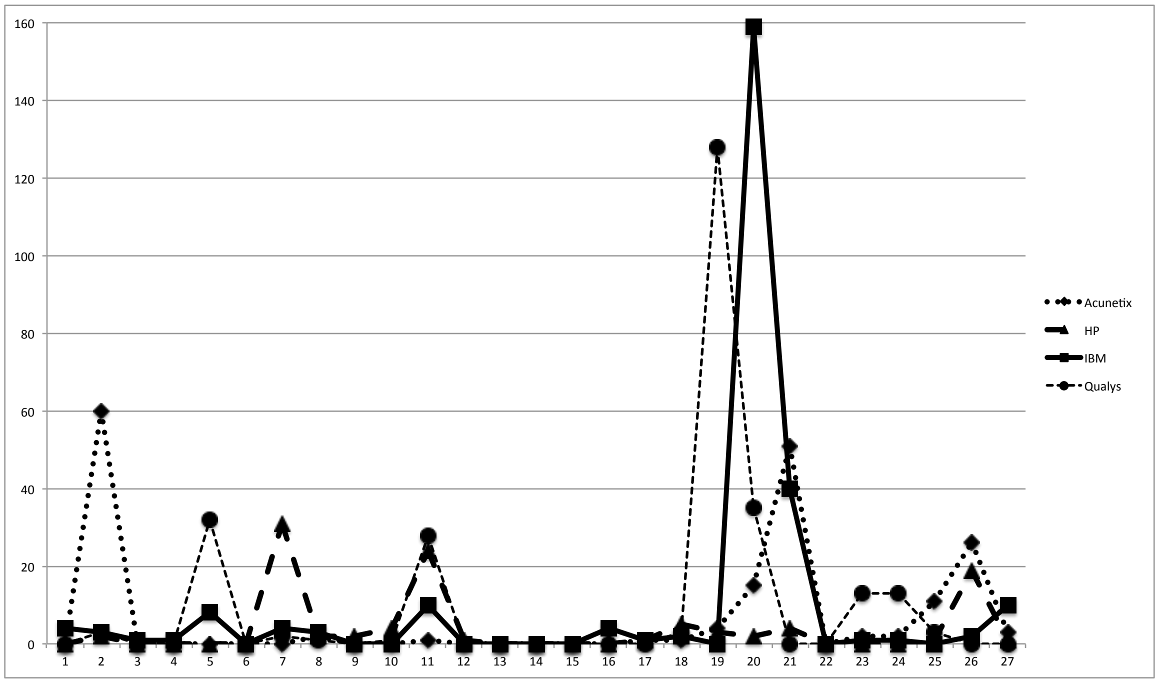


Fig. 4. XSS and Injection vulnerabilities found by scanners for each study participant

TABLE VII
AVERAGE SCANNER VULNERABILITY DETECTION RATE OVER 27 APPS AND STATISTICAL CONFIDENCE OF PAIRWISE MEAN COMPARISON

Category	Acunetix $\frac{vulns\ detected}{1000\ LOC}$	HP $\frac{vulns\ detected}{1000\ LOC}$	IBM $\frac{vulns\ detected}{1000\ LOC}$	Qualys $\frac{vulns\ detected}{1000\ LOC}$	p-value from 1-way ANOVA
XSS+Injection	0.55	0.34	1.47	1.89	0.41
XSS	0.22	0.19	1.15	1.85	0.33
Injection	0.32	0.14	0.32	0.04	0.26

when users interact with it via browser, all of the scanners experienced serious trouble even crawling this application to discover links and user inputs to test. No scanner was able to find any meaningful vulnerabilities in this application, largely due the inability to discover application pages beyond the initial landing page. A second application, written in Java and compiled to Javascript with Google Web Toolkit [15], also caused all scanners to remain stuck on the index page. This behavior stands in contrast from the advertising and interface options for some browsers, which promise Javascript execution in order to find links.

This first anecdote serves to illustrate our observation that the scanners, originally designed as HTTP request fuzzers/response interpreters more naturally suited for a server-reliant Web application model, are still in the process of adapting to more client-Javascript heavy Web 2.0 applications. For instance, the IBM tool includes an entirely separate module that analyzes Javascript code downloaded during scans. While tighter integration of Javascript analysis modules with the crawling engine would likely result in superior site coverage and detection performance, we can report that this code-analysis module was successful in one instance, successfully detecting a Type-0 DOM-based XSS vulnerability, on a site which used a vulnerable version of `swfobject.js` [16] from 2009. (The `swfobject.js` developers later patched the very same vulnerable lines of code that the IBM module flagged [17].)

V. STARTUP DEVELOPERS VS. FREELANCERS

Having used the participating applications together as a testbed for judging individual scanners, we now combine all legitimate vulnerabilities found by all scanners together as a single metric for judging the coding quality of groups of developers categorized by startup versus freelancer, security quiz score, and also programming language usage.

We first describe the results we obtained from the security quiz and scanning of the web applications and highlight some interesting differences in security knowledge between startup developers and freelancers. We also look at the distribution of vulnerabilities and identify correlations between education and vulnerabilities as well as show certain deficiencies in freelancers. We also provide some of our experiences interacting with freelancers and developers.

A. Security Quiz Results

We graded the quizzes on a simple scale in an attempt to reduce noise in the data. For any given question, a person could receive full, half, or no credit. One of the authors graded all the

TABLE VIII
AVERAGE STARTUP AND FREELANCER DEVELOPER SCORE ON SECURITY QUIZ

Category	Startup	Freelancer	p-value of comparison
XSS	39%	39%	0.99
Injection	70%	56%	0.33
Crypto Storage	90%	50%	**0.0004**
SSL	59%	59%	0.99
UI	66%	50%	0.12
Info Leak	68%	44%	0.24
SOP	63%	13%	**0.011**
Total	61%	48%	0.17

quizzes to maintain consistency, and since the questions were gathered from a security class, the answers were compared against the corresponding key. Table VIII presents the average results for startups and freelancers as a group, as well as the p-value calculated for the mean comparison by a student's t-test. The scores are in percentage correct, out of 100.

On average, startup developers performed slightly better than the startup developers on average (around 61 percent to 48). However, the difference is not statistically significant. The range and standard deviation of the scores was greater for the startup developers with some developers receiving perfect or close to perfect and others scoring below 50 percent. We also took a more in-depth look into the scores of the individual categories of questions asked. Startup developers and freelancers both struggled with questions regarding XSS, with a score around 39%. Startup developers performed equivalently or better than freelancers on all categories, only with statistical significance in the cryptographic storage (p=0.0004, more discussion below in Sec. V-D) and same origin policy categories (p=0.011). (The next lowest p-value was .12 for the higher average scores for startups in the UI security category.) We speculate that the significant difference in performance between freelancers and startups may be due to the "same-origin policy" concept and terminology being more familiar to American developers linked to Silicon Valley than to international freelancers.

B. Vulnerability Rate Results

For completeness, Table IX presents the rate of XSS+injection, XSS-only, and injection-only vulnerabilities detected by the 4 scanners as a group for each application. It also serves as the raw data for all the statistical comparisons by category that we perform in Sections V and VI.

Figures 5, 6, and 7 present these vulnerability rates plotted against the developers' score on the security quiz. From Figures 5 and 6, the human eye can discern a loose negative correlation between the number of vulnerabilities per line of code and the security quiz score. (Negative correlation here is the direction that matches intuition, that a higher security quiz score equals fewer security vulnerabilities.) Table X lists the Pearson product-moment correlation coefficients (r) between application vulnerability rates and the security quiz score of their developers, along with the p-value calculated for each coefficient. The correlations are reported for all 27 participant and for the startup and freelancer groups.

TABLE IX
DETECTED VULNERABILITY RATES ($\frac{vulns}{1000LOC}$) BY APPLICATION. THIS DATA IS PLOTTED IN FIGURES 5, 6, AND 7

No.	Provenance	Language	XSS+Injection	XSS	Injection
1	Startup	Java	0.24	0.00	0.24
2	Startup	PHP	0.97	0.97	0.00
3	Startup	PHP	0.10	0.00	0.10
4	Startup	Ruby	0.10	0.00	0.10
5	Startup	PHP	20.00	16.50	3.50
6	Startup	Java	0.00	0.00	0.00
7	Startup	PHP	1.30	0.67	0.63
8	Startup	PHP	7.00	4.00	3.00
9	Startup	Python	0.08	0.08	0.00
10	Startup	Python	0.10	0.06	0.04
11	Startup	PHP	2.11	1.79	0.32
12	Startup	Ruby	0.33	0.33	0.00
13	Startup	Java	0.00	0.00	0.00
14	Startup	Python	0.00	0.00	0.00
15	Startup	Java	0.00	0.00	0.00
16	Startup	Ruby	0.40	0.30	0.10
17	Startup	Python	0.20	0.20	0.00
18	Startup	Ruby	1.10	1.10	0.00
19	Startup	Ruby	25.47	25.09	0.38
1	Freelancer	PHP	37.41	33.69	3.72
2	Freelancer	ASP	3.54	2.12	1.41
3	Freelancer	ASP	0.00	0.00	0.00
4	Freelancer	Java	0.00	0.00	0.00
5	Freelancer	PHP	1.40	1.40	0.00
6	Freelancer	PHP	3.65	0.00	3.65
7	Freelancer	Java	5.88	1.25	4.63
8	Freelancer	Java	1.17	0.50	0.67
Average			4.17	3.34	0.83

TABLE X
CORRELATION BETWEEN APPLICATION VULNERABILITY RATES AND SECURITY QUIZ SCORES OF THEIR DEVELOPERS

Group	XSS+Injection		XSS		Injection	
	r	p	r	p	r	p
All 27	-0.245	0.22	-0.205	0.31	-0.359	**0.066**
Freelancers	-0.079	0.85	-0.068	0.87	-0.102	0.81
Startups	-0.303	0.21	-0.261	0.28	-0.440	**0.059**

We can see that all r values are negative, which at least accords with intuition. We also point out that there exists borderline significant correlation between the rate of injection vulnerabilities and the security quiz score for startups and for all participants, so more knowledgable programmers appear to be somewhat effective in translating their knowledge into prevention of injection vulnerabilities, possibly due to the existence of an effective canonical solution (prepared statements) for SQL injection. Interestingly (but probably without significance), the negative correlation was slightly stronger between injection vulnerability rates and security score over the *entire* quiz, rather than a score just over injection questions.

We also find it noteworthy that the correlation in every category is stronger for startups than for freelancers. We speculate that this may be because startup participants are more invested in the security of their applications (as evidenced by their choice to participate in exchange for security reports) and so put more effort into translating their knowledge

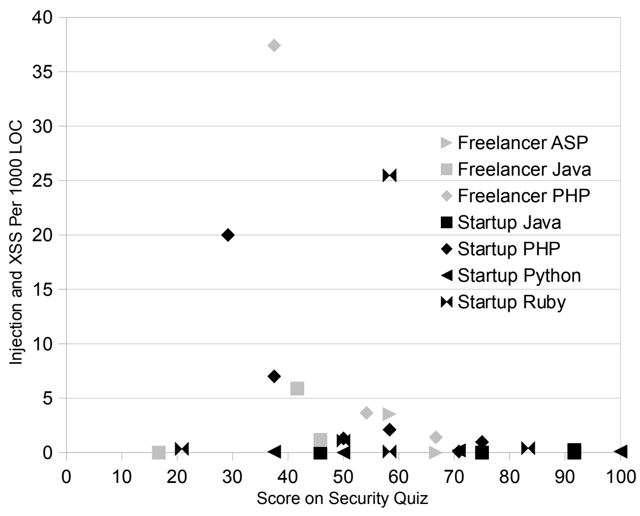


Fig. 5. Rate of injection and XSS vulns versus developer quiz score

TABLE XI
COMPARISON OF AVERAGE VULNERABILITY RATES ($\frac{vulns}{1000LOC}$) BETWEEN FREELANCERS AND STARTUPS

	XSS+Injection	XSS	Injection
Freelancer avg	6.63	4.87	1.76
Startup avg	3.13	2.69	0.44
p-value	0.37	0.54	**0.027**

into practice. We believe freelancers, on the other hand, are incentivised by the bidding process in their hiring towards producing minimally-viable functionality in the shortest time. They also often create sites for customers who cannot verify the quality of their work. Since security features are fairly easy to eliminate without any noticeable difference to the functionality of the application, freelancers looking to make shortcuts may not bother with them, despite knowing the proper techniques. Thus, the chief characteristic that distinguishes between freelancers and startups (shown not to be quiz score) may be motivation. We will provide some further evidence of this divide in the next section.

Overall, the loose correlation between vulnerability rate and quiz score implies some disconnect between security knowledge and its implementation in practice. Even motivated developers might understand the source and solution to these security vulnerabilities but not know how to avoid or solve these vulnerabilities in practice. Or they may not understand the potential vulnerabilities but still avoid them due to framework features (or sheer luck). The underlying reasons for this disconnect are outside the scope of this study. However, we also find the relation between knowledge and implementation for the cryptographic storage area particularly illustrative and will analyze it in more detail below in Sec. V-D.

C. Freelancer and Startup Comparisons

We now look at the effect of the type of developer (startup developer and freelancer) on the average number of severe vulnerabilities per line of code (XSS and injection), summarized

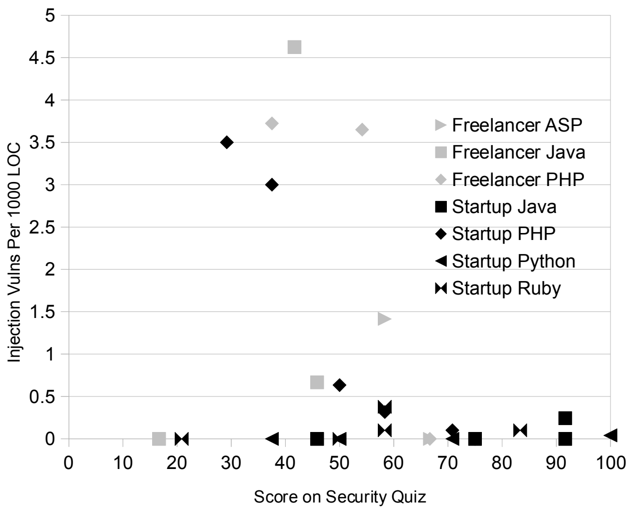


Fig. 6. Rate of injection vulnerabilities versus developer quiz score

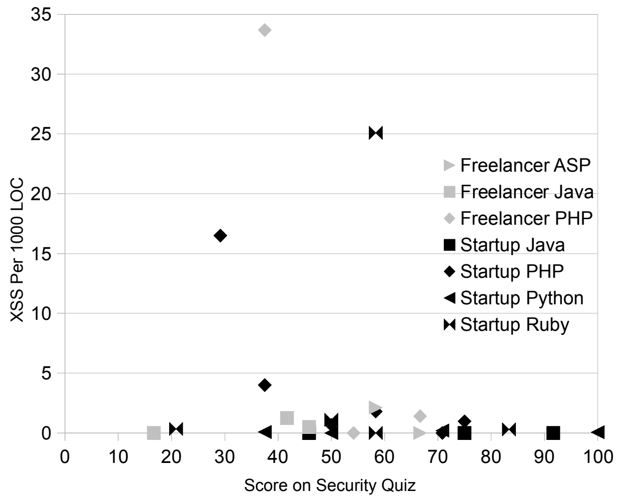


Fig. 7. Rate of XSS vulnerabilities versus developer quiz score

in Table XI. We find no meaningful comparisons for the XSS and XSS+injection categories, but freelancers produced more injection vulnerabilities on average compared to the startup developers with statistical significance of 97 percent (p-value = 0.03, calculated by student's t-test). Freelancers are also worse in a nearly-identical and statistically-significant way if we look at just SQL injections rather than all injections.

Cross-referencing our data about security knowledge in Table VIII, we did *not* find that freelancers knew significantly less than startup developers about avoiding injection vulnerabilities. Given the effectiveness of the textbook solution here, the worse injection vulnerability performance of the freelancers seems again to be attributable to motivation.

D. Freelancer Cryptography Case Study

In our security quiz, we asked the common cryptography question regarding storage of passwords in databases. This question was particularly interesting because we had access to the username-password databases for the freelancers, and

TABLE XII
FREELANCER SECURITY QUIZ RESPONSE AND ACTUAL IMPLEMENTATION FOR PASSWORD DATABASE

Freelancer	Security Quiz Response	Actual Implementation
20	hash only	hash only
21	hash only	hash + salt
22	hash + salt	hash + salt
23	hash only	hash only
24	hash only	hash + same salt
25	use login	hash only
26	hash only	plaintext
27	hash only	plaintext

there was a straight forward way to check if their knowledge matched their actual practice. This is an interesting area to explore because unlike the other vulnerabilities such as XSS and injection, the problem is very specific and isolated in the code base, and the solution to this problem is clearly defined – hashing the password with a unique salt that is stored in cleartext. Their answer on the quiz had some correlation to their practices, but not as strong as we would expect. As seen in Table XII, only one freelancer who knew to hash and salt the password with a unique value actually did that in practice, and one did it despite answering it incorrectly on the quiz. A majority of people who knew to hash the password did so, but two freelancers stored passwords in plaintext despite knowing that the password needed to be hashed. Therefore, this data is more definitive evidence that although developers claim to know correct security practices, it does not always mean these practices will be implemented correctly.

On the other hand, we found the code of one freelancer particularly interesting. Freelancer 21 answered the quiz question incorrectly by saying that the password only needed to be hashed, but in practice, he actually hashed and salted the passwords. Here, he appears to be benefitting from code he does not understand or has not reviewed, similar to how developers benefit from frameworks in general. However, we did not find strong evidence that a framework was incorporated in his source code in a modular fashion, rather, it appears to be copy-and-pasted as a part of a larger library. Thus, the security benefit gained from third-party code in this case is likely at the cost of maintainability.

Although we could not examine the startups' password databases due to privacy and security reasons, we believe that there might be a similar trend. It is also interesting to note that startup developers clearly outperformed freelancers on this quiz question. We believe this might be because the participating startup developers are more attuned to news of recent security breaches involving password databases at LinkedIn, Sony, and others, and also because of their university computer-science education included basic cryptography and security as part of the curriculum.

E. Summary

From our data, we find statistically significant data that freelancers are more prone to producing injection vulnerabilities than startup developers, but in general we find weak correlation between knowing a security vulnerability and actually avoiding it in practice. We did find borderline significant correlation between developer knowledge and resultant application vulnerability rates for injection vulnerabilities, possibly because there is a known solution to a large number of these vulnerabilities. Correlations between knowledge and security were in all cases weaker for freelancers than for startup developers, which may point to a question of motivation. The lack of strong correlation, exemplified by the clearly defined and well-known security problem of storing passwords in a database, shows that knowledge might be a factor but does not strongly determine the prevention of security vulnerabilities. We believe this disconnect between knowledge and result to highlight an interesting area, requiring further research to better understand developers and their practices.

F. Our Experience with Freelancers

We now share some of our dealings with the freelancers in the hope that it is informative for future customers and also researchers pursuing studies of a similar nature.

We found freelancers to be generally unreliable. We received bids from over 100 freelancers, and the average promised delivery time was between 30-40 calendar days. For the freelancers we hired, the average promised delivery period was 35 days. However, only 2 freelancers delivered within 2 days of what they promised. The freelancer as a group missed their promised deadline by 21 days, and one did not finish even 40 days after the promise date and had to be excluded from the study. As a result of this experience, we believe it might be difficult to contact a freelancer to fix a potential security vulnerability after the site is complete. Incidentally, although we intentionally selected freelancers from different price ranges, we did not find any clear correlation between price and vulnerability rate or on-time delivery.

In comparison, our experience with startup developers was far more positive. The startup developers were far more reliable and responsive to our questions and requests, usually responding within a day or two and frequently taking the initiative to inquire whether our results were ready.

VI. LANGUAGES

We now look at the data on programming languages and vulnerabilities for startup developers and freelancers. We begin by reporting preliminary statistics regarding the freelancers and startups use of languages. The distribution of programming languages was predetermined (ASP, JSP, PHP). We received the most bids for ASP (53), with PHP (45) in second, and JSP (15) a distant third. The average prices were very similar. The highest was PHP ($3000) with ASP ($2600) and JSP ($2450) as second and third respectively. The average time for completion had JSP as the shortest with 31 days with ASP and PHP tied around 38 days.

Out of the 19 startups, as seen in Table 1, the most popular language was PHP, followed by Ruby.

Table XIII lists the average vulnerability rates for each language in our study. To test for statistical significance of

TABLE XIII
AVERAGE VULNERABILITY RATES ($\frac{vulns}{1000LOC}$) OF APPLICATIONS BUILT IN EACH LANGUAGE

	XSS+Injection	XSS	Injection
PHP	8.22	6.56	1.66
Java	1.04	0.25	0.79
Ruby	5.48	5.37	0.12
Python	0.10	0.09	0.01
ASP	1.77	1.06	0.71

the mean comparison between the 5 languages, we performed a 1-way ANOVA test and obtained values of (F=0.93 p=0.47) for the XSS+injection category, (F=0.84 p=0.52) for the XSS category, and (F=1.48 and p=0.24) for the injection category. ANOVA results thus state that pairwise comparisons of the mean vulnerability rates between two language groups, such as between PHP applications and Java applications, are not statistically significant.

However, we further try comparisons that partition the entire population into two groups. In particular, we split our entire population into ”PHP” and ”Not PHP”, as PHP has the most number of samples. Viewed in this way, and applying the student's t-test, we find that the average rate of injection vulnerabilities is higher for the PHP group (mean=1.66 vuln/1000LOC) than the non-PHP group (mean=0.42 vuln/1000LOC) with a significant p-value of 0.03. (The PHP group also has an average rate of XSS+injection vulnerabilities (mean=8.22 vuln/1000LOC) higher than the non-PHP group (mean=2.14 vuln/1000LOC) with a borderline p-value of 0.10). No other language groups have a significant comparison to their complement set in the same way. Moreover, this data matches well with our approximation using CVEs from NVD in Section II, which showed that PHP had significantly more reported CVEs compared to other languages. This might imply that vulnerabilities in PHP are a widespread issue, which is magnified since PHP is the most used programming language.

We speculate PHP's statistically-significant higher rate of injection vulnerabilities is due to a propensity for PHP developers to forgo a framework and any associated database interfaces, instead creating their own (improperly sanitized) SQL queries in raw PHP. We did not observe any of the three PHP freelancers using a framework, and only 2 of the 6 startups using PHP reported using a framework (CakePHP and Yii). This lack of framework usage, combined with developer mistakes in manual sanitization in generating HTTP responses, could also explain PHP's higher rate of XSS, leading to the borderline statistically-significant higher rate of all severe vulnerabilities (XSS+injection).

Whether PHP users are naturally disinclined towards frameworks remains an open question. However, in contrast to PHP, *all* users of Python (most often Django) and Ruby (all Rails) in our study reported using a framework. In fact, on the background survey of startups, only developers that used PHP answered ”none” or left the question blank when asked about backend frameworks used, likely because the other languages in our study all originated as general purpose languages that would be quite inconvenient to Web developers without framework support. In any event, we did observe in our study that frameworks were not a complete panacea, as on Ruby-on-Rails application had the second highest number of confirmed XSS in the entire study.

Finally, because we made observations about freelancer and PHP vulnerability rates in the previous section and this, we naturally wondered about the combination of the two. The 3 freelancers using PHP produced an average vulnerability rate 3.39 times the average vulnerability rate for the entire population for all severe vulnerabilities, 3.51 times for XSS, and 2.95 times for injection. Using a z-test to check the significance of this sub-population mean, we found p-values of 0.054, 0.079, and 0.051 respectively, meaning those who hire PHP freelancers would appear headed for a statistically significant bad time.

VII. SUMMARY AND RECOMMENDATIONS

We now summarize the major findings of the paper and provide advice to application owners and scanner consumers based on these findings.
Regarding scanners:

- Due to variability in performance from application to application, it is difficult to predict which scanner will perform best on a single chosen application. We recommend developers thoroughly evaluate scanners on their own environments and not rely on results from testbeds, and possibly use more than one tool.
- Longitudinal results seem to indicate most scanner development effort on crawling dynamic Javascript events and Flash, and on detecting XSS and UI-spoofing vulnerabilities. Conversely, results appear to show stasis for SQL-injection detection and dropping support for crawling Silverlight and Java applets and detecting remote/local file-includes and header injection.
- Scanners' “Information Leak” report items are majority false positives.

Regarding developer selection and education:

- Freelancers are more prone to producing injection vulnerabilities on applications than startup developers, in a statistically-significant way.
- We believe there is evidence in our study to attribute the difference in vulnerability rates between startups and freelancers to “motivation”. Correlations are in all cases stronger between developer knowledge and measured security for startups than for freelancers, and despite their worse results for injection vulnerability rate, freelancers scored similarly to startups on the security quiz questions covering injection.
- Overall, correlation between developer security knowledge and the vulnerability rates of their applications is not as strong as we hoped. This apparent disconnect is interesting and worth further study on developers and their practices.

- Startup developers and freelancers score comparably on the entire security quiz. The difference is not statistically significant.
- Startup developers did score significantly higher on questions covering cryptographic storage and the same-origin policy.
- We found freelancer and their estimates of delivery schedule to be unreliable.

Regarding language selection:

- PHP applications have higher rates of injection vulnerabilities than non-PHP applications, in a statistically-significant way.
- The combination of PHP and freelancers leads to roughly 3x higher rate of vulnerabilities, in a statistically-significant way. We recommend that new application developers avoid this combination or require the use of a development framework. Owners of existing applications of this particular provenance should check their applications for serious vulnerabilities.
- In our study, PHP applications appear much less inclined to use frameworks than applications developed in other languages.

VIII. Related Work

A. Vulnerability Statistics

The MITRE corporation runs large databases for vulnerability statistics and classifications, including the National Vulnerabiiity Database (NVD) [1], and the Common Weakness Enumeration (CWE) [18]. While these databases are valuable repositories of vulnerability data over a large number of applications, their entries are collected from voluntary disclosures by security researchers or companies and so report rough statistics that are not scrutinized to remove duplicates, normalize vulnerability counts to application size, etc. We view the statistics provided by our study, while from a smaller sample, as complementary to these sources, since we provide carefully curated data from a designed experiment on newly created code that attempts to measure vulnerability rates caused by selected developer characteristics.

B. Comparison of Languages

Several academic and industry white papers also provide statistics on Web application vulnerabilities, typically classifying applications by language. Our study is the first of which we are aware that investigates developer education and background as factors in the resultant vulnerability rates of their applications. Finifter et. al. [19] examined 9 applications (3 PHP, 3 Perl, 3 Java) built around the same spec with a single manual reviewer (Finifter) and a single black-box tool (BurpSuite [20]), finding "little evidence that programming language plays a role in application security" but rather placing such influence in frameworks. In our study of 27 samples, we were able to find statistically significant evidence that PHP was more prone to certain vulnerabilities than other languages, which may possibly be caused by the disinclination of PHP developers towards frameworks. Incidentally, Finifter's nine applications had vulnerability rates for XSS+SQLI in the range of 0.07 to 0.91 per 1000 LOC, which is compatible with the range we observed. Walden et al. [21] compared the vulnerability rates of 14 PHP and 11 Java applications and found higher rates for PHP, but without statistical significance.

WhiteHat Security's annual report [22] is another data source for real life vulnerability statistics with a large sample. It classifies vulnerability counts and reports vulnerability and remediation durations by application industry, and type of vulnerabilities, and has classified by language in past editions (though not in 2012). Our work is one of a fairly different nature than theirs, distinguished mainly by our consideration of causal factors for vulnerabilities versus their focus on baseline counts and remediation, which make our reports again complementary.

C. Comparison of Vulnerability Detection Tools

There has also been a developing body of work, from both academic and industry sources, focused on benchmarking and evaluating various techniques for web application vulnerability detection, including black-box scanners, using a testbed or one to two applications. Bau et. al. [5] evaluated 8 commercial scanners for crawling and vulnerability detection performance using a custom testbed, which is reused by our study to provide longitudinal data on scanner developments since 2010–results from that study can be found in Sec. IV-A. Doupé et. al. [6] studied 11 commercial and open-source scanners, also on a single testbed site, and produced a performance ranking. They also found crawling to be a central challenge to scanner performance in general. Austin [23] evaluated the security of two applications with a variety of techniques including manual pen-testing, static analysis (with Fortify 360 [24]), and automated pen-testing (with IBM AppScan) and found that automated pen-testing was most effective in terms of implementation vulnerabilities discovered per hour, but systematic manual pen-testing may be effectively performed in tandem to discover flaws in design.

From industry, the WebGoat [25] and WIVET [26] projects are well-known vulnerability and crawling testbeds respectively often used in scanner development and for cursory scanner comparisons. SecToolMarket [27] is a Website that benchmarks over 60 scanner editions using a single synthetic testbed, with result trends, such as a high relative rate of detection on Type 1 XSS and a much lower one on file-inclusions, that appear to match our findings in this paper.

However, using the broadest testbed (27 organic and 1 synthetic applications) of which we are aware for evaluating vulnerability detection tools, our study was able observe something unique from all previous benchmarking efforts: great variability in the relative performance of scanners from one application to the next. This variability across applications rendered no comparison of average detection rates between scanners statistically meaningful. Application owners should *not* expect comparative results from singular testbeds to hold in their environment, and researchers should similarly **avoid** relying on narrow benchmarks, however well designed they might be, to judge between vulnerability detection techniques.

IX. Conclusion

Our study, undertaken with a breadth of samples in both tools (4 scanners) and applications (19 startups and 8 freelancers), demonstrates that developer background and programming language are both statistically significant factors in certain instances on the vulnerability rates of Web applications. We found that freelancers are significantly more prone to injection vulnerabilities than startups, but that the difference may be more attributable to motivation than education. We also found that PHP applications have significantly more injection vulnerabilities than other languages, which may be explained by low rate of PHP developers in our study using supporting frameworks.

We found an interesting apparent disconnect between developer security knowledge and actual measurable vulnerability rate, which can be the basis of future research into developers and their engineering practices.

Regarding black-box scanning, our study observed evidence that the scanner industry is working on increasing crawling and detection support for client-Javascript heavy applications and cross-site scripting, possibly at the expense of other vulnerabilities with well-understood solutions such as file-inclusion and SQL injection. With the breadth of applications in our study, we demonstrated a large amount of variability of scanner performance across applications. Thus, the size of the testbed required to make statistically meaningful and predictive comparisons of tool performance is greater than the size of our study. Finally, we believe that this variability presents a research opportunity for how to systematically improve the consistency of scanner performance across all applications.

Acknowledgment

The authors would like to thank Qualys, Acunetix, IBM, and HP.

References

[1] National Vulnerability Database. http://nvd.nist.gov/

[2] OWASP Top 10 Project. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[3] Number of Incidents. http://datalossdb.org/statistics

[4] G. Sharath and R. Rashmi, "Analysis of Reliability in Web Service for Fuzzy Keyword Search in Encrypted Data," *International Journal of Computer Trends and Technology*, vol. 3.4, 2012.

[5] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the Art: Automated Black-Box Web Application Vulnerability Testing," vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, May 2010, pp. 332–345.

[6] A. Doupé, M. Cova, and G. Vigna, "Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners," in *Proceedings of the 7th international conference on Detection of intrusions and malware, and vulnerability assessment*, ser. DIMVA'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 111–131. http://dl.acm.org/citation.cfm?id=1884848.1884858

[7] BuiltWith Trends. http://builtwith.com/

[8] Framework Usage Statistics. http://trends.builtwith.com/framework

[9] Vulnerabilities in OSVDB Disclosed by Type by Quarter. http://osvdb.org/

[10] JQuery Usage. http://trends.builtwith.com/javascript/jQuery

[11] SilverLight. http://trends.builtwith.com/framework/Silverlight

[12] Java Applet Usage. http://trends.builtwith.com/docinfo/Applet

[13] Backbone.js. http://backbonejs.org

[14] "The History Interface," *HTML5: Edition for Web Authors*. http://www.w3.org/TR/html5-author/history.html#dom-history-pushstate

[15] Google web toolkit. https://developers.google.com/web-toolkit/

[16] swfobject. http://code.google.com/p/swfobject/

[17] swfobject revision 402. http://code.google.com/p/swfobject/source/detail?r=402

[18] Common weakness enumeration. http://cwe.mitre.org/

[19] M. Finifter and D. Wagner, "Exploring the Relationship Between Web Application Development Tools and Security," in *Proceedings of the 2nd USENIX conference on Web application development*, ser. WebApps'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 9–9. http://dl.acm.org/citation.cfm?id=2002168.2002177

[20] PortSwigger Web Security. Burpsuite. http://portswigger.net/burp/

[21] J. Walden, M. Doyle, R. Lenhof, and J. Murray, "Java vs. PHP: Security Implications of Language Choice for Web Applications," in *International Symposium on Engineering Secure Software and Systems (ESSoS)*, 2010.

[22] (2011) Whitehat security statistics report. https://www.whitehatsec.com/resource/stats.html

[23] A. Austin and L. Williams, "One Technique is Not Enough: A Comparison of Vulnerability Discovery Techniques," in *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 97–106. http://dx.doi.org/10.1109/ESEM.2011.18

[24] HP Fortify. Fortify Source Code Analyzer. https://www.fortify.com/products/hpfssc/source-code-analyzer.html

[25] OWASP WebGoat Project. https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

[26] Web Input Vector Extractor Teaser. http://code.google.com/p/wivet/

[27] Sectoolmarket. http://www.sectoolmarket.com/

Appendix
Security Quiz Questions

1) Most recent browser editions issue security warnings when a website issues a self-signed or unsigned SSL certificate. Aside from avoiding these pesky browser warnings, why should a popular website not use self-signed or unsigned certificates?
2) Many web sites / web applications utilize user logins, and therefore need to store the passwords associated with each user. What standard security measures should be taken when storing user passwords to a database?
3) Why do banking sites display an image chosen by the customer (a firetruck, a tennis racket, etc.) on their login screens?
4) What is the best way to protect against SQL injection?
5) What does the "secure" flag mean? If a cookie is set as "secure", how will it be sent back to the server?
6) What type of attack was the "httponly" cookie designed to protect against?
7) A naive user (or administrator!!) of your web application clicks on a link in his email inbox that takes him to the malicious site attacker.com that is controlled by hackers. What types of threats does this behavior pose to your legitimate web application?
8) Here is some PHP code (remember in PHP file names may be URLs)

```
$dateOfAppointment = $_GET['date'];
include($dateOfAppointment . " events.php");
```

Why is this code a bad idea?

9) Which of the following techniques, if used as the ONLY technique by which user input is introduced to a Web page, is most effective at preventing Javascript cross-site scripting (XSS)?
   a) Filter out all instances of script tags from user input
   b) Encode all user inputs as html entities
   c) Introduce all user inputs into the page using only document.createTextNode
   d) Other
10) Certain unix editors, such as emacs, leave auto-backup files whenever you use them to edit a file. What bad things can happen if these types backup files appear on a production server?
11) There was a recent bug in the implementation of Java applets with regards to redirects. Assume that accessing the URL: http://example.com?redirect=malicious.com/applet will redirect the browser to fetch malicious.com/applet. The bug caused the JVM to consider an applet served with this redirected URL as "belonging" to example.com rather than malicious.com, whose servers actually deliver the applet. What fundamental web security principle does this implementation violate, and how might an attacker exploit this?
12) Assume that example.com allows all of its pages to be shown in a iframe of a page served by a third party domain. What security risks might this pose to example.com?