Effect of the Secure Programming Clinic on Learners' Secure Programming Practices

Melissa Dark
dark@purdue.edu

Lauren Stuart
lstuart@purdue.edu

Ida Ngambeki
ingambek@purdue.edu

Purdue University
West Lafayette, IN 47097

Matt Bishop
mabishop@ucdavis.edu

University of California at Davis
Davis, CA 95616

Abstract - In order to improve the abilities of students to write robust programs ("secure programming") without adding new classes or material in existing classes, a Secure Programming Clinic that functions analogously to an English writing clinic has been developed. This paper reports on preliminary results from an instance of the clinic at a large university. Given the statistics obtained from this trial, the clinic improved the students' secure programming behavior and helped students develop a deeper understanding of secure programming concepts.

Categories and Subject Descriptors

D.1.m [Programming Techniques]: *Miscellaneous.*

D.2.3 [Softwre Engineering]: *Miscellaneous.*

K.3.2 [Computers and Education]: *Coding Tools and Techniques.*

K.6.5 [Management of Computing and Information Systems]: *Security and Protection.*

General Terms:

Experimentation, Human Factors, Security

Keywords

Secure programming, clinic, education, educational metrics

1. INTRODUCTION

A large number of vulnerabilities are present in code written by professional practitioners. Many are code defects that could have been prevented with the application of secure and robust coding principles. "Secure coding", "secure programming", or "robust programming" refers to a software product's robustness against accidental or malicious unexpected behavior causing a problem. One example of the impact of failures in secure programming was the Heartbleed bug in 2014, which impacted services across the Internet. It resulted from a failure to check a length variable for validity [6].

The failure of practitioners to practice this style of programming is not due to incompetence. One factor is simply lack of training. As Evans and Reeder noted [5], "We ... [have a] desperate shortage of people who can design secure systems, write safe computer code, and create the ever more sophisticated tools needed to prevent, detect, mitigate and reconstitute from damage due to system failures and malicious acts." And the shortage is projected to grow. Each year, more programmers enter the workforce. The U.S. Department of Labor projects that the demand for software developers will increase much faster than that for many other occupations due to an increasing demand for software on multiple platforms and in multiple industries [2]. These programmers need to be trained in secure programming.

The failure to practice secure programming is not due to incompetence. One factor is simply lack of training. Despite desire and interest in teaching more secure programming in the college computing and cybersecurity curricula, doing so is hard. Three basic issues underlie the problem of teaching students how to write secure code: the lack of room in the computer science curriculum, the focus of introductory programming courses, and not teaching in a manner that promotes the application of learned techniques of good programming. A glance at the ACM Computing Curricula [1] shows how much material must be compressed into courses for computer science majors. There is already so much content considered essential to the undergraduate curriculum, that secure programming is considered much less essential than other topics. For the many universities and faculty trying to integrate secure programming into the curriculum, there is the challenge of when to infuse it within the sequence. Beginning programming classes typically focus on algorithmic and language issues rather than environmental issues. These classes teach some elements of secure programming, such as good program structure, basic input validation, checking bounds for array references and checking that pointers are non-nil. They do not teach more advanced elements, such as avoiding race conditions and authentication over a network, because those elements involve knowledge that beginning programming students are not expected to have. Classes after the introductory programming class assume that students know, and will apply, principles of good programming. In practice, this is not true. Students tend to focus on what is being taught, and regard the programs they write as instruments to exercise that knowledge. This is appropriate, but—like an English essay comparing Orwell's *1984* to Huxley's *Brave New World*—the expression of the content is as important as the content itself. Unfortunately, graders (and many teachers) ignore the issue of well-crafted, robust code when they grade, and simply check that the program works. The result is that the application of learned techniques of robust programming is not reinforced, leaving students with the implicit belief that security does not matter as much as functionality.

2. THE SECURE PROGRAMMING CLINIC

The Secure Programming Clinic (SPC) serves as a resource where students can learn about and apply

aspects of secure programming in a manner that does not add extra courses to the curriculum and, indeed, allows for the integration of secure programming concepts in multiple contexts across several courses. The main goals of the project are:

1. To develop a clinic to enhance **student learning and expertise** in writing robust, secure software;
2. To develop and validate **measures that reflects the ability of learners with regard to writing robust, secure software;**
3. To implement and evaluate the **efficiency and efficacy of the clinic**; and
4. To disseminate the clinic procedures, instructional modules, and measures to the educational community.

3. BACKGROUND

The Secure Programming Clinic project seeks to develop an educational innovation that enhances students' learning and expertise in writing robust, secure software. An *expert* is somebody who obtains concrete, tangible results that are vastly superior to those obtained by the majority of the population. Conventional thinking has been that "geniuses are born, not made". However, research on expertise and its development shows that expertise is made [4]. Furthermore, expertise is not a state that is attained permanently; when knowledge stores are not put into practice, they can and do go dormant. Conversely, when performance picks back up, knowledge stores are reactivated.

When expertise is conceptualized (as it should be) as attainable and results-oriented, two important implications follow. First, experiences that help build expertise can be designed and developed. Second, expertise and its development can be measured, and those measures can be used to improve experiences designed to develop expertise, creating a feedback loop. This is foundational to the design of the SPC.

3.1 Experiences that Help Build Expertise

The journey to truly superior performance is characterized by several distinctive features. Experts accumulate knowledge over numerous learning experiences, many of which are not traditional classroom experiences. Developing expertise is time intensive, and full of struggles and mistakes. Developing expertise involves both improving the knowledge skills one has, and extending the reach and range of those skills. The path to expertise involves learners testing the veracity and robustness of that knowledge as a way of honing what is known. However, this honing is not a solitary activity. Research shows the critical role of expert mentors can help accelerate the learning process. These mentors serve an essential role in giving constructive, corrective feedback. This deliberate practice at tasks just beyond one's current level of competence and confidence, with meaningful feedback, is how stronger mental models are built and increasingly more capable performances realized. Expertise involves complete acts of thought where the expert fluently moves from specific to general cases and back [3], and building expertise requires practicing movement in both inductive and deductive reasoning.

While it is clear that practice, practice, and more practice are essential for developing expertise, it is equally clear that not all practice is equally effective. Research finds that practice with the following features is superior. The practice must be deliberate, meaning that considerable, specific, and sustained efforts must be undertaken to do something that one cannot do well; purposefully working at what one

cannot yet do well is essential. Effective practice also engages learners in thinking with deliberation (*deliberate thinking*). Deliberate thinking requires a planned approach to learning. This includes intent to improve through a plan, but at that same time the planned approach is adaptive. Deliberate thinking explores possibilities and thinks through consequences *in situ*. Afterwards, deliberate thinking involves reflective processing about when, how, and why a course of action did not work as planned, and corrective assessment. The third essential feature of effective practice is that it be habitual. The development of expertise happens when deliberate thinking and acting occurs in regular, chunked increments. For example, we instinctively know that if one wants to improve at baseball, one must practice intensively. Two hours of practice a day, 6 days a week, is likely to be much more effective than 12 hours of practice on Saturday. It is just as important that deliberate thinking be habitual. Deliberate practice without deliberate thinking can slip into routine doing, with no thought. When this happens, an essential characteristic of expertise building has been lost. And finally, efforts to hone existing knowledge, as well as efforts to choose to move outside one's comfort zone, are habitually practiced.

3.2 Clinical Education as a Framework for Building Expertise

Clearly, a foundational aspect of developing expertise in learning is doing (and redoing). One of our first decisions was to ground the SPC in the clinical education model, and to specifically approach our clinical experiences with the goal of building toward expertise. The principle of clinical education is the integration of theory and guided practice so the students have the opportunity to immediately implement or apply in practice the theoretical knowledge gained in class. The development of expertise will be based on numerous, scaffolded learning experiences with opportunities for deliberate practice and deliberate thinking as discussed above. While there are many educational experiences we can envision for the clinic, we are taking a deliberate approach in developing and testing the interventions of the clinic so that we know which interventions to keep, modify, or stop. This paper reports on results from the first instantiation of the clinic.

4. RESEARCH METHODS FOR TESTING THE EFFICIENCY OF THE SPC

This paper reports on data collected at the initial iteration of the clinic at a large university in the spring of 2015. At this iteration of the clinic, the research team primarily focused on building experiences for deliberate practice with opportunities for guided feedback from clinicians during clinical visits. Throughout the academic term, secure programming concepts were covered in class, and students were given homework assignments that required them to apply the secure programming concepts (learning by doing). Students could visit the clinic (meet with the expert clinicians) to identify mistakes made, discuss corrections needed, and why (the latter is especially important for developing students' ability to test the veracity of their knowledge) before submitting their homework. Once the homework was submitted and graded, students could resubmit the program for a re-grade, in which they recover up to 80% of the points deducted for non-robustness. They were free to visit the clinic before resubmitting. Thus, there were four types of students:

1. Proactive visitors, who visited the clinic before the first submission;
2. Reactive visitors, who visited the clinic after the first submission;
3. Consistent visitors, who visited the clinic both before and after the first submission; and
4. Non-visitors, who did not visit the clinic at all.

4.1 Research Question

The research question is: how does level of engagement in the clinic, as defined by those four types of students, affect students' robust programming behavior?

4.2 Population and Sample

There were a total of 104 students enrolled in the class. Robust programming behavior was analyzed by examining student code on homework 2. The researchers were able to analyze code for 42 students; table 1 shows the breakdown by group.

	N
Non Visitors	6
Reactive Visitors	17
Proactive Visitors	14
Consistent Visitors	5

Table 1: Breakdown of students who visited and did not visit the clinic

4.3 Variables

The students were asked to write a program that checks the status of a second program that performed some privileged function. The students were to write code that found the second program; verified that program had the specified owner, permissions, and other attributes; and if so, modified the permissions to enable the program to run with sufficient privileges to perform the privileged function. The program had to run on designated class resources and to be robust.

Within the program, the researchers looked for indicators that the students recognized and adequately addressed possible vulnerabilities. These vulnerabilities and their mitigation are categorized in five variables:

1. Comment the code. Appropriate comments contribute to robustness of the code by allowing future users and maintainers of the code to understand the code better.

2. Use file descriptors or some equivalent way to maintain access to a file object; furthermore, use these to make changes to the file object to ensure the proper object was modified. Using file descriptors eliminates possible time of check to time of use of race conditions. The improper use of file descriptors fails to achieve the robustness objective as the code is still open to the same compromises as it would have been without any use of file descriptors.

3. Sanitize the environment for spawning sub-processes that executed other programs. One sub-process was required for the assignment; a second was appropriate for the extra credit; and in both cases, spawning the correct sub-process required the environment to be reset to a pristine, known safe state. The failure to do so enables an attacker to cause unexpected behavior, including privilege escalation. A student can fail to clean the environment for just one of these external programs; in this case, there seems to be some disconnect or failure to understand or translate the concept into action in slightly different situations.

4. Safely handle strings and buffers through proper allocation, termination, and bounded operations that do not depend on user input (and, thus, a well-behaved user) to prevent reading beyond the end of the object (which can access additional information) or writing beyond the end of the object (which can change additional information and potentially change the behavior of the program).

5. Check the returns of calls for unexpected values, proper ranges, and failure indicators, to be sure the program is behaving correctly.

Researchers looked for evidence that students implemented the appropriate mitigations for vulnerabilities using indicators. The resulting score ranged from –16 to 16.[1] Students were awarded one point if they implemented the appropriate behavior consistently and completely, and one half-point if they implemented the behavior somewhat or sporadically. If the student's program was vulnerable but the student did not implement a mitigation, the student lost one point. If the student's program was not vulnerable, no points were awarded or deducted; the student might have consciously or unknowingly chosen an implementation that avoids the vulnerability. The sixteen variables are described in table 2.

Comment Code	1	The code is accompanied by comments where the student describes the behavior of the code. Commented-out code does not count.
File Descriptors	2	Use (1): At least one consistent reference to a file is maintained, whether as a file descriptor or as an equivalent approach in a non-C language.
	3	Consistency (1): The same file descriptor is used throughout the program.
	4	All (1): In every case in which the file descriptor could be used, it is used.
	5	Get Status (1): The student used the file descriptor, and the appropriate function (*fstat*(2) in C, or its equivalent in another language), to check the status of the file.
	6	Change Permissions (1): The student used the file descriptor, and the appropriate function (*fchmod*(2) in C, or its equivalent in another language) to change the permissions of the file.
External Calls	7	Change Owner (1): The student took appropriate steps to provide a clean environment for the execution of a program to change ownership (*chown*(1)).
	8	Execute Call (1): The student took appropriate steps to call the Change Owner program (*execve*(2) in C, or its equivalent in another language).
String and buffer handling	9	Allocation (1): If the student set up a string buffer, the buffer is protected from reading or writing beyond the end of the buffer by a length variable. Students could reference a system-set **MAX_LENGTH** value, or a hardcoded length for their check.
	10	Termination (1): If the student used a string buffer, the buffer is protected from reading or writing beyond the end of the buffer by proper string termination (a terminal '\0').
	11	Bounded print (1): If the student printed the contents of a string buffer to the standard output, the student chose a function that limits the number of bytes written (for example, *snprintf*(3) in C, or its equivalent in another language).
	12	Bounded construction (1): If the student constructed strings dynamically, the student chose functions that limit the number of bytes written or read (for example, *strncat*(3) or *strncpy*(3) in C, or their equivalent in another language).
Return value checks	13	All system calls (1): The student checked the return values of all system calls.
	14	Change Permissions (1): The student checked the return value of the permission-

[1] These scores were not related to the scores assigned to the homework.

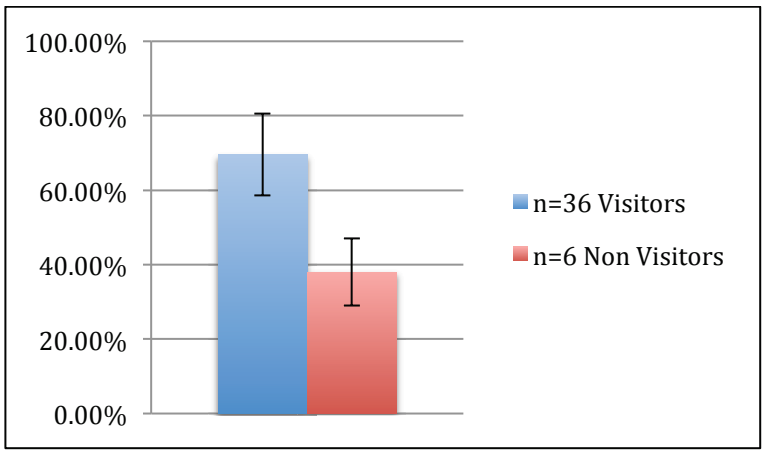
		changing function in particular (*chmod*(2) or *fchmod*(2) in C, or their equivalent in another language).
	15	Execute Call (1): The student checked the return value of the command-execution function in particular (*execve*(2) in C, or its equivalent in another language).
	16	Open (1): The student checked the return value of the function to open a file and obtain the file descriptor (open(2) in C, or its equivalent in another language).

Table 2: Secure programming variables examined in students' code

Students' code was compared at all available time points to see whether and how the code changed in response to clinic visits, grading, or both. The majority of students used C and C++ as their primary programming language.

5. RESULTS

The descriptive data for the final score (the score for the last program submitted by the students) for both groups are summarized in figure 1.



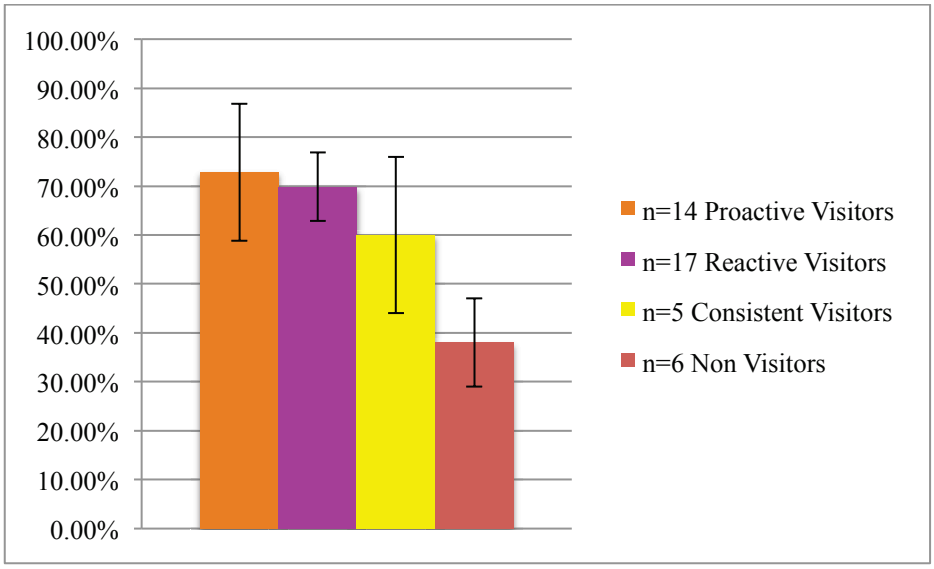
	N	Average	Std Dev.
Visitors	36	69.61 %	23.24%
Non Visitors	6	38.02 %	17.50%

Figure 1: Comparing final scores for students who did or did not visit the clinic

An independent sample t-test was conducted to compare the final scores for visitors and non-visitors to the clinic. There was a significant difference in final scores between visitors (M=69.62%; SD=23.24%) and non-visitors (M=38.02%; SD=17.50%); $t(40) = -3.89$, $p = 0.004$.

Of the 36 students who visited the clinic, 14 were proactive visitors, meaning that they visited the clinic *before* the homework was due. Seventeen students were reactive, meaning that they visited the clinic *after* the homework was graded (and returned to them) to understand how to fix their mistakes before submitting the revised homework for a re-grade (it should be noted that resubmission for a higher score was optional). Five students visited the clinic both before the homework was due and after the graded homework was returned in order to make modifications to their work prior to resubmission; these five students are called *consistent* visitors. The final score averages for the students in all four groups (proactive visitors, consistent visitors, reactive visitors, and non visitors) are summarized in figure 2 below.

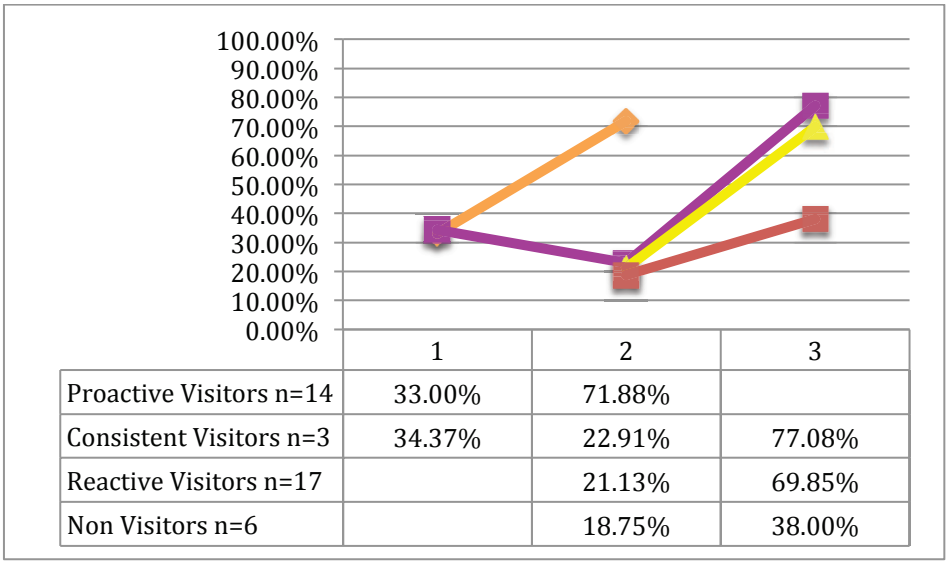
In figure 2, we can see that the proactive visitors had the highest final scores on the assignment, the reactive visitors had the second highest final scores, the consistent visitors the third highest, and the non visitors had the lowest final scores. A one-way between-subjects ANOVA was conducted to compare the effect of timing and frequency of clinic visits on students' final scores. There was a significant effect of the timing and frequency of the visits at the $p < 0.05$ level for the four visitor conditions [$F(3, 38) = 3.67$, $p = .021$]. Post hoc comparisons using the Tukey HSD test indicated that the mean score for the non-visitors (M = 38.02%, SD = 17.51%) was significantly different from the final grades of proactive visitors (M = 72.77%, SD = 28.49%) and reactive visitors (M = 69.85%, SD = 14.78%).



	N	Final grade	Std. Dev.
Non Visitors	6	38.02%	17.51%
Reactive Visitors	17	69.85%	14.78%
Proactive Visitors	14	72.77%	28.49%
Consistent Visitors	5	60.00%	32.58%

Figure 2: Comparing final scores for students who visited the clinic at different time points

Next we considered what growth in knowledge looked like across these different groups. Figure 3 below shows the differences between T1 (pre-homework submission score), T2 (homework score), and T3 (homework resubmission score).



	1	2	3
Proactive Visitors n=14	33.00%	71.88%	
Consistent Visitors n=3	34.37%	22.91%	77.08%
Reactive Visitors n=17		21.13%	69.85%
Non Visitors n=6		18.75%	38.00%

Figure 3. Comparing average grades across time points

We only have T1 data for proactive and consistent visitors (the fact that we have these data is what puts them into their respective categories). As can be seen, the proactive and consistent visitors start out very similarly (33.00% and 34.37% respectively). The mean score for proactive visitors on the homework (T2) increases notably to 71.88%. The T2 scores of non visitors, reactive visitors and consistent visitors are clustered tightly at 18.75%, 21.13%, and 22.91% respectively (keep in mind that the T2 scores for reactive visitors and non visitors is their first grade). However, by T3, the scores of the reactive group are up to 69.85%, and the scores of the 3 participants in the consistent group who submitted for re-grade are up to 77.08%, while the non-visitors are only up to 38.00%.

6. DISCUSSION AND CONCLUSION

It appears that the clinic is having a big impact in helping students learn. All students who visited the clinic scored significantly better on the sixteen instances of five specific applicable robust programming behaviors measured by the researchers. All groups showed gains when the assignment was submitted for re-grade; however, groups who visited the clinic demonstrated significantly more gains than those who did not visit the clinic. For the proactive visitors, the gain is before the homework is submitted. These students use the clinic to troubleshoot and improve their work prior to submission. For the reactive visitors, the gain is after the homework is submitted. These students turn in the homework to find out what needs to be fixed, and then use the clinic to address those gaps. It should be noted that the number of participants in each group is small, and especially so in the consistent and non visitor groups.

The consistent visitors exhibit an interesting pattern. Unlike the other groups who visited the clinic, they demonstrate a loss in secure programming behavior between their first and second submission. This is most likely due to the very small sample size in consistent visitors; one student in that group who did very poorly on the homework submission skewed the results. The three consistent visitors who submitted at T3 demonstrate substantial gains between the second and third submissions. This may be because students in this group were less likely to submit complete or close to complete programs on their initial clinic visit. This suggests that students in this group first visited the clinic not to troubleshoot (which is more common among proactive visitors) but because they were struggling with the assignment. It would then follow that they did not focus on secure programming issues in their program until their second clinic visit. Again, it should be emphasized that the size of this group was very small, so we can only conjecture, not conclude.

The results reported here indicate that the clinic is succeeding in both improving the students' secure programming behavior and helping students develop a deeper understanding of secure programming concepts. Further data collected at this clinic and other secure programming clinics at other institutions as the project proceeds will be analyzed to determine whether the observed results of patterns and timing of clinic visits is replicated.

Acknowledgements. This material is based upon work supported by the National Science Foundation under Grant Number DGE-1303211 and Grant Number DGE-1303048. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] ACM. *Computing Curricula 2001 Computer Science*. New York NY, USA (Dec. 2001).

[2] Bureau of Labor Statistics, U.S. Department of Labor. *Occupational Outlook Handbook*, 2016-17 Edition, Software Developers. URL: http://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm (visited April 13, 2016).

[3] Dewey, J. *How We Think*. Dover Publications, Inc. Mineola, NY, USA (1910).

[4] Ericsson, K., Charness, N., Feltovich, P. and Hoffman, R. (Eds.), *Cambridge Handbook of Expertise and Expert Performance.* Cambridge University Press, Cambridge, MA, USA (2006).

[5] Evans, K., and Reeder, F. *A Human Capital Crisis in Cybersecurity: Technical Proficiency Matters*. Center for Strategic and International Studies, Washington, DC, USA (July 2010).

[6] Durumeric, Z., Kasten, J., Adrian, D., Halderman, J. A., Bailey, M., Li, F., Weaver, N., Amann, J., Beekman, J., Payer, M., and Paxson, V. The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (IMC '14). ACM, New York, NY, USA. pp. 475-488 (Nov. 2014).