C + Go = An Alternate Approach Toward the Linux Programming Course

Saverio Perugini
saverio@udayton.edu
University of Dayton
Dayton, Ohio

Brandon M. Williams
brandonwilliamscs@gmail.com

ABSTRACT

The use of the C programming language in a Linux programming course—common in most undergraduate computer science programs—has been the standard practice for nearly thirty years. The use of C is appropriate because Linux is written in C and, thus, programming with the system (i.e., accessing operating system structures and making calls to the kernel) is natural in C. However, this seamless integration of Linux and C can inhibit student assimilation of course concepts—through conflation of concepts with language (e.g., system calls with C, or libraries with `gcc`, respectively)—and, ultimately, student learning outcomes for a variety of reasons. We challenge the idea of the exclusive use of C in the Linux programming course and alternatively propose the use of the Go programming language, in strategic conjunction with C, to both achieve student learning outcomes and address some of the issues with an exclusive C approach. We explored and studied this approach—the use of Go in the Linux course—over the course of seven consecutive offerings of it. We present our experience with this approach including a collection of desiderata resulting from it as well as student survey data as an evaluation of its use in practice. Overall, the results indicate this approach is feasible, is no worse than an exclusive C approach, and yields advantages. We anticipate this experience report will inspire adoption of the use of Go in similar Linux programming courses.

CCS CONCEPTS

• **Social and professional topics** → **Computer science education**; **Computer engineering education**; • **Software and its engineering** → **General programming languages**; *Operating systems*; Concurrent programming languages;

KEYWORDS

C programming language; filters; Go programming language; Linux kernel; Linux programming; Linux programming course; pipes; system calls; toolchains

SIGCSE '20, March 11–14, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6793-6/20/03…$15.00
https://doi.org/10.1145/3328778.3366944

ACM Reference Format:
Saverio Perugini and Brandon M. Williams. 2020. C + Go = An Alternate Approach Toward the Linux Programming Course. In *The 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20), March 11–14, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3328778.3366944

1 INTRODUCTION

The use of the C programming language in a Linux programming course—common in most undergraduate computer science programs—has been the standard practice for nearly thirty years. The undergraduate *Linux Programming* course at the University of Dayton is a three-credit hour, programming-intensive course that introduces students to programming within the Linux environment and programming with the Linux kernel (e.g., using the system call interface and interprocess communication mechanisms). It is not a course on elementary Linux usage (e.g., file management and manipulation), which is covered in our *Operating Systems* course—the prerequisite for *Linux Programming*. While the use of C is appropriate because Linux is written in C and, thus, programming with the system (i.e., accessing OS structures and making calls to the kernel) is natural in C, it can also be a vice because this seamless integration of Linux and C can inhibit student assimilation of course concepts—through conflation or over-association of concepts with language (e.g., system calls with C, or libraries with `gcc`)—and, ultimately, student learning outcomes for a variety of reasons. Three student learning outcomes in a typical Linux course of this nature are:

(1) An understanding of *Linux system calls*, their residence in the kernel, and programming with them.
(2) An understanding of *toolchains* (i.e., compiling, statically- and dynamically-linking, library/API construction, shell variables controlling their behavior), how to use them, and their role in software development.
(3) An understanding of the *Linux filter style of concurrent programming* as a composition of multiple atomic processes dynamically connected through interprocess communication mechanisms (e.g., FIFOs/pipes) (see Fig. 1—right):

> If you give me the right kind of Tinker Toys, I can imagine the building. I can sit there and see primitives and recognize their power to build structures a half mile high, if only I had just one more to make it functionally complete. — Ken Thompson, creator of UNIX and the 1983 ACM A.M. Turing Award Recipient, quoted in *IEEE Computer*, 32(5), 1999.

Approaches to realizing these three student learning outcomes have traditionally included, respectively:

(1) The use of the system call interface provided by C (e.g., `unistd.h`: `fork`/`wait`/`execvp`, `read`/`write`; `open`/`close`).
(2) The use of the `gcc` toolchain and associated shell variables (e.g., `C_INCLUDE_PATH` and `LIBRARY_PATH`) controlling its behavior.
(3) The use of unnamed pipes (i.e., FIFOs) as provided by the shell (i.e., |) in conjunction with a variety of built-in Linux tools (e.g., `detex paper.tex | aspell list | sort | uniq | wc -l`).

In teaching this course for sixteen consecutive offerings of it, save one, we have experienced the following phenomena, respectively:

(1) Students associate system calls with calls to a C library rather than calls to the Linux kernel for a core OS service. In other words, students tend to think that system calls are a service provided by a C language library rather than a service provided by the Linux kernel. This is primarily because the mechanics of using the system call interface provided by C are the same as a library call. As a result, the idea of context switching between user mode and kernel mode tends to get lost.
(2) Students perceive both the decomposition of a (systems) program into an interface (`.h`), implementation (`.a`), and client application (`.c`) and the toolchain associated with packaging and installing the library and API as features of `gcc` rather than sound software engineering supported by multiple languages from Java and Python to Racket.
(3) Students view the Linux filter style of concurrent programming as a pattern of programming exclusive to the Linux environment rather than an incarnation of a more general, recurring style of programming with connections to pipelines from functional programming (e.g., in Elixir, `expr |> f >| g(a) |> h(b,c) = h(g(f(expr), a), b, c)`) and the generate-filter programming approach made possible through the use of the lazy evaluation parameter-passing mechanism [4] in, e.g., Haskell.

There are two general themes embedded into these three specific issues:

- Conflating or inseparably associating a *concept* (e.g., system calls, a toolchain) with a *language/tool* (e.g., C, `gcc`, respectively).
- Over-learning or inability to abstract away the details. Students tend to think that these ideas encountered in the Linux programming course are specific to Linux and C rather than general computing principles applicable in a variety of software development environments and contexts.

These more general issues are really two sides of the same coin.

In this paper, we challenge the idea of the exclusive use of C in the Linux programming course and alternatively propose the complementary use of the Go programming language, especially as an avenue toward addressing the aforementioned issues. Go was developed by Robert Griesemer, Rob Pike, and Ken Thompson—the latter two were involved in the original work on UNIX and C—as an improved C (see http://golang.org/) [10]. In Fall 2015 we started using Go for a set of programming assignments in the Linux programming course. We explored and studied this approach—the use of Go in the Linux course—over the course of the next seven consecutive offerings of the course (Fall 2015–Spring 2019). We posit that using Go, in strategic conjunction with C, not only achieves the student learning outcomes given above, but also ameliorates the problematic issues associated with the traditional, exclusive C approach. We present our multi-semester experience using this hybrid C/Go approach, including desirable consequences of its use (§ 2) and student feedback (§ 3).

2 ADVANTAGES TO THE USE OF GO

Using this approach for seven consecutive offerings of the course, we have experienced the following desirable consequences of it:

- **Reinforces the idea that Linux system calls are provided by the Linux kernel, not C.** Thus, system calls to the kernel can be called from any programming language, not just C.
- **Contrasts Linux pipes vis-à-vis Go channels.** This approach fosters a salient contrast between concurrent programming with one process and multiple threads of control (see Fig. 1—left) and multiple processes each with one thread of control (see Fig. 1—right) as employed in the Linux filter style of programming. This reinforces the idea that the concept of a Linux pipe is an extension of a Go channel applied to multiple processes, and a precursor to a network socket across multiple computers.
- **No compromise in purity necessary.** This approach involves the use of Go in conjunction with, not in replacement of, C. Thus, students continue to gain experience with the traditional systems programming language.
- **There are benefits to learning multiple languages**, including increased versatility [2], adaptability to change [9], and improved alignment with industrial requirements for employee professional skills and knowledge. Moreover, an environment where student experience concepts through multiple languages inhibits student (over-)association of a concept with a particular language. This approach facilitates student abstraction of the particular details of each individual language from the concept they are experiencing through that language. Lastly, and perhaps most importantly, it is widely believed that one's capacity to express ideas about computation is limited by the programming language through which one describes that computation—the analog of the *Sapir-Whorf Hypothesis* in the context of programming languages [8].
- **Supports ABET Accreditation.** The *Accreditation Board for Engineering and Technology* (ABET), in its program criteria necessary for accreditation, explicitly lists learning multiple programming languages as a requirement necessary for program accreditation (see http://www.abet.org/accreditation/accreditation-criteria/criteria-for-accrediting-computing-programs-2016-2017/). Introducing Go into a Linux programming course is a way to meet this requirement.

Other approaches. There are multiple approaches to a course on Linux programming. One standard approach involves student modifications to the Linux kernel [3]. On the other hand, the use

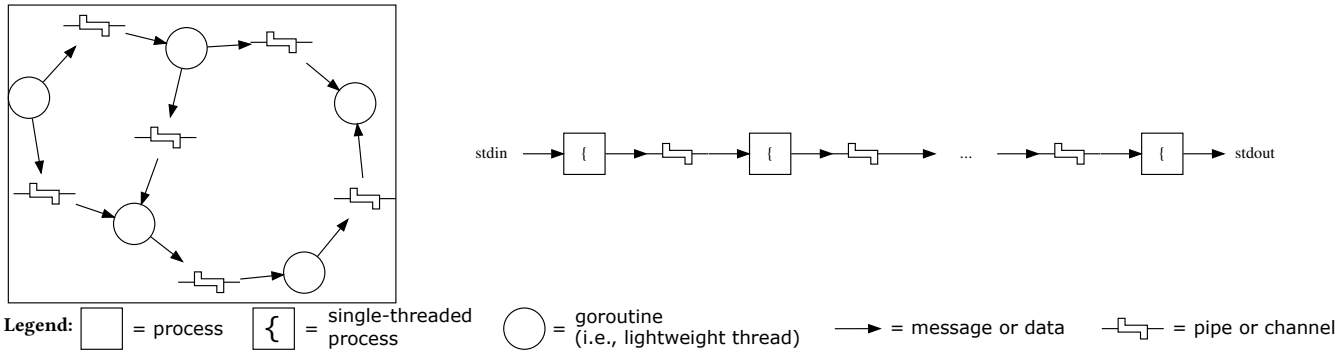

Figure 1: Conceptual differences between the CSP model of concurrent programming/problem solving (depicted left) and the Linux model of concurrent programming/problem solving (depicted right). (left) re-compile threads vs. (right) re-configure processes. © Saverio Perugini.

of interactive learning and assessment tools such as *uAsssign* [1] is effective. Lastly, the incorporation of cybersecurity and/or mobile devices/OS into the course is increasing [5].

3 EXPLORATORY STUDY

To explore how the use of Go in this manner works in practice we collected data in a longitudinal study over the course of seven consecutive offerings of the course (Fall 2015–Spring 2019).

3.1 Demographics of Participants

The *Linux Programming* course at the University of Dayton is a cross-listed undergraduate and graduate course taken primarily by senior-level computer science and computer engineering majors, and is a required course for all computer engineering students. The survey data collected (in all semesters save for Fall 2017 and Fall 2018) is from 70 students (i.e., 62 senior-level undergraduate students and 8 graduate students; 48 computer-engineering students and 22 computer science students). Individual n's for student participants from which the survey data for each semester was collected are given in Tables 1–3. The webpage for the course—the central location for all items course related—is available at http://perugini.cps.udayton.edu/teaching/courses/Spring2019/cps444/.

3.2 Programming Assignments Studied

Four (of approximately ten) programming assignments in this course provide scope from which to observe the use of Go for the benefits outlined above. The assignments are i) a modified version of the Linux `diff` file comparison utility; ii) the `env` command used in shell scripts (e.g., `#!/usr/bin/env ksh`); iii) 'library/toolchain': developing a message logging library and application; and iv) building a synchronization barrier as a client/server application with named pipes as the interprocess communication mechanism.

The `env`, library/toolchain, and client/server barrier assignments are programming exercises from [7], albeit to be programmed in Go here, with other minor modifications. The `env`, library/toolchain, and client/server barrier assignments involve the use of Linux system calls and the system call interface in Go (i.e., `package syscall`) and give students experience with spawning and executing processes. For a detailed description of both the anatomy of the course and the homeworks assigned therein, see [6]. Most students had some programming experience in Go from their OS course, but that was limited to concurrent programming using the *Communicating Sequential Processes* (CSP) model of concurrency used in Go, and did not involve any type of programmer interaction with the Linux kernel, the Go toolchain, or packages.

3.3 Methods

After the completion of each course, we solicited feedback from student participants through questionnaires to ascertain the effect of the use of Go on self-reported student learning and preference. Questionnaires solicited both quantitative, ordinal ratings (on a scale of 1–5) and follow-up, free-form responses explaining some of those ratings.

Since rating data is typically not normally distributed, median most likely represents the center of the distribution of the data. For these reasons, we use *non-parametric* tests, which do not rely on any underlying assumptions about the probability distribution of the sample, can handle ordinal data, and are not seriously affected by outliers. The non-parametric statistical test we used is the χ^2 test.

3.4 Quantitative Results: Student Ratings

The independent variables are the use of Go and C. We explored two dependent variables: *improved learning* (in the Spring 2018 and Spring 2019 offerings) and *preference* (in the Fall 2015, Fall 2016, and Spring 2017 offerings).

Improved learning. The results in Table 1 (top) reveal that both C and Go had an approximately similar effect on students' self-reported improvement in learning in cumulative frequency, with Go experiencing a slightly larger frequency. Table 1 (bottom) reveals that of the four combinations of languages for learning and programming all, save for 'learning in Go, homeworks in C,' are approximately similar in effect on students' self-reported improvement in learning in cumulative frequency, with 'learning in C, homeworks in Go' experiencing a slightly larger frequency. Aggregating languages across learning and programming reinforces this result (see the bottom-most row of Table 1).

Table 1: Results of χ^2 tests regarding *improved learning* from Spring 2018 and Spring 2019. Legend: number* = 'statistically significant result ($p < 0.05$).'

Semester	Question	n	Language	Freq.	χ^2	p
Spring 2018	Which of the follow two languages improved your learning more?	13	C Go	7 6	0.08	0.78
Spring 2019		12	C Go	2 10	5.33	0.02*
Cumulative		25	C Go	9 16	1.96	0.16

Semester	Question	n	Learning/Homework Language	Freq.	χ^2	p
Spring 2018	This semester, class lectures primarily focused on C, but implementation in homeworks was often in Go, especially since systems calls are an aspect of the Linux OS and not specific to a particular programming language (i.e., system calls can be called from any language). Which of the following did improve or would have improved your learning more?	13	Learning in C, homeworks in C. Learning in C, homeworks in Go. Learning in Go, homeworks in C. Learning in Go, homeworks in Go.	6 3 1 3	3.92	0.27
Spring 2019		12	Learning in C, homeworks in C. Learning in C, homeworks in Go. Learning in Go, homeworks in C. Learning in Go, homeworks in Go.	2 7 0 3	8.67	0.03*
Cumulative		25	Learning in C, homeworks in C. Learning in C, homeworks in Go. Learning in Go, homeworks in C. Learning in Go, homeworks in Go.	8 10 1 6	7.16	0.07
Cumulative (aggregate)			Learning in C Learning in Go Homeworks in C. Homeworks in Go.	18 7 9 16		

Preference. The results in Table 2 (top) reveal that there is no strong preference for either C or Go as used to learn in class (i.e., 13 to 13). However, the data reveals that students prefer to program in Go for homeworks (13 to 5). Table 2 (bottom) reveals that of the four combinations of languages for learning and programming, again, all, save for 'learning in Go, homeworks in C,' are approximately similar in effect on students' self-reported preference in cumulative frequency, with 'learning in C, homeworks in C' experiencing a slightly larger frequency.

3.5 Qualitative Results: Student Comments

In our post-course questionnaires, we also solicited free-form, open-ended explanations from students of their responses to each of the questions in Tables 1–3. The following are a sampling of those qualitative responses supporting the effectiveness of the approach as well as those identifying issues associated with it:

Supportive Comments. "Go was always light[-]years easier to implement the concepts in. It may have been helpful to learn in Go or at least have access to more Go examples dealing with the topic but I also think it's important to be exposed to different solutions in different languages."

"Giving examples in C allowed the students to have a decent idea of what the software will look like without giving them the code. It was tough but fair."

"Go is very simple once you get past the learning curve and C is never really that simple. Go has really good documentation. C has a lot of examples."

"C is very basic and useful and I learned it before. It is easy to understand C. Then I can practice with Go."

"Go is similar to C but Go is easier than C to learn and can help [one] understand C."

"Learning another language to put on a resume and now I have a more diverse career path."

"I felt that by doing projects in Go, it was a lot easier to understand the information rather than trying to work our way around the tricky C implementation."

"I think Go is a much more developer[-]friendly language and for the things we developed in class, Go would be the obvious choice for most programmers. However, I believe understanding the lower level on which C operates makes you a better programmer."

Identified Issues. "Something that I struggled with this semester was taking what I learned in C and applying it to Go. I think I got a lot better as the semester went on, but it wasn't a skill I picked up on within the first few weeks. It took me most of my time in the class to start getting good at it."

"Learning a concept in both languages made it harder for me to understand the concepts."

Table 2: Results of χ^2 tests regarding *preference* from Fall 2015, Fall 2016, and Spring 2017.

Semester	Question	n	Language	Freq.	χ^2	p
Fall 2016	Did you prefer studying/learning C or Go?	20	C Go	10 10	0.00	1.00
Spring 2017		6	C Go	3 3	0.00	1.00
Cumulative		26	C Go	13 13	0.00	1.00
Fall 2015	Would you have preferred to complete homeworks 1-4 (i.e., `diff`, `env`, `lib`, client/server) in C or Go?	18	C Go	5 13	3.56	0.06

Semester	Question	n	Learning/Homework Language	Freq.	χ^2	p
Fall 2016	This semester class lectures primarily focused on C, but implementation on homeworks was often in Go. Which of the following did/would you have preferred?	20	Learning in C, homeworks in C. Learning in C, homeworks in Go. Learning in Go, homeworks in C. Learning in Go, homeworks in Go.	8 7 0 5	7.60	0.06
Spring 2017		6	Learning in C, homeworks in C. Learning in C, homeworks in Go. Learning in Go, homeworks in C. Learning in Go, homeworks in Go.	2 2 1 1	0.67	0.88
Cumulative		26	Learning in C, homeworks in C. Learning in C, homeworks in Go. Learning in Go, homeworks in C. Learning in Go, homeworks in Go.	10 9 1 6	7.54	0.06
Cumulative (aggregate)			Learning in C Learning in Go Homeworks in C. Homeworks in Go.	19 7 11 15		

Table 3: Results of χ^2 tests regarding the use of Go (top) and C (bottom) from Fall 2015, Spring 2018, and Spring 2019. Legend: number* = 'statistically significant result ($p < 0.05$).'

Semester	Question	n	Rating	Freq.	$\bar{x}$	$\tilde{x}$	χ^2	p
Fall 2015	On a scale of 1–5, please rate the use of Go in this class (5 being the highest).	18	1 2 3 4 5	0 1 6 8 3	3.72	4	12.56	0.01*
Spring 2018	On a scale of 1-5, indicate how much the use of C in lectures, on homework 9 (`lex`), and on the final project (`lex` and `yacc`) improved your learning in this class (5 being the highest).	13	1 2 3 4 5	0 2 3 5 3	3.69	4	5.08	0.28
Spring 2019		12	1 2 3 4 5	0 1 1 2 8	4.42	5	17.17	0.00*

Table 4: Student homework scores from Fall 2015, Fall 2016, Spring 2017, Fall 2017, Spring 2018, Fall 2018, and Spring 2019.

Semester	Modified `diff` utility					env command					Library/toolchain					Client/server barrier				
	n	Min	$\bar{x}$	*s*	Max	*n*	Min	$\bar{x}$	*s*	Max	*n*	Min	$\bar{x}$	*s*	Max	*n*	Min	$\bar{x}$	*s*	Max
Fall 2015	19	0.00	75.61	28.81	100.00	19	58.00	80.53	12.38	100	19	25.00	91.23	19.76	100	19	0.00	80.33	29.49	100.00
Fall 2016	20	0.00	62.17	34.01	96.67	19	40.00	90.21	18.46	100	20	0.00	91.08	22.64	100	20	0.00	74.19	27.39	100.00
Spring 2017	6	0.00	51.77	44.56	95.74	6	60.00	84.33	17.91	100	6	0.00	50.00	31.62	100	6	0.00	52.08	42.88	87.50
Fall 2017	15	0.00	67.78	30.56	100.00	15	64.00	86.40	13.92	100	15	6.67	83.33	28.40	100	15	3.75	70.67	41.37	100.00
Spring 2018	13	0.00	60.38	34.58	100.00	12	0.00	59.33	41.85	100	13	0.00	71.41	41.11	100	13	0.00	59.62	49.19	100.00
Fall 2018	25	25.00	81.33	22.35	100.00	24	40.00	86.17	16.47	100	25	15.00	83.40	23.79	100	25	12.50	80.75	23.93	100.00
Spring 2019	12	0.00	74.86	32.49	96.67	12	76.00	89.50	8.14	100	12	66.67	88.89	12.30	100	12	70.00	87.08	12.10	100.00
Cumulative	110	0.00	**70.22**	31.25	100.00	107	0.00	**83.18**	21.32	100	110	0.00	**83.50**	27.07	100	110	0.00	**74.74**	32.77	100.00

"I have no issue learning a new language, that's part of being in [computer science]. But, if the objective of the class is to learn some theory that is language independent, it's easier to focus on that theory by sticking with one language. I can't focus on why I'm doing something if I can't get my something to compile because the new language has a few more funky rules I didn't know."

"I spent a lot of time on the homework just struggling with Go and learning how to do certain things in Go. I like Go better than C and I think its more useful although C is very common. You should assign homework in the same language that you teach otherwise I end up spending time understanding the other language instead of the homework itself."

3.6 Performance Results: Student Grades

Grades on the studied assignments were analyzed from approximately 110 students in all seven offering of the course involving the approach described here. The number of student submissions in each the seven course offerings studied is given in Table 4, which also provides descriptive statistics (i.e., mean and standard deviation) on the grades from the four assignments studied. These grades establish that any difficulties identified above associated with learning and programming two languages did not have a negative impact on performance (i.e., means/medians are in the B–C range).

3.7 Summary of Results

The following is a summary of these results.

- Observable from both the improved learning and preference aspects of the survey data is the result that 'learning in Go, and programming in C' is not a worthwhile option for further consideration.
- Students in the Spring 2019 offering of the course were particularly supported through the use of Go. This may be due to the particular group of students in that section or due to the fact that Spring 2019 was the seventh consecutive offering of this course involving this approach, which may imply a cumulative refinement/improvement of the approach across the semesters.
- The main take away from the quantitative survey data is that the use of Go in the course is worthwhile, for either learning in class or for programming on homeworks or both.
- Most of the issues that students conveyed through the qualitative survey data were about having to learn/use two languages rather than about the specifics of each individual language. Quantitative survey results in Table 3 reinforced this result—mean and median overall ratings of Go and C are relatively similar.

4 CONCLUSIONS

We enumerate the desiderata of this approach that we have experienced in § 2. We collected quantitative data, on both student self-reported improvement in learning and preference, and qualitative data to complement both the quantitative data and our experience of this approach, and further ascertain how the approach operates in practice. The survey data illustrates that students are not adverse to the use of Go itself, but grappled with the peculiarities of learning the syntax of two languages. However, the resulting grades on programming assignments indicate that students are successfully able to learn in one language (in this case, C) and programming in another (here, Go). That combination is an important result because a strong dislike for Go could have an effect on student attitude toward learning the concepts and, thus, inhibit the benefits of this approach given in § 2. Instructors can explore a variety of options to support students who find the use of two languages particularly challenging, including offering Go help sessions outside of class, providing access to helpful resources (e.g., quick references sheets), or the use of active learning.

While the data indicates that a hybrid C/Go approach is no worse than an exclusive C approach, the incorporation of Go brings the advantages given in § 2. Overall, the results indicate that this approach is feasible, has benefits, and is not disliked by students and is, thus, worthy of both additional and wider consideration and study. The continuous learning goals, irrespective of a particular course, are ultimately to: i) develop and improve students' ability to generalize patterns from the examples given in class; and subsequently ii) develop their aptitude and intuition for quickly recognizing new instances of these self-learned patterns when faced with similar problems in domains/contexts in which they have little experience. Our experience with this approach indicates that the use of the Go programming language in the Linux programming course moves us closer to the realization of these two related goals. We anticipate this experience report will inspire adoption of a similar use of Go in Linux programming courses.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant Numbers 1712406 and 1712404. 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] J. Bailey and C. Zilles. 2019. uAssign: Scalable Interactive Activities for Teaching the Unix Terminal. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE)*. ACM Press, New York, NY, 70–76.

[2] K. Bouwkamp. [n. d.]. Why You Must Learn Many Coding Languages. *Course Report*, 18 February 2016. Retrieved from https://www.coursereport.com/blog/4-reasons-to-learn-multiple-programming-languages [Last accessed: 30 August 2019]. ([n. d.]).

[3] R. Hess and P. Paulson. 2010. Linux Kernel Projects for an Undergraduate Operating Systems Course. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE)*. ACM Press, New York, NY, 485–489.

[4] J. Hughes. 1989. Why Functional Programming Matters. *Comput. J.* 32, 2 (1989), 98–107. Also in: D.A. Turner (ed.), Research Topics in Functional Programming, Addison-Wesley, 1990, pp. 17–42.

[5] J.-F. Lalande, V. Viet Triem Tong, P. Graux, G. Hiet, W. Mazurczyk, H. Chaoui, and P. Berthomé. 2019. Teaching android mobile security. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE)*. ACM Press, New York, NY, 232–238.

[6] S. Perugini. 2019. Revitalizing the Linux programming course with Go. *Journal of Computing Sciences in Colleges* 35, 5 (2019), 59–67.

[7] K.A. Robbins and S. Robbins. 2003. *UNIX Systems Programming: Communication, Concurrency, and Threads* (second ed.). Prentice Hall, Upper Saddle River, NJ.

[8] R.W. Sebesta. 2015. *Concepts of Programming Languages* (eleventh ed.). Addison Wesley, Boston, MA.

[9] S. Smith. [n. d.]. The Advantages of Knowing Many Programming Languages. *Small Business: Chron.com*. Retrieved from https://smallbusiness.chron.com/advantages-knowing-many-programming-languages-27623.html [Last accessed: 30 August 2019]. ([n. d.]).

[10] M. Summerfield. 2012. *Programming in Go: Creating applications for the 21st century*. Addison Wesley, Boston, MA.