


# A New Programming Environment for Teaching Secure C Programming and Assessment

Dieter Pawelczak <sup>a</sup>

*Institute of Software Engineering, University of the Bundeswehr München, Germany*

**Keywords:** Tool based Learning, Teaching Programming, Secure Programming, Programming Environments, Automated Assessment System, Computer-supported Collaborative Learning, Computer Science Education.

**Abstract:** Learning programming is a barrier for many students enrolled in engineering degree programs. In addition, students need to develop an awareness of security aspects in programming, especially with respect to robustness and correctness. Professional integrated development environments might overwhelm students with many options and features and distract them from learning. In order to lower the burden for novice programmers, we developed the Virtual-C IDE, a programming environment designed for programming beginners, which embeds some rules of the CERT secure C coding standard, provides memory visualizations to foster the students' understanding of the memory model of C and integrates a testing framework that enables programming exercises and automated assessment. The paper shows the benefits of learning and teaching with the Virtual-C IDE, describes our experience with integrating secure coding in an introductory course and presents the students' evaluation of that course.

## 1 INTRODUCTION

Many students enrolled in our electrical engineering bachelor's degree program struggle with learning programming. Independent from our university, these difficulties are widespread and range from lack of understanding syntax to conceptual and strategic misconceptions (Qian and Lehman, 2017). In addition, new challenges arrive with our emergent technologies: learning programming requires addressing security aspects (Tabassum et al., 2018).

Although modern integrated development environments (IDE) support software developers with template-based programming and code completion (Vihavainen et al., 2014), the high number of features, menus and dialogs in those IDEs might overburden students and divert effort from the actual learning target (Dillon et al., 2012). Furthermore, modern IDEs print many notifications on errors or give advice for alternations during the editing, which might irritate novice programmers.


To lower the burden for our students we developed a programming environment called the

*Virtual-C IDE*<sup>1</sup>, which runs C programs in a virtual machine, allows to easily include secure programming aspects in our introductory course, and supports teaching by visualizing many concepts during the execution of a C program. The main requirements for developing a new IDE were:

- a single installation without any configuration,
- same behavior on different computer systems,
- easy to use for students and teachers,
- support for secure programming aspects,
- integration of exercises with direct feedback for individual learning,
- automatic assessment for lab work,
- support for collaborative learning,
- available for standard computer platforms.

Other aspects for creating our own programming environment were changes in license models of existing professional IDEs as well as platform specific behaviour or fast-moving modifications from one version to the next version of an IDE.

We have been using the Virtual-C IDE in the introductory programming course for more than 5 years now and are continuously developing it further.

<sup>a</sup>  <https://orcid.org/0000-0002-1742-2520>

<sup>1</sup> The IDE is freely available for Windows, macOS and Linux at: <https://sites.google.com/site/virtualcide/>

This paper discusses related work (Section 2), the benefits for teaching and learning (Section 3), a short evaluation of the IDE and the new course design with respect to secure C coding (Section 4) and concludes with an outlook (Section 5).

## 2 RELATED WORK

### 2.1 Educational Programming Environments

Many educational programming environments have evolved to counteract the difficulties for programming beginners. The most common are *BlueJ*, *Alice*<sup>2</sup> and *Scratch*<sup>3</sup>. *BlueJ*<sup>4</sup> allows users to write Java code by guiding the programmer with a model of the program structure; thus, focusing on the modelling aspect during learning programming. *Alice* and *Scratch* are block-based and more designed for children's education. Block-based programming is performed visually by drag and drop of specific blocks and therefore does not require any knowledge about a language syntax, which is a typical barrier for learning programming (Lahtinen et al., 2005). Block-C incorporates this idea for an introductory C course: programs can be dragged together on a block base, the resulting functions can be exported to C program code, edited and reversely translated back to blocks (Kyfonidis et al., 2017). Although this concept sounds promising, we prefer students to work with an IDE, which differs not too much from available professional IDEs (compare Vihavainen et al., 2014).

Another interesting project is *ICE*, an automated tool for teaching advanced programming with an integrated assessment system, which provides quite similar functionality compared to the Virtual-C IDE, yet with the focus on advanced topics (Gonzalez, 2017). Other introductory courses use professional IDEs configured or modified by plugins. An extensive overview is found in (Luxton-Reilly, 2018).

### 2.2 Integration of Secure Coding Rules

Since the turn of the millennium, the ACM (Association for Computing Machinery) has been calling for computer science education to be adapted to secure software development. Usually, this takes place in advanced courses such as IT security or secure software engineering (ACM, 2016). Due to the high importance of the topic (Williams et al., 2014) suggest

introducing security aspects already in introductory programming courses. These aspects mainly focus on robustness and correctness. It is important to teach students programming with security awareness from the beginning, because it is difficult to adapt bad habits or to eliminate misunderstandings later. In addition, many textbooks on programming provide little information on security or may even contain vulnerabilities (Zhu et al., 2013). In addition, more and more compilers print warnings about security issues, so students need to learn how to deal with them. ASIDE - an eclipse plugin for secure coding in Java addresses this subject by explaining such compiler warnings and showing proper solutions to fix the code (Zhu et al., 2013).

Of course, dealing with security aspects in the introductory programming course cannot replace an advanced course on IT security (Bandi et al., 2019). The security subjects have to be carefully chosen, to fit in with the scope of known concepts for the novice programmers. Although previous research reports on successful integration of secure coding into introductory courses (Williams et al., 2014), even without changing the workload of the students (Bandi et al., 2019), we propose that some security issues might even help with understanding the execution of a computer program.

## 3 BENEFITS OF THE IDE

### 3.1 Benefits for Teaching

#### 3.1.1 Usability and Scalability

The Virtual-C IDE starts with a clearly arranged set of windows for editing and debugging C programs. Each view has a zoom-in and zoom-out option and is freely locatable to easily fit to different video beamer solutions. Window arrangements can be stored for different applications or varying lecture rooms. Compiler, linker and debugger are configured for instant debugging – thus a lecturer can directly debug a correct C program without the need for a project configuration. While for many examples in introductory programming a single source file is sufficient, the Virtual-C IDE also supports projects with multiple files. We use the IDE for beginners and advanced programming courses, as well as for the compiler construction course. Therefore, the compiler generators *flex* and *bison* can be directly integrated. The IDE

<sup>2</sup> <https://www.alice.org/>

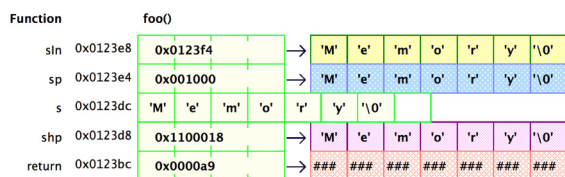
<sup>3</sup> <https://scratch.mit.edu>

<sup>4</sup> <https://bluej.org>

supports platform independent ISO C18, including C18 threads and provides a subset of the *Simple DirectMedia Layer* to work with hardware access (graphics, keyboard and mouse events). Therefore, it offers a wide variety of possibilities for course design.

### 3.1.2 Memory Visualization

Understanding different memory segment types is far more important for learning C as compared to other programming languages like e.g. Java or Python, which hide low level memory operations from the programmer or provide a garbage collector.



a) Stack visualization of function foo() at the return statement

```

5 char* foo(const char *sIn) {
6     const char *sp = "Memory";
7     char s[] = "Memory";
8     char *shp = malloc(strlen(s) + 1);
9     strcpy(shp, s);
10    return shp;
11 }
    
```

b) Corresponding code snippet

Figure 1: Examples of memory visualization.

During debugging, the memory is visualized in the IDE with a colour scheme for memory segments; (Figure 1) shows a code snippet in which a variety of different memory segments are addressed through pointers: light green/ yellow marks visible variables on the stack (e.g. s), whereas valid stack in general is shown in yellow. This applies to sIn (Figure 1) which is passed as a parameter. Invalid memory is painted in red, constant memory in blue (contents of sp) memory on the heap in purple (contents of shp) and memory in the data segment in green, which applies to global variables like e.g. stdin (not shown in Figure 1).

### 3.1.3 Secure Programming

In addition to the compiler, a static code analyser is enabled per default, which currently checks about 17 rules from the SEI CERT C Coding Standard (Cert, 2016) and is extended with every new version. As the usage of the static code analyser is optional, the lecturer can show insecure code first and then adapt it

<sup>5</sup> <https://code.google.com/p/googletest/>

to fit the requirements from (Cert, 2016) by enabling the checks, which are reported as warnings (compare Figure 2).

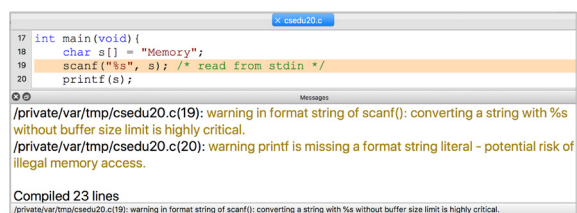


Figure 2: Security warning example for STR31-C and FIO47-C (Cert, 2016).

### 3.1.4 Simple Exercises

The Virtual-C IDE integrates a testing framework which allows lecturers to hand out simple exercises to their students. The testing framework is adapted from the *Google C++ Testing Framework*<sup>5</sup> for C and provides additional test methods to simplify writing tests. In contrast to standard software tests, randomized test data in particular can be used to prevent students from cheating in tests (compare e.g. Kratzke, 2019).

In addition to standard assertions, the testing framework supports reference tests, which compare the return value or output parameters of a function under tests with a given reference function. Simple I/O-tests stimulate the standard input and checks the standard output against simple text content or regular expressions. A unique feature of the Virtual-C IDE is, that all functions of the program under test (PUT) are linked dynamically. Thus, functions can be relinked during the test in order to inject mock functions or to test if functions are called with the right parameters. Although the use of mock functions is common for testing, mock functions are usually linked statically<sup>6</sup>. Re-linking during the test is possible without any modifications of the PUT as it runs in a virtual machine. Additionally for each test case, global variables are re-linked dynamically, and the heap is reset in order to provide a consistent test environment.

(Figure 3) shows the description of two test cases with function and I/O tests; the exemplary result of such a test run is depicted in (Figure 4).

<sup>6</sup> Compare e.g. CMocka: <https://cmocka.org>

```

1 /* a simple test with a reference procedure */
2 void refStrAppend(char* x, const char* append) {
3     strcat(x, append);
4 }
5 /* a test case with variable s as test setup */
6 _testPrologue("Append string test", { char s[128] = "Hello "; });
7
8 /* a function test with reference function */
9 _funcRefTest(strAppend, refStrAppend, 5,
10 ARG(char*, s, sizeof(s)), /* tested argument */
11 "World" );
12
13 /* same test with ARG parameter */
14 _funcTest(strAppend, 0,
15 ARG(char*, s, sizeof(s), "Hello World"), /* tested argument */
16 "World");
17
18 /* tests with strings of random size */
19 _funcRefTest(strAppend, refStrAppend, 5,
20 ARG(char*, s, sizeof(s)), /* tested first argument s */
21 ARGR(char*, 0, 120)); /* a random string */
22
23 _testEpilogue();
24 _testPrologue("I/O test");
25 _IOTest(main(), RANDS('a','z'), $rands,
26 "You're program should print the input");
27 _IOTest(main(), "Sophia\n", "Hello", "Sophia",
28 "Expected output is 'Hello Sophia'");
29 _IOTest(main(), "David\n", "Hello", !"Sophia", "David",
30 "Expected output is 'Hello David'");
31 _testEpilogue();
    
```

Figure 3: Example for two test cases with reference function and I/O tests.

The screenshot shows a test report for a file named `/private/var/tmp/csedu20.tsc`. At the top, a progress bar indicates 66% completion, with a sub-report of 'Passed: 4 / 6' and 'Failed: 2 / 6'. The report is organized into sections: 'A - Static Tests' (1 - Compiler Output, 2 - Linker Output) and 'B - Dynamic Tests' (1 - Append string test, 2 - I/O test). The I/O test section shows three sub-items: '2.1 - main(): stdin = "o"' (passed), '2.2 - main(): stdin = "Sophia"' (failed), and '2.3 - main(): stdin = "David"' (failed). A detailed view of the failed test shows the program's output 'Hallo Sophia' and the error message: 'error: Expected output is 'Hello Sophia' failure: For input: "Sophia" your output should contain: Hello'.

Figure 4: Exemplary test report for the test cases described in (Figure 3).

The ARG macro defines arguments for testing. The macro can either specify the expected result or address the result from a reference function, as for instance used (in line 10, Figure 3) with the reference function `refStrAppend()`. The ARGR macro and the RANDS macro generate random data. The !-operator in the I/O tests defines strings or regular expressions, that shall not match the output. Otherwise, students can simply print all of the expected output and pass without providing the requested functionality (compare e.g. Kratzke, 2019).

Beyond simple tests, the testing framework can access a lot of statistical information about the PUT, e.g.: loop depth, number of execution steps per test, number of function calls, etc.

The screenshot shows an exercise dialog box titled "Exercise 'Hello'". It contains the following text: "The following short exercise develops a tiny C-program, that reads your name and prints a greeting. The learning object is reading and printing strings in C. Your program's in-and output could look like this: Please enter your name: Erica Hello Erica, welcome to C-Programming." Below this, it says "The program prints out a prompt and asks the user to enter their names." Under the heading "Getting started", there are three bullet points: "Create a new C-Module.", "Add a local char-array called name with a size of 32 bytes to the main()-function.", and "Enter the required code to read in a name with a maximum of 32 characters from the console into your local char-array name." A hint follows: "Hint: Use either scanf(), scanf\_s(), or fgets() to read the name. Be careful with scanf() not to read more than 32 characters. Pay attention to fgets(): remove a line feed, that might be added." At the bottom, there is a "Check your code!" link.

Figure 5: Excerpt from an example exercise dialog.

### 3.1.5 Complex Exercises and Assessment

The Virtual-C IDE is equipped with a web interface, that allows exercises from a web server to be downloaded into the IDE. While simple test files require a separate description, the web interface enables the combination of an exercise description and the execution of automated tests in a single view. (Figure 5) shows an example, which guides the user step by step through an exercise. Almost all functions of the IDE can be automated through the web interface, so that exercises can provide assistance to the students such as the "Let me help"-link (Figure 5).

We use the web interface in our introductory course to embed the programming assignments directly into the IDE: the students log in to the assignments from within the Virtual-C IDE, solve their tasks and get feedback on their solutions. Once registered the students can access their course checklist and select their exercises for the assignments. The web server is responsible for the user management.

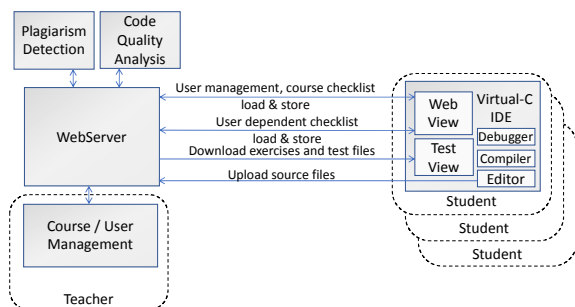


Figure 6: Automated assessments by integrating the Virtual-C IDE into a web environment.

The IDE runs tests on the students' source files and loads the sources to the server. This allows students to interrupt their work and to continue later, (compare



Figure 6). We also use the uploaded files to check for plagiarism. Advantages of an automatic plagiarism detection system are that teachers are released from manually checking the sources and that each student is treated equally. A disadvantage of such a system is that it prevents students from sharing or discussing their solutions. Details about the plagiarism detection system are found in (Pawelczak, 2018).

## 3.2 Benefits for Learning

### 3.2.1 Focus on Learning Programming

In contrast to professional IDEs, which focus on fast development and teamwork, the programming environment provides a pure and intuitive interface for learning programming. A program skeleton helps students to start writing their first programs. The IDE does not provide autocompletion or automatic error fixes and checks for errors only, when the user compiles or debugs the program in order to give the students control on their individual workflow.

On the other hand, students get everything they need by installing the Virtual-C IDE: there is no need to install compilers, plugins, or to configure projects etc. Programs run in a virtual machine; thus, all programs behave equally independent of the platform, the IDE runs on. Students therefore can concentrate on the C programming language and do not need to take platform specific concepts into account.

### 3.2.2 Individual Time Management

Students can work on exercises or tests independent of the course hours, as the automatic generated test reports provide feedback to them. We often experience that students access our exercises on the server once more before the course examinations.

### 3.2.3 Competition and Collaboration

Automated assessment systems decide on students' solutions in a rather binary way, i.e. a certain number of tests must pass before the assignment is marked as passed. Unfortunately, even clumsy solutions often pass, as the tests are primarily based on input-output-behaviour and less on performance characteristics. Although the IDE supports performance tests, such kind of tests are elaborate to develop and teachers can usually assess a student's solution much better (compare Pieterse and Liebenberg, 2017). So ideally, we should combine the automatic assessment with a teacher's review. With this combination, we experienced that students are often sufficiently content

when they pass the assignment and are not willing to accept change proposals to their programs by the teachers. Besides, we also found out, that students spend more time on their programming assignment, when they notice, that their fellow students get better results. In our case, the programming assignments report the results in percent and whereas above 80 % is sufficient to pass, some students still want to reach 100 %, because they see the automatic assessment system more as a computer game and want to receive the best results.

In order to utilize the competitive aspects on the one hand and on the other hand not to hinder weaker students, we extended our assessment system with a code quality analysis module (compare Figure 6), which is located on the server. This module runs the compiler and the static code analysis first, and then executes the students' program with a given set of test data. During the run, it monitors the execution steps for each function and checks the memory allocation on the heap. From this data collection it generates a report on the quality of the program with respect to coding style, security (static code analysis), efficiency (required execution steps for a given input, cyclomatic complexity) and memory management. In addition to the report, the metrics are stored per student on a function basis. This allows links to be included into the report to solutions of other students with a better quality (compare Figure 11 in the appendix). We intend students to use this as an exchange platform and to foster the competition among the students. To avoid a conflict with plagiarism, the students must pass the plagiarism check once before they can access the quality report.

## 4 EVALUATION

### 4.1 General Data

About 65 students enrol in the introductory course each year. In addition to the standard course evaluation, we asked our students in 2019 for their opinion about the programming environment and secure C coding. We received feedback from 45 students in total. About 80 % of the students stated that they had already gathered experience with programming and programming environments before their studies. Figure 7 gives an overview of the programming languages students encountered before enrolment to university.

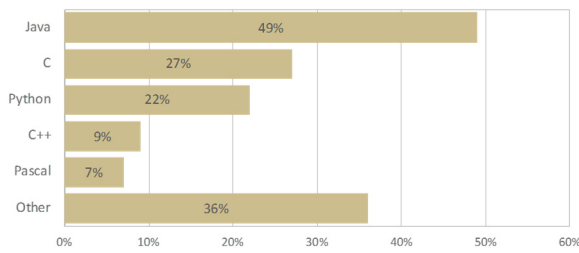


Figure 7: Knowledge of different programming language before enrolment.

Although the percentage of students with previous knowledge is higher compared to the years before, only 16 % assessed themselves as skilled programmers before the course, while 58 % had little or no previous knowledge. As the feedback from students with only little experience in programming is particularly interesting, we divided our results into two groups: 1. good previous knowledge and 2. little or no previous knowledge.

### 4.2 Evaluation of the IDE

With respect to the programming environment we asked our students to answer four questions on a 5 level Likert scale:

- A) First steps in the programming environment were generally easy.
- B) I like the automated assessment system.
- C) The test dialog helped to understand implementation errors.
- D) The plagiarism check is useful, and its usage should be extended.

The results from the students’ evaluation are shown in (Figure 8). For the first two questions, the answers did not vary much within the groups: About 87 % agreed or strongly agreed that the first steps with the programming environment were easy (Question A). All students from group 1 strongly agreed that they liked the automated assessment system. In total, 89 % agreed on that (Question B). For students with little or no previous knowledge about programming (group 2), the majority of this group agreed or strongly agreed that the test dialog was helpful (Question C). During the analysis of the students’ answers we found an ambiguity in (Question C): students might answer in a sense, that they solved their implementation errors without the help of the test dialog, that they made no implementation errors, or that the dialog was not useful for them. This might explain the big deviation for the answers from the experienced group 1.

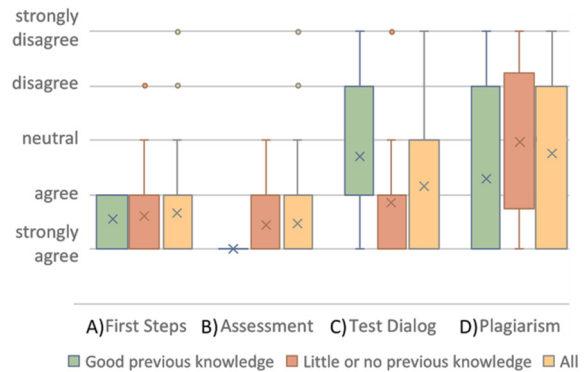


Figure 8: Results from the students’ evaluation questions A-D for the two groups and the whole class (Likert scale); the “x” represents mean values and “o” outliers.

The highest diversity of answers was found in (Question D) about the plagiarism checks. On the one hand, some students with good programming skills strongly appreciate them for the sake of justice, whilst other students from the same group feel restricted by their freedom. Over 70 % of this group disagreed on using solutions from their fellow students, while in group 2 about 54 % agreed on that. As stated before (Section 3.2.3), we encourage students to look at solutions from others as reading and understanding them helps to develop better programming skills.

### 4.3 Evaluation of Secure Coding

Students gave us feedback with respect to secure coding on the following four questions (Figure 9):

- E) Course contents regarding secure coding are very important.
- F) Examples of security vulnerabilities deepens my understanding how C works.
- G) Course content regarding secure coding complicate my understanding on C programming.
- H) Compiler warnings on security issues are more distracting.

Most students (83 %) were aware of the importance of security aspects in the C programming (Question E). Students with good programming knowledge (group 1) all agreed that examples of security vulnerabilities are helpful to explain how C programs are executed (Question F). The answer of the group 2 show a big deviation in their answer. Of course, knowledge on C and knowledge about possible vulnerabilities are closely interwoven; knowing the memory model of C allows to easily understand a buffer overrun.

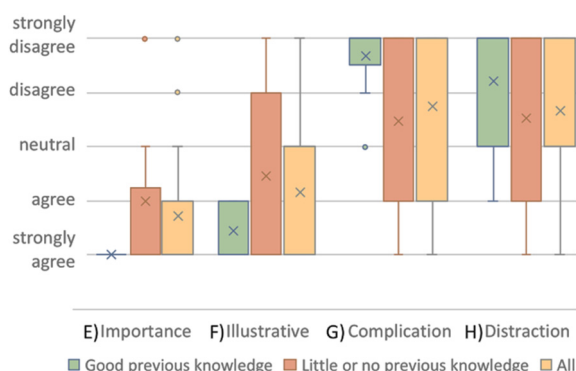


Figure 9: Results from the students' evaluation questions E-H for the two groups and the whole class (Likert scale); the "x" represents mean values and "o" outliers.

About 27 % of group 2 disagreed with our proposal that showing e.g. a buffer overrun helps to understand the C memory model. Still, 67 % of that group agreed with that. There is a big difference in answers between both groups with respect to (Question G). While almost all students of group 1 disagreed that additional course content about secure coding complicates their understanding on C programming, about 33 % of the second group agreed. Some students even state in their evaluation, that this is an add-on, they have to learn for the examination. The answers to (Question H) showed the biggest deviation in both groups. Even some students from group 1 agreed that warnings about security issues distracts them during programming. Still, over 54 % of all students disagreed on this point. Asking the students about their self-assessment before and after the course, group 2 declared the highest learning output: In average, they estimated on a 5 level scale from 1 (excellent) to 5 (insufficient) their programming skills 1.6 grades better (Figure 10).

Adding security aspects to our introductory course had no directly measurable effect on the overall examination results. Still, the results were not inferior to the preceding examinations, although additional subject matters were assessed.

## 5 CONCLUSION AND OUTLOOK

We have been using the Virtual-C IDE successfully in our introductory C programming course for over five years now. An important factor is the integration of all course activities into a single tool: teaching with live coding, exercises at home and automated assessment of programming assignments.

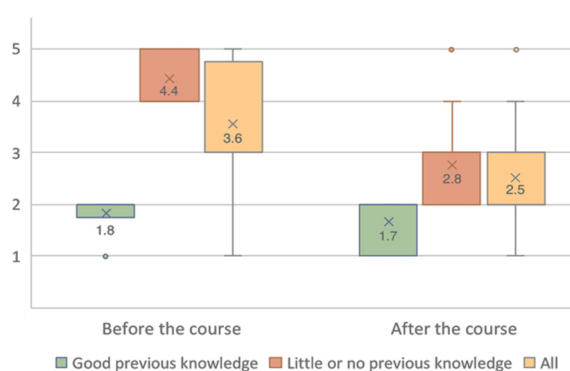


Figure 10: Self-assessment in grades from 1 (excellent) to 5 (insufficient) before and after the course; the "x" represents mean values and "o" outliers.

Including secure coding into introductory courses is challenging while it is doubtless necessary (Williams et al., 2014). The latest integration of some rules of the CERT C secure coding standard into the Virtual-C IDE supports teaching security aspects. Although a few students felt distracted by these additional warnings, the majority of the students agreed that showing vulnerabilities can illustrate the memory model of C and help consolidate their knowledge on programming. To enhance the automatic assessment system, we added a quality report for the submitted solutions that allows student to view and analyse better rated solutions from their fellow students. Our research at the *Institute of Software Engineering* is focused on teaching methods and tools that help students to better grasp programming concepts and to build better programming skills with less burden. In future we want to analyse the effects of the quality report on students' learning behaviour during the lab work, and enhance the feedback of the compiler, the test and the assessment system.

## REFERENCES

ACM, 2016. Computer Engineering Curricula 2016 – Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering. *Tech. Rep.*, Assoc. for Comp. Machinery (ACM)/ IEEE Comp. Society.

Bandi, A., Fella, A., Bondalapati, H., 2019. Embedding security concepts in introductory programming courses. *J. Comput. Sci. Coll.* 34, 4 (April 2019), 78-89.

Cert, 2016. SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems. Online available: <https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=454220> [accessed 2020-02-27]

Dillon, E., Anderson, M., Brown, M., 2012. Comparing feature assistance between programming environments

and their "effect" on novice programmers. *J. Comput. Sci. Coll.* 27, 5 (May 2012), 69-77

Gonzalez, R., 2017. ICE: An Automated Tool for Teaching Advanced C Programming. In *Proc. Int. Conf. on Educational Technologies*, Sydney, Australia, 137-144

Kratzke, N., 2019. Smart Like a Fox: How Clever Students Trick Dumb Automated Programming Assignment Assessment Systems. In *Proc. of the 11<sup>th</sup> Int. Conf. on Computer Supported Education*, SCITEPRESS

Kyfonidis, C., Moumoutzis, N., Christodoulakis, S., 2017. Block-C: A block-based programming teaching tool to facilitate introductory C programming courses. In *IEEE Global Eng. Educ. Conf. (EDUCON)*, Athens, 570-579.

Lahtinen, E., Ala-Mutka, K., Järvinen, H.-M., 2005. A study of the difficulties of novice programmers. *SIGCSE Bull.* 37, 3 (June 2005), 14-18

Luxton-Reilly, A., Simon, Albluwi, I., Becker, B. A., Giannakos, M., Kumar, A. N., Ott, L., Paterson, J., Scott, M. J., Sheard, J., Szabo, C., 2018. Introductory programming: a systematic literature review. In *Proc. Companion of the 23<sup>rd</sup> ACM Conf. on Innovation and Technology in Comp. Science Educ.* (ITiCSE Companion), ACM, 55–106

Jansen, P., 2019. The TIOBE Quality Indicator - a pragmatic way of measuring code quality. TIOBE Software BV, Netherlands.

Pawelczak, D., 2018. Benefits and drawbacks of source code plagiarism detection in engineering education. In *IEEE Global Eng. Educ. Conf. (EDUCON)*, Santa Cruz de Tenerife, 1048-1056.

Pieterse, V., Liebenberg, J., 2017. Automatic vs manual assessment of programming tasks. In *Proc. of the 17<sup>th</sup> Koli Calling Int. Conf. on Computing Educ. Research (Koli Calling '17)*, ACM, 193-194.

Qian, Y., & Lehman, J., 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.* 18 (1), 1-24

Tabassum, M., Watson, S., Chu, B., Richter-Lipford, H., 2018. Evaluating Two Methods for Integrating Secure Programming Education. In *Proc. of the 49<sup>th</sup> ACM Technical Symp. on Comp. Science Educ. (SIGCSE '18)*, ACM, 390-395.

Vihavainen, A., Helminen, J., Ihanola, P., 2014. How novices tackle their first lines of code in an IDE: analysis of programming session traces. In *Proc. of the 14<sup>th</sup> Koli Calling Int. Conf. on Computing Educ. Research (Koli Calling '14)*. ACM, 109-116.

Williams, K. A., Yuan, X., Yu, H., Bryant, K., 2014. Teaching secure coding for beginning programmers. *J. Comput. Sci. Coll.* 29, 5 (May 2015), 91–99

Zhu, J., Lipford-Richter, H., Chu, B., 2013. Interactive support for secure programming education. In *Proc. of the 44<sup>th</sup> ACM technical Symp. on Comp. Science education (SIGCSE '13)*, ACM, 687-692.

APPENDIX

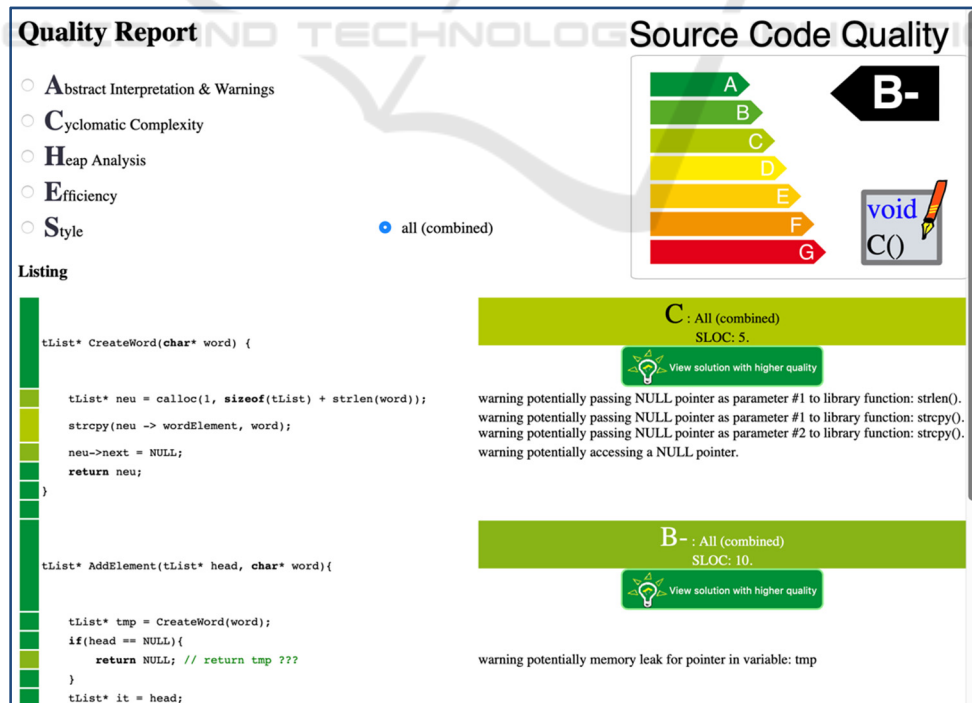


Figure 11: Excerpt from an example quality report (adapted from Jansen, 2019).