Designing Programming Assignments to Reduce the Likelihood of Cheating

Simon
University of Newcastle
Australia
simon@newcastle.edu.au

ABSTRACT
Understanding that there will always be some students who would rather cheat on their assessments than complete them with integrity, a number of authors have proposed rules of thumb for writing assignments that will reduce the incidence of cheating. Unfortunately, these rules are so general as to be of little help when it comes to actually designing an assignment, and then varying it from one course offering to the next. This paper uses a case study, a programming assignment in cryptography, to propose specific guidelines that can be applied to the design of programming assignments to reduce the chance that students will be able to copy from students in prior offerings of the course, and to reduce the chance that students will be able to copy from programs found on the web. The guidelines illustrated by the case study are to begin with something basic, to add hidden complexity, to add manifest complexity, to add levels of complexity, to vary the assignment substantially in each successive offering, and to have multiple test plans.

CCS Concepts
• **Social and professional topics~Computing education**

Keywords
Programming assignments; academic integrity; cheating; computing education; cryptography.

1. INTRODUCTION
Cheating in university assessments is widespread and growing, and it is a problem. Sheard et al [7] explain:

> *If we assume that assignment and class assessment tasks are designed by educators to give students particular learning experiences then it follows that students who cheat on these tasks miss out on valuable learning experiences, which in turn will impact on learning outcomes. Students who engage in these practices are exhibiting poor learning tendencies in their worst forms.*

For well over four decades computing educators have been coming up with ways of diminishing the incidence of cheating. There are many different ways of cheating – Dick et al [3] list 53 – but some are more prevalent than others, and not all pertain to all types of assessment item. For example, copying from another student in an exam or altering an official university document is unlikely to be particularly pertinent to the completion of a programming assignment, while collaborating with other students on work that is meant to be done individually, or copying material from the internet without referencing, will be much more applicable to this form of assessment.

2. CHEATING IN PROGRAMMING ASSIGNMENTS
A number of authors have explicitly addressed the topic of cheating in programming assignments. Dadamo [1] observes that

> *The problem with students cheating on out-of-class programming assignments is a common one . . . In the light of the tendency of faculty to reassign similar assignments and the wealth of programming examples in the literature, or from past courses, the instructor is faced with the impractical task of developing original assignments each term.*

The clear message from this observation is that some students will, where possible, source their assignments from former students in the same course, or from programs found on the web, rather than designing and writing the programs themselves. This message is supported by Sheard et al [7]:

> *If students are given tasks for which solutions are readily available to copy from textbooks or lecture notes they are tempted to take short cuts and avoid the intended learning experience.*

In these circumstances, it is the responsibility of instructors to set assignments that they have not set before, at least recently, and for which solutions are unlikely to be found on the web. I know an instructor who, in an introductory programming course, asked students to write a program to play noughts and crosses. He was surprised when many students handed in programs that they did not appear to have written themselves.

The same responsibility is discussed by other writers. Dick et al [3] suggest that cheating can be reduced by using 'quality assessment items'. Unfortunately, they do little to explain this, except in saying that their survey respondents change assignments from term to term:

> *If assignments remain the same from term to term, students (who may have very little free time in their busy schedules) will be tempted to cheat by copying another student's work, since the probability of being caught will be lower (the work was from a different term). By changing the assignments the instructor can not only reduce cheating but can also develop better assignments from term to term. This does require more effort for the instructor, as new assignments need to be developed each time a course is taught.*

The message from these authors is that instructors should set quality assignments, with little guidance as to how to do that; and that they should not use the same assignment from term to term, with an acknowledgment that this requires substantial time and effort.

Compounding the problem, most instructors like to set authentic assignments, tasks that might actually be encountered in the real world; but so many of these assignments have already been set and solved numerous times at other institutions, and some of those solutions have undoubtedly made their way into the public domain.

Many students do not see it as cheating to copy substantially from code that they find in the public domain [10]. I am familiar with instructors who, when students can't work out how to achieve some programming goal, tell them to look it up on the web. 'Coding by Google' is becoming increasingly common, among practising programmers as well as students. In a sense, though, it doesn't matter whether the students think of it as cheating. So long as the goal is to see what students can design and write from scratch, rather than how they can modify and adapt existing code, that goal is best served by setting assignment tasks for which solutions are unlikely to exist on the web.

The purpose of this paper is to add to the understanding of what constitutes a quality assignment, in the sense of a plausible task, which can be varied substantially from one term to the next, and for which it is unlikely that there are existing solutions freely available at the touch of a search button. By way of illustration, the proposed guidelines will be explained in the context of a particular programming assignment.

3. THE CONTEXT: CRYPTOGRAPHY

The assignment used to illustrate the guidelines is in the area of classical cryptography: cryptography as it was applied to written text long before the advent of computing.

Typical university cryptography courses [4, 11] begin with classical text cryptography before moving on to contemporary computational cryptography – although at least one [6] appears to go direct to the computational material. Classical cryptography is a fine source of material for text-processing programming assignments. At the simplest level, the Caesar shift simply replaces each letter with the letter that is a specified number further on in the alphabet, cycling back to the start of the alphabet as necessary. For example, *nunquam ubi sub ubi* with a Caesar shift of 1 would become *ovorvbn vcj tvc vcj*. If the key (the shift) is known, the message can be decrypted simply by applying the corresponding negative shift. If the key is not known, decryption is achieved by a simple brute force method: apply all 25 possible shifts in turn to the ciphertext, and see which one results in recognisable text.

The key for the Caesar cipher can be described either as a number or as a letter. The former is the number of places by which letters are to be shifted; the latter is the first letter of the shifted alphabet. A shift of 4 can be easily implemented by writing the alphabet beginning with E under the normal alphabet, then simply looking up each letter of the message in the cipher alphabet.

Caesar cipher with a shift of 4 (or E)	
Plain alphabet:	`abcdefghijklmnopqrstuvwxyz`
Cipher alphabet E:	`efghijklmnopqrstuvwxyzabcd`
Sample plaintext:	`nunquam ubi sub ubi`
Ciphertext:	`ryruyeq yfm wyf yfm`

The Vigenère cipher is like a series of different Caesar shifts, controlled by a keyword. With a keyword of *notepad*, for example, the first letter of the plaintext will be shifted by 13 (the shift that replaces A with N), the second by 14 (A-O), the third by 19 (A-T), and so on. The keyword is applied cyclically, so the shift of 3 (A-D) is followed by further shifts of 13, 14, 19, etc.

Vigenère cipher with a key of *dog*	
Plain alphabet:	`abcdefghijklmnopqrstuvwxyz`
Cipher alphabet d:	`defghijklmnopqrstuvwxyzabc`
Cipher alphabet o:	`opqrstuvwxyzabcdefghijklmn`
Cipher alphabet g:	`ghijklmnopqrstuvwxyzabcdef`
Sample plaintext:	`message for encryption`
Keyword repeated:	`dogdogd ogd ogdogdogdo`
Ciphertext:	`psyvomh tuu stffeshorb`

The Vigenère cipher was famously cracked by Charles Babbage. The method is somewhat laborious, but because of Babbage's high place in computing history it is not uncommonly set as an assignment in cryptography courses. Therefore, of course, it is not too difficult to find programs written by others that apply Babbage's method to text encrypted with the Vigenère cipher.

The book cipher is somewhat similar to the Vigenère cipher, but relies on an agreed passage of a specified text rather than a repeating keyword. The key passage is longer than the message to be encrypted, so there is no need to return to the start of the key.

Unlike the three methods mentioned so far, a substitution cipher works not by shifting letters but by simple replacement of letters in the known alphabet with the corresponding letters in a specific jumbled alphabet. An arbitrarily jumbled alphabet is not at all easy for sender and receiver to remember, but an agreed key phrase can be used to generate a suitably jumbled alphabet. The key alphabet is formed by taking each letter of the key phrase in the order of its first occurrence, then the remaining letters of the alphabet in alphabetic order. So, for example, the key phrase *zoological gardens* would generate the key alphabet *zolgicardensbfhjkmpqtuvwxy*, and the message would be enciphered by replacing every A with Z, every B with O, every C with L, and so on.

Substitution cipher with a key of *zoological gardens*	
Plain alphabet:	`abcdefghijklmnopqrstuvwxyz`
Substitution a'bet:	`zolgicardensbfhjkmpqtuvwxy`
Sample plaintext:	`android or iphone`
Ciphertext:	`zfgmhdg hm djrhfi`

These and many more classical ciphers give rise to a wealth of programming tasks. Generally speaking, encryption of a plaintext message is relatively simple; decryption of a ciphertext is equally simple if the key is known; and, except for the Caesar cipher, decryption when the key is not known is orders of magnitude more demanding.

The assignment used to illustrate this paper comes from the area of classical cryptography, but not from a cryptography course. The course, called Information Technology Applications, gives a brief overview of three very different applications of computing, to try to help students appreciate the breadth of uses to which programming can be applied. In recent offerings the course has covered computer vision, cryptography, and computational modelling. The cryptography section of the course deals with classical cryptography in the first week, the mechanisation of cryptography in the second, and computational cryptography in

the third. This compressed schedule means that there is little scope for a major project. The cryptography assignment is effectively restricted to classical cryptography – yet, according to the principles espoused in section 2, it must change from term to term, and must be written in some confidence that students will not find existing packaged solutions. The contribution of this paper is an explication of the approach that was taken to achieving this goal.

4. TECHNIQUES TO MAKE AN ASSIGNMENT LESS STANDARD

Within the constraints of the course under discussion, there are limitations on what can be asked of students in a programming assignment in the cryptography component. It might be interesting to ask students to write a program to crack a cipher; but this tends to require a great deal of analysis, probably more than is appropriate for an assignment associated with just three weeks of classes. Furthermore, the proper use of such a program would be highly interactive: it could take the students many hours to demonstrate that it works, and the marker would probably not be able to assess its full capability without the students present.

At the other extreme, one might consider asking for a program that encrypts or decrypts text with a specified key. This is perhaps too simple; indeed, the tutorial exercises in cryptography had students doing this with a spreadsheet rather than a program.

A compromise, perhaps more in line with the time spent on the topic in classes, would be a program that decrypts a ciphertext, but without having to work out the key from scratch. This is the assignment that will be discussed in the remainder of this section.

It must be emphasised that while the case study presented here is of a highly specific programming assignment in cryptography, the intention is to present general principles or guidelines, some or all of which can be applied to any programming assignment of reasonable scope.

4.1 Start with something basic

Beginning with perhaps the simplest of classic ciphers, the Caesar cipher, consider asking students to decrypt a Caesar-shifted ciphertext with an unspecified key. This is an easy task: they decrypt it in turn with each of the 25 possible keys, stopping when they find the one that works, the one that produces recognisable text. Ciphertexts are normally rendered without correct spaces or punctuation, so they will have to recognise, say, *eatdr inkan dbefa tandd runk* (enciphered text is often written in five-letter blocks), or perhaps *eatdrinkandbefatanddrunk*, as *Eat, drink, and be fat and drunk*; but with a little effort most students should be able to do this. However, those who do not wish to write the program, or cannot write it, can easily find programs on the web that will do the same thing.

4.2 Add hidden complexity

Text cryptography works with letters. With a Caesar shift, A might shift to E, B to F, and so on. Other characters are simply not considered. Therefore the input and output streams are just letters, as in the example above. When inspecting deciphered text to see if it looks right, the lack of spacing and punctuation can actually make it harder for some people to recognise the correctly deciphered text. So we make it easier for the students by writing the text in mixed case and leaving in all punctuation and spacing. Rather than *iexhvmroerhfijexerhhvyro* or *iexhv mroer hfije xerhh vyro*, the ciphertext will be *Iex, hvmro, erh fi jex erh hvyro*. No interpretation will be required once the correct key has been applied; the plaintext will appear as *Eat, drink, and be fat and drunk.*

In fact, though, this makes the programming task significantly more difficult. While the inspection is easier, the program now has to preserve all non-letters, to shift the upper-case letters to upper-case equivalents, and to shift the lower-case letters to lower-case equivalents. While there will be many programs available on the web to do the basic decryption, there will be fewer that can deal with this extra feature.

4.3 Add manifest complexity

With the 'simplification' suggested above, students will have no trouble recognising when they have tried the correct key, because the plaintext message will be displayed in plain English.

Of course this will change if not all of the plaintexts are in English. Students who will confidently recognise a passage in English might not be so confident when the plaintext is in French, or German, or some language less widely known. As soon as students are told that some of the plaintexts will be in other languages, the idea of trying every key and expecting to recognise the correct output becomes less appealing.

There is a further complexity to this decision, one that many students do not at first appreciate. Different languages have different alphabets. So long as we are using the English alphabet, we can shift letters up and down the alphabet using their ASCII codes. For other alphabets, this is not possible. For example, in a language with no J, such as Italian, H with a shift of 5 becomes N, whereas in English, H with a shift of 5 becomes M. So the alphabet for each language has to be provided for the students, and their programs have to manage the shifts making explicit use of that alphabet.

If students were encrypting their own texts and then decrypting them, this problem would not be apparent; but when they are decrypting provided ciphertexts, it will. A ciphertext produced using one alphabet cannot be properly deciphered using another.

A further complexity associated with this choice is that some alphabets have letters not found in English, such as the å, æ, and ø found in Danish. Students are at first tempted just to ignore these characters; but they are part of the 29-letter alphabet used to perform the encryption, and they must therefore be used when doing the decryption.

A related complexity is that the plaintext might include letters from the English alphabet that are not members of the alphabet in question. A passage in Italian might include J in a word borrowed from another language. If so, the J must be treated just as the spaces and punctuation marks: transmitted unchanged. If students are using the provided alphabet as the basis for their decryption, this will happen automatically; but if they are using the English alphabet it will not, and they will not be able to successfully decrypt the ciphertext.

When setting this assignment we provide ciphertexts and alphabets for half a dozen languages, such as perhaps English, French, German, Danish, Italian, and Malay, and tell students that their programs should be able to successfully decipher all of them.

While there might be programs on the web that decipher encrypted text, it is rather less likely that there will be programs that do so on the basis of a provided alphabet, overlooking any characters not in that alphabet.

4.4 Add levels of complexity

To encourage incremental development, the assignment specification tells students that most solution approaches can be placed into one of four categories. The simplest, with which they will undoubtedly begin, is solution by inspection: they decrypt the ciphertext with each key in turn, inspecting the output to decide which was the correct key. Students are told that this method will clearly work for texts in English and other languages with which they are familiar, and will quite possibly work for additional languages if they are willing to make an informed guess.

Students are also told that they will be asked to demonstrate their programs on three files that they have not seen before. The first will be in English, and they should be able to decrypt it by inspection. The second will be in a language that most of them will never have seen, and decrypting this by inspection will require a good understanding of the way words are formed in languages generically. (One student some years ago agonised for fully ten minutes over a Welsh text that he had decided was Flemish. He narrowed it down to two possible keys, and ended up picking the wrong one of the two.) The third file will be an encryption of a text that has already been encrypted by some other method, and that will therefore read like gibberish. With this file it will be impossible to determine the correct key by inspection except by a purely random guess.

The remaining three levels of approach involve increased automation of the task, and rely on the fact that the supplied alphabet for each language also includes the relative frequency of each letter in the language. For example, E is generally accepted to be the most frequent letter in English, making up 12.7% of letters in a large enough passage of representative text; T is next, at 9.1%; and Z is the least frequent, at less than 0.1%.

A level 2 approach will decipher the ciphertext by each key in turn, and simply assign the most frequent letter in each of the deciphered passages to the letter known to be most frequent in the language. This will work surprisingly often in English, but less often in a language such as Italian, whose most frequent letters are E at 11.8%, A at 11.7%, and I at 11.3%. Even in English the approach can be easily fooled. One of the sample English passages provided for the students to practise with is an excerpt from Georges Perec's book 'La Disparition', translated into English by Gilbert Adair as 'A Void'; both the French novel and the English translation are written entirely without the letter E.

A level 3 approach forms letter frequency tables both of the known language and of each decryption in turn, and presents them to the user for comparison, either as numerical tables or as histograms. In examining the tables or histograms, the user is able to deal with such problems as a zero-frequency E in English or a jumbling of the most frequent vowels in Italian, because the pattern of the remaining frequencies remains convincing.

A level 4 approach automates this comparison of frequency tables using a chi-square approach or something similar. Such an approach is impressively robust. The marvellous book 'Eunioa' by Christian Bök has a chapter in which the only vowel is A, another in which the only vowel is E, and so on:

> *Troop doctors who stop blood loss from torn colons or shot torsos go to Kosovo to work pro bono for poor commonfolk, most of whom confront horrors born of long pogroms. Good doctors who go to post-op to comfort folks look for sponsors to sponsor downtrod POWs from Lvov or Brno.*

Excerpts from this book will clearly have letter frequencies dramatically different from the accepted English distributions; but a chi-square analysis of different decryptions invariably finds the correct one.

Students who wisely choose to apply some form of automated recognition can of course find chi-square programs on the web; indeed, they are encouraged to do so. The goal of the exercise is not to prevent them using any resources that they can find: it is to ensure that whatever they might find, they will still have to do substantial programming of their own in order to produce something that meets the specifications.

4.5 Vary the assignment between offerings

A great deal of effort can go into devising a good programming assignment, and academics are understandably reluctant to give up a good assignment after a single use. But repeating an assignment is a sure way to encourage cheating – not, now, from the web, but from students who have already completed the course. One obvious solution is to vary the assignment in such a way that it retains most of its positive features, but looks substantially different to the students. In cryptography this can be achieved by changing the method of encryption.

If the first offering of the assignment used a Caesar shift, as in the preceding descriptions, the next one might use the Vigenère cipher.

The Caesar cipher has (in English) 25 possible shifts, and the assignment relies on the idea of trying each shift in turn and determining which is correct. So for a corresponding Vigenère assignment we postulate a group of cryptographers who tend to use the same keys over and over, a set of keys that is provided to the students. Then the solution method becomes the same as the solution for the Caesar cipher: try each key in turn and check the output. The experienced programmer will see the great similarity between these two assignments, noting that only the decryption module needs rewriting. But it seems that the students immediately notice the change in encryption method, and conclude that there is nothing to gain from using a previous assignment. There has been at least one exception to this observation: I did once have a repeating student who used the previous year's Caesar assignment on Vigenère ciphertexts. Not surprisingly, it failed to decrypt any of them correctly.

There are many other classical text ciphers available: book ciphers, rail-fence ciphers, substitution ciphers, Playfair ciphers, autokey ciphers, and more. Each of them is susceptible to the same approach, of providing a set of keys, telling the students that the ciphertext has been produced using one of these keys, and asking them to determine which one.

4.6 Have multiple test plans

The bulk of the marks for the cryptography assignment are awarded in an interactive session in the lab class. Students get 15% for correctly deciphering each of the three test files, 15% for the interface, and 20% for the level of their solution (from 5% for solving by inspection to 20% for fully automated decryption). The remaining 20%, for programming style and documentation, is allocated outside the class sessions.

As the instructor goes from student to student (or, in this case, pair to pair), watching as each program is applied to the test files, it would be very easy for one pair to tell another 'the key for the second file was cyffwrdd'. This is remedied by having a different set of test files for each program to be assessed. The marker

carries a USB drive with 20 or 30 'mystery bundle' folders on it, and chooses a different mystery bundle for each student or pair.

Each mystery bundle contains versions of the three ciphertexts, the three corresponding alphabet files with letter frequencies, and the three files of known keywords. The programs are expected, for example, to decipher *ciphertextMystery2.txt* with one of the keys in *keysMystery2.txt* using the alphabet in *letfreq2.txt.*

While of course this approach involves substantial extra preparation time, the advantage is that every mystery bundle is encrypted with a different set of three keys, so there is no benefit to students in knowing which keys succeeded for their colleagues.

5. A DIFFERENT CONTEXT

The guidelines in the preceding section are presented in the context of a programming assignment in classical cryptography; however, they are not limited to that context. In this section we will briefly consider a different assignment in a different context, showing how some or all of the same principles can be applied. Ultimately, of course, it is up to individual educators to decide whether and how to apply the guidelines to their own assignments.

For the second example, consider an assignment that requires students to program a game, in this case a dice game. Hakulinen notes that "The popularity of games has led to the idea of using them in education and taking advantage of the engaging features of games" [5 p26]. Accordingly, computer-based games are used in many educational contexts, including as assessment items in programming courses.

A web search for 'popular dice games' returns surprisingly few games, even fewer of which involve complexity of the level required in a reasonable programming task. So long as one can base an assignment on a game that cannot be found by such searches, there is a reasonable chance that programs to play the game are not readily available on the web. I began many years ago with a game called *Groan* [8], which is not particularly well known. Since then I have invented a number of dice games of my own: once you know what features to include, it's not particularly demanding. The following paragraphs indicate how the guidelines of the preceding section might be applied to a dice game programming assignment.

Start with something basic. Selection of a random number in a specified range is a standard exercise in introductory programming courses. If the language in use permits simple drawing, it is no great extension to simulate the throw of a die by displaying the die face corresponding to the resulting number. In the course in which I use this assignment, the weekly exercises include both of these tasks, so they should come at little or no cost to students in the subsequent assignment.

Add manifest complexity. Games have rules, and these can be made almost arbitrarily complex. For example, I have written a game called *Six of one* in which there are six dice, with ones playing a number of important roles. Players have a cumulative score for the game, and at each turn can choose how many of the six dice to roll – all at the same time. If their roll includes a single one, they score nothing for the turn. If it includes two ones, they score nothing for the turn and lose whatever score they have accumulated in previous turns. If it includes three ones, they lose the game outright. But if it includes four or more ones, they win the game outright. If their roll includes no ones, the sum of the dice is added to the cumulative score; except that if any three of the dice show the same face value, the sum is doubled before being added to the cumulative score. This complexity is clearly evident to the students when they read the specification.

Add hidden complexity. Most dice games involve players taking turns. This appears normal to students, but many fail to overlook its intricacies. Even if the end of a turn is completely straightforward (as in *Six of one*, where a turn ends after one roll of the chosen number of dice), there is the matter of adjusting the score of the active player, making the other player active, and initiating the new turn. In other games, though, the change of turn can be far more complex. In *Groan*, for example, a turn involves repeatedly rolling a pair of dice until the turn ends. This can be when the player chooses to end the turn, or when the player's roll includes a one, or when the player's roll includes two ones – all of which have different consequences. Like *Six of one*, the game is won when a player's score reaches or exceeds a specified goal; another complexity that appears to elude many students is that this should be detected as soon as the sum of the cumulative score and the running score reaches the goal: the player should not have to do anything, such as ending the turn, to tell the program to check for the win.

Add levels of complexity. Some dice games can be played by a single player, seeking ever better scores, in which case single-player mode is generally the simplest form of the game to program. A two-player game involves more complexity, and a game in which the program plays against a human opponent is more complex still. Then there are levels of complexity in the strategy applied by the program. These levels can be presented to students at the outset, with an indication that programs achieving higher complexity levels will score more marks.

Vary the assignment substantially in each successive offering. As mentioned earlier, my first dice game assignment was the existing game of *Groan* [8]. I was preparing to use it again, in a different class on a different campus, when I discovered that one of my struggling students was being tutored by a friend who had completed the earlier course. In the few remaining days before the assignment was released to students I devised a completely new dice game. Since then I have created several more games, and can reuse the same sequence of assignments in a cycle of five years, which appears to be sufficient to ensure the disappearance of most copies of each specific game. Of course all dice games have something in common, but, despite first appearances, their overlaps are substantially less than their differences.

Have multiple test plans. This guideline does not apply for dice game assignments, as they are run interactively rather than from prepared input files.

There is never any assurance that all six guidelines can be applied to any one assignment. However, their expected effect is cumulative rather than integral, so even if only four or five of them can be applied, the assignment is still less likely to be found on the web or in the work of students who have completed the course in recent years. Indeed, in 2016 two repeating students expressed disappointment that they could reuse so little of their assignment from the previous year.

6. DISCUSSION

When students have little to lose and a great deal to gain by cheating, some of them will do so [9]. The guidelines described in section 4 are designed with the intention of reducing two specific forms of cheating: substantially copying solutions to the assignment produced in preceding terms, and substantially

copying programs found on the web. In summary, the approaches are:

- start with something basic;
- add hidden complexity;
- add manifest complexity;
- add levels of complexity;
- vary the assignment substantially in each successive offering; and
- have multiple test plans.

The cryptography assignment has never been conducted in its most basic form, and no version has been repeated less than five years after a prior offering, so it is not possible to compare academic misconduct rates before and after its introduction. However, very little academic misconduct has ever been detected in the use of this assignment.

It would be naïve to think that this means there is no cheating. As pointed out by D'Souza et al [2], there is an emerging marketplace in customised software solutions that students can use to purchase individually written solutions to assignments. Indeed, a simple web search discovers a page at *freelancer.com* showing a student putting out one version of this cryptography assignment to tender. Although there was sufficient circumstantial evidence to deduce the student's identity, the university was not convinced, and the student was never even asked to explain his outsourcing of this and a number of other assignments. The instructor was left with the minor consolation that at least the student had to pay for the solutions.

Similar arrangements are also available closer to home. Zobel [12] describes in detail a case in which a former student advertised his assignment-writing services on the university's notice-boards, and it took an immense effort to attempt to put him out of business.

While there is no dissuading students who are determined to cheat, there appears to be merit in the suggestions canvassed earlier that instructors write 'quality assignments' and that they vary their assignments each term. What we have done with this case study is illustrate an approach that can be used to achieve these goals.

A possible spin-off of the paper is that readers might become aware of classical cryptography as a rich source of text-processing assignment material. The author will be happy to share the assignment materials (letter frequency files, plaintext passages, enciphered texts, mystery bundles, etc) with members of the computing education community.

7. REFERENCES

[1] Diana T Dadamo (1990). The correlation quiz: an aid in curbing cheating in programming assignments. ACM SIGCSE Bulletin 22:2, 52-54.

[2] Daryl D'Souza, Margaret Hamilton, and Michael Harris (2007). Software development marketplaces – implications for plagiarism. Ninth Australasian Computing Education Conference (ACE2007), Ballarat, Australia, 27-33.

[3] Martin Dick, Judy Sheard, Cathy Bareiss, Janet Carter, Donald Joyce, Trevor Harding, and Cary Laxer. (2003). Addressing student cheating: definitions and solutions. ACM SIGCSE Bulletin 35:2, 172-184.

[4] Sujata Garera and Jorge Vasconcelos (2009). Challenges in teaching a graduate course in applied cryptography. ACM SIGCSE Bulletin 41:2, 103-107.

[5] Lasse Hakulinen (2015). Gameful approaches for computer science education. Doctoral dissertation, Aalto University.

[6] Dulal C Kar (2006). Teaching cryptography in an applied computing program. Journal of Computing in Small Colleges, 21:4, 119-126.

[7] Judy Sheard, Angela Carbone, and Martin Dick (2002). Determination of factors which impact on IT students' propensity to cheat. Fifth Australasian Computing Education Conference (ACE2003), Adelaide, Australia, 119-126.

[8] Simon (1983). Quality programs for the BBC Micro. Micro Press, Tunbridge Wells, UK.

[9] Simon (2005). Assessment in online courses – some questions and a novel technique. Higher Education in a changing world: Research and Development in Higher Education 28, 501-506.

[10] Simon, Beth Cook, Judy Sheard, Angela Carbone, and Chris Johnson (2013). Academic integrity: differences between computing assessments and essays. 13th International Conference on Computing Education Research (Koli Calling 2013), Koli, Finland, 23-32.

[11] Richard Spillman (2004). A software tool for teaching classical & contemporary cryptology. CCSC Northwestern Conference 2004, 114-124, Consortium for Computing Sciences in Colleges.

[12] Justin Zobel (2004). "Uni Cheats Racket": a case study in plagiarism investigation. Sixth Australasian Computing Education Conference (ACE2004), Dunedin, New Zealand, 357-365.