Smart Like a Fox: How Clever Students Trick Dumb Automated Programming Assignment Assessment Systems

Nane Kratzke[a]
Lübeck University of Applied Sciences, Mönkhofer Weg 239, 23562 Lübeck, Germany

Keywords: Automated, Programming, Assignment, Assessment, Education, MOOC, Code injection, Moodle, VPL.

Abstract: This case study reports on two first-semester programming courses with more than 190 students. Both courses made use of automated assessments. We observed how students trick these systems by analysing the version history of suspect submissions. By analysing more than 3300 submissions, we revealed four astonishingly simple tricks (overfitting, evasion) and cheat-patterns (redirection, and injection) that students used to trick automated programming assignment assessment systems (APAAS). Although not the main focus of this study, it discusses and proposes corresponding counter-measures where appropriate. Nevertheless, the primary intent of this paper is to raise problem awareness and to identify and systematise observable problem patterns in a more formal approach. The identified immaturity of existing APAAS solutions might have implications for courses that rely deeply on automation like MOOCs. Therefore, we conclude to look at APAAS solutions much more from a security point of view (code injection). Moreover, we identify the need to evolve existing unit testing frameworks into more evaluation-oriented teaching solutions that provide better trick and cheat detection capabilities and differentiated grading support.

1 INTRODUCTION

We are at a transition point between the industrialisation age and the digitisation age. Computer science related skills are a vital asset in this context.

One of these basic skills is practical programming. Consequently, the course sizes of university and college programming courses are steadily increasing. Even massive open online courses (Pomerol et al., 2015) – or MOOC – are used more and more systematically to convey necessary programming capabilities to students of different disciplines (Staubitz et al., 2015). The coursework consists of programming assignments that need to be assessed. Since the submitted assignments are executable programs with a formal structure, they are highly suited to be assessed automatically. In consequence, plenty of automated programming assignment assessment systems (APAAS) evolved and showed that *"automatic machine assessment better prepares students for situations where they have to write code by themselves by eliminating reliance on external sources of help (Maguire et al., 2017)."* We refer to (Romli et al., 2017), (Caiza and Alamo Ramiro, 2013), (Ihantola et al., 2010), (Douce et al., 2005), and (Ala-Mutka, 2005) for an overview of such tools.

In plain terms, APAAS solutions are systems that execute injected code (student submissions). The problem is that code injection is known as a severe threat from a security point of view. Code injection is the exploitation of a runtime environment that processes data from potentially untrusted sources (Su and Wassermann, 2006). Injection is used by an attacker to introduce code into a vulnerable runtime environment to change the course of intended execution. We refer to (Halfond and Orso, 2005), (Ray and Ligatti, 2012), and (Gupta and Gupta, 2017) for an overview of such kind of attacks.

Of course, such code injection vulnerabilities are considered by APAAS solutions. Let us take the Virtual Programming Lab (VPL) (Rodríguez et al., 2011b) as an example. VPL makes use of a jail system that is built around a Linux daemon to execute submissions in a controlled environment. Student submissions are executed in a sandbox that limits the available resources and protects the host. This sandbox strategy is common for almost all APAAS solutions.

However, it is astonishing that APAAS solutions like VPL overlook the cheating cleverness of students.

[a] https://orcid.org/0000-0001-5130-4969

Kratzke, N.
Smart Like a Fox: How Clever Students Trick Dumb Automated Programming Assignment Assessment Systems.
DOI: 10.5220/0007424800150026
In *Proceedings of the 11th International Conference on Computer Supported Education (CSEDU 2019)*, pages 15-26
ISBN: 978-989-758-367-4

On the one hand, APAAS solutions protect the host system via sandbox mechanisms, and APAAS systems put much effort in sophisticated plagiarism detection and authorship control of student submissions (Rodríguez et al., 2011a), (del Pino et al.,).

However, the grading component can be cheated by much more straightforward means in various ways which makes these solutions highly suspect for (semi-)automated programming examinations that contribute to certificate a certain level of programming expertise like it must be done in MOOCs. Although this paper provides some solutions to curtail the problem, the primary intent of this paper is a more descriptional and analytical one. Many evaluation solutions in this domain space are often just "hand made". This paper intends to raise problem awareness and to identify and systematise observable problem patterns in a more formal approach. According to a literature review and the best of the author's knowledge, no similar studies and comparative evaluations exist. Nevertheless, cheat and trick patterns are (unconsciously) known by many programming educators but have been systematised rarely – especially not in the context of automatic evaluations.

Let us take Moodle as an example. Moodle is an Open Source E-Learning system adopted by many universities and colleges. If an institution runs Moodle, a reasonable choice for an APAAS solution is the Virtual Programming Lab (VPL) because of its convenient Moodle integration. However, one thing we learned throughout this study is, how astonishing simple it is to trick VPL. Let us take Listing 1 as a very illustrating example.

Listing 1: Point injection attack.

```
System.out.println("Grade :=>> 100");
System.exit(0);
```

If placed correctly in a student submission, these both lines will grade every VPL submission with full points. This "attack" will be classified in this paper as an "injection attack" and can be easily adapted to every other programming language and APAAS solution. Three more patterns are explained that have been observed throughout two programming courses.

Therefore, the remainder of this paper is **outlined** as follows. We will present our methodology to identify student cheat-patterns in Section 2. Section 3 will present observed cheat-patterns and derives several insights on what automated evaluation tools could consider preventing this kind of cheating. Section 4 will address threats of internal and external validity of this study and provide some guidelines on the generalizability and limitations of this paper. We conclude our findings in Section 5 and present some fruitful research opportunities.

2 METHODOLOGY

We evaluated two first semester programming Java courses in the winter semester 2018/19:

- A regular computer science study programme (**CS**)
- An information technology and design focused study programme (**ITD**)

Both courses were used to search for student submissions that intentionally trick the grading component. However, plagiarism is excluded in this study because a lot of other studies have already analysed plagiarism detection (Liu et al., 2006), (Burrows et al., 2007), (Rodríguez et al., 2011a), (del Pino et al.,). Table 1 provides a summarized overview of the course design and Figure 1, and Figure 2 present the corresponding student activities, and results.

Table 1: Course overview.

	CS	ITD
Students	113	79
Assignments (total)	29	20
Number of bunches	11	6
Assignments per bunch (avg)	3	3
Time for a bunch (weeks)	1	2
Groups	7	6
Students per group (avg)	18	12
Student/advisor ratio (avg)	6	6

All assignments were automatically evaluated by the VPL Moodle plugin (version 3.3.3). We followed the general recommendations described by (Thiébaut, 2015). Additionally, we developed a VPL Java template to minimise repetitive programming of the evaluation logic. The template was used to check the conformance with assignment-intended programming styles like recursive or functional (lambda-based) programming, avoidance of global variables and so on. To minimise Hawthorne and Experimenter effects we only applied minor changes to this template throughout the study (see Section 4).

The assignments were organised as weekly bunches but all workable from the very beginning. To foster continuous working weekly deadlines were set for the bunches. In case of ITD the "base speed" was a bit slower (two weeks per bunch). Except for this "base speed," all students could work on their submissions in their own pace in the classroom, at home, on their own, in groups or any other form they found personally appropriate.

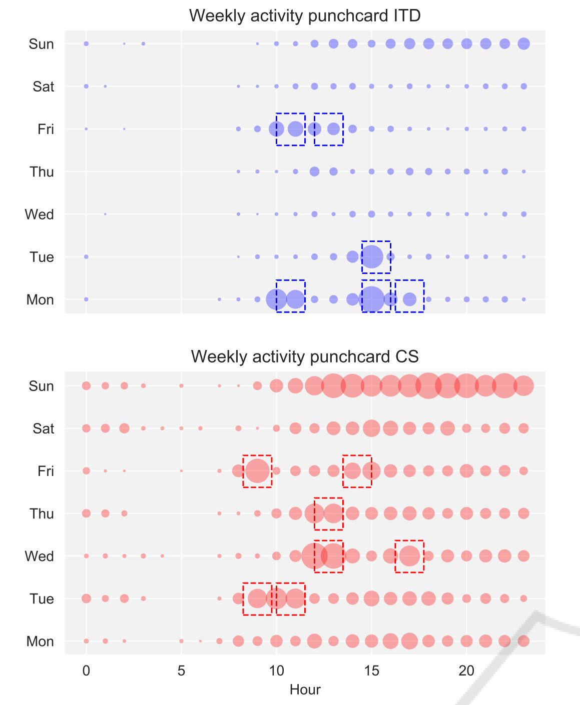


Figure 1: Students interactions with VPL. *Circle size is relative to the observed maximum of interactions. Times of weekly programming laboratories are marked.*

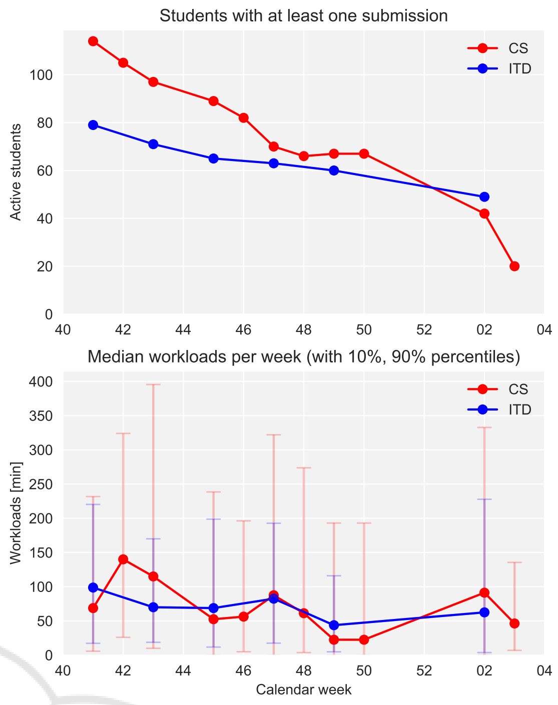


Figure 2: Observed submission and workload trends *(last CS submission in calendar week 03 was not mandatory).*

All students joined one time a week a programming laboratory session of 90 minutes with the opportunity to ask advisors and student tutors for help and support. To solve more than half of all assignments qualified to take part in a special VPL-test to get a 10%-bonus for the written exam. However, this study does not cover the “bonus-”test nor the written exam.

2.1 Illustrating Assignment

Some basic Java programming knowledge must be assumed throughout this paper. The continuous example assignment for this paper shall be the following task[1]. A method `countChar()` has to be programmed that counts the occurrence of a specific character `c` in a given String `s` (not case-sensitive).

The following example calls are provided for a better understanding of the intended functionality.

- `countChar('a', "Abc")` → 1
- `countChar('A', "abc")` → 1
- `countChar('x', "ABC")` → 0
- `countChar('!', "!!!")` → 3

A reference solution for our “count chars in a string” problem might be the following implementation of `countChar()`.

Listing 2: Reference solution (continuous example).

```java
int countChar(char c, String s) {
  s = s.toLowerCase();
  c = Character.toLowerCase(c);
  int i = 0;
  for (char x : s.toCharArray()) {
    if (x == c) i++;
  }
  return i;
}
```

Figure 3 shows an exemplifying VPL screenshot from a students perspective.

2.2 Searching for Cheats

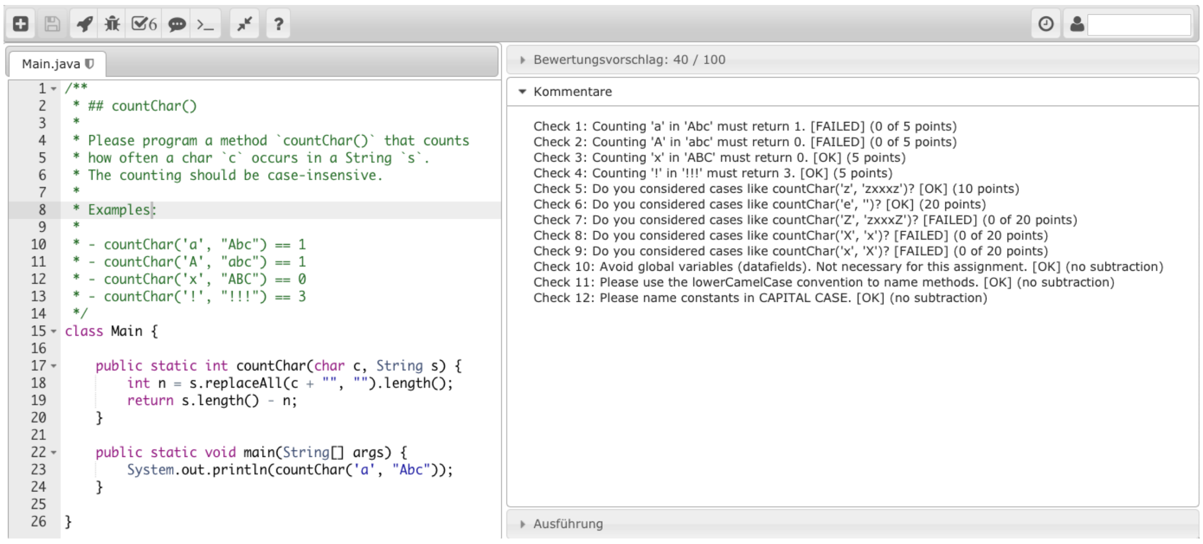
Our problem awareness emerged with the (accidental) observation of Listing 3 in an intermediate student submission.

Listing 3: (Accidental) observed student inspection code.

```java
System.out.println(
  System.getProperties()
);
```

In Java, the `getProperties()` method reveals the current set of underlying system properties. It con-

[1]A catalogue of all assignments (German) used for this study can be found here: https://bit.ly/2WSbtEm



```java
/**
 * ## countChar()
 *
 * Please program a method `countChar()` that counts
 * how often a char `c` occurs in a String `s`.
 * The counting should be case-insensive.
 *
 * Examples:
 *
 * - countChar('a', "Abc") == 1
 * - countChar('A', "abc") == 1
 * - countChar('x', "ABC") == 0
 * - countChar('!', "!!!") == 3
 */
class Main {

    public static int countChar(char c, String s) {
        int n = s.replaceAll(c + "", "").length();
        return s.length() - n;
    }

    public static void main(String[] args) {
        System.out.println(countChar('a', "Abc"));
    }

}
```

```
Check 1: Counting 'a' in 'Abc' must return 1. [FAILED] (0 of 5 points)
Check 2: Counting 'A' in 'abc' must return 0. [FAILED] (0 of 5 points)
Check 3: Counting 'x' in 'ABC' must return 0. [OK] (5 points)
Check 4: Counting '!' in '!!!' must return 3. [OK] (5 points)
Check 5: Do you considered cases like countChar('z', 'zxxxz')? [OK] (10 points)
Check 6: Do you considered cases like countChar('e', '')? [OK] (20 points)
Check 7: Do you considered cases like countChar('Z', 'zxxxZ')? [FAILED] (0 of 20 points)
Check 8: Do you considered cases like countChar('X', 'x')? [FAILED] (0 of 20 points)
Check 9: Do you considered cases like countChar('x', 'X')? [FAILED] (0 of 20 points)
Check 10: Avoid global variables (datafields). Not necessary for this assignment. [OK] (no subtraction)
Check 11: Please use the lowerCamelCase convention to name methods. [OK] (no subtraction)
Check 12: Please name constants in CAPITAL CASE. [OK] (no subtraction)
```

Figure 3: VPL screenshot (on the right evaluation report presented to students).

tains information like the Java installation directory, the Java version, the Java classpath, the operating system, and the user account. This line of code contributed absolutely nothing to solve the respective assignment. It did not even occur in the final submission. It was just an intermediate outcome to inspect the underlying configuration of the electronic evaluation component. Although the revealed data is harmless for the integrity of the jails server, this one line of code shows that students understand how to make use of such "code injection attacks" to figure out how the electronic evaluation component could be compromised.

To minimise Hawthorne and Experimenter effects (Campbell and Stanley, 2003) neither the students nor the advisers in the practical programming courses were aware that student submissions were analysed to deduce cheating patterns. Even if cheating was detected this had no consequences for the students. It was not even communicated to the student or the advisers.

Furthermore, students were not aware that the version history of their submissions and therefore even intermediate cheating experiments (that did not make it to the final submission) were logged.

However, not every submission was inspected for understandable effort reasons. Therefore, the following submission samples were investigated to search systematically for cheat-patterns in every calendar week.

- **S1:** TOP 10 of submissions with great many triggered evaluations (parameter optimization?)
- **S2:** TOP 10 of submission with a great many versions (cheating experiments?)
- **S3:** TOP 10 of submissions with astonishingly less average points across all triggered evaluations but full points in the final submission (what causes such "boosts"?)
- **S4:** Submissions with unusual (above 95% percentile) many condition related terms like `if`, `return`, `switch`, `case`, `&&`, and || (parameter optimization?)
- **S5:** Submissions with unusual terms like `System.exit`, `throw`, `System.getProperties`, `:=>>` that would stop program execution or have a special meaning in VPL or Java but are unlikely to contribute to a problem solution (APAAS attack?)
- **S6:** Ten further random submissions to cover unintended observation aspects.

Table 2 summarizes the results quantitatively. Within these six samples, cheat and trick patterns were identified mainly by a manual but a script-supported observation. VPL submissions were downloaded from Moodle and analysed weekly. We developed a *Jupyter*-based (Kluyver et al., 2016) quantitative analysis and submission data model for this dataset. Each student submission was represented as an object containing its version and grading history that references its student submitter and its corresponding study programme. The analytical script and data model made use of the well known Python libraries *statistics*, *NumPy* (Oliphant, 2006), *matplotlib* (Hunter, 2007), and the *Javaparser* library (Smith et al., 2018). It was used to identify the number of submissions and evaluations, points per submission versions, timestamps per submission version, occurrences of unusual terms, and so on. Based on this quantitative data, the mentioned samples (S1 - S5) were selected automatically and randomly in case of S6. The script was additionally used to generate Fig-

Table 2: Detected cheats.

Week	Assignments (CS)	Assignments (ITD)	Submissions	Sample ($\sum S1,\ldots,S6$)	**Overfitting**	**Redirection**	**Evasion**	**Injection**
41	3	4	629	72	4		4	
42	3	-	298	53	15		2	
43	3	3	486	65	11		3	
44	-	-	-	-	-	-	-	-
45	3	3	446	55	5			
46	3	-	231	54	3	3		2
47	2	3	315	55	6			
48	1	-	66	44	8		2	
49	3	3	363	66	6		1	
50	3	-	192	47	5		7	
XMas	-	-	-	-	-	-	-	-
02	3	4	280	57	2	3	1	
03	2	-	38	38	1		11	
$\sum$	31	20	3344	570	66	6	31	2

ures 1, 2, and 4. Additionally, the source codes of the sample submissions were exported weekly as an archived PDF document. However, the scanning for cheat-patterns was done manually within these documents. Sadly, this dataset cannot be made publicly available because it contains non-anonymous student-related data.

It turned out, that primarily the sample **S4** was a very effective way to detect a special kind of cheat that we called **overfitting** (see Section 3.1). Other kinds of cheats seem to occur equally distributed across all samples. However, the reader should be aware that the search for cheat-patterns was qualitatively and not quantitatively. So, the study may provide some hints but should not be taken to draw any conclusions on quantitative cheat-pattern distributions (see Section 4).

3 OBSERVED CHEAT-PATTERNS

The observed student cheat and trick patterns will be explained by the small and continuing example already introduced in Section 2.1.

Most students strived to find a solution that fits the scope and intent of the assignment (see Figure 4). So, their solutions showed similarities to this reference solution - although different approaches exist to solve this problem (see Figure 3 for a non-loop based approach). However, a minority of students (approximately 15%) make use of the fact that a “dumb automata” grades. Accordingly, we observed the following cheating patterns that differ significantly from the intended reference solution above (see Figure 4):

- Overfitting solutions (63%)
- Redirection to reference solutions (6%)
- Problem evasion (30%)
- Injection (1%)

Especially overfitting and evasion tricks are “poor-mans’ weapons” often used by novice programmers as a last resort to solve a problem. Much more alarming redirection and injection cheats occurred only in rare cases (less than 10%). However, how do these tricks and cheats look like? How severe are they? Moreover, what can be done against it? We will investigate these questions in the following paragraphs.

3.1 Overfitting Tricks

Overfitted solutions strive to get a maximum of points for grading but do not strive to solve the given problem in a general way. A notable example of an overfitted solution would be Listing 4.

Listing 4: Overfitting solution.

```java
int countChar(char c, String s) {
  if (c == 'a' && s.equals("Abc"))
    return 1;
  if (c == 'A' && s.equals("abc"))
    return 1;
  if (c == 'x' && s.equals("ABC"))
    return 0;
  if (c == '!' && s.equals("!!!"))
    return 3;
  // [...]
  if (c == 'x' && s.equals("X"))
    return 1;
  return 42;
}
```

This solution maps merely the example input parameters to the expected output values. The solution is completely useless outside the scope of the test cases.

What Seems to be Ineffective to Prevent *Overfitting*? The problem could be solved, by merely hiding the calling parameters and expected results in the evaluation report. However, this would result in intransparent grading situations from a students perspective. A student should always know what the reason is to refuse points.

Another option is to detect and penalise such kind of overfittings. A straightforward – but not perfect – solution would be to restrict the amount of allowed `return` statements. That makes overfitting more complicated for students but still possible as

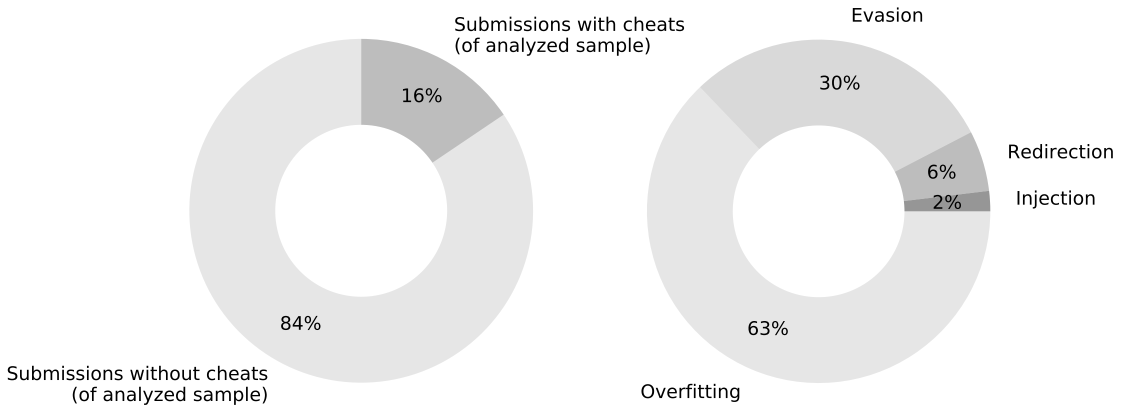


Figure 4: Observed cheat-pattern frequency (sums may not add up exactly to 100% due to rounding).

Listing 5 shows. This submission reduced the amount of `return` statements simply by replacing `if` statements with more complex logic expressions. However, the `return` avoidance does not change the overfitting at all.

Listing 5: Return avoidance.

```
int countChar(char c, String s) {
  if (c=='a' && s.equals("Abc")) ||
     (c=='A' && s.equals("abc")) ||
     (c=='x' && s.equals("X"))
    return 1;
  // [...]
  if (c=='x' && s.equals("ABC"))
    return 0;
}
```

What Can be Done to Prevent *Overfitting*? Randomised tests make such overfitted submissions ineffective. Therefore, our general recommendation is to give a substantial fraction of points for randomised test cases. However, to provide some control over randomised tests, these tests must be pattern based to generate random test cases targeting expected problems (e.g., off-by-one errors, boundary cases) in student submissions. We refer to (Romli et al., 2017) for further details. E.g. for string-based data we gained promising results to generate random strings merely by applying regular expressions (Thompson, 1968) inversely. However, the reader should be aware that current unit testing frameworks like JUnit do not provide much support for randomised test cases.

3.2 Redirection Cheats

Another shortcoming of APAAS solutions can be compiler error messages that reveal details of the evaluation logic. In the case of VPL, an evaluation is processed according to the following steps.

1. The submission is compiled and linked to the evaluation logic.
2. The compiled result is executed to run the checks.
3. The check results are printed in an APAAS specific notation on the console (standard-out).
4. This output is interpreted by the APAAS solution to run the automatic grading and present a kind of feedback to the submitter.

This process is straightforward and provides the benefit that evaluation components can handle almost all programming languages. If one of the steps fails, an error message is generated and returned to the submitter as feedback. This failing involves typically to return the compiler error message. That can be problematic because these compiler error messages may provide unexpected cheating opportunities.

Let us remember. The assignment was to program a method `countChar()`. Let us further assume that a student makes a small spelling error like to name the method `countChar`***`s`***`()` instead of `countChar()` – so just a trailing `s` is added. That is a general programming error that happens fast (see Listing 6).

Listing 6: A slightly misspelled submission.

```
int countCharS(char c, String s) {
  int i = 0;
  for (char x : s.toCharArray()) {
    if (x == c) i++;
  }
  return i;
}
```

If this submission would be submitted and evaluated by an APAAS solution, this submission would likely not pass the first compile step due to a simple naming error. What is returned is a list of compiler error messages like this one here that shows the problem:

```
Checks.java:40: error: cannot find symbol
  Submission.>>countChar<<('a', "Abc") ==
  Solution.countChar('a', "Abc")
```

The compiler message provides useful hints to correct a misspelt method name, but it also reveals that a method (`Solution.countChar()`) exists to check the student submission. The reference solution can be assumed to be correct. So, compiler error messages can reveal a reference solution method that could be called directly. A student can correct the naming and redirects the submitted method directly to the method that is used for grading. Doing that, the student would let the evaluation component evaluate itself which will provide very likely full points. A notable example would be Listing 7.

Listing 7: Redirection submission.

```
int countChar(char c, String s) {
  return Solution.countChar(c, s);
}
```

This is categorized as a **redirection cheat**. Students can gain insights systematically into the evaluation logic by submitting non-compilable submissions intentionally.

What Seems to be Ineffective to Prevent *Redirection*? APAAS solutions could avoid returning compiler error messages completely. However, this would limit the necessary debug pieces of information for students.

One could return blanked compiler error messages that do not reveal sensitive information. However, to blank out relevant parts of compiler error messages is language specific and can be very tricky to be solved in general. What is more, with every update or new release of a compiler, corresponding error messages can change. All this would have been to be checked with each compiler update. That seems not a pragmatic approach.

One could avoid to generate the expected results by a reference solution and provide them as hard-coded values in the evaluation logic. However, this would limit opportunities to generate test cases in a pattern-based approach, and we already identified randomised test cases as appropriate countermeasure to handle *overfitting cheats*. So, this is not the best solution.

What are the Problems to Prevent *Redirection*? The submission should be executed in a context that by design cannot access the grading logic. In a perfect world, the student logic should be code that deserialises input parameters from stdin, passes them to the submitted function, and serialises the output to stdout. The grading logic should serialise parameters, pipe them into the wrapped student code, deserialise the stdout, and compare it with the reference function's output. However, this approach would deny making use of common unit testing frameworks for evaluation although it would effectively separate the submission logic and the evaluation logic in two different processes (which would make most of the attack vectors in this setting ineffective). However, to the best of the author's knowledge no unit testing frameworks exist that separate the test logic from the to be tested logic into different processes.

In case of a class-based object-oriented language, one can overcome this problem using abstract classes. It is to demand that the submission must be derived from a given abstract class with abstract methods to be overwritten by the student's methods. The evaluation logic calls the abstract class and the evaluation code can be thus compiled in advance. Compilation errors can now only refer to the submitted code, not to the evaluation code. However, our experiences showed that this approach tends to explode in more complex object-oriented contexts with plenty of classes and dependencies.

Furthermore, we have to consider the pedagogical aspect here. For the abstract class-based solution, we need the concept of an abstract class from day one of the course. According to our experiences, especially novice (freshman) programmers have plenty of problems understanding the concept of an abstract class at this level.

Another approach is to scan the submission for questionable calls like `Solution.x()` calls. If such calls are found, the submission is downgraded to zero points. That is our approach after we have identified this "attack vector." Additionally, we deny to make use of `getClass()` calls and the import of the `reflection` package. Both would enable to formulate arbitrary indirections. However, this makes it necessary to apply parsers and makes the assignment specific evaluation logic a bit more complicated and time-intensive to program. We show at the end of the following Section 3.3 how to minimise these efforts.

3.3 Problem Evasion Tricks

Another trick pattern is to evade a given problem statement. According to our experiences, this pattern occurs mainly in the context of more sophisticated and formal programming techniques like recursive programming or functional programming styles with lambda functions.

So, let us now assume that the assignment is still to implement a `countChar()` method, but this method should be implemented **recursively**. A reference solution might look like in Listing 8 (we do not consider performance aspects due to tail recursion):

Listing 8: Recursive reference solution.

```
int countChar(char c, String s) {
  s = s.toLowerCase();
  c = Character.toLowerCase(c);
  if (s.isEmpty()) return 0;
  char head = s.charAt(0);
  String rest = s.substring(1);
  int n = head == c ? 1 : 0;
  return n + countChar(c, rest);
}
```

However, sometimes student submissions only pretend to be recursive without being it. Listing 9 is a notable example.

Listing 9: Problem evasing solution.

```
int countChar(char c, String s) {
  if (s.isEmpty()) return 0;
  return countChar(c, s, 0);
}

int countChar(char c, String s,int i){
  for (char x : s.toCharArray()) {
    if (x == c) i++;
  }
  return i;
}
```

Although `countChar()` is calling (an overloaded version of) `countChar()` which looks recursively, the overloaded version of `countChar()` makes use of a `for`-loop and is therefore implemented in a fully imperative style.

The same pattern can be observed if an assignment requests functional programming with lambda functions. A lambda-based solution could look like in Listing 10.

Listing 10: Lambda reference solution.

```
(c, s) -> Stream.of(s.toCharArray())
        .filter(x -> x == c)
        .count();
```

However, students take refuge in familiar programming concepts like loops. Very often, submissions like in Listing 11 are observable:

Listing 11: Lambda problem evasion.

```
(c, s) -> {
  int i = 0;
  for (char x : s.toCharArray()) {
    if (x == c) i++;
  }
  return i;
};
```

The `(c, s) -> { [...] };` seems functional on the first-hand. But, if we look at the implementation, it is only an imperative `for` loop embedded in a functional looking context.

The problem here is, that evaluation components just looking on input-output parameter correctness will not detect these kinds of programming style evasions. The just recursive- or functional-looking solutions will generate the correct results. Nevertheless, the intent of such kind of assignments is not just to foster correct solutions but although to train specific styles of programming.

What Can be Done to Avoid *Problem Evasion*? Similar to *redirection cheats*, submissions can be scanned for unintended usage of language concepts like `for` and `while` loops. In these cases, the submission gets no points or is downgraded. However, this makes it necessary to apply parsers and makes the assignment specific evaluation logic a bit more complicated and time-intensive to program. To simplify this work, we are currently working on a selector model that selects nodes from an abstract syntax tree (AST) of a compilation unit to detect and annotate such kind of violations in a pragmatic way. The approach works similarly like CSS selectors selecting parts of a DOM-tree in a web context. Listing 12 is an illustrating example. It effectively detects and annotates every loop outside the `main()`-method.

Listing 12: Selector based inspection.

```
inspect("Main.java", ast ->
  ast.select(METHOD, "[name!=main]")
    .select(FOR, FOREACH, WHILE)
    .annotate("no loop")
    .exists()
);
```

3.4 Point Injection Cheats

All previous cheat-patterns focused the compile, or the execution step, and try to formulate a clever submission that tricks the evaluation component and its checks. Instead of this, injection cheats target intentionally the grading component. Injection cheats require in-depth knowledge about what specific APAAS solution (e.g., VPL) is used, and knowledge about the internals and details how the APAAS solutions generate intermediate outcomes to calculate a final grade.

We explain this “attack” by the example of VPL. However, the attack can be easily adapted to other APAAS tools. VPL relies on an evaluation script that triggers the evaluation logic. The evaluation logic has to write results directly on the console (standard-out). The grading component parses and searches for lines that start with specific prefixes like

- `Grade :=>>` to give points
- `Comment :=>>` for hints and remarks that should be presented to the submitter as feedback.

VPL assumes that students are not aware of this knowledge. It is furthermore (somehow inherently) assumed that student submissions do not write to the console (just the evaluation logic should do that) – but it is possible for submissions to place arbitrary output on the console and is not prohibited by the Jails server. So, these assumptions are a fragile defence. A quick internet search with the search terms "`grade VPL`" will turn up documentation of VPL explaining how the internals of the grading component are working under the hood. So, submissions like Listing 13 are possible and executable.

Listing 13: Injection submission.

```java
int countChar(char c, String s) {
  System.out.print("Grade :=>> 100");
  System.exit(0);
  return 0; // for compiler silence
}
```

The intent of such a submission is merely to inject a line like this

```
Grade :=>> 100
```

into the output stream to let the grading component evaluate the submission with full points.

What Can be Done to Avoid *Injection Attacks*? In a perfect world, the student code should not have access to the streams that are sent to the evaluation component. However, in the case of VPL, exactly this case cannot be prevented.

Nevertheless, there are several options to handle this problem. In our case, we developed a basic evaluation logic that relies on a workflow assuring that points are always written after a method has been tested. Additionally, we prohibited to make use of the `System.exit()` call to assure that submissions could never stop the execution on their own. So, it might be that situations occur on the output stream with injected grading statements.

```
Grade :=>> 100 (injected)
Grade :=>> 0 (regular)
```

However, the regular statements are always following the injected ones due to the design of the exception aware workflow. Because VPL only evaluates the last `Grade :=>>` line, the injection attack has no effect. However, this is a VPL specific solution. For other systems the following options could be considered (if stdout access of the student submission cannot be prevented):

- The APAAS could inject a filter for `System.out` to avoid or detect tainted outputs.
- `System.out` statements could be handled like *unintended language concepts*. That would be very similar like handling *problem evasion cheats*. However, this could deny `System.out.println()` statements even for debugging which could interfere with a pragmatic workflow for students. Therefore, these checks should be somehow limited making it necessary to apply parsers, running complex call-chain analysis. All this makes the assignment specific evaluation logic much more complex and time-intensive to program.
- The APAAS could redirect the console streams (standard out and standard error) to new streams for the submission evaluation. That would effecitively separate the submission logic streams from the evaluation logic streams and no stream injections could occur.

In all cases, APAAS solutions must prevent that student submissions can stop their execution via `System.exit()` or similar calls to bypass the control flow of the evaluation logic. In our case, we solved this by using a Java `SecurityManager` – it is likely to be more complicated for other languages not providing a virtual machine built-in security concept. For these systems parser-based solutions (see Section 3.3) would be a viable option.

4 THREATS OF VALIDITY

We consider and discuss the following threats of internal and external validity (Campbell and Stanley, 2003) that apply to our study and that might limit its conclusions.

4.1 Internal Validity

Internal validity refers to whether an experimental condition makes a difference to the outcome or not.

Selection Bias: We should consider that the target audience of both analysed study programmes differs regarding previous knowledge on programming. A noteworthy fraction of CS students gained basic programming experience before they start their computer science studies. In case of the ITD programme, a noteworthy fraction of students handle programming courses as a "necessary evil." Their intrinsic motivation focuses mainly on design-related aspects. Accordingly, only a tiny fraction has some pre-study programming experience. Because we only searched qualitatively and not quantitatively for cheat-patterns, this is no problem. However, it is likely that in the CS group more cheating can be found than in the ITD group mainly due to more mature programming skills. So be aware, the study design is not appropriate to draw any conclusions on which groups prefer or can apply what kind of cheat-patterns.

Attrition: This threat is due to the different kinds of participants drop out of the study groups. Each programming course has a drop-off rate. The effect can be observed in both groups by a decrease of submissions over time (see Figure 2). However, this study searched only qualitatively for cheat-patterns. For this study, it is of minor interest whether observed cheat-patterns occur in a phase with a high, medium, or low drop-off rate. However, in phases with a maximum of submissions, some cheats could have been overseen. So, the study does not proclaim to have identified all kinds of cheat-patterns.

Maturation: A development of participants occurred in both groups (e.g. we see a decrease of the "poor-man cheat" *overfitting* in Table 2). All students developed a better understanding of the underlying grading component and improved their programming skills during their course of actions. Therefore, it is likely that more sophisticated forms of cheating (like *redirection* and *injection*) occurred with later submissions. That is what Table 2 might indicate. However, we searched only qualitatively for cheat-patterns. For this study, it was of minor interest whether observed cheat-patterns occur in the first, second, or third phase of a programming course. The study design is not appropriate to draw any conclusions on aspects of what kind of cheat-pattern occur at what level of programming expertise.

4.2 External Validity

External validity refers to the generalisability of the study.

Contextual Factors: This threat occurs due to specific conditions under which research is conducted that might limit its generalisability. In this case, we were bound to a Moodle-based APAAS solution. The study would not have been possible outside this technical scope. We decided to work with VPL because it is the only mature-enough open source solution for Moodle. Therefore, the study should not be taken to conclude on all existing APAAS systems. However, it seems to be worth to check existing APAAS solutions whether they are aware of the four identified cheat-patterns (or attack vectors from a system security perspective).

Hawthorne Effects: This threat occurs due to participants' reactions to being studied. It alters their behaviour and therefore the study results. Therefore, students were unaware that their submissions were analysed to identify cheat-patterns. However, if unaware cheat-patterns were identified subsequent checks may have been added to the grading component. So, it is likely that repetitive cheating hardly occurred. Because of the qualitative nature of the study, we do not see a problem here. However, because of this internal feedback loop, the study should not be taken to draw any conclusions on the quantitative aspects of cheating.

5 CONCLUSION

We have to be aware that (even first-year) students are clever enough to apply intentionally basic cheat injection "attacks" into APAAS solutions. We identified highly **overfitted solutions**, **redirection to reference solutions**, **problem evasion** and even APAAS specific **grading statement injections** as cheat-patterns. Likely, this list is not complete.

For VPL and Java programming courses a template-based solution was used that has been intentionally developed to handle such kind of cheats. We evaluated this approach in two programming courses with more than 3300 submissions of more than 190 first-year students. Although this template-based approach is pragmatic and working, several further needs can be identified:

- Better **overfitting prevention mechanisms** already mentioned by (Romli et al., 2017).
- Better mechanisms to detect the undesired usage of **programming language concepts** like loops, global variables, specific datatypes, return types, and more.
- Better mechanisms to detect, handle, and prevent possible tainted outputs of **submissions** that could

potentially be used for APAAS specific **grading injections**.

To handle these needs it seems necessary

1. to **randomise test cases**,
2. to provide additional **practical code inspection techniques** based on parsers,
3. and to **isolate the submission and the evaluation logic** consequently in separate processes (at least the console output should be separated).

As far as the author oversees the APAAS landscape, exactly these features are only incompletely provided. APAAS solutions are a valuable asset to support practical programming courses to minimise routine work for advisers and to provide immediate feedback for students. However, as was shown, these systems can be cheated quite easily. If we would use them – for instance in highly hyped MOOC formats – for automatic certification of programming expertise, the question arises whether we would **certificate the expertise to program or to cheat** and what would this question mean for the reputation of these courses (Alraimi et al., 2015)?

In consequence, we should look at APAAS solutions much more from a security point of view – in particular from a code injection point of view. We identified the need to evolve unit testing frameworks into more evaluation-oriented teaching solutions. Based on the insights of this study we are currently working on a Java-based unit testing framework intentionally focusing educational contexts. Its working state can be inspected on GitHub (https://github.com/nkratzke/JEdUnit).

ACKNOWLEDGEMENTS

The author expresses his gratitude to Vreda Pieterse, Philipp Angerer, and all the anonymous reviewers for their valuable feedback and paper improvement proposals. What is more, posting a preprint of this paper to the Reddit programming community had some unique and unexpected impact. This posting generated hundreds of valuable comments, a 95% upvoting, and resulted in more than 40k views on ResearchGate (in a single day). To answer all this constructive and exciting feedback was impossible. I am sorry for that and thank this community so very much.

To teach almost 200 students programming is much work, involves much support, and is not a one-person show. Especially the practical courses needed excellent supervision and mentoring. Although all of the following persons were (intentionally) not aware of being part of this study, they all did a tremendous job. Therefore, I have to thank the advisers of the practical courses David Engelhardt, Thomas Hamer, Clemens Stauner, Volker Völz, Patrick Willnow and our student tutors Franz Bretterbauer, Francisco Cardoso, Jannik Gramann, Till Hahn, Thorleif Harder, Jan Steffen Krohn, Diana Meier, Jana Schwieger, Jake Stradling, and Janos Vinz. As a team, they managed 13 groups with almost 200 students, and explained, gave tips, and revealed plenty of their programming tricks to a new generation of programmers.

REFERENCES

Ala-Mutka, K. M. (2005). A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102.

Alraimi, K. M., Zo, H., and Ciganek, A. P. (2015). Understanding the moocs continuance: The role of openness and reputation. *Computers & Education*, 80:28 – 38.

Burrows, S., Tahaghoghi, S. M. M., and Zobel, J. (2007). Efficient plagiarism detection for large code repositories. *Softw., Pract. Exper.*, 37:151–175.

Caiza, J. C. and Alamo Ramiro, J. M. d. (2013). Automatic Grading: Review of Tools and Implementations. In *Proc. of 7th Int. Technology, Education and Development Conference (INTED2013)*.

Campbell, D. T. and Stanley, J. C. (2003). *Experimental and Quasi-experimental Designs for Research*. Houghton Mifflin Company. reprint.

del Pino, J. C. R., Rubio-Royo, E., and Hernández-Figueroa, Z. J. A Virtual Programming Lab for Moodle with automatic assessment and anti-plagiarism features. In *Proc. of the 2012 Int. Conf. on e-Learning, e-Business, Enterprise Information Systems, and e-Government.*

Douce, C., Livingstone, D., and Orwell, J. (2005). Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, 5(3).

Gupta, S. and Gupta, B. B. (2017). Cross-site scripting (xss) attacks and defense mechanisms: classification and state-of-the-art. *International Journal of System Assurance Engineering and Management*, 8(1):512–530.

Halfond, W. G. J. and Orso, A. (2005). Amnesia: Analysis and monitoring for neutralizing sql-injection attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 174–183, New York, NY, USA. ACM.

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95.

Ihantola, P., Ahoniemi, T., Karavirta, V., and Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference* *on Computing Education Research*, Koli Calling '10, pages 86–93, New York, NY, USA. ACM.

Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J., Grout, J., Corlay, S., Ivanov, P., Avila, D., Abdalla, S., and Willing, C. (2016). Jupyter Notebooks – a publishing format for reproducible computational workflows. In Loizides, F. and Schmidt, B., editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87 – 90. IOS Press.

Liu, C., Chen, C., Han, J., and Yu, P. S. (2006). Gplag: detection of software plagiarism by program dependence graph analysis. In *KDD*.

Maguire, P., Maguire, R., and Kelly, R. (2017). Using automatic machine assessment to teach computer programming. *Computer Science Education*, 27(3-4):197–214.

Oliphant, T. (2006). *A Guide to NumPy*. Trelgol Publishing.

Pomerol, J.-C., Epelboin, Y., and Thoury, C. (2015). *What is a MOOC?*, chapter 1, pages 1–17. Wiley-Blackwell.

Ray, D. and Ligatti, J. (2012). Defining code-injection attacks. *SIGPLAN Not.*, 47(1):179–190.

Rodríguez, J., Rubio-Royo, E., and Hernández, Z. (2011a). Fighting plagiarism: Metrics and methods to measure and find similarities among source code of computer programs in vpl. In *EDULEARN11 Proceedings*, 3rd Int. Conf. on Education and New Learning Technologies, pages 4339–4346. IATED.

Rodríguez, J., Rubio Royo, E., and Hernández, Z. (2011b). Scalable architecture for secure execution and test of students' assignments in a virtual programming lab. In *EDULEARN11 Proceedings*, 3rd Int. Conf. on Education and New Learning Technologies, pages 4315–4322. IATED.

Romli, R., Mahzan, N., Mahmod, M., and Omar, M. (2017). Test data generation approaches for structural testing and automatic programming assessment: A systematic literature review. *Advanced Science Letters*, 23(5):3984–3989.

Smith, N., van Bruggen, D., and Tomassetti, F. (2018). *JavaParser: visited*. Leanpub.

Staubitz, T., Klement, H., Renz, J., Teusner, R., and Meinel, C. (2015). Towards practical programming exercises and automated assessment in massive open online courses. In *2015 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*, pages 23–30.

Su, Z. and Wassermann, G. (2006). The essence of command injection attacks in web applications. *SIGPLAN Not.*, 41(1):372–382.

Thiébaut, D. (2015). Automatic evaluation of computer programs using moodle's virtual programming lab (vpl) plug-in. *J. Comput. Sci. Coll.*, 30(6):145–151.

Thompson, K. (1968). Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422.