Musings on Misconduct: A Practitioner Reflection on the Ethical Investigation of Plagiarism within Programming Modules

Michael James Heron
Robert Gordon University
Aberdeen
Scotland
m.j.heron1@rgu.ac.uk

Pauline Belford
Dundee and Angus College
Arbroath
Scotland
pauline.belford@gmail.com

ABSTRACT

Tools for algorithmically detecting plagiarism have become very popular, but none of these tools offers an effective and reliable way to identify plagiarism within academic software development. As a result, the identification of plagiarism within programming submissions remains an issue of academic judgment. The number of submissions that come in to a large programming class can frustrate the ability to fully investigate each submission for conformance with academic norms of attribution. It is necessary for academics to investigate misconduct, but time and logistical considerations likely make it difficult, if not impossible, to ensure full coverage of all solutions. In such cases, a subset of submissions may be analyzed, and these are often the submissions that have most readily come to mind as containing suspect elements. In this paper, the authors discuss some of the issues with regards to identifying plagiarism within programming modules, and the ethical issues that these raise. The paper concludes with some personal reflections on how best to deal with the complexities so as to ensure fairer treatment for students and fairer coverage of submissions.

Categories and Subject Descriptors

K.7.4 [**Professional Ethics**]: Codes of ethics; Codes of good practice; Ethical dilemmas.

General Terms

Security; Human Factors; Legal Aspects

Keywords

Plagiarism; Programming; Teaching; Ethics; Morality; Attribution; Academic Misconduct; Education

1. INTRODUCTION

As a necessary part of evaluating student work, teaching professionals must assess its originality and conformance with institutional rules of attribution. Plagiarism is an unfortunate occurrence within student work, and thankfully as best as can be ascertained still a minority phenomenon. Automated tools such as *turnitin* [3] have allowed for much plagiarism to be automatically identified and the original sources to be located, and the provision of such tools to students for self-assessment even discourages attempts to submit problematic work in the first place [7][8].

Within software engineering, and specifically the field of programming, dealing with plagiarism is much more difficult.

Standard tools such as *turnitin* do not offer facilities for checking the originality of software solutions. While tools such as MOSS [1][5] exist as an attempt to detect similarity in code there are elements that are unique to software development that limit the utility of such automated routines. In the end, it is down to an academic to analyze the code, and usually within the tight time constraints implied by assessment boards and other formal duties. As such, not every submission will receive the same amount of critical attention. Those submissions that are most suspect will receive the greatest amount of effort with regards to investigation. Given the relatively fine-balanced mesh of issues that determine the quality and originality of programming code, the designation of a submission as suspect is often a matter of academic judgment. This presents numerous ethical issues for those who must ensure the integrity of assessments.

In this paper, the authors reflect upon these ethical issues as a professional educator. This paper does not offer a better system for dealing with potential plagiarism in software development modules – there exists, at this time, no obvious alternative to that of relying on the academic judgment of subject matter experts. However, this paper intends for the discussion to help illuminate some of the important considerations that such a state of affairs raises. The authors hope that fuller understanding of the problem helps ensure that students are given the fairest possible consideration when such incidents are investigated.

2. GOOD PRACTISE AS PLAGIARISM

In many ways, it is difficult to truly render a verdict of 'plagiarism' in software development without first invalidating many of the fundamental lessons we attempt to impart to students regarding how programming works in the real world. Some of these issues are already well understood – we work within a medium where vocabulary and syntactic construction of the simplest elements is ritualistic to the point of incantation. Programmers cannot simply extemporize or add lyrical flourishes to an argument to underscore a clever point. We must work within the constraints of the programming language's grammar. Programmers, working on the underlying bones of a program, are limited to three key forms of expression – linear, loop and selection. There are only so many ways to write a for loop or an if statement, and certain standard cultural conventions regarding names for variables and layout of code have taken deep root in both professional and educational instruction. Consider for example the names **i** and **j** as counter variables – while we may all acknowledge how ineffective these names are, as part of a common vocabulary of programming they are hard to ignore.

Not only is the structure of a program often impossible for us to meaningfully alter, but so too often are the unspoken assumptions that we absorb via osmosis through exposure to a larger, established community of practice. As part of that community of practice, we absorb understanding – first informal and then gradually formalized into *standards* – about how code should be written. We discourage experimentation with format, layout and even the name of variables because such things represent **bad practice** [10]. In this way, we sanctify certain plagiaristic practices, turning them from vice to virtue.

Taking a step back from the raw bones of individual statements and structures, a common component of programming courses tends to be some kind of formal instruction in the topic of algorithms. We teach students to understand Big O notation and explain the relative merits of bubble versus quick versus merge sorts. During these discussions, we underscore why we use algorithms. Rather than attempting to reinvent the wheel, we rely on tried and tested solutions to complicated problems because these tend to be more reliable than more original and creative solutions. If the syntax and conventions of programming limit the vocabulary of a programmer, algorithms work to constrain computational creativity.

When students gain a little more appreciation of the way that object orientation works, we may introduce them to the wider world of design patterns, explaining loftily that design patterns are to objects as algorithms are to processes. We go through the classic architectural relationships implied by the most widely used patterns, showing scenarios in which they can be used and encouraging students to consider where to apply them in their own code. 'They may not be the best solutions, but they're good solutions – battle tested solutions', we say. We grade our students abilities to both interpret design patterns in the work of others, and apply them to their own projects. In this way, we place shackles even on the way in which objects within a program are expected to communicate.

We stress the value of **reusability**, often honoured more in the breach than in the observance in the real world, and nod approvingly when reusable code is produced by our students. We favourably grade that code which tightly conforms to general design principles such as encapsulation, remarking with good grace that the submission offers good scope for reuse in other programs. We make a big point of arguing the importance of **maintainability**, stressing that most of software development is in maintenance and that good programs can not only be reused in their current form but refactored to work in other areas too. We encourage, certainly in later years, the use of external libraries to do the heavy lifting in problem domains where students cannot reasonably be expected to 'roll their own' solutions.

We may do all of these things, or some of these things, or none of these things – every academic has their own set of developer battle-scars that experience has cut into their skins, and the way in which particular messages will be emphasised will be a function of this. However, the reality of what it means to develop software, and the lessons we teach students about the development of code, is often starkly incompatible with how we **treat** code which honours the lessons that we have taught. Students often do not realise that what they are doing could be construed as plagiarism because in many ways it's just following the advice they've been given about how code should be written.

3. PLAGIARISM IN PROGRAMMING

It is here where academic judgment becomes an important, and ethically troublesome tool. It is the responsibility of an academic to assess a piece of work in its entirety and form a judgment as to the level of originality shown in a submission. Depending on how strict we wish to be about definitions, it is reasonable to argue that all programming is plagiarism to one degree or another. A University of which I am aware and which will not be named, once formalised its institution-wide plagiarism policy with a requirement that students attribute every single thing that they didn't write themselves. This was held to be the case even if it came from their own lecturer's slides and that failure to do so would be considered a breach of academic conduct.

This policy was constructed without reference to the School of Computing, who would have pointed out that this mean that every single program produced by every single student in every single module would require every single line of code to reference some standard text in programming. This purely as a consequence of the limitations of grammar imposed upon practitioners. Even firm policies regarding attribution of 'anything not in a lecturer's slides' are inconsistently applied – why for example do we need an attribution from Stack Overflow, but not one for using a whole set of Javascript tools such as jQuery? Why do we need to cite a string tokenization tool we grabbed from Unity Answers, but nobody needs us to cite an Abstract Factory? Why is it okay to use a graphical asset from the Unity Store, but not a tutorial on the Unity website? The edge cases here are many because of the need to find the right point in the spectrum between 'attribute every line' and 'attribute nothing'.

The problem is further complicated by the many ways in which plagiarism might be reflected within software code, and the degree to which software development is an incremental process. A program of ten thousand lines may have an incredible structural dependency on a handful of objects at the core of a vast class relationship. Classes may be incredibly light at the core but become much denser the more specialised they become. A piece of code may be complex in its functionality but marginal in its effect, and vice versa. Unlike in an essay where each word should plant a step in one ongoing journey (ideally), a computer program is more like the schematic for a complicated machine through which information will flow in unpredictable ways.

Plagiarism then might be in individual lines of code, in the collection of code into functions or objects, or in the relationship between classes and objects. It might be in the way in which an Application Programming Interface (API) is exposed, and arguments have been made that this should even extend to the order and type of parameters sent into functions [9]. It can also be easily masked by students looking to mislead an academic or just by those who didn't realise that what they were doing strayed into plagiarism at any point. Within my own university, we focus on attribution as the differentiator between 'good software development' and 'plagiarism', but that presupposes that students are aware that there is a need for attribution at all. In the process of diligent software development, a student may refactor a piece of code taken from elsewhere to add in features, remove unnecessary complications, or simply make it consistent with the context in which it is placed. Thus, while they are benefitting from a solution they have found elsewhere, they gradually smudge over its original 'alien' conventions and bring them into line with the conventions they themselves use. The cracks between the two sets of code are plastered over, and if done well there will be no sign that there was ever a crack there to begin with. Thus, the plagiarism becomes, through adaptation, completely invisible. That does not mean it was never there, but the skill and knowledge required to both understand the code and refactor it for consistency is in itself a very valuable programming skill. However, attribution would still be necessary to acknowledge the intellectual debt that the submission owes to the original author, even if there is no trace of the original code left. In my experience students rarely go to the effort of consciously (or indeed, blissfully unconsciously) covering their tracks – they just don't realise that what they are doing constitutes a need for attribution [4].

In many cases, code might not simply be taken from an online or offline source but written collaboratively with other students. In such collaborations, it is rare that the effort is invested equally amongst all participants, and often the stronger students give more assistance than we might desire to their weaker colleagues. However, one of the key benefits that comes from a formal educational experience is the social context in which study is placed. We expect students to discuss their work with each other, plan out solutions, confer on tricky sections, and so on. All of this is useful team-building and group-work – elements that we often work to explicitly stress within core elements of a curriculum. However, we still do expect when work is submitted that it is meaningfully distinct for each individual. Within the constraints of software development though, this can be difficult and students often lack the skills required to make a meaningful judgment on what constitutes distinctly original work. We expect this work to be different in more than just a few variable names or function names, but no matter how we may stress this we are still operating in an environment where the skills to make that judgment may be lacking.

Thus, we see multiple submissions of what is essentially the same code, with only minor surface details changed. Here then is plagiarism which is merely a virtue taken too far into vice – there is often no intent to deceive, or the effort to hide the source of code would be performed more diligently. Students too in these circumstances often confuse the difficulty they had with the work with the academic's likely judgment on individual effort. They may not realise that the fact it took four hours to write a loop doesn't mean that it looks like four hours of effort to the person grading it.

In the experiences I have had with student counts of plagiarism, of which there have been many, only a very small fraction of them have left me feeling that there was a genuine attempt to obtain through deceit credit for work that they had not done. Instead, it tends to be one of the following:

1. Believing they were more responsible for the code that they submitted than a dispassionate review of the contribution would reasonably conclude.
2. Being unaware of the need for attribution in an environment where reuse, generalisation and reliance on external libraries is permitted, and even encouraged.
3. Not appreciating the line between healthy collaboration with colleagues and plagiarising from class-mates.
4. Not fully understanding the expected attributional difference between exemplar material written by their lecturer and provided within the context of a course and those external resources which may be mentioned as 'further reading'.

Clear communication of these issues helps, but it presupposes again that students believe that the communication applies to them, and that they'll remember it when it comes time to submit the work they have done. In the latter case, the stress of deadlines and the worry over the degree to which a submission meets a coursework brief can be distracting enough that attribution may simply be a distant thought in a head already full to bursting.

That is not to say that we should not take a strong position on work that is judged to have been plagiarised, but instead to outline some of the complexities that come with ruling that a piece of work is plagiarised at all. Reasonable people can disagree on the extent to which a piece of programming code represents original work, acceptable modification of the work of others, or outright plagiarism. When it's difficult for subject matter experts to agree on all the details, it is especially difficult for students who lack the training and understanding of the wider context that experience provides in slow, gradual accumulation.

4. IDENTIFICATION OF PLAGIARISM

Within the process of identifying plagiarism, we must resort to academic judgment to determine when a submission has fallen over the line between 'good practice' and 'intentional or unintentional deceit'. The number of submissions that we must routinely analyse along with the intricate complexities of each individual submission mean that we can only ever truly, feasibly, investigate a proportion of these. Tools for automating detection are, for software code, lacking in the sophistication to pick up on anything other than the most overt use of external sources. Thus, we must choose a sample only. In the next section, this paper will discuss some of the ethical implications of this selective analysis.

Informal suspicions may be initially raised in a number of ways. This paper will outline these in turn before moving on to the ways in which the original source for code may be located. Of a necessity, we will not be too specific about the full range of ways in which plagiarism may be identified as the task is already difficult enough without adding additional elements of challenge. Much of the process must be shrouded behind a kind of 'security through obscurity' model.

One of the things that happens for the majority of students within software engineering degrees is that a faculty builds, from the ground up, an understanding of programming. We lay the foundations of their understanding, choose the examples, and structure the assessments. Within a faculty we might cover a broad range of skills and styles, but there is often a link between early and late parts of the curriculum embedded in a single individual. The one that teaches first year programming may also be the one that teaches second year programming. If that is not the case, the necessity of understanding the context of a student's overall experience of a topic means that lecturers will be aware of what is done in the pre and co-requisite modules that describe their own course's academic context.

We also sample the code that students write during practical exercises, often seeing the evolution of coursework as it is moulded from rough sketch to polished artefact. We likely have a hand in that evolution, offering suggestions here, corrections there, and an occasional helpful hand in tracking down misbehaving subsystems. This kind of ongoing familiarity means that we can see the way each individual student writes code, and we can see the degree to which it harmonizes with the way in which we've been teaching the topic. Everyone has their own particular quirks when teaching programming – some favour associative arrays, some prefer arrays of objects. Some prefer the strict architecture of a formally designed class model. Others prefer a looser, ad hoc arrangement of code. Some favour certain design patterns, others make use of language features that obviate their requirement. It is impossible to be a programmer without picking up some developmental quirks that represent the best solutions that have evolved from long, hard experience. On top of these are entirely ornamental quirks such as the way in which variables are named, or the use of camelCase versus underscores_in_names.

Within the courses we teach, many of these quirks will be communicated to students in the form of exemplar code, lecture content, or the occasional aside delivered as part of an informal discussion. These quirks in turn make their way into student submissions to a greater or lesser degree. My own propensity to use the word 'bing' as a temporary variable name has mentally mutilated any number of my students. I apologise if anyone reading this has had to deal with the consequences. As a result, the code that is produced by the students will tend to take on a signature that is similar to the one demonstrated by their instructors, and ongoing familiarity with what they are doing within the labs will make that signature known to their lecturers. It's something like our own personal accent – it doesn't **uniquely** identify us, but it will certainly be something people use to differentiate.

Thus, when code is submitted that doesn't conform to the signature we are expecting, it creates the first sense that something may be wrong with the code that is provided. It might be written with unusual formatting, strange variable names, or even in a structure that is entirely inconsistent with what we may have taught. In a module on using HTML5, we may find jQuery being used rather than the canvas we had been discussing; in a module on PHP, we may find that an old, clunky version of the mysql interface functions were used rather than the up to date mysqli libraries we had advocated. Such things don't necessarily mean that a student has taken their submission from another source, but do raise the suspicion that something unusual has been going on.

Rarely is it the case that such incidents spread throughout the entirety of a submission – what is more common is the discordant tone of two different styles clashing with each other. We are expecting to hear one accent, and suddenly in the middle of a sentence it switches to another – it has exactly that kind of jarring impact when we encounter it, and it too is a sign that something unusual has happened with a submission.

Sometimes it's not an especially jarring accent change, but instead a remarkable quality change – if the majority of a program is of dubious quality, but it surrounds a core that is beautifully written and designed, then we must treat the submission with suspicion. Similarly, if there is a beautifully designed program that just happens to be of dubious quality in those aspects of the brief that were least likely to be present in an online forum, we must consider the possibility of some form of plagiarism. Often, when writing assessments, a lecturer might use a standard 'stock exercise' that is well understood and easily communicated. In such occasions, a common tactic to dissuade students from using the first online solution they can find is to modify the specifics of the exercise to include aspects that are unusual. Thus, students may find the core of the solution but be left with the task of bashing at it until it does what the lecturer has thrown into the brief as a complicating factor. It is at these points of stress that we can often see the suggestion of some kind of code adaption.

Sometimes the suspicious aspect comes in with a student who dramatically over-accomplishes in functional requirements that were never part of the brief, but under-accomplishes in requirements that were. Such unusual prioritization of development time is suggestive that at least some of the submission may have come from a template which did not precisely map on to the requirements as outlined or emphasized.

As a result of familiarity with students during ongoing instruction, we also build up a reasonably good mental profile of which students are especially capable, and which require our additional support. Those students most needing support are usually also those that produce code with which we are the most familiar as we spend a greater proportion of our time working our way through it with them. When a student with whom we have been spending much of our time suddenly submits a piece of work that we strongly suspect is beyond their demonstrated capabilities, then that flags up our interest.

Suspicion however is not sufficient for conviction, and having had their attention drawn to a piece of work a lecturer must ascertain whether their suspicions are grounded. This is often a straightforward matter of finding an especially distinctive piece of code and throwing it into Google. A distinctive piece of code is usually one that is sufficiently complex that its presence acts as a fingerprint for some other project elsewhere on the internet. Students may, as a result of submitting such code, change variable names, the order of invocation of certain statements, or the values associated with variables. However, other pieces of code are less pliable – especially if they implement formulae or make heavy use of structural systems of the host language. In those cases where Google can't throw any light on the matter, the search must move on to other sources such as GitHub or other code archival sites.
If that doesn't work, it's possible to attack the problem from the other direction and execute a search for what you'd look for if you were trying to find a solution to your own coursework exercise. Sometimes the code is taken from a particularly obscure location, but it is rare that it takes too long to track down the original source of the code. In those cases where the code does not seem to exist, then it's necessary to consider the other plausible routes for the source.

More and more commonly these days, we must consider the source of a submission as being that of an essay mill [2][6]. Sadly, in such events where the providence of code may not be identified with online checking we must resort to whatever internal mechanisms we may have available to ascertain student understanding of their own submissions. My own preferred route is through a mini-viva, in which students are asked to explain how their submission works, and to outline the process through which they may have developed it. On occasion, such a mini-viva results in a student giving a considered and confident explanation that resolves any lingering uncertainty about the authorship of the work. Often too, the viva reveals a lack of understanding that likewise settles the issue in the other direction.

5. THE ETHICAL IMPLICATIONS OF INVESTIGATING PLAGIARISM

Having outlined the ways in which plagiarism may manifest itself within programming submissions, and discussed some of the ways in which plagiarism may be detected by academics, we must turn to the ethical implications that are raised by such methods. If we are to truly treat students fairly, we must be aware of the troubling aspects of a process like this and examine where we can make systemic and procedural improvements to alleviate some of the issues.

First we must address the nature of student expectation – as discussed above, my own experience is that by and large students simply do not believe they are doing anything wrong. No matter how we may codify submission requirements, or inculcate a need to attribute, students often have a difficulty in seeing where the line between 'good software engineering' and 'academic misconduct' lies. There is a sector-wide inconsistency in how we teach software engineering principles, and how we treat students who adhere to principles of re-use. It is not that, as a sector, we do not communicate the importance of attribution – it is that students, as a general grouping, are often unaware of what should be attributed. The fact that there is rarely any obvious intention to deceive suggests one of two possibilities. The first is that students simply don't take any pride in their cheating, or have very low expectations of their lecturers. I don't believe, generally speaking, this to be true – it is not that there is little effort to obfuscate code, it is that there is often **no** effort to obfuscate. To the authors of this paper, that argues for the second interpretation – that such submissions are evidence of a lack of understanding, driven in part by the uneasy tension between plagiarism and sensible software engineering.

Solutions to such problems must stem simply beyond lectures on plagiarism and academic misconduct – students can easily give a word for word definition of plagiarism, and their responsibilities in that regard. It is not in the communication of the rules that we find problems, but rather in the interpretation.

With this in mind, we must always, first and foremost, look to whether we are properly contextualising the lessons of software engineering within their academic context. We should include discussions of what authorship means within software engineering and the day to day importance that attribution and sourcing plays in developing computer programs. We must also ensure that students take the necessary time to reflect upon the implications of their own submissions. Requiring students to formally acknowledge that the code they have submitted is entirely their own work, perhaps via a formal cover sheet, gives an opportunity for pause before uploading or sending the work. That pause might be what's needed to make them think 'Oh, I'll just put that attribution in, just in case'.

When assessing a submission for discords and disharmony as discussed above, we must also be mindful of the fact that in some cases we may have had only a small impact on the development of a student's personal signature. Students may have learned how to code outside our classes, and may indeed have arrived in the classroom with their own largely fully formed signature. If a signature is comprised of bad practice, our job may be to break it down and rebuild it in a better form. We must be mindful that any disharmonious elements in a code submission may be as a result not of external parties influencing a submission, but instead our own influence impacting on an already existing coding style. We must be careful to assess all submissions on their own merits, in the context of a student's own academic journey. Failing to do so could potentially subject a student to a harrowing hearing on academic misconduct where their own lack of a confident voice is used as evidence against them.

Similarly, as part of regular lab exposure to students we may find our own code making its way gradually into a submission as we explain how to address a problem or deal with a persistent error. Such *ad hoc* instruction tends to make the rounds amongst other students within that social circle, as it is usually perceived to be a 'lecturer approved' solution. It's important that as we provide such additional support to students that we realize that it is likely to be repeated in other submissions as the work is discussed and analyzed. If we are forgetful of what we have told our students, this can look very much like a whole group of students copying each other. In reality it is a piece of ad hoc support that we ourselves provided that has been traded around a class in response to others having the same problem. In such cases, we must be careful of alleging any misconduct at all – in real terms, there is little difference between students using our lecture notes and using the code that we may have provided, in passing, as part of private classroom discussions. Consider if we might, under other circumstances, have simply written the code out on a whiteboard for the class rather than doled it out to one or two individuals in the course of class discussions. In such cases, how do we even attribute authorship when it was not actually the student who was the source of the code?

When identifying submissions as being suspect or including elements worthy of deeper investigation, we must consider whether or not our own investigation has a bias built into it. We are unlikely to indifferently find submissions where the plagiarism has been done well - when students have managed to successfully marry disparate elements into a coherent and harmonious whole. When we identify work that seems to be sourced from elsewhere, we must be mindful to not simply focus on the low-hanging fruit else we run the risk of punishing those who try the least to obfuscate a submission. In addition to picking up on courseworks that are problematic, I advocate subjecting an additional random sampling of all submissions to an in-depth investigation even where there is no suspicion of wrong-doing. While such investigation rarely yields results, it has on occasion uncovered an especially clever piece of academic misconduct that would otherwise have gone unchallenged. In addition, it ensures that it is not only academic suspicion that leads to investigation – while such judgment is vital in uncovering plagiarism like this, it is also difficult to disassociate from the context of a student cohort. We cannot be sure that we are not letting personal likes or dislikes have influence on the investigation process. We cannot know how widespread plagiarism is within our modules – we can only say how often we notice it. By ensuring we sample beyond the obvious suspects, we can build our own confidence that the work we would otherwise have passed without comment is academically sound.

We must also be careful in ensuring that we are fairly representative of how we search out plagiarism. As discussed above, it may be extremely difficult to source code that comes from an essay mill, whereas a standard online tutorial may take only a few minutes of searching. This creates something of a class divide in investigating plagiarism, where those who can afford to buy 'off the shelf' solutions to class exercises are simultaneously inoculated against proper academic inquiry into the providence of code. When searching out the sources of work, we should look not only for the source of code, but also for incidences in which our course-works themselves have been floated online. Often, a search for a few indicative phrases from our own coursework briefs will reveal a request for a solution on an essay mill site, and this can be enough to raise real concerns regarding the authorship of submissions. An in-depth investigation of a student submission involves taking in a number of sources, and it would be unethical to do so without considering what financial solvency may permit in terms of covering the true source of code authorship.

In the sourcing process too, we must be mindful of the fact that there may be several sources which have been synthesized into a single submission – it's unusual that only one source is ever the single canonical reference point for all incidences of plagiarism. It's not enough to simply find a bit of code and say 'gotcha'. It's necessary to forensically outline the source of code statements and consider whether the welding of disparate elements may reflect sufficient mastery of the topic in and of itself to be worth credit. For my own purposes, when I suspect plagiarism I will go through each line of a submission and comment out those that come from an external source. Where adjustments have been made, such as changing the name or value of variables, I will comment those changes too. The result is a review of the code that allows for me to specifically reference lines of code and link them back to their original source. That which is left is, as best I can tell, the student's original contribution to the work. On occasion, when mitigating factors have been taken into account, that original contribution can turn out to be sufficient to pass a module. Whether that is an appropriate outcome is something that must be assessed on a case by case basis, but this forensic deconstruction is a process that both serves to solidify an argument for academic misconduct as well as more effectively frame the student's own contribution to the work.

This forensic examination of the code can serve as a valuable part of a formal or informal viva on the providence of a submission. However, here we must be careful – academic regulations may not permit a viva to be used as an additional, unannounced format of assessment. Often as part of an academic misconduct hearing there will be some viva element in which students may be asked to explain their code, but this is different to simply getting people in to ask about what they did. The possibility of later examination via viva should be announced in course books and module descriptors. It would be unethical to assess based on hidden criterion within a course, and likewise unethical to offer no guidance as to who is likely to be selected to perform. Linked to this is an issue of stigma if the only people asked to present their work orally are those who have likely plagiarized – in such cases, the invitation alone is enough to overlay a degree of suspicion amongst students. Thus, if vivas are to be conducted they should include a random sampling of students who are under no suspicion of plagiarism. Not only does this mitigate the stigmata issue, it also ensures that there is a control group against whom performance can be calibrated. The fact that a student cannot communicate clearly the code they are suspected of having not written may not mean anything when students under no suspicion also cannot clearly communicate! We must be careful to not prejudge the result, and equally careful not to stack the deck against students.

In the event that a student's work fails all possible good faith considerations, my own preference during hearings of academic misconduct is that the student be provided access to the full annotated transcripts of their code. While to a certain extent this allows an opportunity for students to shape the narrative of their explanation, in most cases the evidence is reasonably cut and dried. All that providing the code ahead of time does in that respect is allow for students to consider the evidence outside of the fraught, and often stressful, environment of an academic misconduct hearing. My own feelings on this matter is that it is much better to hear a considered explanation, even when it may be manufactured. The alternative is an explanation that is a result of stress, worry and the discomfiture that comes from misconduct being alleged. In the latter cases, we cannot reasonably expect that students can acquit themselves under such conditions even in those situations where they may have a reasonable explanation. In none of the academic misconduct hearings I have been responsible for initiating has there been an explanation that made me feel as if the student had been incorrectly targeted. However, if there was such a plausible explanation, I would like to hear it rationally put forward without the additional stresses implied by a formal academic hearing. In some cases, being presented with the evidence alone may be sufficient to make a student acknowledge the work that they submitted was substantively influenced by external sources.

6. CONCLUSION

The lack of any realistically effective mechanism for algorithmically detecting code plagiarism means that even now the process of identifying academic misconduct is one tied up in issues of academic judgment. However, in identifying submissions that have the hallmarks of external influence, we must be careful not to allow our own plagiarism antennae to override our ethical duty of care to our students.

Within software engineering as a discipline, and particularly within the topic of programming, many of the normal conventions of plagiarism simply do not hold – we work within very limited vocabularies and even within limited structural flexibility. The good practice of software engineering too is in many ways an exhortation towards plagiarism – we endorse, as a field, principles of re-use and the application of generalized solutions to problems rather than encouraging individuals to solve them anew each time. Our reference to algorithms, standardized data types, and design patterns creates a powerful impression that we have a preference, as a field, for standard solutions. Our own conventions regarding attribution too are loose and ill-defined, and may even be impossible to honour within complex environments where authorship may be an emergent property. None of this excuses academic misconduct, but it does help situate it within a context that makes it easier to explain.

When we are suspicious of a submission, the process through which we go is often bespoke and ad hoc – we perceive disharmony in submissions, or find things written in ways that are entirely alien to the structure we have inculcated into our students. It is those submissions which most trigger those reactions that are likely to receive the most attention in terms of further investigation, and this has a risk of skewing the results towards those who are least likely to be intentionally attempting to mislead.

We must be mindful then to ensure that the plagiarism investigations that we perform are not only focused where we have suspicions, but also where we have no reason to assume dishonesty at all. Plagiarism which is the best well-executed is almost by definition the least likely to trigger our initial suspicions. As a result we should be careful not to give the clever plagiarists a free ride while we focus our attention on those who simply did not understand the obligations of attribution.

7. REFERENCES

[1] Aiken, A. (2005). Moss: A system for detecting software plagiarism. University of California–Berkeley. [Available from See www.cs.berkeley.edu/aiken/moss. html].

[2] Bartlett, T. (2009). Cheating goes global as essay mills multiply. The Chronicle of Higher Education, 55(28), A1.

[3] Batane, T. (2010). Turning to Turnitin to Fight Plagiarism among University Students. Educational Technology & Society, 13(2), 1-12.

[4] Gullifer, J. M., & Tyson, G. A. (2014). Who has read the policy on plagiarism? Unpacking students' understanding of plagiarism. Studies in Higher Education,39(7), 1202-1218.

[5] Kim, D., Han, Y., Cho, S. J., Yoo, H., Woo, J., Nah, Y., ... & Chung, L. (2013, March). Measuring similarity of windows applications using static and dynamic birthmarks. In Proceedings of the 28th Annual ACM Symposium on Applied Computing (pp. 1628-1633). ACM.

[6] Mahmood, Z. (2009). Contract cheating: a new phenomenon in cyber-plagiarism. Communications of the IBIMA, 10(12), 93-97.

[7] Marsh, B. (2004). Turnitin.com and the scriptural enterprise of plagiarism detection. Computers and Composition, 21(4), 427-438.

[8] Savage, S. (2004). Staff and student responses to a trial of Turnitin plagiarism detection software. In Proceedings of the Australian Universities Quality Forum.

[9] Turner, J. (2012). Developer Week in Review: Are APIs Intellectual Property? In Radar. [Available online at http://radar.oreilly.com/2012/05/api-oracle-googlecopyright-c.html]

[10] Zoubi, Q., Alsmadi, I., & Abul-Huda, B. (2012, May). Study the impact of improving source code on software metrics. In Computer, Information and Telecommunication Systems (CITS), 2012 International Conference on (pp. 1-5). IEEE.