Securely Autograding Cybersecurity Exercises Using Web Accessible Jupyter Notebooks

Mac Malone
tydeu@cs.unc.edu
University of North Carolina
Chapel Hill, United States

Yicheng Wang
yicheng@cs.unc.edu
University of North Carolina
Chapel Hill, United States

Fabian Monrose
fabian@ece.gatech.edu
Georgia Institute of Technology
Atlanta, United States

ABSTRACT

The rapidly growing demand for computer science expertise combined with the pandemic era forced much education into large hybrid or fully remote learning environments, placing new emphasis on online learning platforms and automatic grading. Jupyter notebooks are a popular way to teach coding skills, as they provide an online way to distribute assignments with a low-cost Python coding environment to students and are also heavily used in data science, making the skills learned transferrable to the real world. However, autograding Jupyter notebooks is challenging, and contemporary tools have a number of pitfalls that make it difficult to integrate into a larger learning platform. As such, we implement our own grading system for Jupyter notebooks within the context of a broader gamified learning platform used in a cybersecurity course. Significant emphasis is given to the design, feedback, and security, as we often wish to introduce vulnerabilities within the student's learning environment for them to exploit while simultaneously protecting the system from misuse. We evaluate the system during its use in the Fall 2021 semester, discussing both its successes and failures, and provide transferrable lessons other instructors can use in their own systems, as the autograding systems used by many instructors are home-grown.

CCS CONCEPTS

• **Security and privacy**; • **Applied computing** → **Interactive learning environments**; *Distance learning*;

KEYWORDS

Automated Assessment; Learning Platform; Cybersecurity; Distance Learning

ACM Reference Format:
Mac Malone, Yicheng Wang, and Fabian Monrose. 2023. Securely Autograding Cybersecurity Exercises Using Web Accessible Jupyter Notebooks. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023), March 15–18, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3545945.3569862

SIGCSE 2023, March 15–18, 2023, Toronto, ON, Canada.

ACM ISBN 978-1-4503-9431-4/23/03…$15.00
https://doi.org/10.1145/3545945.3569862

1 INTRODUCTION

As our world grows ever more technologically oriented, the demand for computer science expertise has similarly grown more intense. Student enrollment in computer science courses across universities has surged dramatically, leading instructors to teach larger and larger classes [3]. Simultaneously, the pandemic era forced many teachers into a remote learning environment, requiring them to quickly adapt to the new paradigm [1]. Teaching and assessment now often need to be done online to hundreds of students, placing new emphasis on learning platforms and automatic grading. Careful application of these technologies can even improve student achievement and reduce costs [2, 18, 23].

In computer science, Jupyter notebooks are a convenient one-stop shop to distribute in-class labs and take-home assignments, provide students a platform to write and execute code, and collect and grade submissions [4]. Jupyter notebooks are also popular professionally, especially in data science [19], providing students the opportunity to transfer the skills they learn in class to the real world. However, the standard automated assessment tool for Jupyter notebooks, `nbgrader` [8], has a number of pitfalls that make it difficult to integrate into a broader learning platform [14].

The greater use of technology in student assessment also creates new opportunities for academic dishonesty and cybersecurity vulnerabilities. For example, the number of student requests posted on the homework sharing site Chegg for five STEM subjects (including computer science) increased by 196.25% from April-August 2019 to April-August 2020 (the height of the pandemic) [12]. The cybersecurity concerns with automatic graders have also been highlighted in previous work [10, 24] with Peveler *et al.* comparing the relative security and efficiency of different solutions [20].

Our Contributions: We present our design of an automated assessment system for Jupyter notebooks situated within a larger gamified learning platform and report on its use in a cybersecurity course at our university. We combine previous work on the types of automated assessment, grading Jupyter notebooks, giving rich feedback, and securing such systems into a single approach, providing a top-to-bottom analysis of the challenges and pitfalls involved in creating a successful, secure autograding system. What makes our approach unique is its focus on gamified cybersecurity, which complicates the security concerns of our system. We often wish to introduce vulnerabilities within the student's learning environment for them to exploit while simultaneously protecting the system from misuse (be it academic dishonesty or malicious code). In outlining our contributions, we focus on design principles and transferrable lessons from our experience that other instructors (cybersecurity or not) can apply to their own systems, as the autograding systems used by many instructors are home-grown [24].

2 BACKGROUND

A variety of different technical strategies for autograding assignments are outlined by Wilcox [24]. These can be collected into three broad categories: black box analysis, source code analysis, and invasive analysis.

Black box analysis treats student code as a black box, interacting with the program only through standard process I/O. This can be done in a number of ways: from simply piping in template input and comparing the output to some fixed reference (*e.g.,* via `diff`) to fully custom test scripts that construct input and analyze output in complex ways. On the other hand, *source code analysis* does not run the student programs at all, but instead statically analyzes student code for style and functionality. *Invasive analysis* serves as something of a midpoint between the two approaches: test code interacts with student code directly, allowing the grader to use builtin features of the underlying language to assist in grading. This can be done via instrumentation (*e.g.,* tracking student code via provided libraries or by automatically injecting grader code into the program), through reflection (*e.g.,* using metaprogramming to list student definitions), or by utilizing and invoking student-defined abstractions and procedures directly from the test code.

Each approach has strengths and weaknesses, often balancing power with security concerns. This balancing act means that a sophisticated approach will often mix all three strategies. For example, invasive analysis, while powerful, also poses the greatest security concerns – as the metaprogramming techniques it utilizes can also be used by students to find the grader code and reverse engineer assignment solutions. Thus, extreme care must be taken not to leak too much information, which often limits what can be done. In contrast, source code analysis is easily secured, but makes analyzing runtime behavior infeasible. Conversely, black box analysis supports runtime analysis but presents all the security concerns common to any black box unknown program. However, if run within a secure environment, such concerns can be mitigated.

Securing Grading Systems

One way to secure a student program to run it from within a virtual machine (VM). This almost completely isolates the program environment from the host machine, providing strong security guarantees. However, provisioning VMs is time and resource intensive and also makes communication between the grader and the student program extremely difficult.

A simpler solution is to use containers. Containers are a lightweight alternative to virtual machines, providing a similar level of isolation of the internal environment from that of the host machine but with a much lower setup cost. Virtualization is also performed at the system level, enabling test code and the isolated black box program to communicate through standard process I/O. However, spinning up a container still takes time and the isolation provided by a container, while great for security, also limits the student program's interoperability with other elements of the host machine, which can be undesirable.

Thus, another alternative solution is to use a jailed sandbox. With a jailed sandbox, the student program is run in the same environment as grader but with fewer privileges. For example, the grader run as root, but runs the student program through a unprivileged student account. The grader can also use tools like `chroot`, `rlimit`, and `seccomp` to limit student processes' access to the file system, memory and CPU, and system calls, respectively. However, unlike containers and virtual machines, constructing a sufficiently secure jailed sandbox is difficult, as it is easy to overlook some aspect of the environment a student program can improperly leverage to gain an advantage.

Peveler *et al.* [20] compare using jailed sandboxes versus Docker containers for autograding. They demonstrate the feasibility of both approaches, but note that jailed sandboxes do provide some performance advantages when one needs to quickly spin up and tear down many such environments.

Kinds of Feedback

Previous work by Narciss [16] and Keuning *et al.* [11] outlines many kinds of feedback a system can provide. There are six common categories: knowledge about performance, knowledge about results, knowledge about mistakes, knowledge about task constraints, knowledge about how to proceed, and knowledge about concepts.

Knowledge about performance (KP) and *knowledge about results* (KR) cover simple measures of performance and task completion (*e.g.,* grades and rubrics). *Knowledge about mistakes* (KM) includes test failures, compile errors, runtime errors, style issues, and performance issues. Such information can be limited (*e.g.,* pass/fail of a meaningfully named test) or detailed (*e.g.,* full stack traces and error messages). *Knowledge about task constraints* (KTC) and *knowledge about how to proceed* (KH) provides hints to the learner. KTC hints are reminders that list the steps needed to complete a task and highlight missing task requirements. KH hints are suggestions that help with error correction and figuring out next steps. Finally, *knowledge about concepts* (KC) covers what a tutor might provide: explanations of the relevant subject matter and examples illustrating certain concepts. Such help can either be provided when the student requests it or automatically offered when the system detects the student is struggling.

An ideal, fully automated feedback system would be able to provide each kind of feedback, but most systems focus on KP/KR/KM. One reason for this that the other kinds of feedback usually require custom scripting on the part of the instructor to identify relevant mistakes and provide intelligent hints. KM feedback, however, can be templated via example I/O, error reporting, and specific resource limits. Futhermore, it is already common in computer science outside education in the form of test suites and linters. Also, in the case of KC, many autograding systems are used within the context of a conventional class with an instructor that provides such content, creating little demand for it in autograding systems.

3 APPROACH

Our grading system is built into an online, gamified learning platform we previously designed – Riposte [13] – that we use to teach the introductory computer security course at our university. We gamified our learning platform to foster competition (an important aspect of cybersecurity), motivate students, and improve learning outcomes [2, 15]. The platform has a number of separate modules including a missions page (which tracks assignment progress in a gamified manner with achievements), a leaderboard (which compares students against one another), an embedded 2D action game, and the focus of this paper – an integrated Jupyter notebook server.

Infrastructure Setup

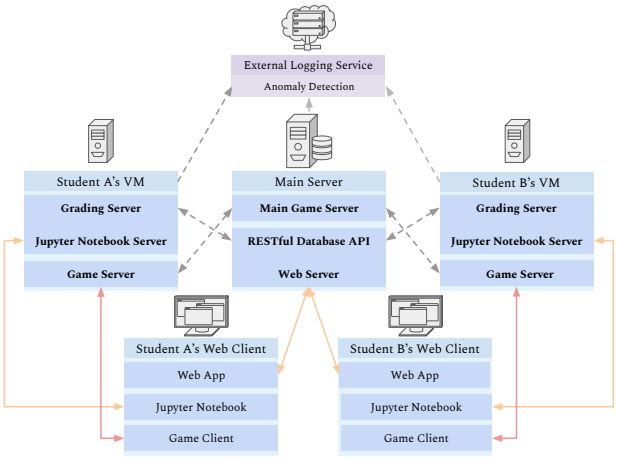

Figure 1: *Overview of the server architecture (best viewed in color).* Boxes with bold services are run together within a single isolated Docker container. Arrows represent communications between services. Red solid ones are those that the students can monitor and are encouraged to explore and exploit. Yellow solid ones can technically be seen but are not the focus, and gray dashed ones are invisible to students.

Each student we teach is provisioned a cloud VM. This VM hosts multiple Docker containers which provide various services for the student, including the Jupyter notebook environment and the game server (see Figure 1). We chose this approach because it enables strong isolation of student activity from their peers and simplifies the administration of students environments. Alternative solutions, such as running multiple different student Docker containers on one machine and/or sharing environments between students do not easily scale well and create administrative headaches and security risks. However, our solution is also more resource intensive and requires substantial existing infrastructure, so in cases were such an approach is infeasible, it makes sense to adopt such alternatives.

The Docker container with the Jupyter notebook environment is split in two: a Jupyter notebook server runs under an unprivileged user with the relevant tools, libraries, and Python packages installed for the tasks we give students, while the grading server is run separately under `root`. Once finished with an assignment, students submit their notebook to the grading server, which runs their solutions within a jailed sandbox running under the same unprivileged user they used for development. This sharing of same system environment was designed to prevent common grading failures caused by unanticipated incongruities between the student and the grading environment. However, it also means that students have more control over the grading environment and can install packages and tools the instructor may not want them to use. For us, this is not a problem, as the open-ended, challenge-based, gamified nature of our assignments allows us to give the students significant leeway to solve problems their way and minimizes the chances of pre-packaged solutions being readily available.

To access this environment, students log into the central web portal and select the Code module tab. This page has an IFrame that links to the Jupyter notebook server in the environment (see Figure 2). This is achieved by having the Docker container expose the server's ports to the VM and making the VM accessible from a subdomain of the learning platform's website. The link to each student's Jupyter server is then stored in the platform's database and used by the website to generate the IFrame presented to students. The end result provides the student with a online IDE with no setup cost on their side, making onboarding students as easy and pain-free as possible.

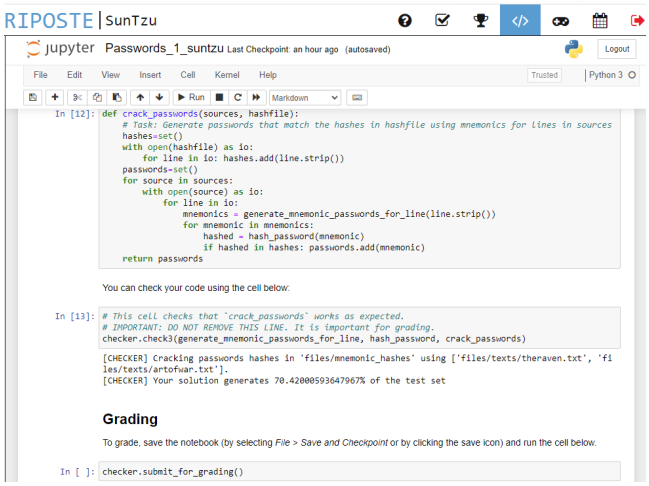

Figure 2: A Jupyter notebook in the learning platform.

Grading System

Our grading system has three parts: a per-assignment inline aide, a general grading server, and a per-assignment grader. The aide submits a student's notebook to the grading server, which runs the assignment grader on it and reports the results to the learning platform. This pipeline is diagramed in Figure 3.

The aide is an insecure Python module that is accessible to students while they are coding. It provides an assortment of quick checks to help students verify that their code satisfies basic task requirements before they formally submit their notebook. Its invocations also serve as instrumentation points for the grading server that students are informed not to delete. Students can submit multiple times, as previous research shows this to be beneficial [6].

On submission, the grading server converts the notebook into a Python script using the standard `nbconvert` tool provided by Jupyter. This script is then run in the jailed sandbox but with a different aide module. This module, instead of performing checks, uses reflection to extract the student's code into separate distinct scripts that are augmented to pass command-line input to specific functions and output their results. Afterward, the assignment's individual grader executes these scripts in the jailed sandbox with different inputs and checks the outputs for correctness.

The assignment grader is a mix of black box and invasive analysis. Invasive analysis (*e.g.*, reflection and instrumentation via the aide library) is used to split the notebook into multiple tiny scripts

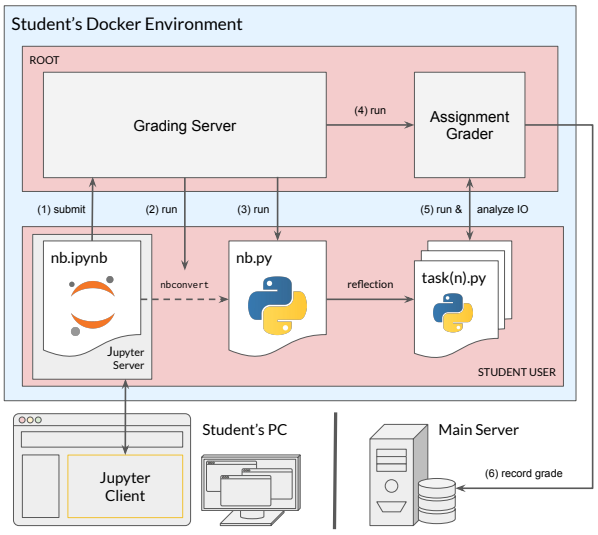

Figure 3: *Overview of the grading architecture.* Students edit their Jupyter notebook through the web client embedded in the learning platform. When they are ready for grading, they (1) submit the notebook to the grading server, which (2) runs `nbconvert` to transform it into a single Python file, which is (3) run by the grading server to produce small tasklet Python files via reflection in the aide. The server then (4) runs the assignment grader, which (5) runs and analyzes the tasklets for correctness and (6) records the grade in the main database.

that can be black box analyzed (in the jailed sandbox) for correctness. The black box analysis ensures user code cannot discover and exploit the grader code to achieve improper assessments while the invasive analysis allows students to write relatively free-form code that is smartly transformed into something more amendable to black box analysis. What makes the invasive analysis secure here is that it is run separately from the grader in the jailed sandbox and it has no knowledge of the grader's strategies.

Results from the grader are sent to the learning platform to record and display in its various interfaces – *e.g.,* awarding trophies on the missions page and ranking up on the leaderboard. Once a student has completed a goal and received the corresponding trophy, the goal remains completed even if student breaks their solution in future submissions. A student may thus be considered to have completed all tasks even if their final submission does not.

Automated Feedback

The feedback provided by the aide is substantially more detailed than the grader. The grader just provides basic KP/KR/KM. It notes which tasks were completed, if there were runtime errors or incorrect outputs, and assigns an overall grade. This limited detail is designed to prevent students from using the grader as an oracle to incrementally fix their code and/or figure out test cases the grader is using and hard code solutions to them without actual solving the problem and completing the learning objective of the assignment.

In contrast, the aide provides detailed KM (*e.g.,* what output was expected versus what was actually produce), general KTC, and some KH (*e.g.,* hints about edge cases they may have missed or strategies for improving their solutions to address task constraints). An experienced student can even use Python's reflection utilities to print out the contents of the aide library for themselves and see what test cases it is using. Solutions, however, are pre-computed to prevent them from easily discovering the algorithms they need to write. As the aide's test cases are just a subset of what the grader checks, hard coding these solutions will not help the students pass the grader and thus mitigates the dangers of such leakage.

Other forms of feedback, such as knowledge about concepts (KC), are provided statically through the assignment instructions embedded within the notebook or dynamically through lectures and office hours. In the future, we are considering better integrating such feedback into the platform itself so that it could function as independent learning tool separate from the classroom, but that is not the current focus of our system. Instead, our approach simply augments existing instruction, it does not fully replace it.

4 CASE STUDY

We have been using the broader learning platform that the grader is situated in multiple iterations of the undergraduate "Introduction to Computer Security" class at our university. The course is higher-level course generally taken by juniors and seniors and some graduate students. In the Fall 2021 semester, we introduced the new grader and evaluated its utility as part of the course. In this iteration, there were 59 students (51 males, 8 females), and the course was taught in a hybrid in-person / remote environment, with regular in-person lectures and remote gamified labs.

As an introduction, the course touches on a wide variety of computer security topics and corresponding exercises, including online and offline password cracking, web client modification, network traffic inspection, SQL injection, and cryptanalysis. Some of these exercises (*e.g.,* offline password cracking) are conventional programming assignments that make use of our Jupyter notebook and grader setup. Other assignments (*e.g.,* web client modification) are fully gamified and do not. Some assignments (*e.g.,* cryptanalysis) are a mixture, where students write code in the Jupyter notebook that they then use in the game to earn marks.

Our major research questions this time were: (RQ1) Do students like our system and does its use impact performance? (RQ2) Do the grader and aide provide sufficient feedback on student work?

As to RQ1, we believe tha Jupyter notebook is a convenient coding environment for students and thus hypothesized that *students will like that Jupyter notebook was used for their assignments* (**Hypothesis 1a**). We also believe that *the new system will have a positive impact on student performance* (**Hypothesis 1b**) as previous work [14] has produced similar results. For RQ2, the aide is designed to provide detailed feedback while the grader is not, thus we expected that *students will find the aide to provide sufficient feedback and the grader less so but still reasonable* (**Hypothesis 2a**). As the grader gives immediate feedback as to their success, their impression of it is likely to be tainted by their final grade so we also hypothesized that *students with a higher grade will find the grader to provide more sufficient feedback* (**Hypothesis 2b**).

To test these hypotheses and to generally measure the quality of the course, students are asked to complete a questionnaire after each assignment, which includes 5-point Likert-scale disagree/agree

assessments of their opinions. Of relevance to this research, we ask whether "[Students] liked that Jupyter notebook was used for this assignment" and whether "The [aide/grader] gave [them] sufficient feedback on their work." Students are also asked to provide qualitative feedback about their experience. Grades are recorded in the learning platform on a 5-point letter scale – F, C, B, A, S – where S means they completed every task for an assignment (including bonus tasks) and rewards them extra points on the leaderboard.

Results

To answer RQ1 and verify **Hypothesis 1a**, we found that students consistently agreed with the statement that they "liked that Jupyter notebook was used for this assignment" across every exercise and the final exam. The median was 5 (strongly agree) for each instance, the mean ranged from 4.19 to 4.44, and the standard deviation ranged from 0.77 to 1.01. Thus we have strong evidence to suggest that Hypothesis 1a is true – students do like the use of Jupyter notebooks in the course.

To verify **Hypothesis 1b**, we compared the student's performance on an offline password cracking assignment across three iterations of the course using the metric of percentage of password cracked (a consistent goal across the different versions of the assignment). For the Fall 2021 semester with the new grading system, the mean was 81.36% and the standard deviation was 12.76%. For Fall 2020, where we used an older grading system, it was 71.86% / 21.41%, and for Fall 2019, where we did not use Jupyter or automated grading at all, it was 75.61% / 9.49%. Using a two-sample, unpaired, unequal variances t-test, we found a statistically significant ($p < 0.05$) positive difference in the mean between Fall 2021 (new system) and Fall 2019 (no system), a statistically weak positive difference ($p < 0.15$) between Fall 2021 (new system) and Fall 2020 (old system), and no statistically significant difference ($p \gg 0.15$) between Fall 2020 (old system) and Fall 2019 (no system). This provides good evidence that the Jupyter notebook combined with the new grading system had a positive impact on student performance, a similar result to Manzoor *et al.* [14].

For RQ2, we analyzed the survey responses to whether "The [aide / grader] gave [students] sufficient feedback on their work" on the offline password cracking assignment. A histogram of the responses can be seen in Figure 4. The responses to the aide had a mean of 3.35 and a standard deviation of 1.2 while the grader had a mean of 2.75 and and a standard deviation of 1.28. Both had a median of 3. As can been seen in the figure, the response to the aide leaned positive while the response to the grader leaned negative.

To formally verify **Hypothesis 2a**, we performed both a Student's t-test (two tailed, paired) and a Wilcoxon signed-rank test (matched) comparing the aide and grader responses for the assignment. We did both as Likert scales produce ordinal data and there is a debate on whether parametric or non-parametric tests are more appropriate for such data [22]. Both tests showed a statistically significant difference ($p < 0.01$), indicating that, as expected, the aide's feedback was deemed more sufficient then that of the grader.

For **Hypothesis 2b**, we computed the correlation between students grade (1/F-5/S) and their response to grader (where grade is the independent variable) using both Pearson's r (parametric) and Spearman's ρ (non-parametric). We found a statistically insignificant ($p \gg 0.15$) and slightly **negative** correlation (r, ρ = -0.069),

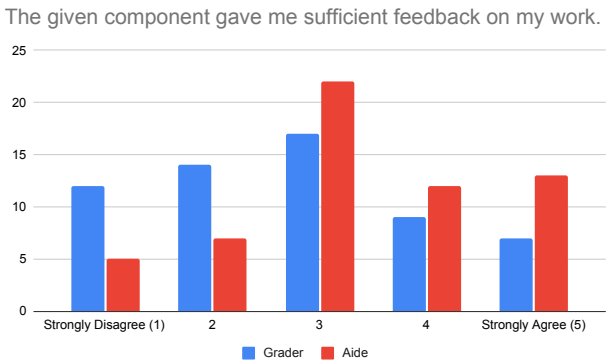

Figure 4: *Grading System Survey Responses.* Students' views on whether the grader/aide gave them sufficient feedback.

which was rather surprising, and provides no evidence for Hypothesis 2b, suggesting that students' grades were not a factor in their opinion of the grader.

5 DISCUSSION

While we expected students to find the grader's feedback less sufficient than that of the aide, the response to both were less stellar than we hoped. However, listening to students' qualitative reactions provided us insight into their reasoning, which we discuss here.

Feedback Expectations

Many students expressed confusion about the the grader/aide dichotomy – *"The difference between the [aide] and the grader was frustrating," "the [aide] implements [grading] differently than the autograder."* When the grader did not mirror the aide, they viewed the aide as providing insufficient feedback. Furthermore, even students who understood the dichotomy still felt that grader could provide more feedback. For example, *"I'd simply say a little more specificity on output specifications, errors/specification violations, and reports would be helpful, but I also understand if that compromises the integrity of the assignment."*

Students provided a number of suggested feedback improvements. For example, they wanted the grader to show the exact number of passing test cases, rather than the thresholds (*e.g.,* low, medium, and high) we had decided on. It made them feel better to see the number slowly climb up as they improved their code rather than jump up at unknown intervals. As one student stated: *"it was a lot more encouraging to see the percent [in] the [aide] slowly getting [higher] and mildly discouraging to see 'Low' in the grader."* While such a suggestion is reasonable, others wanted more details. For example, one student asked for the *"[s]pecification of inputs [for] the grader."* Reading between the lines, some students seemed to be trying to use the grader as an oracle to improve their code and were unhappy it was too difficult to do so.

Another indication that the grading system was serving as a crutch was students' tendency to never test their code themselves, but rather to rely entirely on feedback (*i.e.,* KM/KTC/KH) of the grader and aide – a finding similar to Chen [5]. They often attempted to divine from the grader whether their code had a specific

behavior (*e.g.,* worked on particular edge case). We, the instructors, then had to highlight (*e.g.,* in office hours) that they could just write a simple test themselves to see if their code was behaving.

Previous research is split on how much feedback is too much and whether the potential for abuse is significant. Some authors note that greater feedback improves performance and achievement – a consistent finding in the research [17] – and that this is indicative of the approach's success [9]. Others argue that this performance is merely a by-product of the student relying on the feedback itself to derive solutions and that they will thus have a hard time accomplishing real-world tasks where such feedback is inaccessible. For example, in Rao *et al.*'s intervention, 60% of students reported they had come to rely on the autograder to verify solutions [21]. This view is further supported by the common finding that students make significant progress in their learning even when external feedback is quite impoverished [17].

As our educational topic was cybersecurity, a field where feedback on the success or failure of an approach is often minimal, we leaned towards too little rather than too much feedback. However, this can, as observed, frustrate students, so areas in which the real-world applications are more prone to well-structured problems with rich feedback, it may be wise to provide more. Regardless, how much feedback is appropriate should be a question an instructor carefully considers when designing a grading system. One should also inform students exactly how much feedback they should expect from the system to avoid the confusion we observed.

Resource Limitations

Another major area of concern was resource management. The assignment in which the grader was analyzed (offline password cracking) is very sensitive to memory and CPU overuse. Students found optimizing their programs difficult and the errors provided by the grader opaque. Some specifically cited *"clearer expectations on the parameters of the memory cutoff"* as a pain point. This was not helped by the grader initially producing just a generic "grader crashed unexpectedly" message due to its restricted error reporting.

Furthermore, in our setup, the Jupyter notebook is accessed indirectly through the web and students do not have access to the machine it is running on. Therefore, resource overuse by student's code in the grader or in the notebook can cause the entire system to hang. This results in the student losing access to their Jupyter notebook and requires them to wait for the resource monitor to restart the system before they can resume coding. As solutions can be both time and memory intensive, it can also take a long while (tens of minutes) before the problem emerges and resolves itself.

One potential solution is to give the Docker environment a fixed resource limit that is low enough to allow the VM to continue handling connections and a UI action can be added to the learning platform to reset the environment as a student's last resort. However, this is still very disruptive to a student's workflow, so resource limits within the environment itself can be instituted to try and avoid this situation. Such limits should ideally be the same between the grader and the notebook itself to make sure code that works in the notebook works in the grader. However, generous limits means the grader can not grade much code in parallel (*e.g.,* run multiple test cases simultaneously), reducing turnaround time (another common complaint of students).

Therefore, when using autograding system for exercises that may be resource intensive, it is important to carefully consider how the resources for the grader and the student's coding environment are managed. In particular, if reasonable solutions are expected to be resource intensive, one must carefully consider the trade-offs between strict limits that can prohibit solutions and generous limits that making grading slower.

6 RELATED WORK

Most germane is the approach of Manzoor *et al.* [14] for autograding Jupyter notebooks within the Web-CAT autograding framework [7] and the Canvas learning management system. Like us, they wish to incorporate the user-friendly environment of Jupyter within a larger learning platform but discover a number of pitfalls with the stock solution for notebook grading, `nbgrader` [8].

Specifically, while `nbgrader` does provide a convenient UI for building assignments and denoting tests, the auto-grading it does must be in the form of unit tests and has to be applied manually to the code. Thus, it does not address how notebooks should be submitted for grading, how its results should be reported to students, or how to secure the system from misuse, and it still restricts the types of grading analysis that can be done to student code. Manzoor *et al.* address these issues by using `nbgrader` to create assignments, Web-CAT to autograde them, and Canvas to collect the results. Their end product was quite popular, with 75% of students stating they would recommend others to use a similar approach. They also observed a statistically significant increase in mean student grades, indicating automation had a positive effect on student performance.

We were encouraged by these results, but could not directly apply them. Like many other instructors who use home-grown systems [23], we already had a custom gamified learning platform we needed to use (for other assignments in the course, which focus on the embedded game). Thus, we could not switch to the Web-CAT and Canvas system used by Manzoor *et al.* Instead, we took cues from their design and related research to create our own autograding framework, which we tailored to our use case. This also allowed us to focus more on the *security* of the system, something Manzor *et al.* did not address, but was of particular concern to us given our educational topic of cybersecurity.

7 CONCLUSION

We presented an architecture for an automatic grading system for web accessible Jupyter notebooks. Our approach highlighted specific concerns about security and feedback, areas of particular relevance to our use case of cybersecurity. Upon evaluation, we found that students like the use of Jupyter notebook and the system improved their performance, but they had mixed feelings about the feedback of the grader and aide. Further discussion revealed that the students wanted to rely more on the grader to help fix their code, the merits of which are debatable. They also had a hard time optimizing their code to avoid resource limits, which was acerbated by the minimal detail initially provided on them. We hope the lessons we learned and the recommendations we made can help others avoid such pitfalls in the future and inspire further work – both inside and outside the specific topic of cybersecurity.

REFERENCES

[1] Wahab Ali. 2020. Online and remote learning in higher education institutes: A necessity in light of COVID-19 pandemic. *Higher education studies* 10, 3 (2020), 16–25. https://eric.ed.gov/?id=EJ1259642

[2] Shurui Bai, Khe Foon Hew, and Biyun Huang. 2020. Does gamification improve student learning outcome? Evidence from a meta-analysis and synthesis of qualitative data in educational contexts. *Educational Research Review* 30 (2020), 100322. https://doi.org/10.1016/j.edurev.2020.100322

[3] Tracy Camp, W. Richards Adrion, Betsy Bizot, Susan Davidson, Mary Hall, Susanne Hambrusch, Ellen Walker, and Stuart Zweben. 2017. Generation CS: The Growth of Computer Science. *ACM Inroads* 8, 2 (May 2017), 44–50. https://doi.org/10.1145/3084362

[4] Alberto Cardoso, Joaquim Leitão, and César Teixeira. 2019. Using the Jupyter Notebook as a Tool to Support the Teaching and Learning Processes in Engineering Courses. In *The Challenges of the Digital Transformation in Education (Advances in Intelligent Systems and Computing, Vol. 917)*, Michael E. Auer and Thrasyvoulos Tsiatsos (Eds.). Springer, Cham, 227–236. https://doi.org/10.1007/978-3-030-11935-5_22

[5] P.M. Chen. 2004. An automated feedback system for computer organization projects. *IEEE Transactions on Education* 47, 2 (2004), 232–240. https://doi.org/10.1109/TE.2004.825220

[6] Amy Cook, Alina Zaman, Eric Hicks, Kriangsiri Malasri, and Vinhthuy Phan. 2022. Try That Again! How a Second Attempt on In-Class Coding Problems Benefits Students in CS1. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1* (Providence, RI, USA) *(SIGCSE '22)*. Association for Computing Machinery, New York, NY, USA, 509–515. https://doi.org/10.1145/3478431.3499362

[7] Stephen H. Edwards and Manuel A. Perez-Quinones. 2008. Web-CAT: Automatically Grading Programming Assignments. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education* (Madrid, Spain) *(ITiCSE '08)*. Association for Computing Machinery, New York, NY, USA, 328. https://doi.org/10.1145/1384271.1384371

[8] Jessica B. Hamrick. 2016. Creating and Grading IPython/Jupyter Notebook Assignments with NbGrader. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (Memphis, Tennessee, USA) *(SIGCSE '16)*. Association for Computing Machinery, New York, NY, USA, 242. https://doi.org/10.1145/2839509.2850507

[9] Qiang Hao, David H. Smith IV, Lu Ding, Amy Ko, Camille Ottaway, Jack Wilson, Kai H. Arakawa, Alistair Turcan, Timothy Poehlman, and Tyler Greer. 2022. Towards understanding the effective design of automated formative feedback for programming assignments. *Computer Science Education* 32, 1 (2022), 105–127. https://doi.org/10.1080/08993408.2020.1860408

[10] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 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, Finland) *(Koli Calling '10)*. Association for Computing Machinery, New York, NY, USA, 86–93. https://doi.org/10.1145/1930464.1930480

[11] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2016. Towards a Systematic Review of Automated Feedback Generation for Programming Exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education* (Arequipa, Peru) *(ITiCSE '16)*. Association for Computing Machinery, New York, NY, USA, 41–46. https://doi.org/10.1145/2899415.2899422

[12] Thomas Lancaster and Codrin Cotarlan. 2021. Contract cheating by STEM students through a file sharing website: a Covid-19 pandemic perspective. *International Journal for Educational Integrity* 17, 1 (04 Feb 2021), 3. https://doi.org/10.1007/s40979-021-00070-0

[13] Mac Malone, Yicheng Wang, and Fabian Monrose. 2021. An Online Gamified Learning Platform for Teaching Cybersecurity and More. In *Proceedings of the 22st Annual Conference on Information Technology Education* (SnowBird, UT, USA) *(SIGITE '21)*. Association for Computing Machinery, New York, NY, USA, 29–34. https://doi.org/10.1145/3450329.3476859

[14] Hamza Manzoor, Amit Naik, Clifford A. Shaffer, Chris North, and Stephen H. Edwards. 2020. Auto-Grading Jupyter Notebooks. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 1139–1144. https://doi.org/10.1145/3328778.3366947

[15] Cristina Ioana Muntean. 2011. Raising engagement in e-learning through gamification. In *Proceedings of the 6th International Conference on Virtual Learning ICVL*, Vol. 1. University of Bucharest, Bucharest, Romania, 323–329. http://icvl.eu/2011/disc/icvl/documente/pdf/met/ICVL_ModelsAndMethodologies_paper42.pdf

[16] Susanne Narciss. 2008. Feedback strategies for interactive learning tasks. In *Handbook of research on educational communications and technology* (3rd ed.), David Jonassen, Michael J. Spector, Marcy Driscoll, M. David Merrill, Jeroen van Merrienboer, and Marcy P. Driscoll (Eds.). Routledge, New York, NY, USA, 125–143. https://doi.org/10.4324/9780203880869

[17] David J. Nicol and Debra Macfarlane-Dick. 2006. Formative assessment and self-regulated learning: a model and seven principles of good feedback practice. *Studies in Higher Education* 31, 2 (2006), 199–218. https://doi.org/10.1080/03075070600572090

[18] Edeh Michael Onyema, Nwafor Chika Eucheria, Ezeanya Christiana Uchenna, Eziokwu Patricia Nkiruka, and Ani Ukamaka Eucheria. 2020. Impact of e-learning platforms on students' interest and academic achievement in data structure course. *CCU Journal of Science* 1, 1 (2020), 1–16.

[19] Jeffrey M Perkel. 2018. Why Jupyter is data scientists' computational notebook of choice. *Nature* 563, 7732 (Nov 2018), 145–147. https://www.nature.com/articles/d41586-018-07196-1

[20] Matthew Peveler, Evan Maicus, and Barbara Cutler. 2019. Comparing Jailed Sandboxes vs Containers Within an Autograding System. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 139–145. https://doi.org/10.1145/3287324.3287507

[21] Dhananjai M. Rao. 2019. Experiences With Auto-Grading in a Systems Course. In *2019 IEEE Frontiers in Education Conference (FIE)* (Covington, KY, USA). Institute of Electrical and Electronics Engineers, New York, NY, USA, 1–8. https://doi.org/10.1109/FIE43999.2019.9028450

[22] Gail M. Sullivan and Jr Artino, Anthony R. 2013. Analyzing and Interpreting Data From Likert-Type Scales. *Journal of Graduate Medical Education* 5, 4 (12 2013), 541–542. https://doi.org/10.4300/JGME-5-4-18

[23] Chris Wilcox. 2015. The Role of Automation in Undergraduate Computer Science Education. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) *(SIGCSE '15)*. Association for Computing Machinery, New York, NY, USA, 90–95. https://doi.org/10.1145/2676723.2677226

[24] Chris Wilcox. 2016. Testing Strategies for the Automated Grading of Student Programs. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (Memphis, Tennessee, USA) *(SIGCSE '16)*. Association for Computing Machinery, New York, NY, USA, 437–442. https://doi.org/10.1145/2839509.2844616