ABSTRACT

BRUCE is a distributed system that compiles, runs, tests, and grades arbitrary code segments based on a given set of input and output. BRUCE uses multiple workers to handle large numbers of simultaneous submissions during major events such as Google Code Jam, TopCoder contests, and homework submission deadlines for large classes.

1. INTRODUCTION

Many computer science courses use automated grading mechanisms to test student assignment submissions and assign scores. Similarly, many online “programming challenge” sites (such as HackerRank) use similar automated grading mechanisms to test challenge submissions. These systems tend to be unique per class with no overlap in design or structure; for example, many computer science courses at the University of Pennsylvania rely on completely different systems to test and grade student submissions. Furthermore, they tend to be heavily language-dependent; a class that uses multiple languages (for example, CIS 120 uses OCaml and Java) would effectively need to create two separate backend grading systems for each language, even though the front-end, student-facing side is the same. This situation not only makes designing and creation of the system difficult, but also causes many maintenance problems as the system gets handed down to the new course staff each semester. Similarly, websites such as HackerRank must waste engineering man-hours creating these systems and maintaining them, much like how a course staff must do so for a university.

The goal of BRUCE is to fix this problem by creating a distributed back-end service that allows clients (such as the course staff or a programming challenge site) to run a suite of tests on submitted code and provide results. To the client, the system would be a “black-box” API and/or web interface that would simply require inputs, outputs, and submitted code. It would enable the same functionality as existing systems, but with no designing, creation, or maintenance on behalf of the client. Fundamentally, this would allow clients to focus on providing their core service to end-users, and also create a standardized automated distributed testing service for all use cases.

2. RELATED WORK

BRUCE’s related work falls into four categories. The first is open-source software, such as PC2, independently developed for use by other institutions. The second is proprietary submission software developed by the maintainers of programming contests such as Project Euler and Google Code Jam. The third is code sandboxing, or the act of ensuring that the possible damage done by malicious or ill-formed code is limited. The fourth is master/worker systems, a type of distributed system in which work is distributed to n slave machines under the coordination of a master machine.

2.1 Code Submission Systems

There currently exist several popular broadly-accessible code submission systems, usually aimed at programming contests, as well as many automatic code testing systems proprietary to the universities that built them.

Marmoset Project [9]
Developer: University of Maryland

The Marmoset Project is a system for handling student programming project submission, testing and code review. It was developed at the University of Maryland five years ago and recently handled 74,000 submissions over the course of a semester. Marmoset is language-agnostic, working with all programming languages, and is robust enough to handle both large and small submissions. With BRUCE, we want to emulate the way that Marmoset can handle any programming language. Given the wide variety of programming contests that are currently run (and especially given that the frequency of these contests will only increase given the recent positive national attention computer science has been receiving), we believe that it is crucially important to handle all major programming languages. It would be a significant barrier to adoption if only certain languages were supported.

Marmoset never runs student code on the machines that host the submit server and the database. Build servers instead connect to the submit server and present credentials for the University of Maryland courses that they are authorized to build projects for. If there are any appropriate submissions, then the build server downloads them before
disconneting from the submit server and building and testing the submission. Finally, the results of the tests are uploaded to the submit server. Thus, Marmoset can run multiple build servers for a course to provide both redundancy and quick turnaround time for project testing. We also want to emulate Marmoset in this area. Security and privacy, especially for universities (which have to comply with national laws about the nondisclosure of sensitive student data such as their student IDs), are paramount concerns for code submission systems.

**PC$^2$ [6]**

**Developer: California State University, Sacramento**

PC$^2$, or the Programming Contest Control System, was developed at California State University, Sacramento (CSUS) to support the activities of the ACM International Collegiate Programming Contest and its regional contests around the world. PC$^2$ allows contestants in these contests to submit their code segments to contest judges, who can recompile and execute the code as well as view test results alongside the code itself. PC$^2$ also has a feature called “automated judging” that grades software algorithmically. BRUCE will emulate PC$^2$ in this regard. We hope to add a number of plugins during our subsequent development next semester, some of which will focus on this ability to grade software. Having this functionality will make BRUCE even more useful during programming contests.

The PC$^2$ system automatically timestamps and archives submitted runs. It also supports contests being held simultaneously at different sites by automatically transmitting information and generating a single contest-wide page for each remote site. PC$^2$ can handle any language and any language development tool. PC$^2$ has several other features that are beyond the scope of the infrastructure service portion of BRUCE, such as contest submission scoreboard and communication tools for teams and judges, but could easily be built on a web application using BRUCE as a service.

However, PC$^2$ requires each contestant to install and configure the system. Some newer operating systems, such as Windows 8, have trouble running the software. This downside is something we certainly want to avoid with BRUCE. Programming contest organizers need to have the latest technology for their submission systems. As the contests become more popular, they may try to differentiate themselves by ease of entry for participants. Participants will enjoy using systems that are fast and cutting-edge, so we want BRUCE to be in both of those categories.

**University of Pennsylvania CIS 1xx Autograder [15]**

**Developer: University of Pennsylvania**

Penn uses an automatic grading infrastructure for its CIS 110, CIS 120, and CIS 121 courses. This system can hypothetically support any language via implementing a custom back-end. The system can perform either real-time grading (CIS 120) or after-the-fact grading (CIS 110 and 121).

The system sandboxes code by running it on a low-privileged user account that can only modify files in its own directory. The account can only communicate via writing text files to that directory, which it does in order to return the test case results.

The main spot for improvement of the system is that it is somewhat poorly designed; it is held together by several bash scripts. It is consequently hard to modify and debug, which is a potential pain point for adopting the system. We want the ease of iteration in development to be high for BRUCE.

**Vocareum [10]**

A recent entrant into the Code Submission System space is a company called Vocareum. Vocareum is a learning management system (LMS) similar to Canvas, which is used at the University of Pennsylvania. The difference is that Vocareum places an explicit emphasis on computer science courses. The system allows for functionality such as assignment authoring and management, in-context communication, analytics, and grading automation. Some of the University of Pennsylvania computer science classes are considering using Vocareum in the future.

While Vocareum is a robust solution with many features, it is not clear whether or not it executes code the way BRUCE intends to. Vocareum is more frontend- and user-focused, while BRUCE is more backend- and API-focused. In some ways, we see BRUCE being the eventual code execution backend for a service like Vocareum.

The largest difference between Vocareum and BRUCE is a matter of scope. BRUCE is a niche service targeted specifically for code execution, while Vocareum is an end-to-end service that provides functionality for multiple facets of a computer science class. For example, Vocareum contains a portal for professor-student communication and tools for teaching assistants to manage information about the class, including grades.

**Sphere Research [8]**

Sphere Research is similar to BRUCE in that it is a backend for code execution. Sphere has its own API that allows developers to plug-in and run code on their own systems, as BRUCE does. We have based some of our system architecture on that of Sphere Research.

There are two main problems, however, with Sphere Research that BRUCE hopes to improve upon. First, Sphere Research currently uses a SOAP API, which is an outdated paradigm compared to REST APIs used today. BRUCE plans to use the more modern REST API approach. Second, the documentation for Sphere Research is poor, making it clunky and hard to use. We plan on creating extensive documentation as well as client libraries and sample applications to remedy this issue.

### 2.2 Websites with Code Submission

Many websites exist that use code submission as part of their product. These range from online programming contests to coding scratchpads to interview-specific websites. Each one of these bespoke systems has a feature set that BRUCE should be an ideal replacement for.

**HackerRank [5]**

HackerRank is an online platform that allows coders to refine their skills and solve puzzles that earn them the attention of tech companies such as Facebook and Yahoo.
HackerRank offers a wide variety of services, such as tutorials, communication tools with companies, leaderboards, and coding challenges. The company built its own proprietary submission system.

Their system handles all programming languages. It optimizes security by limiting system resources (CPU time, memory, disk usage) per submission, limiting access to the filesystem so the user cannot read the submissions of other users, and disabling network access. All of these features will be essential for BRUCE to emulate.

During periodic coding challenges in which submission volume is high, HackerRank adds multiple code-checking servers behind a load balancer and scales multiple simultaneous submissions on the same machine. The system generally ensures that all submissions get an equal amount of system resources.

Google Code Jam [4]

Google Code Jam is an annual worldwide coding competition run by, as the name suggests, Google. The challenge makes its way around the world and is supported by a central website. In addition to allowing participants to submit code, the Code Jam website provides game elements like a timer and a leaderboard. When a contest is not in progress, aspiring participants can solve practice problems, which also uses the submission system.

Google developed its own submission software for the Code Jam website. It accepts all languages that have both a free compiler and a free interpreter. Participants can use any development environment that they choose, free or otherwise.

2.3 Code Sandboxing

Code grading system must have proper sandboxing for security reasons, especially when used at academic institutions, which must abide by federal student privacy laws, or for coding challenges, which usually have some explicit monetary compensation. A lot of research has been done on the subject of code sandboxing, with many different techniques developed to do so. For example, one technique may involve the use of the chroot command to “change” the filesystem in which the application is running [13]. The following is an analysis of another online submission system and its code sandboxing system.

Codepad [2]

Codepad.org is an online compiler and interpreter aimed at easing collaboration between people seeking to share code with each other. It runs arbitrary code of any language that a user enters and then generates a shortened URL that the user can share.

Codepad takes several steps to sandbox effectively. It disallows most system calls. Both the compiler and final executables are executed in a chroot jail with strict resource limits. A chroot jail is an artificial root directory which “locks down” a given process so that all the process can see is the directory in which it is running in [3]. The supervisor, which handles code execution, runs on virtual machines firewalled to prevent outgoing connections.

2.4 Master/Worker Systems

BRUCE will model its distributed systems on both academic literature and popular implementations, allowing it to handle an arbitrary amount of simultaneous code submission. There has been much work done on distributed systems so the information presented it is what is relevant to our planned system in particular.

When first building a system, the most important thing to consider is what kind of guarantees to make about the system’s consistency, availability, and partition handling (the CAP theorem). This is most famously discussed by Gilbert and Lynch [11] and states that any system can only have either two of the three options, and must make some sacrifices on the third. The CAP theorem is always important to take into account when designing a system; the creator must always consider the needs of the end-user and what sacrifices the system can make with regards to these three features.

The second big piece of related work regarding master/worker systems is how they will communicate. There has been much research done on message queues, which are the most common way of communication between nodes in a system. Many messaging queue systems have been created, each with advantages and disadvantages. Examples include RabbitMQ, ZeroMQ, and Kestrel, as well as proprietary software developed at IBM and Microsoft. On top of the message queue, the system will have a distributed task queue such as Celery to facilitate the task management that the master/worker system will use. What follows is a deeper analysis of two of the aforementioned systems: RabbitMQ and Celery.

RabbitMQ [7]

RabbitMQ is an open-source messaging broker - an intermediary for messaging. It implements the Advanced Message Queuing Protocol (AMQP), which was developed in the mid-2000s. The main purpose of AMQP was to address the scenario where there is likely to be some economic impact if the message is lost, does not arrive on time, or is not processed properly, as well as to create a standard for all such middleware [12]. RabbitMQ, built on top of this protocol, is known for its reliability, security, and strong queuing and routing capabilities. However, it does not function well in network partitions, which is something to take into account when creating a system around this. The system also has strong guarantees on reliable delivery of messages and has many built in measures and configurations to ensure that this happens. Lastly, RabbitMQ will gracefully handle network failures, an inevitable occurrence in a distributed system.

Celery [1]

Celery is a distributed, asynchronous task/job queue based on distributed message passing. It allows the system to distribute work across machines. It effectively adds another layer on top of distributed message brokers such as RabbitMQ, making it a middleware between the broker and the actual application. Celery is also highly available as workers and clients will automatically retry in the event of failure. The entire system is also extremely fast as a single Celery process can go through millions of tasks a minute, with less-
than-one millisecond roundtrip latency. It is currently used by companies such as Instagram and Mozilla to drive their back-end systems.

3. SYSTEM MODEL

BRUCE at its core is an API plus a distributed system underneath that API. BRUCE also includes an application built on top of the API for administering programming contests, and documentation for how to use and build on the API.

BRUCE uses a use-case-agnostic code execution platform underneath a simple API through which code can be submitted. An application using BRUCE’s API can focus on its own use case, be it administering a programming contest, conducting technical interview challenges, or collecting homework submissions, while BRUCE takes care of running and evaluating the code submissions. This will benefit application developers by enabling them to make use of automatic code execution and scoring, without requiring them to reinvent any wheels or worry about security. This will let developers focus on their user experience, instead of on their backend architecture.

Our system comprises five components:

1. Worker pool
2. Task queue
3. Database
4. API server
5. Use-case-specific web applications

First, some terminology: we segregate code executions into “problems” and “submissions.” A problem is defined by a configuration of how to run and judge the correctness of a submission for that problem. For example, a homework assignment can be configured as a problem, and each student would make a separate submission, all to be run against the same problem configuration. Problem configurations also include parameters like maximum execution time, point values for each test case, etc.

A problem is defined through the API and stored in the database, after which it will be possible to make submissions to that problem. A problem submission given to the API is then be turned into a job and passed to the task queue, where it waits for a worker to process it. The API server acts as the only point of entry to the system from the outside world and ultimately orchestrates the submission and reporting process via all the internal components.

The worker pool consists of multiple identical machines that are capable of executing code and determining results, and which are safely sandboxed to preserve system integrity. These machines are virtual cloud servers on services such as Amazon Elastic Compute Cloud or DigitalOcean. These services have APIs which will enable BRUCE to automatically shrink or expand the worker pool based on the size of the task queue.

The workers pull jobs from the task queue, and then query the database for the corresponding problem configuration in order to set up the code execution and its associated test cases. The worker then executes the code and logs the result in the database, creating a history of all code submissions and executions for each problem. After that process is done, the worker sends back a message to let the API server know that the results are ready.

The database is used as a shared data store by the API server, worker pool, and the task queue server. It stores problem configurations, submission and execution logs, information about the current state of the task queue and worker pool, and information about accounts registered to interact with the API. This database is the intermediary from the API reporting information and results to the clients and the workers doing the majority of the difficult work.

Finally, BRUCE implements a web application on top of this architecture, tailored to specific use cases. This application interacts with the API server to create problems and run submissions. One of BRUCE’s goals is to power the administration and scoring of a programming contest, and the web application facilitates this functionality. The web app enables team tracking, problem viewing, submission monitoring, and leaderboard viewing. The application also enforces rules that are beyond the scope of the API, such as submission restrictions and contest time limits.

The lifecycle of a submission is visualized in Figure 1.

4. SYSTEM IMPLEMENTATION

For the API itself, we decided to program the entire server using Node.js. Although we had originally planned on Python, we realized that the asynchronous nature of BRUCE gave
5. SYSTEM PERFORMANCE

For the database, we decided to use MongoDB over a more traditional SQL-like database. The reasons for this decision were three-fold. First, MongoDB's document-like data model fit well with how we planned on modeling our data. Second, we found that our data was not highly relational, and thus we would have not been properly taking advantage of relational database design if we had decided to go with a highly relational database. Third, MongoDB has an extension called GridFS, which is a built in file-system. This allows us to easily store and access the files that are being submitted along with submission records themselves. When looking at performance, although MongoDB is not as strong in some areas, and does run into some issues such as data loss at high load and scale, we decided that for our use case, this would be ok.

A crucial part of our system's infrastructure is the use of Docker. Docker is a open-source piece of software built on top of Linux containers that allows developers to effectively automate full deployment of any type of applications inside a container. There are many reasons we decided to use Docker, the most important of which being the ease of use of the software. Docker comes with many pre-packaged "images," which allow us to easily start up various applications using Docker. Second, Docker makes the sandboxing and security of our API quite easy, as we can simply start a new Docker container upon execution. The Docker containers guarantee complete isolation from all other parts of the application, which is exactly what we were looking for when searching for technologies to satisfy this challenge. The only other option to solve this problem would be to use virtual machines. However, VMs are much more heavyweight than the use case we were looking for; thus, Docker was a perfect choice.

For our task queue, we decided to use Redis to handle messaging. Redis is a simple key-value store with strong guarantees on persistence, reliability, replication, and performance, which suited our needs. Originally, we had planned on using RabbitMQ and Celery on top of Redis, but this did not integrate well with our Node.js/Docker system. Since our use case for the task queue was relatively simple, we found a Redis-based task queue library for Node.js, which was easy to use and lightweight, perfect for BRUCE.

In production, BRUCE is deployed on a group of Amazon Web Services (AWS) Elastic Cloud Compute (EC2) instances. Since BRUCE is entirely deployed using the Docker container system, there is very little setup overhead required to install BRUCE on a given instance, meaning that it is quite easy to spawn up new instances as BRUCE scales. Furthermore, we have made the production system secure by adding an SSL certificate and forcing all communication to use HTTPS.

5.1 PClassic Fall 2014

On December 6, we ran a trial run of BRUCE's performance at the PClassic programming contest. PClassic is a programming contest for high schoolers hosted by Penn where students answer problems using Java. While our system supports other languages like Python and Javascript currently, Java is the most complicated language we support thus far, which made this a good test. The contest organizers used BRUCE for all of their code execution needs over the course of the Saturday event. BRUCE ran over 1300 submissions across a span of nearly four hours in a production environment. Our system was able to determine the scores of over 100 teams at PClassic.

There were several periods of time during which we received ten submissions over the course of ten seconds. In all of these cases, our parallel worker pool handled the load with ease. We ran the system on a quad-core 8GB cloud server and found that the typical submission processing time was just under one second in addition to the actual runtime of the arbitrary code segments. That processing time was divided between Java compilation and startup of the Docker containers. In the upcoming weeks, we can reduce this time by keeping Docker containers waiting and ready to run rather than creating the container after the request comes in. Despite this, we are happy with our decision to go with Docker over using standard virtual machines; it is hard to imagine what the runtime performance would be if we were starting up a new virtual machine upon every execution.

This experience was valuable to our development. We encountered several problems at the beginning of the contest with race conditions related to multiple teams submitting at the same time (as expected in a distributed system at scale), and some unhandled exceptions that we did not foresee arising. Thankfully, we were able to fix these issues over the course of the day, which led to a better experience for all those involved.

Overall, we were encouraged by our trial run at the fall iteration of PClassic. It certainly gave us a long list of elements of BRUCE on which we needed to improve.

5.2 PClassic Spring 2015

On March 21, a second iteration of PClassic was held, and BRUCE was used again. BRUCE was set up with four worker cores, as in the previous iteration, and deployed with SSL on a dedicated usebruce.org domain (as opposed to an IP address in the fall). Although smaller in size (200 participants and 900 submissions processed), only one bug was discovered, as opposed to many in the fall iteration. Finally, the administrators created a much stronger admin console on top of BRUCE, which allowed for faster processing of submissions.

Overall, although the number of submissions processed was lower than in the fall, the version of BRUCE run at the spring PClassic fared much better than the fall version.

5.3 CIS 197

BRUCE was also used in the Spring 2015 iteration of CIS 197: Javascript at Penn. Compared to the four hours for each of the PClassic competitions, we had to make sure that BRUCE was up and able to process submissions for the entire semester without failure. Of course, although we had found some bugs while running BRUCE at PClassic, there
were some issues that arose while supporting BRUCE for CIS 197, especially given that we had to add support for Javascript (PClassic had only used Java up to that point). For example, we had to configure BRUCE to accept and run submissions containing zip files since some assignments were quite complex and required an undetermined number of files (for example, one assignment required creating an AngularJS application).

There were several periods of time (especially close to deadlines), during which many students would be submitting concurrently in short periods of time. By scaling the number of workers (see section 5.4), we were able to handle this load appropriately while maintaining a reasonable end-to-end processing time dependent on the complexity of the submissions. Furthermore, we were able to enhance the class experience for students by allowing them to receive real-time test and lint (style-checking) results on assignments. This allowed students to quickly iterate and improve on their submissions and re-submit, as opposed to the older model of creating one final submission and waiting for a long period of time (up to a couple of weeks) to receive feedback.

In terms of uptime, as of the 3/4 point in the semester, BRUCE had processed over 2800 total submissions for CIS 197; the system did go down at five points throughout the semester, but they were all within the first two months and all bugs were fixed. BRUCE has been running successfully without any failures for CIS 197 since February 2015.

Finally, we realized that running BRUCE for CIS 197 gave us an invaluable feedback loop with the instructors as they continuously gave comments on the system itself as well as how to improve the API to allow developers to easily create front-end systems on top of BRUCE, as was done for CIS 197. According to the estimates of the CIS 197 teaching assistants, BRUCE saved the teaching team roughly 100 hours of manual labor this semester thanks to its case of running submissions and reviewing test output to assign grades [14]. Overall, we believe that the success of BRUCE in CIS 197 bodes well for future use in other computer science classes.

5.4 Load Testing

A key metric of a distributed system is performance with many concurrent requests. Given that BRUCE is designed as a submission system, it is easy to envision scenarios where there may be many concurrent requests (i.e. close to a submission deadline). A graph of BRUCE’s performance with up to 10 concurrent requests can be seen in Figure 2. The test ran here measures end-to-end response time in milliseconds for a sample BRUCE submission that takes 200 milliseconds to run. The test suite was run on a backend system that consisted of 4 CPUs each running a worker.

These results are consistent with our expectations for the given system for two main reasons. The first is that much of the end-to-end response time (apart from running the program itself) is due to the overhead of creating an execution environment for the code itself. Since BRUCE has strict security requirements, the system is very careful in setting up the proper execution environment (namely, a new Docker container), which ends up accounting for most of the overhead (500ms). The second reason has to do with the shape of the graph. Up to the first four concurrent requests, the graph is flat, which makes sense given that the system consisted of 4 CPUs, so each CPU could be activated at a given time. The steady climb between five and eight concurrent requests is due to the fact that each processor core is alternating between executing one submission as well as setting up the execution environment for the next queued submission. Finally, there is a much steeper curve past 9 submissions because the processor cores are alternating between three or more activities, such as running multiple submissions or setting up multiple Docker containers at the same time.

The results from our load tests are in-line with what we expected: they indicate that performance is a function of the number of workers present in the system. This means that BRUCE administrators can handle load simply by increasing (or decreasing) the number of workers.

6. FUTURE WORK

Our trial run at both PClassics as well as in CIS 197 has shown us that we have a functioning service that has succeeded in a rigorous and mid-scale test. In the coming months, we will refine BRUCE by making it more technically robust and creating a more visually appealing web application. We will also add more functionality to the web application so it can be easily implemented by programming contest organizers and course staff.

More specifically, we will improve our handling of Docker containers such that we draw from a pool rather than create one when a request comes in. This choice will save us several hundred milliseconds of latency, according to our current estimates. The only downside is that we will need to be careful about how we monitor the pool to account for containers going down and other possible issues. We will also ensure that each worker handles more than one execution container at a time. Currently, each worker handles only one execution container, which means that not as much code can be run in parallel as we would like. For our current purposes, this is ok, but as our API scales to more and more users, this increases end-to-end wait time, which is not good. If we can have multiple workers running multiple containers, then we can greatly multiple the number of submissions that are running in parallel. Again, this will require more careful monitoring than the simple one-to-one worker to container model, but we are confident that we can circumvent this issue.

Additionally, we will make small improvements in the areas of code comparison and failure handling. We will implement hash comparison of arbitrary code segments such that if a file has already been uploaded that the system has already executed, CloudSubmit will simply return the results again instead of re-executing the code, so as not to waste processing time and instead move on to another submission. We will also account for the failure condition that we experienced at PClassic in which a code segment outputs large amounts of text and the system strains to record all of it. Of course, it is hard to anticipate every kind of error that could occur when a file is run, but we will design our system so that in case an unanticipated error does occur, the system handles it gracefully and continues on.

As we work to achieve these remaining tasks, we will use the following as evaluation criteria to gauge our progress and effectiveness at implementation.

1. Ease of integration into new or existing applications.
2. Quick processing of numerous simultaneous submissions.
3. Ability to scale worker pool up and down automatically.

4. Extensibility with plugins.

One can evaluate BRUCE by the complexity of the onboarding process for potential clients. It should be as easy as possible to set up an API account and integrate an application with the system. These processes should be well-documented to facilitate setup.

BRUCE should quickly evaluate a large sustained volume of submissions. The system itself should minimize its overhead, instead trying to limit the total turnaround time of a submission to the CPU time required by the submission code, plus a very small nearly-fixed factor for the latency of passing data around the distributed system.

An ideal outcome would be, for code submissions which run near-instantaneously, to turn the results around fast enough for interfaces like to provide near-real-time results. This would enable use cases such as in-browser code editors to provide the end user with a short iteration cycle.

Finally, BRUCE should be extensible. It should accept plugins to extend the functionality of the overall system. It should enable seamless scaling of the system to multiple API, database, and task queue servers. It should provide an easy means of updating the deployed system with changes to the codebase. There are also numerous possible extensions to the project - data analysis, automated style grading, and cheating detection are immediate possibilities. BRUCE’s system architecture needs to make it easy to head in new directions.

7. ETHICS

If BRUCE were to grow at scale, three major ethical issues would arise.

The first issue is a heightened security issue. Security is always an ethical issue, no matter what the scale, but BRUCE’s widespread success might make it a greater target for malicious attackers. We would need to take additional precautions to secure our system against intensified attacks. A breach of our system could result in the leaking of sensitive and private student data. Our users trust us to keep this information contained.

The second issue is the possibility that a course instructor manipulates the grades of students through the BRUCE dashboard. Unfortunately, there is not much we can do to prevent against this threat. It exists for any teacher who is assigning grades in any way to students.

The third issues is the most serious — the restriction of access to grade and score information to only those who need it. The course instructor obviously needs access to this information for logistical reasons, but technical admins who configured BRUCE for a course may have access that they could abuse. For example, if the IT department at a university sets up BRUCE (e.g. CETS here at Penn) and fails to explain the nature of our system fully to the teacher, IT employees could unethically access student data without the instructor’s knowing at a future point in time. Our main approach here is to define clear admin roles and educate users on this dangerous possibility in our documentation.

We experienced several of these issues when we had to get CETS certification before becoming the official submission system for CIS 197 during the spring semester.

8. CONCLUSIONS

Our team members are still strong believers in the “submission-as-a-service” platform that BRUCE represents. As we continue to see more learn-to-code, scratchpad, and code puzzle websites arise, we continue to be amazed by the fact that each of them has created its own code submission system. We think that BRUCE can be the start of a centralized and unified code submission system that is rapidly and easily deployable for all developers.

Furthermore, since our team has over twenty semesters of Penn teaching assistant experience, we believe even more
strongly that BRUCE has the potential to improve computer science education. We estimate that over 70,000 middle school and high school students each year will study computer science. Middle school and high school teachers often find themselves in a difficult dilemma: their schools do not have enough budget to buy them an enterprise code submission system, while grading assignments themselves (i.e. having students email them all homework files manually, downloading them, and then compiling and running them all locally) is both insecure and inefficient. Running arbitrary code locally can result in disaster depending on the type of assignment, and it takes a while to execute the overhead of opening emails, downloading files, running code, and then manually keeping track of submission results. We think that BRUCE offers an immediate solution to this problem, allowing middle school and high school teachers to improve their workflows and focus on the most important part of their jobs — teaching.

9. REFERENCES