Acta Informatica Pragensia, 2015, 4(3): 226–241
DOI: 10.18267/j.aip.71
Peer-reviewed paper

Security Measures in Automated Assessment System for Programming Courses

Jana Šťastná[*], Ján Juhár[*], Miroslav Biňas[*], Martin Tomášek[*]

Abstract

A desirable characteristic of programming code assessment is to provide the learner the most appropriate information regarding the code functionality as well as a chance to improve. This can be hardly achieved in case the number of learners is high (500 or more). In this paper we address the problem of risky code testing and availability of an assessment platform Arena, dealing with potential security risks when providing an automated assessment for a large set of source code. Looking at students' programs as if they were potentially malicious inspired us to investigate separated execution environments, used by security experts for secure software analysis. The results also show that availability issues of our assessment platform can be conveniently resolved with task queues. A special attention is paid to Docker, a virtual container ensuring no risky code can affect the assessment system security. The assessment platform Arena enables to regularly, effectively and securely assess students' source code in various programming courses. In addition to that it is a motivating factor and helps students to engage in the educational process.

Keywords: Automated assessment, Programming assignment, Unsafe code, Virtual environment, Docker, System availability.

1 Introduction

Automated assessment in programming assignments has been a center of attention in various studies which differ in applied strategies or technologies. Some focused on deployment of separate tools run from command line, others described complex systems providing a user interface (UI) over Internet, and finally, some are available as web services. As mentioned in a work of Ihantola et al. (2010) or Pears et al. (2007), one common problem is availability of particular assessment tool. Mostly, such tools are created to solve local problems, e.g. automated evaluation of student projects.

Programming courses and innovative assessment approaches have been discussed since at least 1997, when Thorburn and Rowe (1997) described in their work an automated system used for assessment of students' programs. They called it PASS (**P**rogram **A**ssessment using **S**pecified **S**olutions) and it checked if a student's program matches a solution plan provided to the system by a teacher. The solution plan was a description of proper program functionality at a higher level of abstraction – some kind of pseudo-code of desired solution. The representative solution plan was compared with a solution plan extracted from the student's

[*] Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9, 042 00 Košice, Slovak Republic
✉ jana.stastna@tuke.sk, jan.juhar@tuke.sk, miroslav.binas@tuke.sk, martin.tomasek@tuke.sk

program and if they matched, it indicated that the student's solution is implemented as desired and with full score. The score together with more detailed feedback were provided to the student.

It seems that PASS, the system mentioned above, inspired many other researchers and programming teachers, if not in functionality then at least in naming. Wang and Wong (2008) deal with computer-assisted learning in programming courses and they describe how they used the **P**rogramming **A**ssignment a**S**sessment **S**ystem (PASS) in their classes. Automatic assessment systems are used not only as a means to simplify the assessment. Law, Lee, & Yu (2010) provide a study which shows how PASS encourages and motivates students in learning.

Problems of automated assessment are also discussed in a work of Pieterse (2013). The author provides an extensive summary of issues regarding programming assignments assessment, especially in massive open online courses (MOOC) focused on teaching programing. With high numbers of attendees it is crucial to provide fast feedback and fair assessment of assignments, which is not possible to achieve without automation of the assessment process. The educational view on the evaluation of programming assignments is also discussed in a work of Biňas and Pietriková (2014) and Pietriková, Juhár, and Šťastná (2015).

Regardless of the preferred assessment tool it is crucial to discuss security issues associated with testing by execution of unknown and probably risky code submitted by students. Novice programmers will probably make mistakes in their code which could cause problems on the testing system or even render it inoperable. Automatic assignments evaluation platform should be resistant to such effects of risky code execution and availability of the evaluation service needs to be sustained also in case of numerous requests.

In this article we describe potential security risks related to verification of students' code, weaknesses of evaluation systems and our experiences with automatic evaluation platform *Arena* that is being developed at our department. We present one of several approaches for coping with security risks in *Arena* and describe how we ensured the system's confidentiality, integrity and availability.

2 Motivation

Our assessment platform *Arena* is based on web services with focus on its reusability and availability. However, execution of students' programming solutions may pose a problem unless appropriate security measures are applied.

2.1 The Arena platform

In our programming courses, we use an approach of programming custom computer games (Biňas & Pietriková, 2014). Generally, aspects of a programming language along with an individual game-related problems are dispensed gradually. This way students remain motivated while they are not potentially overburdened neither by the complexity of the language nor by the complexity of a particular problem. It also opens a way to get into the depth of language concepts as well as principles of given programming paradigm.

Our main intention with *Arena* platform is to build an effective learning environment leading towards training of good programmers. Maintaining attractive learning environment and its services is a contributing factor in increasing of students' motivation and enhancement of their engagement during the entire semester. Version control system in combination with

automated evaluator provide an effective as well as a fair assessment of large number of students.

Over the last year, *Arena* has been experimentally used with the programming languages C and Java within three courses, together covering over 1100 students. We currently have following two use cases in which *Arena* is being used.

- *Scheduled assessment* of students' projects they push to Git repository. This assessment is scheduled in a *cron* task that every few hours sends repositories matching a given name pattern for evaluation to *Arena*. Result of assessment is made available on a web page of *Arena* platform accessible with the knowledge of a randomly generated student identifier that is sent to the student by email after first assessment of his or her project.
- *Real-time assessment* during final exam. Students access a web page containing description of programming tasks and a web-based code editor where they implement their solutions. These solutions can be assessed on-demand multiple times during the exam - whenever student press the "submit" button. In this use case it is especially important to achieve as short evaluation times as possible.

With the help of *Arena* platform, students have a chance to fix errors and improve their overall score prior to a deadline. In the case of scheduled assessment our intention is to identify specific problems as early as possible rather than to assess a black-box once by the end of semester. For final exam, our approach gives students enough space for improvement within the exam time limit.

2.2 Services of the Arena platform

The top-level view on the *Arena* platform reveals a set of separate web services designed to cooperate with each other through REST-like[2] interfaces. They are displayed in Fig. 1 and can be characterized by the following description.

- *Arena*, as a service, represents user-facing web application through which the assessment results of students' projects are presented. This service is dependent on the *Gladiator*, as it expects the input data (the assessment results) in the format produced by this service.
- *Gladiator* service represents the key part of the *Arena* platform, its test runner. This service is responsible for running sets of tests provided by lecturers against students' projects and grade the results according to rules provided with tests. *Gladiator* is designed to be independent of the other services and to be usable for assessment of programs written in (almost) any programming language.
- *Conductor* is a small standalone service that checks the structure of a project against the declarative description of its expected content.
- *Spartan* is a web application providing real-time evaluation results of programming tasks presented and solved within a web browser. The main use case of this service is the automated assessment and grading of course exams. It depends on the *Gladiator* service for actual evaluation. The interface displays only the summary result of each evaluation and provides a URL address to full results available in *Arena* service.

[2] REST – Representational State Transfer, a software architectural style for web application interfaces.

These services, developed mostly in Python, have already been used in our courses. The separation of the processes involved in the automatic assessment of programming assignments that resulted in this set of services is a continuation of our work presented in (Biňas, 2014).

Although the platform is designed to be universal regarding the tests it is able to run, it is also designed for the needs of an educational environment. This is manifested for example in the terms *testcase*, *problemset*, and *submission* that have following meanings: a testcase represents a single task to be assessed and graded. In the most common case it is a test represented as a command to be executed with additional information as title, description, expected results and score assigned for correct solution. All the testcases form a problemset configuration. Problemset package provided by lecturer contains this configuration in a JSON (JavaScript Object Notation) file. It also contains all the files that are required to actually run the tests, e.g., implementation of unit tests and required libraries. Finally, a submission refers to the result of assessing student's project against a problemset that contains structured output of each testcase. It usually consists of standard output, error output and return code of a testcase command. More details on problemset package preparation and presentation of submission outputs are published in (Pietriková, Juhár & Šťastná, 2015).

Testcase runners in *Gladiator* are implemented as plugins. Currently there are two such plugins: *executable* testcase runner that runs a specified local command and *web service* testcase runner that calls remote service through HTTP (HyperText Transfer Protocol). The web service testcase runner is currently used only for checking structure of student's project with the *Conductor* service. In the following, we focus on the executable testcase runner, as there were several challenges in its implementation in both secure and performing way.

2.3 Security issues

Our evaluation platform works on a base of software testing (e.g., unit testing). In order to find out if a solution is correct, the source code from the submission is executed with predefined inputs. The first version of the executable testcase runner used system calls executed directly from within Gladiator's web server processes to run the tests specified in the problemset configuration. This posed two significant security problems:

- *Risky code problem* - unknown code was executed directly under operating system of the server (although under user with restricted system rights).
- *Availability problem* - during execution of tests the server process was unavailable to serve other requests. Server configuration was set up for 4 worker processes, which presented limit of 4 concurrently processable HTTP requests, while assessing one submission takes - depending on the test complexity - several seconds to complete.

Execution of unknown code poses a risk that students unwittingly or, in a worse case scenario, on purpose submit for evaluation code that performs harmful operations or that may subvert fair grading of test results or even damage the evaluation system. Even if we do not expect many submissions to include harmful operations, for security reasons we should handle students' programs as potentially malicious software. After all, code we evaluate comes from students learning to program and erroneous implementations should be expected. By taking some precautions it should be possible to prevent collapse of the assessment platform.

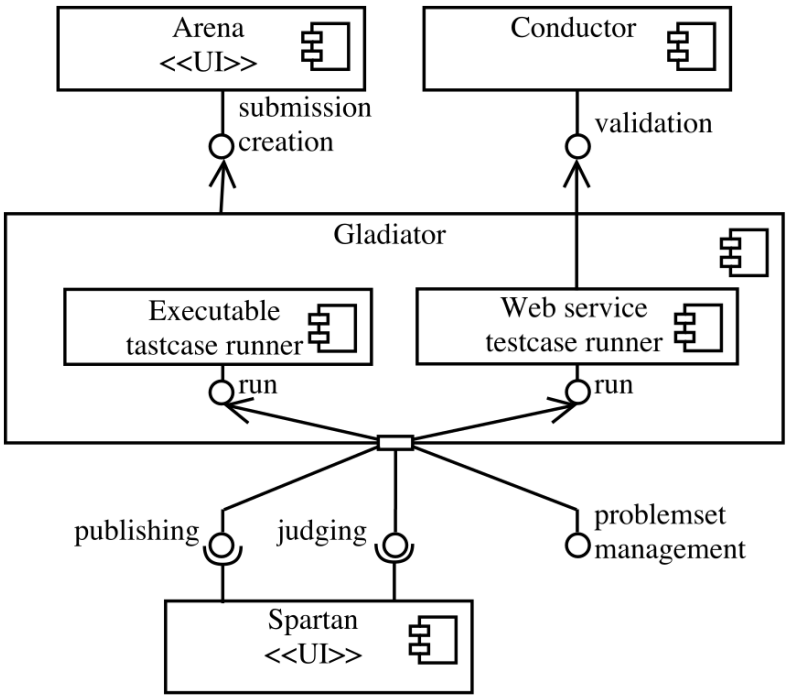

***Fig. 1**. Components of the Arena platform. Source: Authors.*

As for the *availability problem*, first usage of *Arena* was during scheduled assessment of students' project from single course and thus it was not affecting the system performance. Scheduled nature of this kind of assessment prevented the problem from occurring even when two courses were using the *Arena* during the next semester. However, first exposure of *Spartan* to real-time use case scenario, in which around 80 students had the possibility to evaluate their current solution on-demand, confirmed that the design is inappropriate and problem needs prompt solution. Web server running *Arena* services was easily overloaded with incoming requests [3] which caused long evaluation times and even developed into complete unavailability of the service.

Our motivation for this work lies in the effort to explore possible solutions of the above outlined problems. We also want to show how the selected technologies were integrated with our solution.

3 Execution of unknown programs

Depending on the specific harmful code fragments, execution of the program may have negative consequences on the system's primary security principles:

- Confidentiality – it is necessary to ensure that a submission and test results are available to the student-author and a lecturer. No other user should have access to these data by exploiting the evaluation system.
- Integrity – one student's submission cannot be modified in an undesired manner – neither by another student nor by other person involved in the course. Such undesired

[3] There were two requests for each submission, because *Spartan* communicated with *Gladiator* through another HTTP request.

alterations would lead to violation of information integrity core attributes (Boritz, 2005).

- Availability – sometimes even unintentional error in submitted source code, for example mistake in the implementation of the *while-do* cycle producing an infinite loop, can obstruct the evaluation process and render the evaluation system practically inoperable. This situation should be prevented in order to sustain availability of the evaluation service, mainly in the use case of course exams evaluation and grading.

Secure execution of potentially harmful software is an issue not only when assessing students' projects in programming courses but also in computer security research, especially in the field of malicious software analysis.

There are two general approaches to analysis of unknown code functionality:

1. static analysis, and
2. dynamic analysis.

3.1 Static analysis

Static analysis of source code as simply defined in the work of Landi (1992) is "the processes of extracting semantic information about a program at compile time". Determining program's functionality from its syntactic representation is a difficult task. In the case of students' submission analysis we consider static analysis for the purpose of determining program's partial correctness, since the total correctness of a program is generally an unsolvable problem (Wögerer, 2005).

Static analysis methods can be helpful in detection of various errors in code, however, checking program's functionality seems to be more complicated. For every testcase a valid program structure would have to be defined, not to mention the issue that some functionality may be programmed in numerous different ways.

In general, every program can comprise numerous execution paths, also called execution traces. The disadvantage of dynamic analysis is that only one execution trace can be observed at a time. On the other hand, static analysis can handle all of the traces, but this is viewed also as a disadvantage of static analysis (Beaucamps, Gnaedig, & Marion, 2012), because inspecting large number of program traces requires more processing time and memory space than only one execution trace.

While static analysis is practically safe, it does not meet the needs of our evaluation platform, although it may be useful as a minor analytic technique, for checking e.g. the submission's structure and proper program's construction.

3.2 Dynamic analysis

Techniques of dynamic program analysis require execution of the analyzed program, which is a drawback from security point of view, but offer results relatively quickly and in an easier way. Several security researchers consider dynamic analysis more reliable in obtaining program's real functionality than static analysis (Egele et al., 2012).

Concerning dynamic analysis, a good practice among malicious software researchers is to use separated environment for the purpose of programs' execution. This can be achieved by allocating either a physical system or a virtual system for this special purpose. Considering our resources the virtual system was the best choice.

3.3 Secure environments for unknown code testing

We looked at solutions used by malware analysts in search of suitable environment for execution and testing of unknown code. A lot of researchers rely on an analytic setting of virtual machines, as described in a work e.g. by Wagener, State, & Dulaunoy (2008). A potentially malicious code is executed and analyzed in a virtual environment without a risk of damaging the host system. After the analysis the environment can be safely destroyed and re-created with the initial settings. There are paid solutions available as well as free and open-source systems.

3.3.1 General and special-purpose virtualization system

One of the possibilities for establishing a secure environment for code execution, known as sandbox, is to use full virtualization technology. The advantage is that the environment is fully isolated from hosting operating system and such environment allows to run programs without any modifications, as if running directly in a normal operating system. Virtualization software like VirtualPC, VMWare and VirtualBox are built for general-purpose virtualization.

Alongside full virtualization sometimes a paravirtualization technology is used, also called "hardware-bound". Xen is one of the systems using that technology (Ormandy, 2007). The main difference, comparing with full virtualization, is that instructions of a program are executed by the physical central processing unit. In order to effectively share physical resources between the host and the guest operating system, several modifications need to be made on the virtual environment (Ferrie, 2007). These changes are detectable by programs running on the guest system, which is often undesired when analyzing malicious software.

Especially for the purpose of malware analysis and research some virtualization systems have evolved into specialized systems with analytical features. CWSandbox is a sandbox which enables automatization of malware behavior analysis by implementing hooking of Windows API (Application Programming Interface) functions calls (Willems, Holz, & Freiling, 2007). A similar analytic environment is Anubis (Bayer et al., 2009), which has formed into an advanced dynamic malware analysis platform. In addition to Windows API hooking, Anubis is able to inspect data flows and network traffic of analyzed samples and in this way collect information about program's behavior.

3.3.2 Hybrid virtual machine

The problem of executing unsafe programs is addressed in an unusual way in a work of Nishiyama (2012) which deals with source code written in programming language C. The code is executed through Java Native Interface which allows C functions execution from Java methods. Nishiyama proposes an improved Virtual Machine execution mechanism - a Hybrid Virtual Machine - which is a combination of Java bytecode interpreter and an engine for emulation of native code execution in a sandbox-like environment. Every time a native code is to be executed, a context manager changes the execution context for the emulation engine and so the code executes in a separate context. Then when Java code is to be executed again the context is switched back from emulation engine to the original Java thread.

Hybrid Virtual Machine is able to check and limit unsuspected system calls which are able to violate consistency of Virtual Machine. It is achieved either by preventing the system call to even execute or by limiting external resources used by it, e.g. by disabling access to certain system directories or limiting data transfer speed (Nishiyama, 2012). It is an interesting approach, however, it is limited only to C programming language, so it could not be used in our evaluation platform in the future for courses using different programming languages.

Even if virtualization provides a relatively safe separated environment for execution of unknown code, virtual systems and sandboxes have also bugs and weak points (Ray & Schultz, 2009) which generate opportunities for exploitation (Payer, Hartmann, & Gross, 2012). Therefore it is necessary to count with the fact that even the best virtualized environment is not 100% secure.

3.3.3 Virtual containers

A significant disadvantage of full virtualization technology, mentioned in sections above, is the time required to start virtual operating system inside the virtual machine, which can be in tens of seconds. This makes it not usable for our use cases.

A more lightweight solution is a container-based virtualization, of which Docker represents an increasingly popular option. It provides fast and secure virtualization as well as extensive API for working with virtual containers. As we can see in Fig. 2, the main difference compared to full virtualization is that container virtualization does not include full guest operating system in virtual environment.

Docker containers are built on top of *Linux containers*. When a container is started, *kernel namespaces* and *control groups* are created for it (Petazzoni, 2014). The provided virtual environment runs as an isolated process on the host operating system, shares kernel with the host OS and other containers, and comprises only the application and its dependencies. Control groups ensure that no single container can exhaust all system resources. Resource isolation and allocation benefits of virtualization are (for the most part) preserved while the solution is significantly more efficient. Container starts more quickly - usually in the sub-second range - and requires less system resources.

However, shared OS kernel adds one attack vector that is not present in full virtualization, specifically if a kernel vulnerability can be exploited or container is misconfigured and allows for elevation of user permissions. Also the Docker API needs to be secured to prevent its usage by unauthorized users.

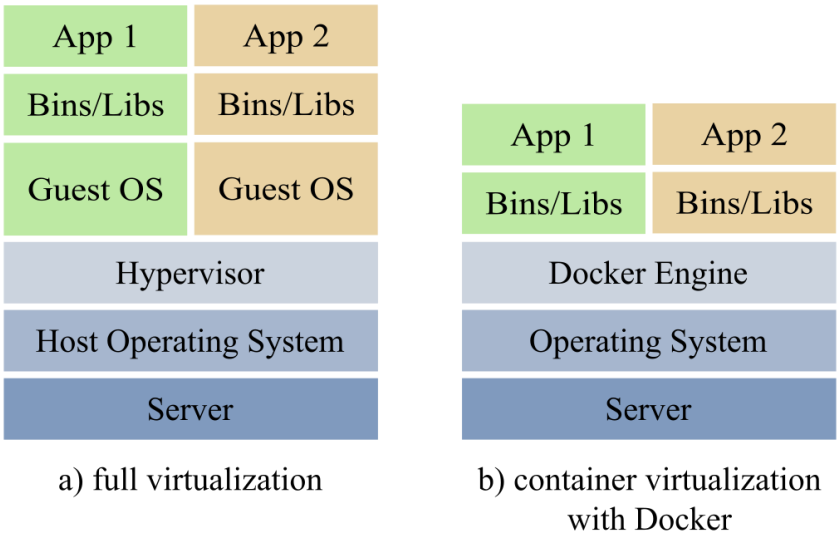

Fig. 2. *Difference between full virtualization and container virtualization. Source: Docker Inc. (2015)*

Besides extending Linux containers with more secure defaults, Docker containers provide benefits in terms of container image management. An image template can be specified within `Dockerfile`. It contains a sequence of commands used to assemble an image of container.

These commands can specify a base image to build upon (either from local or remote image repository), run commands to install required system packages, set up working directories, and more.

Based on discussed properties of multiple virtualization solutions, we decided to use Docker virtual containers. The decision was mainly influenced by its short startup times, good environment isolation, and convenient automation features, like `Dockerfile`.

4 Distributed task queues

Based on the behavior of *Arena* platform during real-time assessment use case and on the subsequent *availability problems*, we concluded that there are two specific changes to the platform architecture that should resolve these problems. The first is to move submission assessment execution off the main server processes. Such execution "in the background" will not block server processes and they will be able to serve other requests. The second change is to introduce a reliable mechanism that even under higher load will ensure that all received submissions will be evaluated. Properties of *distributed task queues* meet requirements of these changes.

Distributed task queues enable remote execution of tasks through message passing. Various implementations of this technique provide a wide range of capabilities, which include task scheduling, re-running of failed tasks or persistently storing results of tasks. As the *Arena* platform is implemented in Python, we looked at some Python task queue projects. According to Makai (2015) the following represent the most used ones:

- *Celery* – arguably the most advanced solution for Python. Celery supports many features like scheduling, handling of synchronous and asynchronous task, message routing, result storing. It also supports multiple message brokers and storage backends, like RabbitMQ[4] and Redis[5], which can be selected with regard to different features they provide.
- *RQ* (Redis Queue) – library for queueing tasks and processing them in background. As name implies, this library is backed by Redis.
- *Taskmaster* – task queue designed to handle large numbers of one-off tasks that are characterized by large amount of transferred data.
- *Huey* – simple task queue that depends only on Redis as its backend.

As Makai (2015) also notes, Celery is generally the library to go for, even though its usage is more complicated (due to larger number of included features) than with the other libraries. As we wanted to select solution that would provide enough possibilities for future extension, we decided to use this task queue in our implementation.

5 Integrating Docker and Celery with Gladiator

First version of executable testcase runner used direct system calls from the context of web server process. In POST request to `judge` REST endpoint of Gladiator service with student's submission and problemset identifier the problemset configuration was used to read

[4] Message broker with focus on reliability and high availability, see https://www.rabbitmq.com/

[5] Data structure storage server supporting publish/subscribe pattern, see http://redis.io/

individual commands and these were executed with Python's `subprocess` library. The only security measures applied were execution of the tests under user with restricted permissions and specification of maximal allowed duration for single testcase execution.

To solve the *availability problem* and the *problem of risky code* that we described earlier, we transformed *Gladiator* service into Celery workers that run executable testcases within Docker container.

5.1 Running tests with Docker

First added was the Docker support. Problemset configuration JSON file can specify the name of Docker image that will be used during assessment. It is also possible to include Dockerfile within the problemset package, which provides convenient way specifying requirements for test execution. This way, a lecturer - author of a problemset package - can set up the whole testing environment without needing direct access to the server where test are executed.

Within Gladiator we implemented a new class `Dockerizer` that serves for checking availability of configured images on the system, for building those images from `dockerfile` in case they are not available, and for starting containers with these prepared images. `Dockerizer` is implemented with the help of Docker API client for Python called docker-py[6]. Before the container is created, the content of problemset package is copied into temporary directory that is then configured as a *volume* on the container. This means that content of this directory is made available to otherwise isolated file system of the container. Container is also created with limited available memory.

The last required step for docker integration was to update executable testcase runner to use prepared Docker container. Main modification consisted of changing direct testcase command execution to sending the command into running Docker container. However, we still set the desired user and maximal allowed execution time of the test.

5.2 Gladiator as Celery worker

Integration of distributed task queue Celery with *Gladiator* started with configuration of the so-called *Celery application*. Its configuration consists of specifying message broker (we choose RabbitMQ) and task result backend (in our case Redis) and of various parameters related to message passing. We also needed to refactor code related to submission evaluation into functions independent on the web server-related code of *Gladiator* service that could be then marked as Celery tasks.

To minimize amount of data transferred through RabbitMQ queue, we set up a shared network storage server accessible through SSH. This storage is used for problemset packages as well as students' submissions retrieved by *Gladiator*, which are thus accessible for workers running on multiple different hosts.

[6] An official Docker API client for Python, see https://github.com/docker/docker-py

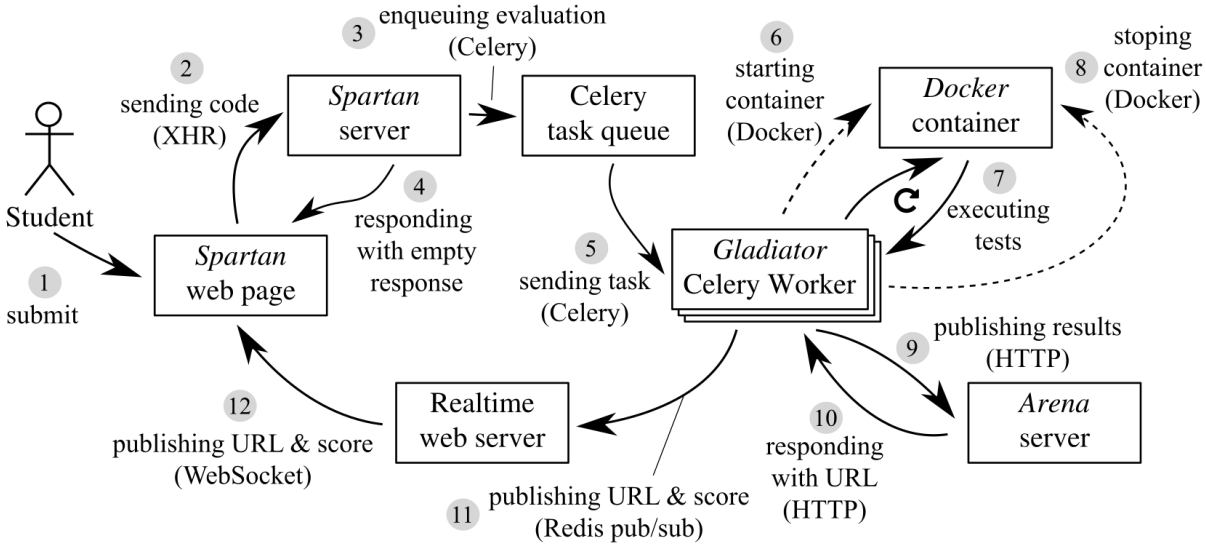

Fig. 3. *Submission evaluation in the real-time assessment scenario. Source: authors.*

REST API endpoint `judge` of the Gladiator's web server (the one that performs submission evaluation) was modified to call appropriate task of the Celery application. The call places a new task on the Celery queue. Three arguments are passed to this call: original name of submitted file, unique identifier of that file stored in shared storage, and identifier of problemset against which the submission should be evaluated. These arguments are retrieved by worker that is (by Celery internal mechanisms) selected to process the task, it can then fetch the submission and problemset files from shared storage and perform the evaluation of the submission.

5.3 Updated real-time assessment scenario

However, for all these changes in the implementation, there was still one obvious weak spot that prevented full benefit of task queue to show up. Even though submission evaluation was moved to one of the distributed celery workers, if the server needs to wait for the result the situation regarding availability is not really improved. And in a situation where all server workers are waiting for celery workers to finish their jobs, task queue is not going to help much either.

As we wanted to preserve REST-like *Gladiator* interface, it appeared that the problem would require usage of asynchronous web server. Such server would be able to wait for result of a worker job without blocking the process of handling other requests and to respond with the result after it is received. This would, of course, mean a significant rewrite of the whole web server part of *Gladiator*. However, as we stated in section 2.2, for *real-time assessment* use case we needed some more quickly realizable solution. Thus we decided to implement an alternative asynchronous communication channel usable with combination of the web browser and leave REST services synchronous for the moment (they would not be used). To this end a simple real time web server was created, which serves as a mediator for delivering submission result to browser through Redis publish-subscribe channel and WebSocket connection.

The whole process of on-demand submission evaluation during real-time scenario (final exam) is captured in Fig. 3 and can be described as follows.

1. A Student clicks on the "submit" button when he or she wants to evaluate their current solution.
2. A request with the student's implemented source code is sent to *Spartan* web server through XMLHttpRequest[7] (XHR) API.
3. Spartan saves source code to shared storage and enqueues submission evaluation and publication tasks directly in Celery task queue.
4. *Spartan* server sends an empty response to the original XHR request. This ensures that the connection is quickly closed.
5. Eventually, the tasks from the queue gets to one of running workers. Receiving worker retrieves submission code from shared storage.
6. Worker starts Docker container from image specified in problemset configuration.
7. Actual evaluation of the submission proceeds as a sequence of commands from problemset configuration is sent to the running Docker container and results of these command calls (i.e., stdout, stderr, return code) are collected for assessment.
8. Worker stops running Docker container.
9. Worker sends results of submission assessment to *Arena* service, where it is stored for viewing in accessible form by student.
10. Response from *Arena* service contains URL (Uniform Resource Locator) address where the results are available.
11. The URL address and the achieved submission score are combined into a simple JSON-formatted string and this message is sent through Redis publish/subscribe, where the real-time server can pick it up.
12. Real-time server sends the received JSON to student's browser through WebSocket connection that is open for entire duration of student's session on Spartan web page.

5.4 Results

Implementation changes described above were tested in an artificial situation where 48 computers were scripted to open *Spartan* web page and repeatedly send evaluation requests of tasks prepared for C programming course exam. *Gladiator* was configured to use 4 workers. At first, responses of the system (retrieval of the gained score and URL to full submission results) were in 3-4 second range. As the queue began to fill up with unprocessed tasks, responses slowed down and stabilized at 45-50 seconds. Although such times are not exactly acceptable for final exam, the experiment setup was rather unrealistic, because scripted *Spartan* page issued new request as soon as it retrieved response to the previous one. Moreover, response times would be easily reduced with more configured workers. On the positive side, we did not observe any request that would fail to be evaluated or its result to be delivered to the browser during the entire duration of the experiment (30 minutes). During the real final exam evaluation times rarely raised above 5 seconds. This showed significant improvement of platform availability over the original setup.

Evaluation of Docker integration benefits from the risky code problem perspective is more complicated to approach. Docker containers themselves do not prevent erroneous or malicious

[7] An API available in web browsers that enables in-background HTTP requests to a server.

code execution, but they should provide enough isolation from the rest of the system that any problems would have effect only on the container. And although we have not yet detected any attempts to compromise the system by submitting malicious code, by taking basic precautions against students' errors through limiting execution time and available memory, we managed to ensure better stability of the evaluation platform. Moreover, automation features of Docker, like `Dockerfile`, provide convenient means to configure and manage execution environments for different problemsets.

6 Related work and discussion

The approach to evaluation of students' programs presented by Thorburn and Rowe (1997) surely inspired other authors who needed to deal with the same problem. The idea of so called solution plan as a simple representation of program's functionality at a higher level of abstraction is interesting. We believe that simple programs in procedural languages can be safely checked whether they match a specific solution plan, however, in case of object-oriented programming complications should be expected. The nature of object-oriented languages gives programmers more freedom in how some functionality is expressed and usually these languages are used in larger projects which would be difficult to check with the technique of Thorburn and Rowe. Despite all this, their solution enables safe inspection of unknown code and could be considered in the future.

Concerning security issues with automatic evaluation of programming assignments, Pieterse (2013) assumes based on her previous experience that only a low amount of students' assignments are intentionally malicious, however, the potential threat remains in erroneous programs. While programming courses provided for university students have a limited amount of participants, massive open online courses have potentially unlimited attendance, so especially in the latter case even serious security breaches need to be considered and prevented.

A system developed by Pieterse (2013) and her colleagues is described in her paper. Similarly as in our approach they used a sandbox as a separated secured environment, in which functionality of students' programs is tested with a set of testcases. The testing procedure is described in general, but neither details concerning the testing environment nor specific security measures that they used are provided in the paper.

A problem with students' assignments evaluation, similar to ours, is addressed in the work of Špaček, Sohlich, and Dulík (2015). For secure testing of unknown students' programs they incorporated separated environment into their evaluation system called APAC. Similarly as in our work they chose Docker container for secure testing of programs.

There are several differences worth mentioning between our *Arena* platform and APAC. Beside the difference in programming language used for implementation and architecture of these systems, it seems that in their case Docker container and APAC run on different systems. It can improve the isolation of Docker from the main system, on the other hand, such isolation causes a communication delay.

Concerning students' submissions assessment process, Špaček et al. (2015) mention that the submitted source code is compiled still on the host system on which APAC application is running. Only then is the compiled program transferred to the Docker container. In contrast with their solution, our Docker container takes care also of the compilation of source code submitted by a student. This ensures greater level of protection against errors or attacks possible in compilation phase. Moreover, they use pool of prepared Docker containers that are

repeatedly used while they are available. Although in this way they may manage to shorten evaluation time (there are always some containers prepared to execute tests), our implementation creates new container for each submission being evaluated. The result is that each submission starts in an identical testing environment, unaffected by any changes that may remain from previous evaluations.

7 Conclusion

Beginner programmers many times learn on their own mistakes. Errors and bugs simply belong to programming. Even if targeted attacks are rare in students' source codes, we can learn from security experts who cope with malicious programs every day. Looking at students' programs as potentially malicious code led us to important security improvements in our automated assessment platform *Arena*.

By integrating Docker container into *Arena* the testing environment is separated from the rest of the system and thereby better secured. In this way the *risky code problem* is resolved. This solution is promising since it seems that employment of Docker is moving towards computer security domain, especially malicious software analysis (Zeltser, 2015). With growing popularity of containers like Docker also their implementation will improve, so their disadvantages against full virtualization solutions may eventually disappear.

Availability of platform services is another area in which we made improvements. These were achieved by using distributed task queue and workers that now handle scheduling of tasks and their actual execution, respectively.

There are still some unresolved issues that we want to address in our future work. Best practices for using Docker container securely are developing together with the technology itself. We will seek to adapt those to our implementation to ensure secure and stable operation of *Arena* platform. Also, the solution of *availability problem* presented in this work was focused on our real-time use case and scheduled assessment use case cannot yet fully benefit from the presence of task queue. Already outlined transformation of *Gladiator* web server to asynchronous operation is another of our future plans.

Another direction for future work is to enrich the *Arena* platform with additional methods for fair assessment of students' assignments. Beside the functionality of tested code, its quality can be evaluated also based on programmers' profiles (Pietriková & Chodarev, 2015).

We believe that *Arena* will provide the most appropriate information regarding functionality and quality of students' code as well as a chance to improve their programming skills.

Acknowledgement

This work was supported by project KEGA No. 019TUKE-4/2014 Integration of the Basic Theories of Software Engineering into Courses for Informatics Master Study Programmes at Technical Universities – Proposal and Implementation.

References

Bayer, U., Habibi, I., Balzarotti, D., Kirda, E., & Kruegel, C. (2009). A view on current malware behaviors. In *Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and m*ore (p. 8). Berkeley: USENIX Association Berkeley.

Beaucamps, P., Gnaedig, I., & Marion, J. Y. (2012). Abstraction-based malware analysis using rewriting and model checking. In *Proceedings of the 17th European Symposium on Research in Computer Security* (pp. 806-823). Berlin: Springer. doi: 10.1007/978-3-642-33167-1_46

Biňas, M., & Pietriková, E. (2014). Useful recommendations for successful implementation of programming courses. In *Proceedings of the 12th International Conference on Emerging eLearning Technologies and Applications* (pp. 397-401). New York: IEEE. doi: 10.1109/ICETA.2014.7107618

Biňas, M. (2014). Identifying web services for automatic assessments of programming assignments. In *Proceedings of the 12th International Conference on Emerging eLearning Technologies and Applications* (pp. 45-50). New York: IEEE. doi: 10.1109/ICETA.2014.7107547

Boritz, J. E. (2005). IS practitioners' views on core concepts of information integrity. International *Journal of Accounting Information Systems*, 6(4), 260-279. doi: 10.1016/j.accinf.2005.07.001

Docker Inc. (2015). *What is Docker*. Retrieved from https://www.docker.com/what-docker

Egele, M., Scholte, T., Kirda, E., & Kruegel, C. (2012). A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys*, 44(2), 6. doi: 10.1145/2089125.2089126

Ferrie, P. (2007). *Attacks on more virtual machine emulators.* Retrieved from https://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf

Ihantola, P., Ahoniemi, T., Karavirta, V., & Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (pp. 86–93). doi: 10.1145/1930464.1930480

Landi, W. (1992). Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems*, 1(4), 323-337. doi: 10.1145/161494.161501

Law, K. M.Y., Lee, V. C.S., & Yu Y.T. (2010). Learning motivation in e-learning facilitated computer programming courses. *Computers & Education*, 55 (1), 218-228. doi: 10.1016/j.compedu.2010.01.007

Makai, M. (2015). *Task Queues - Full Stack Python*. Retrieved from http://www.fullstackpython.com/task-queues.html

Nishiyama, H. (2012). Improved sandboxing for java virtual machine using hybrid execution model. In *Proceedings of the 6th International Conference on New Trends in Information Science and Service Science and Data Mining* (pp. 173-178). New York: IEEE.

Ormandy, T. (2007). *An empirical study into the security exposure to hosts of hostile virtualized environments.* Retrieved from http://taviso.decsystem.org/virtsec.pdf

Payer, M., Hartmann, T., & Gross, T.R. (2012). Safe Loading - A Foundation for Secure Execution of Untrusted Programs. In *Proceedings of the IEEE Symposium on Security and Privacy* (pp. 18-32). New York: IEEE. doi: 10.1109/SP.2012.11

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., & Paterson, J. (2007). A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin*, 39(4), 204-223. doi: 10.1145/1345375.1345441

Petazzoni, J. (2014). *Containers & Docker: How Secure Are They?* Retrieved from http://blog.docker.com/2013/08/containers-docker-how-secure-are-they/

Pieterse, V. (2013). Automated Assessment of Programming Assignments. In *Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research* (pp. 45-56). New York: ACM. doi: 10.1145/1559755.1559763

Pietriková, E., & Chodarev, S. (2015). Profile-driven Source Code Exploration. In *Proceedings of the IEEE Federated Conference on Computer Science and Information Systems* (pp. 929-934). New York: IEEE. doi: 10.15439/2015F238

Pietriková, E., Juhár, J., & Šťastná, J. (2015). Towards Automated Assessment in Game-Creative Programming Courses. *Proceedings of the 13th International Conference on Emerging eLearning Technologies and Applications* (pp. 307-312). Košice: TUKE.

Ray, E., & Schultz, E. (2009). Virtualization Security. In *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies* (pp. 42:1-42:5). New York: ACM. doi: 10.1145/1558607.1558655

Špaček, F., Sohlich, R., & Dulík, T. (2015). Docker as Platform for Assignments Evaluation. *Procedia Engineering*, 100, 1665-1671. doi: 10.1016/j.proeng.2015.01.541

Thorburn, G., & Rowe, G. (1997). PASS: An automated system for program assessment. Computers & Education, 29 (4), 195-206. doi: 10.1016/S0360-1315(97)00021-3

Wagener, G., State, R., & Dulaunoy, A. (2008). Malware behaviour analysis. Journal in Computer Virology, 4(4), 279-287. doi: 10.1007/s11416-007-0074-9

Wang, F.L., & Wong, T.-L. (2008), Designing Programming Exercises with Computer Assisted Instruction. In J. Fong, R. Kwan, & F.L. Wang (Eds.), *Lecture Notes in Computer Science: Hybrid Learning and Education* (pp. 283-293). Berlin: Springer. doi: 10.1007/978-3-540-85170-7_25

Willems, C., Holz, T., & Freiling, F. (2007). Toward automated dynamic malware analysis using CWSandbox. *IEEE Security & Privacy*, (2), 32-39.

Wögerer, W. (2005). *A Survey of Static Program Analysis Techniques*. Retrieved from http://www.ics.uci.edu/~lopes/teaching/inf212W12/readings/Woegerer-progr-analysis.pdf

Zeltser, L. (2015). *Security Risks and Benefits of Docker Application Containers*. Retrieved from https://zeltser.com/security-risks-and-benefits-of-docker-application/